Classes of Errors
BIOS Error-Handling Functions
Practical Error Handling
Character I/O Errors
Disk Errors
Improving Error Messages

Dealing with Hardware Errors

This chapter describes the enhancements you can make to improve CP/M's somewhat primitive error handling. It covers the general classes of errors that the BIOS may have to handle. It describes some of the underlying philosophical aspects of errors, how to detect them, and how to correct them or otherwise make the best of the situation.

At the end of the chapter are some example error-handling subroutines. Some of these have already been shown in the previous chapter as part of the enhanced BIOS (Figure 8-10); they are repeated here so that you can see them in isolation.

Classes of Errors

Basically, the user perceives only two classes of errors — those that are user-correctable and those that are not. There is a third, almost invisible class of errors—those that are recoverable by the hardware or software without the user's intervention.

The possible sources for hardware errors vary wildly from one computer system to another, since error detection is heavily dependent on the particular logic in the hardware. The BIOS can detect some hardware-related errors — mainly errors caused when something takes too long to happen, such as when a recalcitrant printer does not react in a specified length of time.

The BDOS has no built-in hardware detection code. It can detect *system* errors, such as an attempt to write to a disk file that is marked "Read-Only" in the file directory or attempts to access files that are not on the disk. These BDOS-detected errors, however, generally are unrelated to the well-being of the hardware. For example, a disk controller with a hardware problem could easily overwrite a sector of the directory, thereby deleting several files. This error would not show up until the user tried to use one of the now-departed files.

BIOS Error-Handling Functions

The error-handling code in the BIOS has to serve the following functions:

- Detection
- Analysis
- Indication
- Correction.

Error Detection

Clearly, before any later steps can be taken, an error must be detected. This can be done by the software alone or by the BIOS interacting with error-detecting logic in the hardware. In general, the only errors that the BIOS can detect unassisted are caused when certain operations take longer to complete than expected. Because the writer of the BIOS knows the operating environment of the specific peripherals in the system, the code can predict how long a particular operation should take and can signal an error when this time is exceeded. This would include such problems as printers that fail to react within a specified time period.

The BIOS can work in cooperation with the hardware to determine whether the hardware itself has detected an error. Armed with the hardware's specifications, the BIOS can input information on controller or device status to trigger error-detecting logic. How this should be done depends heavily on the peripheral devices in your computer system and the degree to which these devices have "smart" controllers capable of processing independently of the computer. Unfortunately, many manufacturers document the significance of individual status bits that indicate errors, but not combinations of errors, or what to do when a particular error occurs.

Error Analysis

Given that your BIOS has detected an error, it must first determine the class of error; that is, whether or not the error can be corrected by simply trying the operation again. Some errors appear at first to be correctable, but retrying the operation several times still fails to complete it. An example would be a check-sum error while reading a disk sector. If several attempts to read the sector all yield an error, then it becomes a "fatal" error. The code in your BIOS must be capable of initial classification and then subsequent reclassification if remedial action fails.

Other types of errors can be classified immediately as fatal errors—nothing can be done to save the situation. For example, if the floppy disk controller indicates that it cannot find a particular sector number on a diskette (due to an error in formatting), there is nothing that the BIOS can do other than inform the user of the problem and supply other helpful information.

Analysis of errors may require some basic research, such as inducing failures in the hardware and observing combinations of error indicators. For example, some printers (interfaced via a parallel port) indicate that they are "Out of Paper" or "Busy" when, in fact, they are switched off. The BIOS should detect this condition and tell the user to switch the printer on, not load more paper.

Error Indication

An incomplete or cryptic error message is infuriating. It is the functional equivalent of saying, "There has been an error. See if you can guess what went wrong!"

An error message, to be complete, should inform the recipient of the following:

- The fact that an error has occurred.
- Whether or not automatic recovery has been attempted and failed.
- The details of the error, if need be in technical terms to assist a hardware engineer.
- What possible choices the user has now.

To put these points into focus, consider the error message that can be output by CP/M after you have attempted to load a program by entering its name into the CCP. What you see on the console is the following dialog:

```
A>myprog<cr>
BAD LOAD
A>
```

All you know is that there has been an error, and you must guess what it is, even though the specific cause of the error was known to CP/M when it output the message. This error message is output by the CCP when it attempts to load a

".COM" file larger than the current transient program area. The message "BAD LOAD" is only understandable *after* you know what the error is. Even then, it does not tell you what went wrong, whether there is anything you can do about it, and how to go about doing it.

To be complete, this error message could say something like this:

```
A>myprog<cr>
     "MYPROG.COM" exceeds the available memory space by
     1,024 bytes, and therefore cannot be loaded under the
     current version of CP/M.
```

Notice how the message tells you what the problem is, and even quantifies it so that you can determine its severity (you need to get 1K more memory or reduce the program's size). It also tells you how you stand—you cannot load this program under the current version of CP/M, so retrying the operation is futile.

Not many systems programmers like to output messages like the example above. They argue that such a message is too long and too much work for something that does not happen often. Admittedly, the message *is* too long. It could be shortened to read

```
(131) Program 1,024 bytes too large to load.
```

This conveys the same information; the number in parentheses can serve as a reference to a manual where the full impact of the message should be described.

The major problem with the way error messages are designed is that they usually are written by programmers to be read by nontechnical lay users, and programmers are notoriously bad at guessing what nonexperts need to know.

Error indications you design should address the following issues, from the point of view of the user:

- The cause of the error
- The severity of the error
- The corrective action that has and can be taken.

Examine the error messages in the error processor for the example BIOS in Figure 8-10, from line 03600 onward. Although these are an improvement on the BDOS all-purpose

```
BDOS Error on A: Bad Sector
```

even these messages do not really meet all of the requirements of a good error message system.

Another often overlooked aspect of errors is that most hardware errors form a pattern. This pattern is normally only discernible to the trained eye of a hardware maintenance engineer. When these engineers are called to investigate a problem,

they will quiz the user to determine whether a given failure is an isolated incident or part of an ongoing pattern. This is why an error message should contain additional technical details. For example, a disk error message should include the track and sector used in the operation that resulted in an error. Only with these details can the engineer piece together the context of a failure or group of failures.

Error Correction

Given that a lucid error message has been displayed on the console, the user is still confronted with the question: "Now what do I do?" Not only can this be difficult for the user to answer, but also the particular solution decided upon can be hard for the BIOS to execute.

Normally, there are three possible options in response to errors:

- Try the operation again
- Ignore the error and attempt to continue
- Abort the program causing the error and return to CP/M.

For some errors, retrying can be effective. For example, if you forget to put the printer on-line and get a "Printer Timeout" error message, it is easy to put the printer back on-line and ask the BIOS to try again to send data to the printer.

Seldom can you ignore an error and hope to get sensible results from the machine; many disk controllers do not even transfer data between themselves and the disk drive if an error has been detected. Only ignorant users, or brave ones in desperation, ignore errors.

Aborting the program causing the error is a drastic measure, although it does escape from what could otherwise be a "deadly embrace" situation. For example, if you misassign the printer to an inactive serial port and turn on printer echoing (with the CONTROL-P toggle), you will send the system into an endless series of "Printer Timeout" messages. If you abort the program, the error handler in the BIOS executes a System Reset function (function 0) in the BDOS, CP/M warm boots, and control is returned to the CCP. In the process, the printer toggle is reset and the circle is broken.

Practical Error Handling

This section discusses several errors, describing their causes and the way in which the BIOS and the user can handle them when they occur.

Character I/O Errors

At the BIOS level, most detectable errors related to character input or output will be found by the hardware chips.

Parity Error

Parity, in this context, refers to the number of bits set to 1 in an 8-bit character. The otherwise unused eighth bit in ASCII characters can be set to make this number always odd, or alternatively, always even. Your computer hardware can be programmed to count the number of 1 bits in each character and to generate an error if the number is odd (odd parity) or, alternatively, if it is even (even parity). If the hardware on the other end of the line is programmed to operate in the same mode, parity checking provides a primitive error-detection mechanism—you can tell that a character is bad, but not what it should have been.

CP/M does not provide a standard mechanism for reporting a parity error, so your only option is to reset the hardware and substitute an ASCII DEL (7FH; delete) character in the place of the erroneous character.

If your BIOS is operating in a highly specialized environment, you may need to count the number of such parity errors so that a utility program can report on the overall performance of the system.

Framing Error

When an 8-bit ASCII character is transmitted over a serial line, the eight bits are transmitted serially, one after the other. A *start* bit is transmitted first, followed by the data character and then a *stop* bit. If the hardware fails to find the stop and start bits in the correct positions, a *framing error* will occur. Again, the only option available to the BIOS is to reset the hardware chip and substitute an ASCII DEL.

Overrun Error

This error occurs when incoming data characters arrive faster than the program can handle them, so that the last characters overrun those being processed by the hardware chip. This error can normally be avoided by the use of serial line protocols, such as those in the example BIOS in Figure 8-10.

An *overrun error* implies that the protocol has broken down. As with the parity and framing errors, almost the only option is to reset the hardware and substitute a DEL character.

Printer Timeout Error

This is one of the few errors where the BIOS can sensibly attempt an error recovery. The error occurs when the BIOS tries to output a character to a serial printer and finds that the printer is not ready for more than, say, 30 seconds. The most common cause of this error is that the user forgets to put the printer on-line. Many printers require that they be off-line during a manual form feed, and users will often forget to push the on-line button afterward.

After a 30-second delay, the BIOS can send a message to the console device(s) informing the user of the error and asking the user to choose the appropriate course of action. Note that console output can be directed to more than one device.

Parallel Printers

Printers connected to your system by means of a parallel port can indicate their status to the computer much more easily than can serial printers. They can communicate such error states as "Out of Paper," "End of Ribbon," and "Off-line."

These single-error indicators can also be used in combination to indicate whether the printer cable is connected, or even whether the printer is receiving power. You need to experiment, deliberately putting the printer into these states and reading status in order to identify them. It is misleading to indicate to the inexperienced user that the printer is "Out of Paper" when the problem is that the data cable has inadvertently become disconnected.

However, each of these errors can be dealt with in the same way as the serial printer's timeout problem: display an error message and request the user's choice of action.

Example Printer Error Routine

Figure 9-1 shows an example of a program that handles printer errors. It consists of several subroutines, including

- The error detection classification and indication routine
- The error correction routine.

It uses other subroutines that are omitted from the figure to avoid obscuring the logic. These subroutines are listed in full in the example BIOS in Figure 8-10.

```
                ;       This example shows, in outline form, how to handle the
                ;       situation when a serial printer remains busy for too long.
                ;       It is intended that this generic example show how to
                ;       deal with this class of errors.
                ;
                ;       The example presupposes the existence of a clock interrupt
                ;       every 16.666 milliseconds (1/60th of a second), and that
                ;       control will be transferred to the Real Time Clock service
                ;       routine each time the clock "ticks".
                ;
                ;       Figure 8-10 shows a more complete example, installed in a real
                ;       BIOS.
                ;
                ;
0000 =          B$System$Reset          EQU     0       ;BDOS system reset function
0005 =          BDOS                    EQU     5       ;BDOS entry point
                ;
0000 00         Printer$Timeout$Flag:   DB      0       ;This flag is set by the interrupt
                                                        ;  service subroutine that is called
                                                        ;  when the watchdog timer subroutine
                                                        ;  count hits zero (after having
                                                        ;  counted down a 30-second delay)

0708 =          Printer$Delay$Count     EQU     1800    ;Given a clock period of 16.666 ms
                                                        ;  this represents a delay of 30 secs
```

Figure 9-1. Serial printer error handling

```
                ;
000D =          CR                      EQU     0DH     ;Carriage return
000A =          LF                      EQU     0AH     ;Line feed
                ;
                Printer$Busy$Message:
0001 0D0A               DB      CR,LF
0003 5072696E74         DB      'Printer has been busy for too long,',CR,LF
0028 436865636B         DB      'Check that it is on-line and ready.',CR,LF,0
                ;
004E 00         Printer$Character:      DB      0       ;Save area for the data character
                                                        ;  to be output
                ;
                ;
                LIST:                                   ;<=== Main BIOS entry point
                        ;.......                        ;<=== I/O redirection code occurs here

004F 79                 MOV     A,C                     ;Save the data character
0050 324E00             STA     Printer$Character

                Printer$Retry:
0053 010807             LXI     B,Printer$Delay$Count   ;This is the count of the number
                                                        ;  of clock ticks before the watchdog
                                                        ;  subroutine call
0056 217E00             LXI     H,Printer$Timed$Out     ;  <== this address
0059 CDA300             CALL    Set$Watchdog            ;Sets the watchdog running

                Printer$Wait:
005C CDA300             CALL    Get$Printer$Status      ;See if the printer is ready to
                                                        ;  accept a character for output
                                                        ;  This includes checking if the printer
                                                        ;  is "Busy" because the driver is
                                                        ;  waiting for XON, ACK, or DTR to
                                                        ;  come high
005F C26C00             JNZ     Printer$Ready           ;The printer is now ready

0062 3A0000             LDA     Printer$Timeout$Flag    ;Check if the watchdog timer has
                                                        ;  hit zero (if it does, the
                                                        ;  watchdog routine will call
                                                        ;  the Printer$Timed$Out code
                                                        ;  that sets this flag)
0065 B7                 ORA     A
0066 C28400             JNZ     Display$Busy$Message    ;Yes, so display message to
                                                        ;  indicate an error has occurred
0069 C35C00             JMP     Printer$Wait            ;Otherwise, check if printer is
                                                        ;  now not busy

                Printer$Ready:                          ;The printer is now ready to output
                                                        ;  a character, but before doing so,
                                                        ;  the watchdog timer must be reset
006C F3                 DI                              ;Ensure no false timeout occurs
006D 010000             LXI     B,0                     ;This is done by setting the count
0070 CDA300             CALL    Set$Watchdog            ;  to zero
0073 FB                 EI

0074 3A4E00             LDA     Printer$Character       ;Get character to output
0077 11A300             LXI     D,Printer$Device$Table  ;DE -> device table for printer
007A CDA300             CALL    Output$Data$Byte        ;Output the character to the printer

007D C9                 RET                             ;Return to the BIOS's caller
                ;
                ;
                Printer$Timed$Out:                      ;Control arrives here from the
                                                        ;  watchdog routine if the
                                                        ;  watchdog count ever hits zero
                                                        ;  This is an interrupt service
                                                        ;  routine
                                                        ;All registers have been saved
                                                        ;  before control arrives here
007E 3EFF               MVI     A,0FFH                  ;Set printer timeout flag
0080 320000             STA     Printer$Timeout$Flag
0083 C9                 RET                             ;Return back to the watchdog

                                                        ;Interrupt service routine
```

Figure 9-1. (Continued)

```
                    ;
                    Display$Busy$Message:                           ;Printer has been busy for
                                                                    ;  30 seconds or more
0084 AF                     XRA     A                               ;Reset timeout flag
0085 320000                 STA     Printer$Timeout$Flag

0088 210100                 LXI     H,Printer$Busy$Message          ;Output error message
008B CDA300                 CALL    Output$Error$Message

008E CDA300                 CALL    Request$User$Choice             ;Displays a Retry, Abort, Ignore?
                                                                    ;  prompt, accepts a character from
                                                                    ;  the keyboard, and returns with the
                                                                    ;  character, converted to upper
                                                                    ;  case in the A register
0091 FE52                   CPI     'R'                             ;Check if Retry

0093 CA5300                 JZ      Printer$Retry
0096 FE41                   CPI     'A'                             ;Check if Abort
0098 CA9E00                 JZ      Printer$Abort
009B FE49                   CPI     'I'                             ;Check if Ignore
009D C8                     RZ
                    ;
                    Printer$Abort:
009E 0E00                   MVI     C,B$System$Reset                ;Issue system reset
00A0 C30500                 JMP     BDOS                            ;No need to give call as
                                                                    ;  control will not be returned
                    ;
                    ;
                    ;       Dummy subroutines
                    ;       These are shown in full in Figure 8-10. The line numbers in
                    ;       Figure 8-10 are shown in the comment field below
                    ;
                    Printer$Device$Table:           ;Line 01300 (example layout)
                    Request$User$Choice:            ;Line 03400
                    Output$Error$Message:           ;Line 03500
                    Get$Printer$Status:             ;Line 03900 (similar code)
                    Output$Data$Byte:               ;Line 05400 (similar code)
                    Set$Watchdog:                   ;Line 05800
```

Figure 9-1. Serial printer error handling (continued)

Disk Errors

Disks are much more complicated than character I/O devices. Errors are possible in the electronics and in the disk medium itself. Most of the errors concerned with electronics need only be reported in enough detail to give a maintenance engineer information about the problem. This kind of error is rarely correctable by retrying the operation. In contrast, media errors often can be remedied by retrying the operation or by special error processing software built into the BIOS. This chapter discusses this class of errors.

Media errors occur when the BIOS tries to read a sector from the disk and the hardware detects a check-sum failure in the data. This is known as a *cyclical redundancy check* (CRC) error. Some disk controllers execute a read-after-write check, so a CRC error can also occur during an attempt to write a sector to the disk.

With floppy diskettes, the disk driver should retry the operation at least ten times before reporting the error to the user. Then, because diskettes are inexpensive and replaceable, the user can choose to discard the diskette and continue with a new one.

With hard disks, the media cannot be exchanged. The only way of dealing with bad sectors is to replace them logically, substituting other sectors in their place.

There are two fundamentally different ways of doing this. Figure 9-2 shows the scheme known as sector sparing—substituting sectors on an outer track for a sector that is bad.

The advantage of this scheme is that it is dynamic. If a sector is found to be bad in a read-after-write check, even after several retries, then the data intended for the failing sector can be written to a spare sector. The failing sector's number is placed into a spare-sector directory on the disk. Thereafter, the disk drivers will be redirected to the spare sector every time an attempt is made to read or write the bad sector.

The disadvantage of this system is that the read/write heads on the disk must move out to the spare sector and then back to access the next sector. This can be a problem if you attempt to make a high-speed backup on a streaming tape drive (one that writes data to a tape in a single stream rather than in discrete blocks). The delay caused by reading the spare sector interrupts the data flow to the streaming tape drive.

You need a special utility program to manipulate the spare-sector directory, both to substitute for a failing sector manually and to attempt to rewrite a spare sector back onto the bad sector.

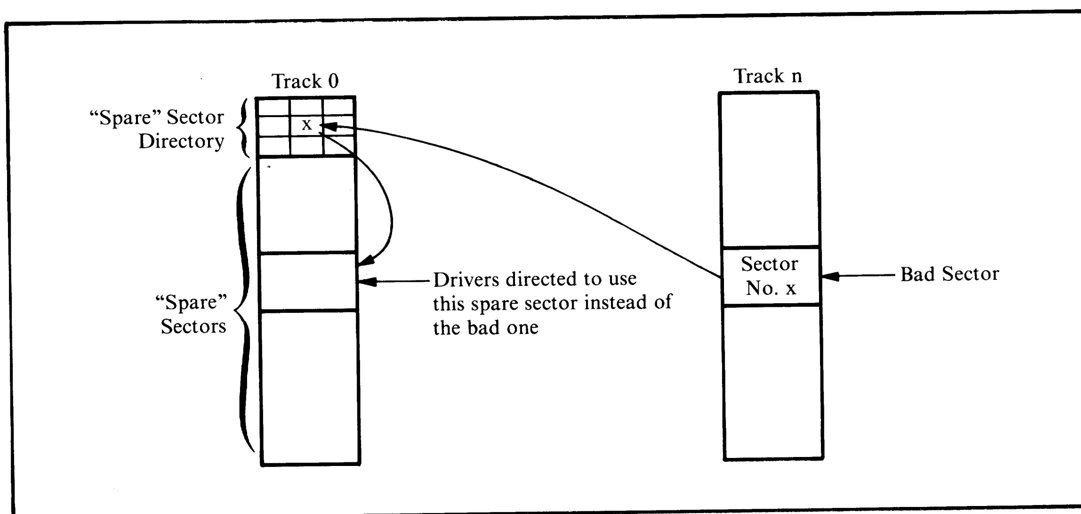

Figure 9-2. Sector sparing

Figure 9-3 shows another scheme for dealing with bad sectors. In this method, bad sectors are skipped rather than having sectors substituted for them.

The advantage of sector skipping is that the heads do not have to perform any long seeks. The failing sector is skipped, and the next sector is used in its place. Because of this, sector skipping can give much better performance. Data can be read off the disk fast enough to keep a streaming tape drive "fed" with data.

The disadvantage of sector skipping is that it does not lend itself to dynamic operation. The bad sector table is best built during formatting. Once data has been written to the disk, if a sector goes bad, all subsequent sectors on the disk must be "moved down one" to make space to skip the bad sector. On a large hard disk, this could take several minutes.

Example Bad Sector Management

Sector sparing and sector skipping use similar logic. Both require a spare-sector directory on each physical disk, containing the sector numbers of the bad sectors. This directory is read into memory during cold start initialization. Thereafter, all disk read and write operations refer to the memory-resident table to see if they are about to access a bad sector.

For sector sparing, if the sector about to be read or written is found in the spare directory, its position in the directory determines which spare sector should be read.

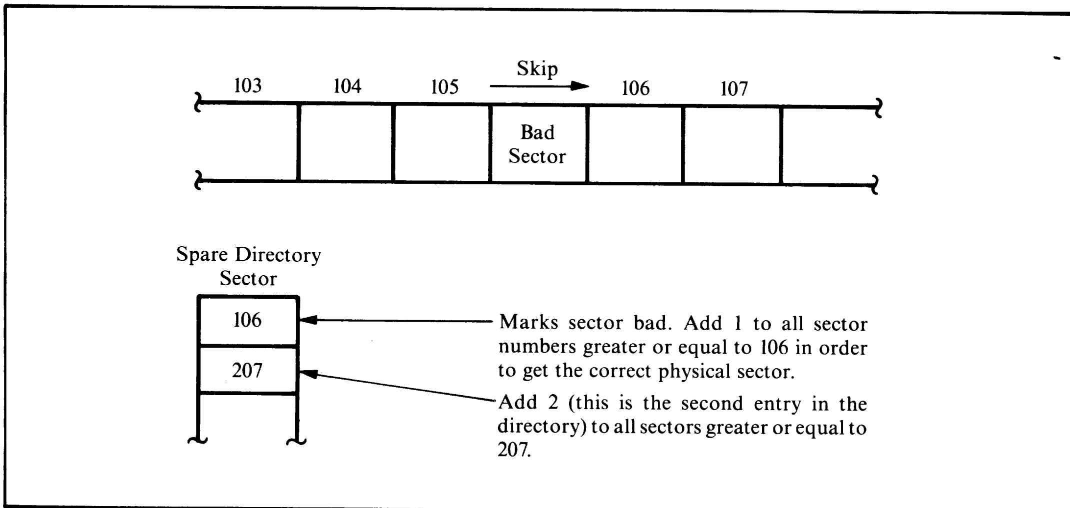

Figure 9-3. Sector skipping

In the case of sector skipping, every access to the disk makes the driver check the bad sector directory. The directory is used to tell how many bad sectors exist between the start of the disk and the failing bad sector. This number must be added to the requested track and sector to compensate for all the bad sectors.

The physical low-level drivers need four entry points:

- Read the specified sector without using bad sector management. This is used to read in the spare directory itself.
- Write the specified sector without using bad sector management. This is used to write the spare directory onto the disk, both to initialize it and to update it.
- Read and write the sector using bad sector management. These entry points are used for normal disk input/output.

Figure 9-4 shows the code necessary for both sector sparing and (using conditional code) sector skipping.

```
                ;       This example shows the modifications to be made in order
                ;       to implement bad sector management using sector sparing
                ;       and sector skipping.
                ;
0000 =          False           EQU     0
FFFF =          True            EQU     Not False
                ;
0000 =          Sector$Sparing  EQU     False
FFFF =          Sector$Skipping EQU     Not Sector$Sparing
                ;
                ;
                ;       Additional equates and definitions
                ;
                Spare$Directories:              ;Table of spare directory addresses
                                                ;Note: The directories themselves
                                                ;  are declared at the end of the
                                                ;  BIOS
0000 D500               DW      Spare$Directory$0       ;Physical disk 0
0002 9701               DW      Spare$Directory$1       ;Physical disk 1
                ;
                Spare$Dir$In$Memory:            ;Flags used to indicate whether spare
0004 00                 DB      0               ;  directory for a given physical disk
0005 00                 DB      0               ;  has been loaded into memory. Set by SELDSK
                ;
                ;
0000 =          Spare$Track             EQU     0       ;Track containing spare directory
                                                        ;  sectors
0004 =          Spare$Sector            EQU     4       ;Sector containing directory
0005 =          First$Spare$Sector      EQU     Spare$Sector + 1
                ;
                ;       Variables set by SELDSK
                ;
                Selected$Spare$Directory:
0006 0000                               DW      0       ;Pointer to directory
0008 00         Selected$Disk:          DB      0       ;Logical disk number
0009 00         Disk$Type:              DB      0       ;Floppy/hard disks
000A 00         Deblocking$Required:    DB      0       ;Deblocking flag
000B 00         Selected$Physical$Disk: DB      0       ;Physical disk number
                ;
000C 0000       Disk$Track:     DW      0       ;) These variables are part of the command
000E 00         Disk$Sector:    DB      0       ;) block handed over to the disk controller
```

Figure 9-4. Bad sector management

```
                        ;
8000 =                  Maximum$Track           EQU     32768   ;Used as a terminator
0012 =                  Sectors$Per$Track       EQU     18
0000 =                  First$Sector$On$Track   EQU     0
                        ;
                        ;
                        Disk$Parameter$Headers:
                                ;-------------------------
                                ;Standard DPH Declarations
                                ;-------------------------
                        ;
                        ;
                        ;       Equates for disk parameter block
                        ;
                        ;       The special disk parameter byte that precedes each disk
                        ;       parameter block, needs to be rearranged so that a
                        ;       physical disk drive number can be added.
                        ;
                        ;       Disk types
                        ;
                        ;                               vvvv--- Physical disk number
0010 =                  Floppy$5        EQU     0$001$0000B     ;5 1/4" mini floppy
0020 =                  Floppy$8        EQU     0$010$0000B     ;8" floppy (SS SD)
0030 =                  M$Disk          EQU     0$011$0000B     ;Memory disk
0040 =                  H$Disk$10       EQU     0$100$0000B     ;Hard disk - 10 megabyte
                        ;
0070 =                  Disk$Type$$Mask         EQU     0$111$0000B     ;Masks to isolate values
000F =                  Physical$Disk$Mask      EQU     0$000$1111B
                        ;
                        ;       Blocking/deblocking indicator
                        ;
0080 =                  Need$Deblocking EQU     1$000$0000B     ;Sector size > 128 bytes
                        ;
                        ;
                        ;       Disk parameter blocks
                        ;
                                ;----------------------------
                                ; Standard DPB's for A: and B:
                                ;----------------------------
                        ;
                                                        ;Logical disk C:
                                                        ;Extra byte indicates disk type
                                                        ; deblocking requirements and physical
                                                        ; disk drive.
000F C0                         DB      H$Disk$10 + Need$Deblocking + 0 ; Physical drive 0
                        Hard$5$Parameter$Block$C:
                                ;------------------------------
                                ;Standard format parameter block
                                ;------------------------------
                        ;
                        ;
0010 C0                         DB      H$Disk$10 + Need$Deblocking + 0 ; Physical drive 0
                        Hard$5$Parameter$Block$D:
                                ;------------------------------
                                ;Standard format parameter block
                                ;------------------------------
                        ;
                        ;
0004 =                  Number$of$Logical$Disks         EQU     4
                        ;
                        ;
                        SELDSK:                 ;Select disk in register C
                                                ;C = 0 for drive A, 1 for B, etc.
                                                ;Return the address of the appropriate
                                                ; disk parameter header in HL, or 0000H
                                                ; if the selected disk does not exist.
                                                ;
0011 210000                     LXI     H,0             ;Assume an error
0014 79                         MOV     A,C             ;Check if requested disk valid

0015 FE04                       CPI     Number$of$Logical$Disks
0017 D0                         RNC                     ;Return if > maximum number of disks
```

Figure 9-4. (Continued)

```
0018 320800             STA     Selected$Disk      ;Save selected disk number
                                                   ;Set up to return DPH address
001B 6F                 MOV     L,A                ;Make disk into word value
001C 2600               MVI     H,0
                                                   ;Compute offset down disk parameter
                                                   ; header table by multiplying by
                                                   ; parameter header length (16 bytes)
001E 29                 DAD     H                  ;*2
001F 29                 DAD     H                  ;*4
0020 29                 DAD     H                  ;*8
0021 29                 DAD     H                  ;*16
0022 110F00             LXI     D,Disk$Parameter$Headers       ;Get base address
0025 19                 DAD     D                  ;DE -> appropriate DPH
0026 E5                 PUSH    H                  ;Save DPH address

                                                   ;Access disk parameter block in order
                                                   ; to extract special prefix byte that
                                                   ; identifies disk type and whether
                                                   ; deblocking is required
                                                   ;
0027 110A00             LXI     D,10               ;Get DPB pointer offset in DPH
002A 19                 DAD     D                  ;DE -> DPB address in DPH
002B 5E                 MOV     E,M                ;Get DPB address in DE
002C 23                 INX     H
002D 56                 MOV     D,M
002E EB                 XCHG                       ;DE -> DPB

                SELDSK$Set$Disk$Type:
002F 2B                 DCX     H                  ;DE -> prefix byte
0030 7E                 MOV     A,M                ;Get prefix byte
0031 E670               ANI     Disk$Type$Mask     ;Isolate disk type
0033 320900             STA     Disk$Type          ;Save for use in low-level driver
0036 7E                 MOV     A,M                ;Get another copy of prefix byte
0037 E680               ANI     Need$Deblocking        ;Isolate deblocking flag
0039 320A00             STA     Deblocking$Required    ;Save for use in low-level driver

                                                   ;Additional code to check if spare
                                                   ;  directory for given disk has already
                                                   ;  been read in.

003C 7E                 MOV     A,M                ;Get physical disk number
003D E60F               ANI     Physical$Disk$Mask
003F 320B00             STA     Selected$Physical$Disk  ;Save for low-level drivers

0042 5F                 MOV     E,A                ;Make into word
0043 1600               MVI     D,0
0045 210400             LXI     H,Spare$Dir$In$Memory   ;Make pointer into table
0048 19                 DAD     D

0049 7E                 MOV     A,M                ;Get flag
004A B7                 ORA     A
004B C27700             JNZ     Dir$In$Memory      ;Spare directory already in memory
004E 34                 INR     M                  ;Set flag

004F 210000             LXI     H,Spare$Directories     ;Create pointer to spare
0052 19                 DAD     D                       ;  spare directory (added twice
0053 19                 DAD     D                       ;  as table has word entries)
                                                        ;HL -> word containing directory addr.
0054 5E                 MOV     E,M
0055 23                 INX     H
0056 56                 MOV     D,M                     ;Spare directory address in DE
0057 EB                 XCHG                            ;HL -> spare directory

0058 220600             SHLD    Selected$Spare$Directory ;Save for use in physical
                                                         ;  drivers later on

005B 110000             LXI     D,Spare$Track           ;Track containing spare directory
005E 3A0B00             LDA     Selected$Physical$Disk
0061 47                 MOV     B,A
0062 3E04               MVI     A,Spare$Sector          ;Sector containing spare directory
0064 0E18               MVI     C,Spare$Length/8        ;Number of bytes in spare directory / 8
0066 CDD500             CALL    Absolute$Read           ;Read in spare directory - without
                                                        ;  using bad sector management
```

Figure 9-4. (Continued)

```
0069 2A0600            LHLD    Selected$Spare$Directory ;Set end marker
006C 11C000            LXI     D,Spare$Length          ;  at back end of spare directory
006F 19                DAD     D
0070 110080            LXI     D,Maximum$Track         ;Use maximum track number
0073 73                MOV     M,E
0074 23                INX     H
0075 3602              MVI     M,D

               Dir$In$Memory:
0077 E1                POP     H               ;Recover DPH pointer
0078 C9                RET
               ;
               ;
               ;       In the low-level disk drivers, the following code must be
               ;       inserted just before the disk controller is activated to
               ;       execute a read or a write command.
               ;
0079 2A0C00            LHLD    Disk$Track              ;Get track number from disk
                                                       ;  controller command table
007C EB                XCHG                            ;DE = track
007D 2A0600            LHLD    Selected$Spare$Directory ;HL -> spare directory
0080 2B                DCX     H                       ;Back up one entry
0081 2B                DCX     H                       ;  (3 bytes)
0082 2B                DCX     H

0083 3A0E00            LDA     Disk$Sector             ;Get sector number
0086 4F                MOV     C,A                     ;Save for later

0087 06FF              MVI     B,0FFH                  ;Set counter (biased -1)

               Check$Next$Entry:
0089 23                INX     H                       ;Update to next (or first) entry
               Check$Next$Entry1:
008A 23                INX     H
               Check$Next$Entry2:
008B 23                INX     H

008C 04                INR     B                       ;Update count

                       IF      Sector$Sparing
                                                       ;If sparing is used, the
                                                       ;  end of the table is indicated
                                                       ;  by an entry with the track number
                                                       ;  = to maximum track number
                       LXI     D,Maximum$Track         ;Get maximum track number
                       CALL    CMPM                    ;Compare DE to (HL), (HL+1)
                       JZ      Not$Bad$Sector          ;End of table reached
                       ENDIF

                                                       ;Note: For sector skipping
                                                       ;  the following search loop will
                                                       ;  terminate when the requested track
                                                       ;  is less than that in the table.
                                                       ;This will always happen when the
                                                       ;  maximum track number is encountered
                                                       ;  at the end of the table.
008D EB                XCHG                            ;DE -> table entry
008E 2A0C00            LHLD    Disk$Track              ;Get requested track
0091 EB                XCHG                            ;DE = req. track, HL -> table entry
0092 CDCD00            CALL    CMPM                    ;Compare req. track to table entry


                       IF      Sector$Sparing          ;Use the following code for
                                                       ;  sector sparing
                       JNZ     Check$Next$Entry        ;Track does not match
                       INX     H                       ;HL -> MS byte of track
                       INX     H                       ;HL -> sector
                       MOV     A,C                     ;Get requested sector
                       CMP     M                       ;Compare to table entry
                       JNZ     Check$Next$Entry2       ;Sector does not match

                                                       ;Track and sector match, so
                                                       ;  substitute spare track and
                                                       ;  appropriate sector
```

Figure 9-4. (Continued)

```
                        LXI     H,Spare$Track           ;Get track number used for spare
                                                        ;  sectors
                        SHLD    Disk$Track              ;Substitute track

                        MVI     A,First$Spare$Sector    ;Get first sector number
                        ADD     B                       ;Add on matched directory
                                                        ;  entry number
                        STA     Disk$Sector             ;Substitute sector
                        ENDIF

                        IF      Sector$Skipping         ;Use the following code for
                                                        ;  sector skipping
                                                        ;The object is to find the
                                                        ;  entry in the table which
                                                        ;  is greater or equal to the
                                                        ;  requested sector/track

0095 CA9E00             JZ      Tracks$Match            ;Possible match of track and sector
0098 D2AC00             JNC     Compute$Increment       ;Requested track < table entry
009B C38900             JMP     Check$Next$Entry        ;Requested track > table entry

                Tracks$Match:
009E 23                 INX     H                       ;HL -> MS byte of track
009F 23                 INX     H                       ;HL -> sector
00A0 77                 MOV     M,A                     ;Get sector from table

00A1 B9                 CMP     C                       ;Compare with requested sector
00A2 CAAB00             JZ      Sectors$Match           ;Track/sector matches
00A5 D2AC00             JNC     Compute$Increment       ;Req. trk/sec < spare trk/sec
00A8 C38B00             JMP     Check$Next$Entry2       ;Move to next table entry

                Sectors$Match:
00AB 04                 INR     B                       ;If track and sectors match with
                                                        ;  a table entry, then an additional
                                                        ;  sector must be skipped

                Compute$Increment:
                                                        ;B contains number of cumulative
                                                        ;  number of sectors to skip
00AC 79                 MOV     A,C                     ;Get requested sector
00AD 80                 ADD     B                       ;Skip required number
00AE 0612               MVI     B,Sectors$Per$Track     ;Determine final sector number
                                                        ;  and track increment
00B0 CDC300             CALL    DIV$A$BY$B              ;Returns C = quotient, A = remainder
00B3 320E00             STA     Disk$Sector             ;A = new sector number

00B6 59                 MOV     E,C                     ;Make track increment a word
00B7 1600               MVI     D,0
00B9 2A0C00             LHLD    Disk$Track              ;Get requested track
00BC 19                 DAD     D                       ;Add on increment
00BD 220C00             SHLD    Disk$Track              ;Save updated track
                        ENDIF

                Not$Bad$Sector:
                                                        ;Either track/sector were not bad,
                                                        ;  or requested track and sector have
                                                        ;  been updated.
00C0 C3D500             JMP     Read$Write$Disk         ;Go to physical disk read/write
                ;
                        IF      Sector$Skipping
                                                        ;Subroutine required for skipping
                                                        ;  routine

                ;
                ;
                ;       DIV$A$BY$B
                ;       Divide A by B
                ;
                ;       This routine divides A by B, returning the quotient in C
                ;       and the remainder in A.
                ;
                ;       Entry parameters
                ;
                ;               A = dividend
                ;               B = divisor
                ;
                ;       Exit parameters
```

Figure 9-4. (Continued)

```
                 ;
                 ;               A = remainder
                 ;               C = quotient
                 ;
                 DIV$A$BY$B:
00C3 0E00                MVI     C,0             ;Initialize quotient
                 DIV$A$BY$B$Loop:
00C5 0C                  INR     C               ;Increment quotient
00C6 90                  SUB     B               ;Subtract divisor
00C7 F2C500              JP      DIV$A$BY$B$Loop ;Repeat if result still +ve
00CA 0D                  DCR     C               ;Correct quotient
00CB 80                  ADD     B               ;Correct remainder
00CC C9                  RET
                         ENDIF
                 ;
                 ;       CMPM
                 ;       Compare memory
                 ;
                 ;       This subroutine compares the contents of DE to (HL) and (HL+1)
                 ;       returning with the flags as though the subtraction (HL) - DE
                 ;       were performed.
                 ;
                 ;       Entry parameters
                 ;
                 ;               HL -> word in memory
                 ;               DE = value to be compared
                 ;
                 ;       Exit parameters
                 ;
                 ;               Flags set for (HL) - DE
                 ;
                 CMPM:
00CD 7E                  MOV     A,M                     ;Get MS byte
00CE BA                  CMP     D
00CF C0                  RNZ                             ;Return now if MS bytes unequal
00D0 23                  INX     H                       ;HL -> LS byte
00D1 7E                  MOV     A,M                     ;Get LS byte
00D2 BB                  CMP     E
00D3 2B                  DCX     H                       ;Return with HL unchanged
00D4 C9                  RET

                 ;
                 ;
                 Absolute$Read:
                                         ;The absolute read (and write) routines
                                         ;  access the specified sector and track
                                         ;  without using bad sector management.
                 ;       Entry parameters
                 ;
                 ;               HL -> Buffer
                 ;               DE = Track
                 ;               A = Sector
                 ;               B = Physical disk drive number
                 ;               C = Number of bytes to read / 8
                 ;
                 ;       Set up disk controller command block with parameters in
                 ;       registers, then initiate read operation by falling through
                 ;       into Read$Write$Disk code below.
                 ;
                 ;
                 Read$Write$Disk:

                         ;-------------------------------------------------
                         ;The remainder of the low level disk drivers follow,
                         ;  reading the required sector and track.
                         ;-------------------------------------------------

                 ;
                 ;       Spare directory declarations
                 ;
                 ;       Note: The disk format utility creates an initial spare
                 ;       directory with track/sector entries for those track/sectors
                 ;       that it finds are bad. It fills the remainder of the
                 ;       directory with 0FFH's (these serve to terminate the
                 ;       searching of the directory).
```

Figure 9-4. (Continued)

```
                 ;
                 ;
00C0 =           Spare$Length    EQU     64 * 3        ;64 Entries, 3 bytes each
                                                       ;  Byte 0,1 = track
                                                       ;  Byte 2 = sector

                 Spare$Directory$0:
00D5                     DS      Spare$Length     ;Spare directory itself
0195                     DS      2                ;Set to maximum track number by SELDSK as
                                                  ;  a safety precaution. The FORMAT utility
                                                  ;  puts the maximum track number into all
                                                  ;  unused entries in the spare directory.

                 Spare$Directory$1:
0197                     DS      Spare$Length     ;Spare directory itself
0257                     DS      2                ;End marker
```

Figure 9-4. Bad sector management (continued)

Improving Error Messages

The final extension to BIOS error handling discussed here is in disk-driver error-message handling. The subroutine shown in the example BIOS in Figure 8-10, although a significant improvement on the messages normally output by the BDOS, did not advise the user of the most suitable course of action for each error. Figure 9-5 shows an improved version of the error message processor.

```
                 ;       This shows slightly more user-friendly error processor
                 ;       for disk errors than that shown in the enhanced BIOS
                 ;       in Figure 8-10.
                 ;       This version outputs a recommended course of action
                 ;       depending on the nature of the error detected.
                 ;       Code that remains unchanged from Figure 8-10 has been
                 ;       abbreviated.
                 ;
                 ;       Dummy equates and data declarations needed to get
                 ;       an error free assembly of this example.
                 ;
0001 =           Floppy$Read$Code        EQU     01H     ;Read command for controller
0002 =           Floppy$Write$Code       EQU     02H     ;Write command for controller
                 ;
0000 00          Disk$Hung$Flag:         DB      0       ;Set NZ when watchdog timer times
                                                         ;  out
0258 =           Disk$Timer              EQU     600     ;10-second delay (16.66ms tick)
                 ;
0043 =           Disk$Status$Block       EQU     43H     ;Address in memory where controller
                                                         ;  returns status
                                                         ;Values from controller command table
0001 00          Floppy$Command:         DB      0
0002 00          Floppy$Head:            DB      0
0003 00          Floppy$Track:           DB      0
0004 00          Floppy$Sector:          DB      0
```

Figure 9-5. User-friendly disk-error processor

```
0005 00         Deblocking$Required:    DB      0       ;Flag set by SELDSK according
                                                        ;  to selected disk type

0006 00         Disk$Error$Flag:        DB      0       ;Error flag returned to BDOS
                ;
0007 00         In$Buffer$Disk:         DB      0       ;Logical disk Id. relating to current
                                                        ;  disk sector in deblocking buffer
                ;
                ;       Equates for Messages
                ;
0007 =          BELL    EQU     07H     ;Sound terminal bell
000D =          CR      EQU     0DH     ;Carriage return
000A =          LF      EQU     0AH     ;Line feed
                ;
0005 =          BDOS    EQU     5       ;BDOS entry point (for system reset)
                ;
                ;
                ;
                ;
                No$Deblock$Retry:
                        ;------------------------------------------------
                        ; Omitted code to set up disk controller command table
                        ; and initiate the disk operation
                        ;------------------------------------------------
0008 C31500             JMP     Wait$For$Disk$Complete
                ;
                ;
                Write$Physical:                 ;Write contents of disk buffer to
                                                ;  correct sector
000B 3E02               MVI     A,Floppy$Write$Code     ;Get write function code
000D C31200             JMP     Common$Physical ;Go to common code
                Read$Physical:                  ;Read previously selected sector
                                                ;  into disk buffer
0010 3E01               MVI     A,Floppy$Read$Code      ;Get read function code
                Common$Physical:
0012 320100             STA     Floppy$Command  ;Set command table

                ;
                Deblock$Retry:                  ;Re-entry point to retry after error
                        ;------------------------------------------------
                        ; Omitted code sets up disk controller command block
                        ; and initiates the disk operation
                        ;------------------------------------------------
                ;
                Wait$For$Disk$Complete:         ;Wait until disk status block indicates
                                                ;  operation has completed, then check
                                                ;  if any errors occurred
                                                ;On entry HL -> disk control byte
0015 AF                 XRA     A                       ;Ensure hung flag clear
0016 320000             STA     Disk$Hung$Flag

0019 213100             LXI     H,Disk$Timed$Out        ;Set up watchdog timer
001C 015802             LXI     B,Disk$Timer            ;Time delay
001F CD3B03             CALL    Set$Watchdog
                Disk$Wait$Loop:
0022 7E                 MOV     A,M                     ;Get control byte
0023 B7                 ORA     A
0024 CA3700             JZ      Disk$Complete           ;Operation done

0027 3A0000             LDA     Disk$Hung$Flag          ;Also check if timed out
002A B7                 ORA     A
002B C29F02             JNZ     Disk$Error              ;Will be set to 40H

002E C32200             JMP     Disk$Wait$Loop

                Disk$Timed$Out:                 ;Control arrives here from watchdog
                                                ;  routine itself -- so this is effectively
                                                ;  part of the interrupt service routine.
0031 3E40               MVI     A,40H                   ;Set disk hung error code
0033 320000             STA     Disk$Hung$Flag          ;  into error flag to pull
                                                        ;  control out of loop
0036 C9                 RET                             ;Return to watchdog routine
```

Figure 9-5. (Continued)

```
                   Disk$Complete:
0037 010000                LXI     B,0                      ;Reset watchdog timer
                                                            ;HL is irrelevant here
003A CD3B03                CALL    Set$Watchdog

003D 3A4300                LDA     Disk$Status$Block        ;Complete -- now check status
0040 FE80                  CPI     80H                      ;Check if any errors occurred
0042 DA9F02                JC      Disk$Error               ;Yes
                   ;
                   Disk$Error$Ignore:
0045 AF                    XRA     A                        ;No
0046 320600                STA     Disk$Error$Flag          ;Clear error flag
0049 C9                    RET

                   ;
                   ;       Disk error message handling
                   ;
                   ;
                   Disk$Error$Messages:             ;This table is scanned, comparing the
                                                    ; disk error status with those in the
                                                    ; table. Given a match, or even when
                                                    ; the end of the table is reached, the
                                                    ; address following the status value
                                                    ; points to the correct advisory message text.
                                                    ; Following this is the address of an
                                                    ; error description message.
004A 40                    DB      40H
004B B0019500              DW      Disk$Advice1,Disk$Msg$40
004F 41                    DB      41H
0050 C9019A00              DW      Disk$Advice2,Disk$Msg$41
0054 42                    DB      42H
0055 E301A400              DW      Disk$Advice3,Disk$Msg$42
0059 21                    DB      21H
005A 0702B400              DW      Disk$Advice4,Disk$Msg$21
005E 22                    DB      22H
005F 1B02B900              DW      Disk$Advice5,Disk$Msg$22
0063 23                    DB      23H
0064 1B02C000              DW      Disk$Advice5,Disk$Msg$23
0068 24                    DB      24H
0069 3D02D200              DW      Disk$Advice6,Disk$Msg$24
006D 25                    DB      25H
006E 3D02DE00              DW      Disk$Advice6,Disk$Msg$25
0072 11                    DB      11H
0073 5302F100              DW      Disk$Advice7,Disk$Msg$11
0077 12                    DB      12H
0078 5302FF00              DW      Disk$Advice7,Disk$Msg$12
007C 13                    DB      13H
007D 53020C01              DW      Disk$Advice7,Disk$Msg$13
0081 14                    DB      14H
0082 53021A01              DW      Disk$Advice7,Disk$Msg$14
0086 15                    DB      15H
0087 53022901              DW      Disk$Advice7,Disk$Msg$15
008B 16                    DB      16H
008C 53023501              DW      Disk$Advice7,Disk$Msg$16
0090 00                    DB      0                        ;<== Terminator
0091 53024501              DW      Disk$Advice7,Disk$Msg$Unknown    ;Unmatched code
                   ;
0005 =             DEM$Entry$Size  EQU     5        ;Entry size in error message table
                   ;
                   ;
                   ;       Message texts
                   ;
0095 48756E6700Disk$Msg$40:  DB      'Hung',0         ;Timeout message
009A 4E6F742052Disk$Msg$41:  DB      'Not Ready',0
00A4 5772697465Disk$Msg$42:  DB      'Write Protected',0
00B4 4461746100Disk$Msg$21:  DB      'Data',0
00B9 466F726D61Disk$Msg$22:  DB      'Format',0
00C0 4D69737369Disk$Msg$23:  DB      'Missing Data Mark',0
00D2 4275732054Disk$Msg$24:  DB      'Bus Timeout',0
00DE 436F6E7472Disk$Msg$25:  DB      'Controller Timeout',0
00F1 4472697665Disk$Msg$11:  DB      'Drive Address',0
00FF 4865616420Disk$Msg$12:  DB      'Head Address',0
010C 547261636BDisk$Msg$13:  DB      'Track Address',0
```

Figure 9-5. (Continued)

```
011A 536563746FDisk$Msg$14:    DB      'Sector Address',0
0129 4275732041Disk$Msg$15:    DB      'Bus Address',0
0135 496C6C6567Disk$Msg$16:    DB      'Illegal Command',0
0145 556E6B6E6FDisk$Msg$Unknown:       DB      'Unknown',0
                ;
                Disk$EM$1:                     ;Main disk error message -- part 1
014D 070D0A                    DB      BELL,CR,LF
0150 4469736B20                DB      'Disk ',0
                ;
                                               ;Error text output next
                ;
                Disk$EM$2:                     ;Main disk error message -- part 2
0156 204572726F                DB      ' Error ('
015E 0000       Disk$EM$Status: DB     0,0      ;Status code in hex
0160 290D0A2020                DB      ')',CR,LF,'     Drive '
016E 00         Disk$EM$Drive:  DB     0        ;Disk drive code, A,B...
016F 2C20486561                DB      ', Head '
0176 00         Disk$EM$Head:   DB     0        ;Head number
0177 2C20547261                DB      ', Track '
017F 0000       Disk$EM$Track:  DB     0,0      ;Track number
0181 2C20536563                DB      ', Sector '
018A 0000       Disk$EM$Sector: DB     0,0      ;Sector number
018C 2C204F7065                DB      ', Operation - '
019A 00                        DB      0                ;Terminator
                ;
019B 526561642EDisk$EM$Read:   DB      'Read.',0        ;Operation names
01A1 5772697465Disk$EM$Write:  DB      'Write.',0
                ;
01A8 0D0A202020Disk$Advice0:   DB      CR,LF,'     ',0
01B0 436865636BDisk$Advice1:   DB      'Check disk loaded, Retry',0
01C9 506F737369Disk$Advice2:   DB      'Possible hardware problem',0
01E3 5772697465Disk$Advice3:   DB      'Write enable if correct disk, Retry',0
0207 5265747279Disk$Advice4:   DB      'Retry several times',0
021B 5265666F72Disk$Advice5:   DB      'Reformat disk or use another disk',0
023D 4861726477Disk$Advice6:   DB      'Hardware error, Retry',0
0253 4861726477Disk$Advice7:   DB      'Hardware or Software error, Retry',0
                ;
0275 2C206F7220Disk$Advice9:   DB      ', or call for help if error persists',CR,LF
                ;
                Disk$Action$Confirm:
029B 00                        DB      0       ;Set to character entered by user
029C 0D0A00                    DB      CR,LF,0
                ;
                ;       Disk error processor
                ;
                ;       This routine builds and outputs an error message.
                ;       The user is then given the opportunity to:
                ;
                ;               R -- retry the operation that caused the error
                ;               I -- ignore the error and attempt to continue
                ;               A -- abort the program and return to CP/M
                ;
                Disk$Error:
029F F5                PUSH    PSW             ;Preserve error code from controller
02A0 215E01            LXI     H,Disk$EM$Status        ;Convert code for message
02A3 CD3B03            CALL    CAH                     ;Converts A to hex

02A6 3A0700            LDA     In$Buffer$Disk          ;Convert disk id. for message
02A9 C641              ADI     'A'                     ;Make into letter
02AB 326E01            STA     Disk$EM$Drive

02AE 3A0200            LDA     Floppy$Head             ;Convert head number
02B1 C630              ADI     '0'
02B3 327601            STA     Disk$EM$Head

02B6 3A0300            LDA     Floppy$Track            ;Convert track number
02B9 217F01            LXI     H,Disk$EM$Track
02BC CD3B03            CALL    CAH

02BF 3A0400            LDA     Floppy$Sector           ;Convert sector number
02C2 218A01            LXI     H,Disk$EM$Sector
02C5 CD3B03            CALL    CAH

02C8 214D01            LXI     H,Disk$EM$1             ;Output first part of message
02CB CD3B03            CALL    Output$Error$Message
```

Figure 9-5. (Continued)

```
02CE F1                 POP     PSW                         ;Recover error status code
02CF 47                 MOV     B,A                         ;For comparisons
02D0 214500             LXI     H,Disk$Error$Messages - DEM$Entry$Size
                                                            ;HL -> table -- one entry
02D3 110500             LXI     D,DEM$Entry$Size            ;For loop below
                Disk$Error$Next$Code:
02D6 19                 DAD     D                           ;Move to next (or first) entry

02D7 7E                 MOV     A,M                         ;Get code number from table
02D8 B7                 ORA     A                           ;Check if end of table
02D9 CAE302             JZ      Disk$Error$Matched          ;Yes, pretend a match occurred
02DC B8                 CMP     B                           ;Compare to actual code
02DD CAE302             JZ      Disk$Error$Matched          ;Yes, exit from loop
02E0 C3D602             JMP     Disk$Error$Next$Code        ;Check next code
                ;
                Disk$Error$Matched:
02E3 23                 INX     H                           ;HL -> advisory text address
02E4 5E                 MOV     E,M
02E5 23                 INX     H
02E6 56                 MOV     D,M                         ;DE -> advisory test
02E7 D5                 PUSH    D                           ;Save for later

02E8 23                 INX     H                           ;HL -> message text address
02E9 5E                 MOV     E,M                         ;Get address into DE
02EA 23                 INX     H
02EB 56                 MOV     D,M

02EC EB                 XCHG                                ;HL -> text
02ED CD3B03             CALL    Output$Error$Message        ;Display explanatory text

02F0 215601             LXI     H,Disk$EM$2                 ;Display second part of message
02F3 CD3B03             CALL    Output$Error$Message

02F6 219B01             LXI     H,Disk$EM$Read              ;Choose operation text
                                                            ;  (assume a read)
02F9 3A0100             LDA     Floppy$Command              ;Get controller command
02FC FE01               CPI,    Floppy$Read$Code
02FE CA0403             JZ      Disk$Error$Read             ;Yes
0301 21A101             LXI     H,Disk$EM$Write             ;No, change address in HL
                Disk$Error$Read:
0304 CD3B03             CALL    Output$Error$Message        ;Display operation type

0307 21A801             LXI     H,Disk$Advice0              ;Display leading blanks
030A CD3B03             CALL    Output$Error$Message

030D E1                 POP     H                           ;Recover advisory text pointer
030E CD3B03             CALL    Output$Error$Message

0311 217502             LXI     H,Disk$Advice9              ;Display trailing component
0314 CD3B03             CALL    Output$Error$Message
                ;
                Disk$Error$Request$Action:                  ;Ask the user what to do next
0317 CD3B03             CALL    Request$User$Choice         ;Display prompt and get single
                                                            ;  character response (folded to
                                                            ;  uppercase)
031A FE52               CPI     'R'                         ;Retry
031C CA2C03             JZ      Disk$Error$Retry
031F FE41               CPI     'A'                         ;Abort?
0321 CA3603             JZ      System$Reset
0324 FE49               CPI     'I'                         ;Ignore?
0326 CA4500             JZ      Disk$Error$Ignore
0329 C31703             JMP     Disk$Error$Request$Action
                ;
                Disk$Error$Retry:                           ;The decision on where to return to
                                                            ;  depends on whether the operation
                                                            ;  failed on a deblocked or
                                                            ;  nondeblocked drive
032C 3A0500             LDA     Deblocking$Required
032F B7                 ORA     A
0330 C21500             JNZ     Deblock$Retry
0333 C30800             JMP     No$Deblock$Retry
```

Figure 9-5. (Continued)

```
                ;
                System$Reset:                           ;This is a radical approach, but
                                                        ;  it does cause CP/M to restart
0336 0E00               MVI     C,0                     ;System reset
0338 CD0500             CALL    BDOS

                ;
                ;       Omitted subroutines (listed in full in Figure 8-10)
                ;
                Set$Watchdog:           ;Set watchdog timer (to number of "ticks" in BC, and
                                        ;  to transfer control to (HL) if timer hits zero).
                CAH:                    ;Convert A to two ASCII hex characters, storing
                                        ;  the output in (HL) and (HL+1)
                Output$Error$Message:   ;Display the 00-byte terminated error message
                                        ;  pointed to by HL. Output is directed only to
                                        ;  those console devices not being used for list
                                        ;  output as well.
                Request$User$Choice:    ;Display prompt "Enter R, A, I..." and return
                                        ;  single keyboard character (uppercase) in A
033B C9                 RET             ;Dummy
```

Figure 9-5. User-friendly disk-error processor (continued)

Basic Debugging Techniques
Debug Subroutines
Software Tools for Debugging
Bringing Up CP/M for the First Time
Debugging the CP/M Bootstrap Loader
Debugging the BIOS
Live Testing a New BIOS

10

Debugging A New CP/M System

This chapter deals with some of the problems you will face bringing up CP/M on a computer system for the first time or enhancing it once it is up and running on your system.

In the first case, when CP/M does not yet run on your computer, you may be writing the complete BIOS yourself, although you can model what you do on the example BIOS provided on the CP/M release diskette and the example code from Chapter 6.

In the second case, you can extend the existing BIOS by adding code—from the examples in Chapters 8 and 9, code from computer magazines, or code you create yourself. To do this, you will need access to the BIOS source code—a problem if the manufacturer of your computer does not make it available. In general, however, the BIOS source code is included with the system or can be obtained at nominal or no cost. If you cannot obtain the source code, you can, of

course, take the bull by the horns and reimplement CP/M on your system. This may require many hours of disassembling the current BIOS machine code to find out how to access all the various ports and how to control the devices to which they are connected.

Although the BIOS is the major component of a new CP/M implementation, remember that it is only the beginning—you can spend the same amount of time and effort getting the bootstrap loader and all the utilities to function.

Basic Debugging Techniques

Before getting involved in the details of how to debug a CP/M implementation, it is worth considering the nature of the task. Some quotations that are appropriate here:

> "Program testing can be used to show the presence of bugs, but never to show their absence." —Dijkstra

> "We call them bugs because to call them mistakes would be psychologically unacceptable." —Hopkins

> "Constants aren't, variables won't." —Osborne

Debugging is the name we give to the process of executing programs and ascertaining whether the programs are running correctly. "Correctly" means in accordance with the mental model we have built of how the program should behave, subject to the constraints imposed by the physical hardware. Therein lies the first of the problems; you and the hardware are the arbiters of correct performance. The hardware is usually unforgiving; if there is a flaw in the way you program it, it will either be dramatically "uncooperative" or not work at all. As for how you perceive the system, several fairly simple tests, along with attempts to use the system for useful work for a few days, will shake the system down fairly well. The most difficult problems will be with intermittent failures or logical contradictions.

Computers are deterministic. That is, if you start from a known state and perform a known series of operations, the computer will always yield the same results. To achieve a known state is not so difficult—resetting the system and clearing memory will do it. Performing a known series of operations just means running the program again, although if you are using interrupts, you cannot truthfully say that exactly the same operations are being performed, because the interrupts will not happen at *exactly* the same time as before.

The "Orville Wright" Approach

Your role in debugging a new CP/M system is comparable to the popular, though untrue, idea of the way the Wright brothers developed flying machines:

build a machine, take it to the top of a hill, throw it off, and, when it crashes, examine the debris to discover what went wrong.

Each time you do an assembly and test, you are building the aircraft and lobbing it off the edge of a cliff. Each time it crashes, you examine the wreckage and try to determine the possible cause.

This is a highly inferential process. With the wreckage as a starting point, you use inference and intuition to extrapolate the real problem and the correction for it.

Built-In Debug Code

The single most important concept that you will need in testing CP/M systems is the same as that used in the modern day "black box" flight recorder. This device is essentially a multi-channel tape recorder that records all of the relevant conditions of the aircraft, its height, altitude, throttle settings, flap settings, and even the voice communications among crew members. If the airplane crashes, investigators can replay the information and understand what happened during the flight.

Applying this concept to debugging CP/M means that you must build into your code some method for recording what it is doing, so that if the system crashes, you can see what it was doing. Make the code tell you what went wrong.

The debug code should be designed at the same time as the rest of the program. Plan the debugging code while the design is still on the drawing board. The source code for debugging should be a permanent part of the BIOS. Use conditional assembly to "IF" out most of the debug code from the final version, or make the code sensitive to a flag in the configuration block so that you can re-enable the debug code at a moment's notice if the system begins to behave strangely.

The more meaningful the debug output data, the less you will have to guess at what is wrong, and therefore the less painful and time-consuming the debugging process will be. Make the output intelligible to others who may use it or yourself several months hence. Data that tells you what is happening is more useful than internal hexadecimal values, particularly if someone else must interpret it or relay it to you over the telephone.

Debug Subroutines

Many programmers do their debugging on a casual "catch as catch can" basis because they are overwhelmed by the task of building the necessary tools. Others are too eager to start on a new program to take a few extra hours or days to build debug subroutines.

To help solve this problem, the following section provides some ready-made debugging tools that can be used "as is." Each of these routines has been thor-

oughly debugged (there's nothing worse than debug code with bugs in it!) and has been used in actual program testing.

Overall Design Philosophy

Some common methods run through the examples that follow. These include displaying meaningful "captions" (including the specific address that called the debug routine), grouping all debugging code together, preserving the contents of all registers, and setting up the stack area in a standard way.

Debug Code Captions When the contents of registers or memory are output as part of a debugging process, a caption of explanatory text describing the values should be displayed. For example, rather than displaying the contents of the A register like this,

```
A = 1F
```

you can use a meaningful caption such as:

```
Transaction Code A = 1F.
```

When you write additional debugging code, especially if you need to add it to an existing routine, it is cumbersome to have to write the call to the debug routine and then search through the source code to find a convenient place to put an ASCII caption string. A caption string several pages removed from the point where it is referenced makes for problems when you want to relate the debug output on the screen or listing to the source code itself. Therefore, all of the routines that follow allow you to declare the caption strings "in-line" like this:

```
IF          DEBUG
CALL        Debug$Routine
DB          'Caption string here',CR,LF,0
ENDIF

MVI         .....            ;Next instruction
```

All of the following routines that output a caption recognize one specific 8-bit value in the caption string. If they encounter a value of 0ADH (mnemonic for ADdress), they will output the address of the byte following the call to the debug routine. For example,

```
0210        CALL        Debug$Routine
0213        DB          0ADH,'Caption string',0
```

will cause the routine to display the following:

```
0213 Caption string
```

This identifies the point in your program from which the debug routine was called, and thus avoids any possible ambiguity between different calls to the same debug routine with similar captions.

Grouping Debug Code Grouping all the debug code together lends itself to using conditional assembly with IF/ENDIF statements.

Setting Up the Stack Area All of the following routines preserve the CPU registers so that there are no side effects from using them. All of them assume that they can use the stack pointer and that there is sufficient room in the stack area. Hence you will need to declare adequate stack space for your main code and for the debug routines. Fill the stack area with a known pattern like this:

```
        DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
        DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
        DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
Stack$Area:                     ;Label the upper end of the area
```

Then, during debugging, you can examine the stack area and determine how much of it is unused. For example, if you looked at the stack area you might see something like this:

```
                                    "Low-water mark"
                                          V
99 99 99 99 99 99 99 99 99 99 99 99 09 15 43 42
01 29 00 00 1A 2B 10 FF FF 39 02 ED 11 01 37 44
DD 00 00 11 1A 23 31 00 41 AE FE 00 01 10 70 C9
```

Stack area overflow can give arcane bugs; the program seems to leap off into space in a nondeterministic way. By setting up the stack area in this way, you can recognize an overflow condition easily.

Debug Initialization Before you can execute any of the debug subroutines in this chapter, you must make a call to the initialization subroutine, DB$Init. The DB$Init routine sets up some of the internal variables needed by the debug package. You may need to add some of your own initialization code here.

Console Output

Normally, you can use the CONOUT functions either via the BDOS (Function 2), or via the BIOS by calling the jump vector directly. You cannot do this when you need to debug console routines themselves, nor when you need to debug interrupt service routines. In the latter case, if an interrupt pulled control out of the CONOUT routine in the BIOS, you would get unwanted re-entrancy if the debug code again entered the CONOUT driver to display a caption. Therefore, the debug routines have been written to call their own local CONOUT routine, which is called DB$CONOUT. DB$CONOUT can be changed to call the BDOS, the BIOS, or a "private" polled output routine.

A counterpart DB$CONIN routine for console input is provided for essentially the same reasons.

Controlling Debug Output

All output of debug routines in this chapter is controlled by a single master flag, DB$Flag. If this flag is nonzero, debug output will occur; if zero, all output is suppressed.

This flag can be set and cleared from any part of the program you are testing. It is especially useful when you need to debug a subroutine that is called many times from many different places. You can write additional code to enable debug output when certain conditions prevail; for example, when a particular track or sector is about to be written or when a character input buffer is almost full.

Two subroutines, DB$On and DB$Off, are shown that access the debug control flag. These, as their names suggest, turn debug output on and off.

Turning the debug output on and off from within the program can create a confusing display of debug output, lacking any apparent continuity. DB$Off gives you the option of outputting a character string indicating that debug output has been turned off.

Pass Counters

Another method of controlling debug output is to use a *pass counter,* enabling debug output only after control has passed through a particular point in the code a specific number of times.

Two subroutines are provided for this purpose. DBSetPass sets the pass counter to a specific value. DB$Pass decrements this pass count each time control is transferred to it. When the pass count hits zero, the debug control flag DB$Flag is nonzero and debug output begins.

Using pass counter techniques can save you time and effort in tracking down a problem that occurs only after the code has been running for several minutes.

Displaying Contents of Registers and Memory

Figure 10-2 shows a series of display subroutines, the primary one of which is DB$Display. It takes several parameters, depending on the information you want displayed. The generic call to DB$Display is as follows:

```
CALL     DB$Display
DB       Code        <- Indicates the data to be
                        displayed
{DW      Optional additional parameters}
DB       'Caption string',0
```

The codes that can be used in this call are shown in Table 10-1.

The only function that uses additional parameters is DB$Memory. This displays bytes from memory in hexadecimal and ASCII, using the start and finish

addresses following the call. Here is an example:

```
CALL        DB$Display
DB          DB$Memory
DW          Start$Address,End$Address
DB          'Caption string',0
```

Table 10-1. Codes for DB$Display

Code	Value displayed
8-bit registers	
DB$F	Condition Flags
DB$A	Register A
DB$B	Register B
DB$C	Register C
DB$D	Register D
DB$E	Register E
DB$H	Register H
DB$L	Register L
Memory	
DB$Memory	Bytes starting and ending at the addresses specified by the two word values following the code value.
16-bit registers	
DB$BC	Register pair BC
DB$DE	Register pair DE
DB$HL	Register pair HL
DB$SP	Stack Pointer
Byte values	
DBBBC	Byte addressed by BC
DBBDE	Byte addressed by DE
DBBHL	Byte addressed by HL
Word values	
DBWBC	Word addressed by BC
DBWDE	Word addressed by DE
DBWHL	Word addressed by HL

Debugging Program Logic

In addition to displaying the contents of registers and memory, you need to display the program's execution path, not in terms of addresses, but in terms of the *problem*. You can do this by displaying debug messages that indicate what decisions have been made by the program as it executes. For example, if your BIOS checks a particular value to see whether the system should read or write on a particular device, the debug routine should display a message like this:

```
Entering Disk Read Routine
```

This is more meaningful than just displaying the function code for the drivers— although you may want to display this as well, in case it has been set to some strange value.

Two subroutines are provided to display debug messages. They are DB$MSG and DB$MSGI. Both of these display text strings are terminated with a byte of 00H. You can see the difference between the two subroutines if you examine the way they are called.

DB$MSG is called like this:

```
LXI  H,Message$Text       ;HL -> text string
CALL DB$MSG
```

DB$MSGI is called like this:

```
CALL DB$MSG
DB   0DH,0AH,'Message Text',0 ;In-line
```

DB$MSGI is more convenient to use. If you decide that you need to add a message, you can declare the message immediately following the call. This also helps when you look at the listing, since you can see the complete text at a glance.

Use DB$MSG when the text of the message needs to be selected from a table. Get the address of the text into HL and then call DB$MSG to display it.

Creating Your Own Debug Displays

If you need to build your own special debug display routines, you may find it helpful to incorporate some of the small subroutines in the debug package. The following are the subroutines you may want to use:

DB$CONOUT
Displays the character in the C register.

DB$CONIN
Returns the next keyboard character in A.

DB$CONINU
Returns the next keyboard character in A, converting lowercase letters to uppercase.

DB$DHLH
Displays contents of HL in hexadecimal.

DB$DAH
Displays contents of A in hexadecimal.

DB$CAH
Converts contents of A to hexadecimal and stores in memory pointed at by HL.

DB$Nibble$To$Hex
Converts the least significant four bits of A into an ASCII hexadecimal character in A.

DB$CRLF
Displays a CARRIAGE RETURN/LINE FEED.

DB$Colon
Displays the string " : ".

DB$Blank
Displays a single space character.

DB$Flag$Save$On
Saves the current state of the debug output control flag and then sets the flag "on" to enable debug output.

DB$Flag$Restore
Restores the debug output control flag to the state it was in when the DB$Flag$Save$On routine was last called.

DB$GHV
Gets a hexadecimal value from the keyboard, displaying a prompt message first. From one to four characters can be specified as the maximum number of characters to be input.

DBATo$Upper
If the A register contains a lowercase letter, this converts it to an uppercase letter.

Debugging I/O Drivers

Debugging low-level device drivers creates special problems. The major one is that you do not normally want to read and write via actual hardware ports while you are debugging the code—either because doing so would cause strange things to happen to the hardware during the debugging, or because you are developing and debugging the drivers on a system different from the target hardware on which the drivers are to execute.

Before considering the solution, remember that the input and output instructions (IN and OUT) are each two bytes long. The first byte is the operation code

(0DBH for input, 0D3H for output), and the second byte is the port number to "input from" or "output to."

Debug subroutines are provided here to intercept all IN and OUT instructions, displaying the port number and either accepting a hexadecimal value from the console and putting it into the A register (in the case of IN), or displaying the contents of the A register (for the OUT instruction).

IN and OUT instructions can be "trapped" by changing the operation code to one of two RST (restart) instructions. An RST is effectively a single-byte CALL instruction, calling down to a predetermined address in low memory. The debug routines arrange for JMP instructions in low memory to receive control when the correct RST is executed. The code that receives control can pick up the port number, display it, and then accept a hex value for the A register (for IN) or display the current contents of the A register (for OUT). The example subroutines shown later in this chapter use RST 4 in place of IN instructions, RST 5 for OUT.

Wherever you plan to use IN, use the following code:

```
IF          Debug
RST         4
ENDIF
IF          NOT Debug
DB          IN
ENDIF
DB          Port$Number
```

Note that you can use the IN operation code as the operand of a DB statement. The assembler substitutes the correct operation code.

Use the following code wherever you need to use an OUT instruction:

```
IF          Debug
RST         5
ENDIF
IF          NOT Debug
DB          OUT
ENDIF
DB          Port$Number
```

When the RST 4 (IN) instruction is executed, the debug subroutine displays

```
1AB3 : Input from Port 01 : _
```

The "1AB3" is the address in memory of the byte containing the port number. It serves to pinpoint the IN instruction in memory. You can then enter one or two hexadecimal digits. These will be converted and put into the A register before control returns to the main program at the instruction following the byte containing the port number.

When the RST 5 (OUT) instruction is encountered, the debug subroutine displays

```
1AB5 :  Output to Port 01 : FF
```

This identifies where the OUT instruction would normally be as well as the port number and the contents of the A register when the RST 5 (OUT) is executed.

Debugging Interrupt Service Routines

You can use a technique similar to that of the RST instruction just described to "fake" an interrupt. You preset the low-memory address for the RST instruction you have chosen for the jump into the interrupt service routine under test.

When the RST instruction is executed, control will be transferred into the interrupt service routine just as though an interrupt had occurred. You will need to intercept any IN or OUT instructions as described above—otherwise the code probably will go into an endless loop.

Before executing the RST instruction to fake the interrupt, load all the registers with known values. For example:

```
MVI     A,0AAH
LXI     B,0BBCCH
LXI     D,0DDEEH
LXI     H,01122H
RST     6          ;Fake interrupt
NOP
```

When control returns from the service routine, you can check to see that it restored all of the registers to their correct values. An interrupt service routine that does not restore all the registers can produce bugs that are very hard to find.

Check, too, that the stack pointer register has been restored and that the service routine did not require too many bytes on the stack.

You also can use the CALL instruction to transfer control to the interrupt service routine in order to fake an interrupt. RST and CALL achieve the same effect, but RST is closer to what happens when a real interrupt occurs. As it is a single-byte instruction, it also is easier to patch in.

Subroutine Listings

Figure 10-1 is a functional index to the source code listing for the debug subroutines shown in Figure 10-2. The listing's commentary defines precisely how each debug subroutine is called.

Figure 10-3 shows the output from the debug testbed.

Software Tools for Debugging

In addition to building in debugging subroutines, you will need one of the following proprietary debug programs:

DDT (Dynamic Debugging Tool)

This program, included with the standard CP/M release, allows you to load programs, set and display memory and registers, trace through your program instruction by instruction, or execute it at full speed, but stopping

Start Line	Functional Component or Routines
00001	Debug subroutine's Testbed
00100	Test register display
00200	Test memory dump display
00300	Test register pair display
00400	Test byte indirect display
00500	Test DB$On/Off
00600	Test DBSetPass and DB$Pass
00700	Test debug input/output
00800	Debug subroutines themselves
01100	DB$Init - initialization
01200	DB$CONINU - get uppercase keyboard character
01300	DB$CONIN - get keyboard character
01400	DB$CONOUT - display character in C
01500	DB$On - enable debug output
01600	DB$Off - disable debug output
01700	DBSetPass - set pass counter
01800	DB$Pass - execute pass point
01900	DB$Display - main debug display routine
02200	Main display processing subroutines
02500	DB$Display$CALLA - display CALL's address
02600	DB$DHLH - display HL in hexadecimal
02700	DB$DAH - display A in hexadecimal
02800	DB$CAH - convert A to hexadecimal in memory
02900	DB$Nibble$To$Hex - convert LS 4 bits of A to hex.
02930	DB$CRLF - display Carriage Return, Line Feed
02938	DB$Colon - display " : "
02946	DB$Blank - display " "
03100	DB$MSGI - display in-line message
03147	DB$MSG - display message addressed by HL
03300	DB$Input - debug INput routine
03500	DB$Output - debug OUTput routine
03700	DB$Flag$Save$On - save debug flag and enable
03800	DB$Flag$Restore - restore debug control flag
03900	DB$GHV - get hexadecimal value from keyboard
04100	DBATo$Upper - convert A to upper case

Figure 10-1. Functional index for Figure 10-2

at certain addresses (called breakpoints). It also has a built-in mini-assembler and disassembler so you do not have to hand assemble any temporary code "patches" you add.

SID (Symbolic Interactive Debug)
Similar to DDT in many ways, SID has enhancements that are helpful if you use Digital Research's MAC (Macro Assembler) or RMAC (Relocating Macro Assembler). Both of these assemblers can be told to output a file

```
00001
00002
00003                   ;
00004                   ;       Debug Subroutines
00005                   ;
00006                   ;<----  NOTE:
00007                   ;       The line numbers at the extreme left are included purely
00008                   ;       to reference the code from the text.
00009                   ;       There are deliberately induced discontinuities
00010                   ;       in the numbers in order to allow space for expansion.
00011                   ;
00012                   ;       Because of the need to test these routines thoroughly,
00013                   ;       and in case you wish to make any changes, the testbed
00014                   ;       routine for the debug package itself has been left in
00015                   ;       in this figure.
00016                   ;
00017                   ;       Debug testbed
00018                   ;
00019   0100                    ORG     100H
00020                   START:
00021   0100 316B03             LXI     SP,Test$Stack           ;Set up local stack
00022   0103 CDEA04             CALL    DB$Init                 ;Initialize the debug package
00023   0106 CD1505             CALL    DB$On                   ;Enable debug output
00024                                                           ;Simple test of A register display
00025   0109 3EAA               MVI     A,0AAH                  ;Preset a value in the A register
00026   010B 01CCBB             LXI     B,0BBCCH                ;Prefill all other registers, partly
00027   010E 11EEDD             LXI     D,0DDEEH                ;  to check the debug display, but
00028   0111 2111FF             LXI     H,0FF11H                ;  also to check register save/restore
00100                   ;#
00101                   ;       Test register display
00102                   ;
00103   0114 B7                 ORA     A                       ;Set M-flag, clear Z-flag, set E-flag
00104   0115 37                 STC                             ;Set carry
00105   0116 CD5205             CALL    DB$Display              ;Call the debug routine
00106   0119 00                 DB      DB$F
00107   011A 466C616773         DB      'Flags',0
00108                   ;
00109   0120 CD5205             CALL    DB$Display              ;Call the debug routine
00110   0123 02                 DB      DB$A
00111   0124 4120526567         DB      'A Register',0
00112                   ;
00113   012F CD5205             CALL    DB$Display              ;Call the debug routine
00114   0132 04                 DB      DB$B
00115   0133 4220526567         DB      'B Register',0
00116                   ;
00117   013E CD5205             CALL    DB$Display              ;Call the debug routine
00118   0141 06                 DB      DB$C
00119   0142 4320526567         DB      'C Register',0
00120                   ;
00121   014D CD5205             CALL    DB$Display              ;Call the debug routine
00122   0150 08                 DB      DB$D
00123   0151 4420526567         DB      'D Register',0
00124                   ;
00125   015C CD5205             CALL    DB$Display              ;Call the debug routine
00126   015F 0A                 DB      DB$E
00127   0160 4520526567         DB      'E Register',0
00128                   ;
00129   016B CD5205             CALL    DB$Display              ;Call the debug routine
00130   016E 0C                 DB      DB$H
00131   016F 4820526567         DB      'H Register',0
00132                   ;
00133   017A CD5205             CALL    DB$Display              ;Call the debug routine
00134   017D 0E                 DB      DB$L
00135   017E 4C20526567         DB      'L Register',0
00200                   ;#
00201                   ;       Test Memory Dump Display
00202                   ;
00203   0189 CD5205             CALL    DB$Display
00204   018C 18                 DB      DB$M                    ;Dump memory
00205   018D 08012801           DW      108H,128H               ;Check start/end at nonmultiples
00206   0191 4D656D6F72         DB      'Memory Dump #1',0      ;  of 10H
00207                   ;
00208   01A0 CD5205             CALL    DB$Display
00209   01A3 18                 DB      DB$M                    ;Dump memory
00210   01A4 00011F01           DW      100H,11FH               ;Check start and end on displayed
00211   01A8 4D656D6F72         DB      'Memory Dump #2',0      ;  line boundaries
00212                   ;
```

Figure 10-2. Debug subroutines

```
00213     01B7 CD5205            CALL    DB$Display
00214     01BA 18                DB      DB$M                    ;Dump memory
00215     01BB 01010001          DW      101H,100H               ;Check error handling where
00216     01BF 4D656D6F72        DB      'Memory Dump #3',0      ;  start > end address
00217                   ;
00218     01CE CD5205            CALL    DB$Display
00219     01D1 18                DB      DB$M                    ;Dump memory
00220     01D2 00010001          DW      100H,100H               ;Check end-case of single byte
00221     01D6 4D656D6F72        DB      'Memory Dump #4',0      ;  output
00300                   ;#
00301                   ;        Test register pair display
00302                   ;
00303     01E5 CD5205            CALL    DB$Display              ;Call the debug routine
00304     01E8 10                DB      DB$BC
00305     01E9 4243205265        DB      'BC Register',0
00306                   ;
00307     01F5 CD5205            CALL    DB$Display              ;Call the debug routine
00308     01F8 12                DB      DB$DE
00309     01F9 4445205265        DB      'DE Register',0
00310                   ;
00311     0205 CD5205            CALL    DB$Display              ;Call the debug routine
00312     0208 14                DB      DB$HL
00313     0209 484C205265        DB      'HL Register',0
00314                   ;
00315     0215 CD5205            CALL    DB$Display              ;Call the debug routine
00316     0218 16                DB      DB$SP
00317     0219 5350205265        DB      'SP Register',0
00318                   ;
00319     0225 013203            LXI     B,Byte$BC               ;Set up registers for byte tests
00320     0228 113303            LXI     D,Byte$DE
00321     022B 213403            LXI     H,Byte$HL
00400                   ;#
00401                   ;        Test byte indirect display
00402                   ;
00403     022E CD5205            CALL    DB$Display              ;Call the debug routine
00404     0231 1A                DB      DB$B$BC
00405     0232 4279746520        DB      'Byte at (BC)',0
00406                   ;
00407     023F CD5205            CALL    DB$Display              ;Call the debug routine
00408     0242 1C                DB      DB$B$DE
00409     0243 4279746520        DB      'Byte at (DE)',0


00410                   ;
00411     0250 CD5205            CALL    DB$Display              ;Call the debug routine
00412     0253 1E                DB      DB$B$HL
00413     0254 4279746520        DB      'Byte at (HL)',0
00414                   ;
00415     0261 013503            LXI     B,Word$BC               ;Set up the registers for word tests
00416     0264 113703            LXI     D,Word$DE
00417     0267 213903            LXI     H,Word$HL
00418
00419     026A CD5205            CALL    DB$Display              ;Call the debug routine
00420     026D 20                DB      DB$W$BC
00421     026E 576F726420        DB      'Word at (BC)',0
00422                   ;
00423     027B CD5205            CALL    DB$Display              ;Call the debug routine
00424     027E 22                DB      DB$W$DE
00425     027F 576F726420        DB      'Word at (DE)',0
00426                   ;
00427     028C CD5205            CALL    DB$Display              ;Call the debug routine
00428     028F 24                DB      DB$W$HL
00429     0290 576F726420        DB      'Word at (HL)',0
00500                   ;#
00501                   ;        Test DB$On/Off
00502                   ;
00503     029D CD1D05            CALL    DB$Off                  ;Disable debug output
00504     02A0 CDD607            CALL    DB$MSGI                 ;Display in-line message
00505     02A3 0D0A546869        DB      0DH,0AH,'This message should NOT appear',0
00506
00507     02C4 CD1505            CALL    DB$On
00508     02C7 CDD607            CALL    DB$MSGI
00509     02CA 0D0A446562        DB      0DH,0AH,'Debug output has been re-enabled.',0
00600                   ;#
00601                   ;        Test pass count logic
00602                   ;
```

Figure 10-2. (Continued)

```
00603   02EE CD1D05           CALL    DB$Off                  ;Disable debug output
00604   02F1 CD2405           CALL    DB$Set$Pass             ;Set pass count
00605   02F4 1E00             DW      30
00606                 ;
00607   02F6 3E22             MVI     A,34                    ;Set loop counter greater than pass
00608                                                         ;  counter
00609                 Test$Pass$Loop:
00610   02F8 CD3505           CALL    DB$Pass                 ;Decrement pass count
00611   02FB CDD607           CALL    DB$MSGI                 ;Display in-line message
00612   02FE 0D0A546869       DB      0DH,0AH,'This message should display 5 times',0
00613   0324 3D               DCR     A
00614   0325 C2F802           JNZ     Test$Pass$Loop
00700                 ;#
00701                 ;       Test debug input/output
00702                 ;
00703   0328 CD1D05           CALL    DB$Off                  ;Check that debug IN/OUT
00704                                                         ;  must still occur when debug
00705                                                         ;  output is disabled.
00706   032B E7               RST     4                       ;Debug input
00707   032C 11               DB      11H                     ;Port number
00708   032D EF               RST     5                       ;Debug output (value return from input)
00709   032E 22               DB      22H                     ;Port number
00710
00711   032F C30000           JMP     0                       ;Warm boot at end of testbed
00712                 ;
00713                 ;
00714                 ;       Dummy values for byte and word displays
00715   0332 BC       Byte$BC:        DB      0BCH
00716   0333 DE       Byte$DE:        DB      0DEH
00717   0334 F1       Byte$HL:        DB      0F1H
00718                 ;
00719   0335 0C0B     Word$BC:        DW      0B0CH
00720   0337 0E0D     Word$DE:        DW      0D0EH
00721   0339 010F     Word$HL:        DW      0F01H
00722                 ;
00723   033B 9999999999               DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
00724   034B 9999999999               DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
00725   035B 9999999999               DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
00726                 Test$Stack:
00727                 ;
00728                 ;
00729                 ;
00730   0400                  ORG     400H                    ;To avoid unnecessary listings
00731                                                         ;  when only the testbed changes
00732                 ;------------------------------------------------------------------------
00800                 ;#
00801                 ;
00802                 ;       Debug subroutines
00803                 ;
00804                 ;
00805                 ;       Equates for DB$Display codes
00806                 ;       These equates are the offsets down the table of addresses
00807                 ;       for various subroutines to be used.
00808                 ;
00809   0000 =        DB$F    EQU     00      ;Flags
00810   0002 =        DB$A    EQU     02      ;A register
00811   0004 =        DB$B    EQU     04      ;B
00812   0006 =        DB$C    EQU     06      ;C
00813   0008 =        DB$D    EQU     08      ;D
00814   000A =        DB$E    EQU     10      ;E
00815   000C =        DB$H    EQU     12      ;H
00816   000E =        DB$L    EQU     14      ;L
00817   0010 =        DB$BC   EQU     16      ;BC
00818   0012 =        DB$DE   EQU     18      ;DE
00819   0014 =        DB$HL   EQU     20      ;HL
00820   0016 =        DB$SP   EQU     22      ;Stack pointer
00821   0018 =        DB$M    EQU     24      ;Memory
00822   001A =        DB$B$BC EQU     26      ;(BC)
00823   001C =        DB$B$DE EQU     28      ;(DE)
00824   001E =        DB$B$HL EQU     30      ;(HL)
00825   0020 =        DB$W$BC EQU     32      ;(BC+1),(BC)
00826   0022 =        DB$W$DE EQU     34      ;(DE+1),(DE)
00827   0024 =        DB$W$HL EQU     36      ;(HL+1),(HL)
00828                 ;
00829                 ;
00830                 ;       Equates
00831   0020 =        RST4            EQU     20H     ;Address for RST 4 - IN instruction
```

Figure 10-2. (Continued)

```
00832  0028 =           RST5            EQU     28H     ;Address for RST 5 - OUT instruction
00833                   ;
00834  0001 =           B$CONIN         EQU     1       ;BDOS CONIN function code
00835  0002 =           B$CONOUT        EQU     2       ;BDOS CONOUT function code
00836  000A =           B$READCONS      EQU     10      ;BDOS read console function code
00837  0005 =           BDOS            EQU     5       ;BDOS entry point
00838                   ;
00839  0000 =           False           EQU     0
00840  FFFF =           True            EQU     NOT False
00841                   ;
00842                                                           ;Equates to specify how DB$CONOUT
00843                                                           ;  and DB$CONIN should perform
00844                                                           ;  their input/output
00845  0000 =           DB$Polled$IO    EQU     False           ;)
00846  0000 =           DB$BIOS$IO      EQU     False           ;) Only one must be true
00847  FFFF =           DB$BDOS$IO      EQU     True            ;)
00848                   ;
00849                                                           ;Equates for polled I/O
00850  0001 =           DB$Status$Port  EQU     01H             ;Console status port
00851  0002 =           DB$Data$Port    EQU     02H             ;Console data port
00852                   ;
00853  0002 =           DB$Input$Ready  EQU     0000$0010B      ;Incoming data ready
00854  0001 =           DB$Output$Ready EQU     0000$0001B      ;Ready for output
00855                   ;
00856                                                           ;Data for BIOS I/O
00857  0400 C3          BIOS$CONIN:     DB      JMP             ;The initialization routine sets these
00858  0401 0000                        DW      0               ;  two JMP addresses into the BIOS
00859  0403 C3          BIOS$CONOUT:    DB      JMP
00860  0404 0000                        DW      0
00861                   ;
00862                   ;       Main debug variables and constants
00863                   ;
00864  0406 00          DB$Flag:        DB      0       ;Main debug control flag
00865                                                   ;  When this flag is nonzero, all debug
00866                                                   ;  output will be made. When zero, all
00867                                                   ;  debug output will be suppressed.
00868                                                   ;  It is altered either directly by the user
00869                                                   ;  or using the routines DB$On, DB$Off and
00870                                                   ;  DB$Pass.
00871                   ;
00872  0407 0000        DB$Pass$Count:  DW      0       ;Pass counter
00873                                                   ;  When this is nonzero, calls to DB$Pass
00874                                                   ;  decrement it by one. When it reaches
00875                                                   ;  zero, the debug control flag, DB$Flag,
00876                                                   ;  is set nonzero, thereby enabling
00877                                                   ;  debug output.
00878                   ;
00879                   DB$Save$HL:                     ;Save area for HL
00880  0409 00          DB$Save$L:      DB      0
00881  040A 00          DB$Save$H:      DB      0
00882
00883  040B 0000        DB$Save$SP:     DW      0       ;Save area for stack pointer
00884  040D 0000        DB$Save$RA:     DW      0       ;Save area for return address
00885  040F 0000        DB$Call$Address:        DW      0       ;Starts out the same as DB$Save$RA
00886                                                   ;  but DB$Save$RA gets updated during
00887                                                   ;  debug processing. This value is
00888                                                   ;  output ahead of the caption
00889                   DB$Start$Address:               ;Start address for memory display
00890  0411 0000                        DW      0
00891                   DB$End$Address:                 ;End address for memory display
00892  0413 0000                        DW      0
00893                   DB$Display$Code:                ;Display code requested
00894  0415 00                          DB      0
00895                   ;
00896                   ;
00897                                                   ;Stack area
00898  0416 9999999999                  DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
00899  0426 9999999999                  DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
00900  0436 9999999999                  DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
00901  0446 00          DB$Save$E:      DB      0       ;E register
00902  0447 00          DB$Save$D:      DB      0       ;D register
00903  0448 00          DB$Save$C:      DB      0       ;C register
00904  0449 00          DB$Save$B:      DB      0       ;B register
00905  044A 00          DB$Save$F:      DB      0       ;Flags
00906  044B 00          DB$Save$A:      DB      0       ;A register
00907                   DB$Stack:                       ;Debug stack area
00908                                                   ;  The registers in the stack area are PUSHed
00909                                                   ;  onto the stack and accessed directly.
```

Figure 10-2. (Continued)

```
00910                      ;
00911                      ;       Register caption messages
00912                      ;
00913                      ;       The table below, indexed by the Display$Code is used to access
00914                      ;       the register caption string.
00915                      ;
00916                      DB$Register$Captions:
00917     044C 7204                DW      DB$F$RC         ;Flags
00918     044E 7804                DW      DB$A$RC         ;A register
00919     0450 7A04                DW      DB$B$RC         ;B
00920     0452 7C04                DW      DB$C$RC         ;C
00921     0454 7E04                DW      DB$D$RC         ;D
00922     0456 8004                DW      DB$E$RC         ;E
00923     0458 8204                DW      DB$H$RC         ;H
00924     045A 8404                DW      DB$L$RC         ;L
00925     045C 8604                DW      DB$BC$RC        ;BC
00926     045E 8904                DW      DB$DE$RC        ;DE
00927     0460 8C04                DW      DB$HL$RC        ;HL
00928     0462 8F04                DW      DB$SP$RC        ;Stack pointer
00929     0464 9204                DW      DB$M$RC         ;Memory
00930     0466 A604                DW      DB$B$BC$RC      ;(BC)
00931     0468 AB04                DW      DB$B$DE$RC      ;(DE)
00932     046A B004                DW      DB$B$HL$RC      ;(HL)
00933     046C B504                DW      DB$W$BC$RC      ;(BC+1),(BC)
00934     046E C104                DW      DB$W$DE$RC      ;(DE+1),(DE)
00935     0470 CD04                DW      DB$W$HL$RC      ;(HL+1),(HL)
00936                      ;
00937     0472 466C616773DB$F$RC:      DB      'Flags',0       ;Flags
00938     0478 4100       DB$A$RC:     DB      'A',0           ;A register
00939     047A 4200       DB$B$RC:     DB      'B',0           ;B
00940     047C 4300       DB$C$RC:     DB      'C',0           ;C
00941     047E 4400       DB$D$RC:     DB      'D',0           ;D
00942     0480 4500       DB$E$RC:     DB      'E',0           ;E
00943     0482 4800       DB$H$RC:     DB      'H',0           ;H
00944     0484 4C00       DB$L$RC:     DB      'L',0           ;L
00945     0486 424300     DB$BC$RC:    DB      'BC',0          ;BC
00946     0489 444500     DB$DE$RC:    DB      'DE',0          ;DE
00947     048C 484C00     DB$HL$RC:    DB      'HL',0          ;HL
00948     048F 535000     DB$SP$RC:    DB      'SP',0          ;Stack pointer
00949     0492 5374617274DB$M$RC:      DB      'Start, End Address ',0 ;Memory
00950     04A6 2842432900DB$B$BC$RC:   DB      '(BC)',0        ;(BC)
00951     04AB 2844452900DB$B$DE$RC:   DB      '(DE)',0        ;(DE)
00952     04B0 28484C2900DB$B$HL$RC:   DB      '(HL)',0        ;(HL)
00953     04B5 2842432B31DB$W$BC$RC:   DB      '(BC+1),(BC)',0 ;(BC+1),(BC)
00954     04C1 2844452B31DB$W$DE$RC:   DB      '(DE+1),(DE)',0 ;(DE+1),(DE)
00955     04CD 28484C2B31DB$W$HL$RC:   DB      '(HL+1),(HL)',0 ;(HL+1),(HL)
00956                      ;
00957                      ;       Flags message
00958                      ;
00959     04D9 43785A784DDB$Flags$Msg: DB      'CxZxMxExIx',0  ;Compatible with DDT's display
00960                      ;
00961                      ;       Flags masks used to test user's flag byte
00962                      ;
00963                      DB$Flag$Masks:
00964     04E4 01                      DB      0000$0001B      ;Carry
00965     04E5 40                      DB      0100$0000B      ;Zero
00966     04E6 80                      DB      1000$0000B      ;Minus
00967     04E7 04                      DB      0000$0100B      ;Even parity
00968     04E8 10                      DB      0001$0000B      ;Interdigit carry (aux carry)
00969     04E9 00                      DB      0               ;Terminator
01100                      ;#
01101                      ;       DB$Init
01102                      ;       This routine initializes the debug package.
01103                      ;
01104                      DB$Init:
01105                              IF      DB$BIOS$IO              ;Use BIOS for CONIN/CONOUT
01106                              LHLD    1                       ;Get warm boot address from base
01107                                                              ;  page. H = BIOS jump vector page
01108                              MVI     L,09H                   ;Get CONIN offset in jump vector
01109                              SHLD    BIOS$CONIN + 1          ;Set up address
01110                              MVI     L,0CH                   ;Get CONOUT offset in jump vector
01111                              SHLD    BIOS$CONOUT + 1
01112                              ENDIF
01113
01114                                              ;Set up JMP instructions to receive control
01115                                              ;  when an RST instruction is executed
01116     04EA 3EC3                MVI     A,JMP           ;Set JMP instructions at RST points
```

Figure 10-2. (Continued)

```
01117   04EC 322000             STA     RST4
01118   04EF 322800             STA     RST5
01119   04F2 211A08             LXI     H,DB$Input      ;Address of fake input routine
01120   04F5 222100             SHLD    RST4 + 1
01121   04F8 216C08             LXI     H,DB$Output     ;Address of fake output routine
01122   04FB 222900             SHLD    RST5 + 1
01123
01124   04FE C9                 RET
01200                   ;#
01201                   ;       DB$CONINU
01202                   ;       This routine returns the next character from the console,
01203                   ;       but converting "a" to "z" to uppercase letters.
01204                   ;
01205                   DB$CONINU:
01206   04FF CD0505             CALL    DB$CONIN                ;Get character from keyboard
01207   0502 C31B09             JMP     DB$A$To$Upper           ;Fold to upper and return
01300                   ;#
01301                   ;       DB$CONIN
01302                   ;       This routine returns the next character from the console.
01303                   ;       According to the setting of equates, it uses simple
01304                   ;       polled I/O, the BDOS (function 2) or the BIOS.
01305                   ;
01306                   ;       Exit parameters
01307                   ;
01308                   ;               A = character from console
01309                   ;
01310                   DB$CONIN:
01311                           IF      DB$Polled$IO            ;Simple polled input
01312                           IN      DB$Status$Port          ;Check if incoming data
01313                           ANI     DB$Input$Ready
01314                           JZ      DB$CONIN                ;No
01315                           IN      DB$Data$Port            ;Input data character
01316                           PUSH    PSW                     ;Save data character
01317                           MOV     C,A                     ;Ready for output
01318                           CALL    DB$CONOUT               ;Echo it back
01319                           POP     PSW                     ;Recover data character
01320                           RET
01321                           ENDIF
01322
01323                           IF      DB$BDOS$IO              ;Use BDOS for input
01324   0505 0E01               MVI     C,B$CONIN               ;Read console
01325   0507 C30500             JMP     BDOS                    ;BDOS returns to our caller
01326                           ENDIF
01327
01328                           IF      DB$BIOS$IO              ;Use BIOS for input
01329                           JMP     BIOS$CONIN              ;This was set up during BIOS
01330                                                           ;  initialization
01331                           ENDIF
01332
01400                   ;#
01401                   ;       DB$CONOUT
01402                   ;       This routine outputs the character in the C register to the
01403                   ;       console, using simple polled I/O, the BDOS or the BIOS.
01404                   ;
01405                   ;       Entry parameters
01406                   ;               A = byte to be output
01407                   ;
01408                   DB$CONOUT:
01409   050A 3A0604             LDA     DB$Flag                 ;Check if debug output enabled
01410   050D B7                 ORA     A
01411   050E C8                 RZ                              ;Ignore output if disabled
01412
01413                           IF      DB$Polled$IO            ;Use simple polled output
01414                           IN      DB$Status$Port          ;Check if ready for output
01415                           ANI     DB$Output$Ready
01416                           JZ      DB$CONOUT               ;No
01417                           MOV     A,C                     ;Get data byte
01418                           OUT     DB$Data$Port
01419                           RET
01420                           ENDIF
01421
01422                           IF      DB$BDOS$IO              ;Use BDOS for output
01423   050F 59                 MOV     E,C                     ;Move into correct register
01424   0510 0E02               MVI     C,B$CONOUT
01425   0512 C30500             JMP     BDOS                    ;BDOS returns to our caller
01426                           ENDIF
01427
01428                           IF      DB$BIOS$IO              ;Use BIOS for output
```

Figure 10-2. (Continued)

```
01429                            MOV     A,C                 ;Move into correct register
01430                            JMP     BIOS$CONOUT         ;Set up during debug initialization
01431                            ENDIF
01500                    ;#
01501                    ;
01502                    ;       DB$On
01503                    ;       This routine enables all debug output by setting the
01504                    ;       DB$Flag nonzero.
01505                    ;
01506                    DB$On:
01507   0515 F5                  PUSH    PSW                 ;Preserve registers
01508   0516 3EFF                MVI     A,0FFH
01509   0518 320604              STA     DB$Flag             ;Set control flag on
01510   051B F1                  POP     PSW
01511   051C C9                  RET
01600                    ;#
01601                    ;
01602                    ;       DB$Off
01603                    ;       This routine disables all debug output by setting the
01604                    ;       DB$Flag to zero.
01605                    ;
01606                    DB$Off:
01607   051D F5                  PUSH    PSW                 ;Preserve registers
01608   051E AF                  XRA     A
01609   051F 320604              STA     DB$Flag             ;Clear control flag
01610   0522 F1                  POP     PSW
01611   0523 C9                  RET
01700                    ;#
01701                    ;
01702                    ;       DB$Set$Pass
01703                    ;       This routine sets the pass counter. Subsequent calls to DB$Pass
01704                    ;       decrement the count, and when it reaches 0, debug output
01705                    ;       is enabled.
01706                    ;
01707                    ;       Calling sequence
01708                    ;
01709                    ;               CALL    DB$Set$Pass
01710                    ;               DW      Pass$Count$Value
01711                    ;
01712                    DB$Set$Pass:
01713   0524 220904              SHLD    DB$Save$HL          ;Preserve user's HL
01714   0527 E1                  POP     H                   ;Recover return address
01715   0528 D5                  PUSH    D                   ;Preserve user's DE
01716   0529 5E                  MOV     E,M                 ;Get LS byte of count
01717   052A 23                  INX     H                   ;Update pointer
01718   052B 56                  MOV     D,M                 ;Get MS byte
01719   052C 23                  INX     H                   ;HL points to return address
01720   052D EB                  XCHG                        ;HL = pass counter
01721   052E 220704              SHLD    DB$Pass$Count       ;Set debug pass counter
01722   0531 EB                  XCHG                        ;HL points to return address
01723   0532 D1                  POP     D                   ;Recover user's DE
01724   0533 E3                  XTHL                        ;Recover user's HL and set
01725                                                        ;  return address on top of stack
01726   0534 C9                  RET
01800                    ;#
01801                    ;
01802                    ;       DB$Pass
01803                    ;       This routine decrements the debug pass counter -
01804                    ;       if the result is negative, it takes no further action.
01805                    ;       If the result is zero, it sets the debug control flag nonzero
01806                    ;       to enable debug output.
01807                    ;
01808                    DB$Pass:
01809   0535 F5                  PUSH    PSW                 ;Save user's registers
01810   0536 E5                  PUSH    H
01811   0537 2A0704              LHLD    DB$Pass$Count       ;Get pass count
01812   053A 2B                  DCX     H
01813   053B 7C                  MOV     A,H                 ;Check if count now negative
01814   053C B7                  ORA     A
01815   053D FA4705              JM      DB$Pass$x           ;Yes, take no further action
01816   0540 220704              SHLD    DB$Pass$Count       ;Save downdated count
01817   0543 B5                  ORA     L                   ;Check if count now zero
01818   0544 CA4A05              JZ      DB$Pass$ED          ;Yes, enable debug
01819                    DB$Pass$x:                          ;
01820   0547 E1                  POP     H                   ;Recover user's registers
01821   0548 F1                  POP     PSW
01822   0549 C9                  RET
```

Figure 10-2. (Continued)

```
01823                     ;
01824                     DB$Pass$Ed:                             ;Enable debug
01825   054A 3EFF                 MVI     A,0FFH
01826   054C 320604               STA     DB$Flag                 ;Set debug control flag
01827   054F C34705               JMP     DB$Pass$x
01900                     ;#
01901                     ;
01902                     ;       DB$Display
01903                     ;       This is the primary debug display routine.
01904                     ;
01905                     ;       Calling sequence
01906                     ;
01907                     ;               CALL    DB$Display
01908                     ;               DB      Display$Code
01909                     ;               DB      'Caption String',0
01910                     ;
01911                     ;               Display code identifies which register(s) are to be
01912                     ;               displayed.
01913                     ;
01914                     ;               When the display code specifies a block of memory
01915                     ;               the sequence is:
01916                     ;
01917                     ;               CALL    DB$Display
01918                     ;               DB      Display$Code
01919                     ;               DW      Start$Address,End$Address
01920                     ;               DB      'Caption String',0
01921                     ;
01922                     DB$Display:
01923                     ;
01924                     DB$Display$Enabled:
01925   0552 220904               SHLD    DB$Save$HL              ;Save user's HL
01926
01927   0555 E3                   XTHL                            ;Get return address from stack
01928   0556 220D04               SHLD    DB$Save$RA              ;This gets updated by debug code
01929   0559 E5                   PUSH    H                       ;Save return address temporarily
01930   055A 2B                   DCX     H                       ;Subtract 3 to address call instruction
01931   055B 2B                   DCX     H                       ;  itself
01932   055C 2B                   DCX     H
01933   055D 220F04               SHLD    DB$Call$Address         ;Save actual address of CALL
01934   0560 E1                   POP     H                       ;Recover return address
01935
01936   0561 F5                   PUSH    PSW                     ;Temporarily save flags to avoid
01937                                                             ;  them being changed by DAD SP
01938   0562 210000               LXI     H,0                     ;Preserve stack pointer
01939   0565 39                   DAD     SP
01940   0566 23                   INX     H                       ;Correct for extra PUSH PSW needed
01941   0567 23                   INX     H                       ;  to save the flags
01942   0568 220B04               SHLD    DB$Save$SP
01943   056B F1                   POP     PSW                     ;Recover flags
01944
01945   056C 314C04               LXI     SP,DB$Stack             ;Switch to local stack
01946
01947   056F F5                   PUSH    PSW                     ;Save other user's registers
01948   0570 C5                   PUSH    B                       ;The stack area is specially laid
01949   0571 D5                   PUSH    D                       ;  out to access these registers
01950
01951   0572 2A0D04               LHLD    DB$Save$RA              ;Get return address
01952   0575 7E                   MOV     A,M                     ;Get display code
01953   0576 321504               STA     DB$Display$Code
01954   0579 23                   INX     H                       ;Update return address
01955
01956   057A FE18                 CPI     DB$M                    ;Check if memory to be displayed
01957   057C C29105               JNZ     DB$Not$Memory
01958   057F 5E                   MOV     E,M                     ;Get DE = start address
01959   0580 23                   INX     H
01960   0581 56                   MOV     D,M
01961   0582 23                   INX     H
01962   0583 EB                   XCHG                            ;HL = start address
01963   0584 221104               SHLD    DB$Start$Address
01964   0587 EB                   XCHG                            ;HL -> end address
01965   0588 5E                   MOV     E,M                     ;Get DE = end address
01966   0589 23                   INX     H
01967   058A 56                   MOV     D,M
01968   058B 23                   INX     H
01969   058C EB                   XCHG                            ;HL = end address, DE -> caption
01970   058D 221304               SHLD    DB$End$Address
01971   0590 EB                   XCHG                            ;HL -> caption string
```

Figure 10-2. (Continued)

```
01972                      DB$Not$Memory:
01973                      ;
01974                      ;       Output preamble and caption string
01975                      ;       The format for everything except memory display is:
01976                      ;
01977                      ;       nnnn : Caption String : RC = vvvv
01978                      ;        ^                       ^    ^
01979                      ;       Call Address             :    Value
01980                      ;                         Register Caption (A, B, C...)
01981                      ;
01982                      ;       A carriage return, line feed is output at the start of the
01983                      ;       message - but NOT at the end.
01984                      ;
01985                      ;       Memory displays look like :
01986                      ;
01987                      ; nnnn : Caption String : Start, End  ssss, eeee
01988                      ; ssss : hh hh hh hh hh hh hh hh hh hh hh hh hh hh hh hh : cccc cccc cccc cccc
01989                      ;
01990
01991   0591 E5                    PUSH    H                       ;Save pointer to caption string
01992   0592 CDC107                CALL    DB$CRLF                 ;Display carriage return, line feed
01993   0595 CD7C07                CALL    DB$Display$CALLA        ;Display DB$Call$Address in hex.
01994
01995   0598 E1                    POP     H                       ;Recover pointer to caption string
01996                      DB$Display$Caption:                     ;HL -> caption string
01997   0599 7E                    MOV     A,M                     ;Get character
01998   059A 23                    INX     H
01999   059B B7                    ORA     A                       ;Check if end of string
02000   059C CAA805                JZ      DB$End$Caption          ;Yes
02001
02002   059F E5                    PUSH    H                       ;Save string pointer
02003   05A0 4F                    MOV     C,A                     ;Ready for output
02004   05A1 CD0A05                CALL    DB$CONOUT               ;Display character
02005   05A4 E1                    POP     H                       ;Recover string pointer
02006   05A5 C39905                JMP     DB$Display$Caption      ;Go back for next character
02007                      ;
02008                      DB$End$Caption:
02009   05A8 220D04                SHLD    DB$Save$RA              ;Save updated return address
02010
02011   05AB CDC807                CALL    DB$Colon                ;Display ' : '
02012
02013                                                              ;Display register caption
02014   05AE 3A1504                LDA     DB$Display$Code         ;Get user's display code
02015   05B1 5F                    MOV     E,A                     ;Make display code into word
02016   05B2 1600                  MVI     D,0
02017   05B4 D5                    PUSH    D                       ;Save word value for later
02018
02019   05B5 FE18                  CPI     DB$M                    ;Memory display is a special case
02020   05B7 CACF05                JZ      DB$Display$Mem$Caption  ;Yes
02021
02022   05BA 214C04                LXI     H,DB$Register$Captions  ;Make pointer to address in table
02023   05BD 19                    DAD     D                       ;HL -> word containing address of
02024                                                              ;  register caption
02025   05BE 5E                    MOV     E,M                     ;Get LS byte of address
02026   05BF 23                    INX     H
02027   05C0 56                    MOV     D,M                     ;DE -> register caption string
02028   05C1 EB                    XCHG                            ;HL -> register caption string
02029   05C2 CDEE07                CALL    DB$MSG                  ;Display message addressed by HL
02030
02031   05C5 CDD607                CALL    DB$MSGI                 ;Display in-line message
02032   05C8 203D2000              DB      ' = ',0
02033   05CC C3ED05                JMP     DB$Select$Routine       ;Go to correct processor
02034                      ;
02035                      DB$Display$Mem$Caption:                 ;The memory display requires a special
02036                                                              ;  caption with the start and end
02037                                                              ;  addresses
02038   05CF 219204                LXI     H,DB$M$RC               ;Display specific caption
02039   05D2 CDEE07                CALL    DB$MSG
02040   05D5 CDC807                CALL    DB$Colon                ;Display ' : '
02041
02042   05D8 2A1104                LHLD    DB$Start$Address        ;Display start address
02043   05DB CD8707                CALL    DB$DHLH                 ;Display HL in hex.
02044
02045   05DE CDD607                CALL    DB$MSGI                 ;Display in-line message
02046   05E1 2C2000                DB      ', ',0
02047
02048   05E4 2A1304                LHLD    DB$End$Address          ;Get end address
```

Figure 10-2. (Continued)

```
02049   05E7 CD8707         CALL    DB$DHLH                 ;Display HL in hex.
02050   05EA CDC107         CALL    DB$CRLF                 ;Display carriage return, line feed
02051                                                       ;Drop into select routine
02052                   DB$Select$Routine:
02053   05ED D1             POP     D                       ;Recover word value Display$Code
02054   05EE 210A06         LXI     H,DB$Display$Table
02055   05F1 19             DAD     D                       ;HL -> address of code to process
02056                                                       ;  display requirements
02057   05F2 5E             MOV     E,M                     ;Get LS byte of address
02058   05F3 23             INX     H                       ;Update pointer
02059   05F4 56             MOV     D,M                     ;Get MS byte of address
02060   05F5 EB             XCHG                            ;HL -> code
02061
02062   05F6 11FB05         LXI     D,DB$Exit               ;Fake link on stack
02063   05F9 D5             PUSH    D
02064   05FA E9             PCHL                            ;"CALL" display processor
02065                   ;
02066                   DB$Exit:                            ;Return to the user
02067   05FB D1             POP     D                       ;Recover user's registers saved
02068   05FC C1             POP     B                       ;  on local debug stack
02069   05FD F1             POP     PSW
02070   05FE 2A0B04         LHLD    DB$Save$SP              ;Revert to user's stack
02071   0601 F9             SPHL
02072   0602 2A0D04         LHLD    DB$Save$RA              ;Get updated return address (bypasses
02073                                                       ;  in-line parameters)
02074   0605 E3             XTHL                            ;Replace on top of user's stack
02075   0606 2A0904         LHLD    DB$Save$HL              ;Get user's HL
02076   0609 C9             RET                             ;Transfer to correct return address
02077
02078
02079                   DB$Display$Table:
02080   060A 3006           DW      DP$F            ;Flags
02081   060C 5406           DW      DP$A            ;A register
02082   060E 5A06           DW      DP$B            ;B
02083   0610 6006           DW      DP$C            ;C
02084   0612 6606           DW      DP$D            ;D
02085   0614 6C06           DW      DP$E            ;E
02086   0616 7206           DW      DP$H            ;H
02087   0618 7806           DW      DP$L            ;L
02088   061A 7E06           DW      DP$BC           ;BC
02089   061C 8406           DW      DP$DE           ;DE
02090   061E 8A06           DW      DP$HL           ;HL
02091   0620 9006           DW      DP$SP           ;Stack pointer
02092   0622 9606           DW      DP$M            ;Memory
02093   0624 4907           DW      DP$B$BC         ;(BC)
02094   0626 5007           DW      DP$B$DE         ;(DE)
02095   0628 5707           DW      DP$B$HL         ;(HL)
02096   062A 5E07           DW      DP$W$BC         ;(BC+1),(BC)
02097   062C 6807           DW      DP$W$DE         ;(DE+1),(DE)
02098   062E 7207           DW      DP$W$HL         ;(HL+1),(HL)
02200                   ;#
02201                   ;       Debug display processing routines
02202                   ;
02203                   DP$F:                           ;Flags
02204                                                   ;The flags are displayed in the same way that
02205                                                   ;  DDT uses: C1Z0M0E0I0
02206   0630 3A4A04         LDA     DB$Save$F       ;Get flags
02207   0633 47             MOV     B,A             ;Preserve copy
02208   0634 21DA04         LXI     H,DB$Flags$Msg + 1      ;HL -> first 0/1 in message
02209   0637 11E404         LXI     D,DB$Flag$Masks ;DE -> table of flag mask values
02210                   DB$F$Next:
02211   063A 1A             LDAX    D               ;Get next flag mask
02212   063B B7             ORA     A               ;Check if end of table
02213   063C CA4E06         JZ      DB$F$Display    ;Yes, display the results
02214
02215   063F A0             ANA     B               ;Check if this flag is set
02216   0640 3E31           MVI     A,'1'           ;Assume yes
02217   0642 C24706         JNZ     DB$F$NZ         ;Yes, it is set
02218   0645 3E30           MVI     A,'0'           ;No, it is clear
02219                   DB$F$NZ:
02220   0647 77             MOV     M,A             ;Store '0' or '1' in message text
02221   0648 23             INX     H               ;Update pointer to next 0/1
02222   0649 23             INX     H
02223   064A 13             INX     D               ;Update flag mask pointer
02224   064B C33A06         JMP     DB$F$Next
02225                   DB$F$Display:                   ;Display results
02226   064E 21D904         LXI     H,DB$Flags$Msg
```

Figure 10-2. (Continued)

```
02227  0651 C3EE07           JMP     DB$MSG          ;Display message and return
02228                ;
02229                DP$A:           ;A register
02230  0654 3A4B04           LDA     DB$Save$A       ;Get saved value
02231  0657 C39107           JMP     DB$DAH          ;Display it and return
02232                ;
02233                DP$B:           ;B
02234  065A 3A4904           LDA     DB$Save$B       ;Get saved value
02235  065D C39107           JMP     DB$DAH          ;Display it and return
02236                ;
02237                DP$C:           ;C
02238  0660 3A4804           LDA     DB$Save$C       ;Get saved value
02239  0663 C39107           JMP     DB$DAH          ;Display it and return
02240                ;
02241                DP$D:           ;D
02242  0666 3A4704           LDA     DB$Save$D       ;Get saved value
02243  0669 C39107           JMP     DB$DAH          ;Display it and return
02244                ;
02245                DP$E:           ;E
02246  066C 3A4604           LDA     DB$Save$E       ;Get saved value
02247  066F C39107           JMP     DB$DAH          ;Display it and return
02248                ;
02249                DP$H:           ;H
02250  0672 3A0A04           LDA     DB$Save$H       ;Get saved value
02251  0675 C39107           JMP     DB$DAH          ;Display it and return
02252                ;
02253                DP$L:           ;L
02254  0678 3A0904           LDA     DB$Save$L       ;Get saved value
02255  067B C39107           JMP     DB$DAH          ;Display it and return
02256                ;
02257                DP$BC:          ;BC
02258  067E 2A4804           LHLD    DB$Save$C       ;Get saved word value
02259  0681 C38707           JMP     DB$DHLH         ;Display it and return
02260                ;
02261                DP$DE:          ;DE
02262  0684 2A4604           LHLD    DB$Save$E       ;Get saved word value
02263  0687 C38707           JMP     DB$DHLH         ;Display it and return
02264                ;
02265                DP$HL:          ;HL
02266  068A 2A0904           LHLD    DB$Save$HL      ;Get saved word value
02267  068D C38707           JMP     DB$DHLH         ;Display it and return
02268                ;
02269                DP$SP:          ;Stack Pointer
02270  0690 2A0B04           LHLD    DB$Save$SP      ;Get saved word value
02271  0693 C38707           JMP     DB$DHLH         ;Display it and return
02272                ;
02273                DP$M:           ;Memory
02274  0696 2A1304           LHLD    DB$End$Address  ;Increment end address to make
02275  0699 23               INX     H               ;  arithmetic easier
02276  069A 221304           SHLD    DB$End$Address
02277
02278  069D 2A1104           LHLD    DB$Start$Address
02279  06A0 CD3A07           CALL    DB$M$Check$End  ;Compare HL to End$Address
02280  06A3 DAD106           JC      DB$M$Address$OK ;End > start
02281  06A6 CDD607           CALL    DB$MSGI         ;Error start > end
02282  06A9 0D0A2A2A20       DB      0DH,0AH,'** ERROR - Start Address > End **',0
02283  06CD C9               RET
02284                ;
02285                DB$M$Next$Line:
02286  06CE CDC107           CALL    DB$CRLF                 ;Output carriage return, line feed
02287                DB$M$Address$OK:                        ;Bypass CR,LF for first line
02288  06D1 CDD607           CALL    DB$MSGI                 ;Indent line
02289  06D4 202000           DB      '  ',0
02290  06D7 2A1104           LHLD    DB$Start$Address        ;Get start of line address
02291  06DA CD8707           CALL    DB$DHLH                 ;Display in hex
02292
02293  06DD CDC807           CALL    DB$Colon                ;Display ' : '
02294
02295  06E0 2A1104           LHLD    DB$Start$Address
02296                DB$M$Next$Hex$Byte:
02297  06E3 E5               PUSH    H                       ;Save memory address
02298  06E4 CDD007           CALL    DB$Blank                ;Output a blank
02299  06E7 E1               POP     H                       ;Recover current byte address
02300  06E8 7E               MOV     A,M                     ;Get byte from memory
02301  06E9 23               INX     H                       ;Update memory pointer
02302  06EA E5               PUSH    H                       ;Save for later
02303  06EB CD9107           CALL    DB$DAH                  ;Display in hex.
02304  06EE E1               POP     H                       ;Recover memory updated address
```

Figure 10-2. (Continued)

```
02305   06FF CD3A07         CALL    DB$M$Check$End         ;Compare HL vs. end address
02306   06F2 CAFE06         JZ      DB$M$Display$ASCII     ;Yes, end of area
02307   06F5 7D             MOV     A,L                    ;Check if at start of new line,
02308   06F6 E60F           ANI     0000$1111B             ; (is address XXX0H?)
02309   06F8 CAFE06         JZ      DB$M$Display$ASCII     ;Yes
02310   06FB C3E306         JMP     DB$M$Next$Hex$Byte     ;No, loop back for another
02311                 ;
02312                 DB$M$Display$ASCII:                  ;Display bytes in ASCII
02313   06FE CDC807         CALL    DB$Colon               ;Display ' : '
02314   0701 2A1104         LHLD    DB$Start$Address       ;Start ASCII as beginning of line
02315                 DB$M$Next$ASCII$Byte:
02316   0704 7E             MOV     A,M                    ;Get byte from memory
02317   0705 E5             PUSH    H                      ;Save memory address
02318   0706 E67F           ANI     0111$1111B             ;Remove parity
02319   0708 4F             MOV     C,A                    ;Prepare for output
02320   0709 FE20           CPI     ' '                    ;Check if non-graphic
02321   070B D21007         JNC     DB$M$Display$Char      ;Char >= space
02322   070E 0E2E           MVI     C,'.'                  ;Display non-graphic as '.'
02323                 DB$M$Display$Char:
02324   0710 FE7F           CPI     7FH                    ;Check if DEL (may be non-graphic)
02325   0712 C21707         JNZ     DB$M$Not$DEL           ;No, it is graphic
02326   0715 0E2E           MVI     C,'.'                  ;Force to '.'
02327                 ;
02328                 DB$M$Not$DEL:
02329   0717 CD0A05         CALL    DB$CONOUT              ;Display character
02330   071A E1             POP     H                      ;Recover memory address
02331   071B 23             INX     H                      ;Update memory pointer
02332   071C 221104         SHLD    DB$Start$Address       ;Update memory copy
02333   071F CD3A07         CALL    DB$M$Check$End         ;Check if end of memory dump
02334   0722 CA3707         JZ      DB$M$Exit              ;Yes, done
02335   0725 7D             MOV     A,L                    ;Check if end of line
02336   0726 E60F           ANI     0000$1111B             ; by checking address = XXX0H
02337   0728 CACE06         JZ      DB$M$Next$Line         ;Yes, start next line
02338   072B 7D             MOV     A,L                    ;Check if extra blank needed
02339   072C E603           ANI     0000$0011B             ; if address is multiple of 4
02340   072E C20407         JNZ     DB$M$Next$ASCII$Byte   ;No -- go back for next character
02341   0731 CDD007         CALL    DB$Blank               ;Yes, output blank
02342   0734 C30407         JMP     DB$M$Next$ASCII$Byte   ;Go back for next character
02343
02344                 ;
02345                 DB$M$Exit:
02346   0737 C3C107         JMP     DB$CRLF                ;Output carriage return, line feed
02347                                                      ; and return
02348                 ;
02349                 DB$M$Check$End:                      ;Compares HL vs End$Address
02350   073A D5             PUSH    D                      ;Save DE (defensive programming)
02351   073B EB             XCHG                           ;DE = current address
02352   073C 2A1304         LHLD    DB$End$Address         ;Get end address
02353   073F 7A             MOV     A,D                    ;Compare MS bytes
02354   0740 BC             CMP     H
02355   0741 C24607         JNZ     DB$M$Check$End$X       ;Exit now as they are unequal
02356   0744 7B             MOV     A,E                    ;Compare LS bytes
02357   0745 BD             CMP     L
02358                 DB$M$Check$End$X:
02359   0746 EB             XCHG                           ;HL = current address
02360   0747 D1             POP     D                      ;Recover DE
02361   0748 C9             RET                            ;Return with condition flags set
02362                 ;
02363                 DP$B$BC:              ;(BC)
02364   0749 2A4804         LHLD    DB$Save$C       ;Get saved word value
02365   074C 7E             MOV     A,M             ;Get byte addressed by it
02366   074D C39107         JMP     DB$DAH          ;Display it and return
02367                 ;
02368                 DP$B$DE:              ;(DE)
02369   0750 2A4604         LHLD    DB$Save$E       ;Get saved word value
02370   0753 7E             MOV     A,M             ;Get byte addressed by it
02371   0754 C39107         JMP     DB$DAH          ;Display it and return
02372                 ;
02373                 DP$B$HL:              ;(HL)
02374   0757 2A0904         LHLD    DB$Save$HL      ;Get saved word value
02375   075A 7E             MOV     A,M             ;Get byte addressed by it
02376   075B C39107         JMP     DB$DAH          ;Display it and return
02377                 ;
02378                 DP$W$BC:              ;(BC+1),(BC)
02379   075E 2A4804         LHLD    DB$Save$C       ;Get saved word value
02380   0761 5E             MOV     E,M             ;Get word addressed by it
02381   0762 23             INX     H
```

Figure 10-2. (Continued)

```
02382   0763 56                 MOV     D,M
02383   0764 EB                 XCHG                    ;HL = word to be displayed
02384   0765 C38707             JMP     DB$DHLH         ;Display it and return
02385                   ;
02386                   DP$W$DE:                ;(DE+1),(DE)
02387   0768 2A4604             LHLD    DB$Save$E       ;Get saved word value
02388   076B 5E                 MOV     E,M             ;Get word addressed by it
02389   076C 23                 INX     H
02390   076D 56                 MOV     D,M
02391   076E EB                 XCHG                    ;HL = word to be displayed
02392   076F C38707             JMP     DB$DHLH         ;Display it and return
02393                   ;
02394                   DP$W$HL:                ;(HL+1),(HL)
02395   0772 2A0904             LHLD    DB$Save$HL      ;Get saved word value
02396   0775 5E                 MOV     E,M             ;Get word addressed by it
02397   0776 23                 INX     H
02398   0777 56                 MOV     D,M
02399   0778 EB                 XCHG                    ;HL = word to be displayed
02400   0779 C38707             JMP     DB$DHLH         ;Display it and return
02401                   ;
02500                   ;#
02501                   ;       DB$Display$CALLA
02502                   ;       This routine displays the DB$Call$Address in hexadecimal,
02503                   ;       followed by " : ".
02504                   ;
02505                   DB$Display$CALLA:
02506   077C E5                 PUSH    H               ;Save caller's HL
02507   077D 2A0F04             LHLD    DB$Call$Address ;Get the call address
02508   0780 CD8707             CALL    DB$DHLH         ;Display HL in hex.
02509   0783 E1                 POP     H               ;Recover caller's HL
02510   0784 C3C807             JMP     DB$Colon        ;Display " : " and return
02511                   ;
02600                   ;#
02601                   ;
02602                   ;       DB$DHLH
02603                   ;       Display HL in hex.
02604                   ;
02605                   ;       Entry parameters
02606                   ;
02607                   ;               HL = value to be displayed
02608                   ;
02609                   DB$DHLH:
02610   0787 E5                 PUSH    H               ;Save input value
02611   0788 7C                 MOV     A,H             ;Get MS byte first
02612   0789 CD9107             CALL    DB$DAH          ;Display A in hex.
02613   078C E1                 POP     H               ;Recover input value
02614   078D 7D                 MOV     A,L             ;Get LS byte
02615   078E C39107             JMP     DB$DAH          ;Display it and return
02616                   ;
02700                   ;#
02701                   ;
02702                   ;       DB$DAH
02703                   ;       Display A register in hexadecimal
02704                   ;
02705                   ;       Entry parameters
02706                   ;
02707                   ;               A = value to be converted and output
02708                   ;
02709                   DB$DAH:
02710   0791 F5                 PUSH    PSW             ;Take a copy of the value to be converted
02711   0792 0F                 RRC                     ;Shift A right four places
02712   0793 0F                 RRC
02713   0794 0F                 RRC
02714   0795 0F                 RRC
02715   0796 CDB407             CALL    DB$Nibble$To$Hex        ;Convert LS 4 bits to ASCII
02716   0799 CD0A05             CALL    DB$CONOUT               ;Display the character
02717   079C F1                 POP     PSW                     ;Get original value again
02718   079D CDB407             CALL    DB$Nibble$To$Hex        ;Convert LS 4 bits to ASCII
02719   07A0 C30A05             JMP     DB$CONOUT               ;Display and return to caller
02800                   ;#
02801                   ;
02802                   ;       DB$CAH
02803                   ;       Convert A register to hexadecimal ASCII and store in
02804                   ;       specified address.
02805                   ;
02806                   ;       Entry parameters
02807                   ;
```

Figure 10-2. (Continued)

```
02808                       ;               A = value to be converted and output
02809                       ;               HL -> buffer area to receive two characters of output
02810                       ;
02811                       ;       Exit parameters
02812                       ;
02813                       ;               HL -> byte following last hex.byte output
02814                       ;
02815                       DB$CAH:
02816   07A3 F5                     PUSH    PSW             ;Take a copy of the value to be converted
02817   07A4 0F                     RRC                     ;Shift A right four places
02818   07A5 0F                     RRC
02819   07A6 0F                     RRC
02820   07A7 0F                     RRC
02821   07A8 CDB407                 CALL    DB$Nibble$To$Hex        ;Convert to ASCII hex.
02822   07AB 77                     MOV     M,A                     ;Save in memory
02823   07AC 23                     INX     H                       ;Update pointer
02824   07AD F1                     POP     PSW                     ;Get original value again
02825   07AE CDB407                 CALL    DB$Nibble$To$Hex        ;Convert to ASCII hex.
02826   07B1 77                     MOV     M,A                     ;Save in memory
02827   07B2 23                     INX     H                       ;Update pointer
02828   07B3 C9                     RET
02900                       ;#
02901                       ;
02902                       ;       Minor subroutines
02903                       ;
02904                       ;
02905                       ;       DB$Nibble$To$Hex
02906                       ;       This is a minor subroutine that converts the least
02907                       ;       significant four bits of the A register into an ASCII
02908                       ;       hex. character in A and C
02909                       ;
02910                       ;       Entry parameters
02911                       ;
02912                       ;               A = nibble to be converted in LS 4 bits
02913                       ;
02914                       ;       Exit parameters
02915                       ;
02916                       ;               A,C  = ASCII hex. character
02917                       ;
02918                       DB$Nibble$To$Hex:
02919   07B4 E60F                   ANI     0000$1111B      ;Isolate LS four bits
02920   07B6 C630                   ADI     '0'             ;Convert to ASCII
02921   07B8 FE3A                   CPI     '9' + 1         ;Compare to maximum
02922   07BA DABF07                 JC      DB$NTH$Numeric  ;No need to convert to A -> F
02923   07BD C607                   ADI     7               ;Convert to a letter
02924                       DB$NTH$Numeric:
02925   07BF 4F                     MOV     C,A             ;For convenience of other routines
02926   07C0 C9                     RET
02927
02928
02929                       ;
02930                       ;       DB$CRLF
02931                       ;       Simple routine to display carriage return, line feed.
02932                       ;
02933                       DB$CRLF:
02934   07C1 CDD607                 CALL    DB$MSGI         ;Display in-line message
02935   07C4 0D0A00                 DB      0DH,0AH,0
02936   07C7 C9                     RET
02937                       ;
02938                       ;       DB$Colon
02939                       ;       Simple routine to display ' : '.
02940                       ;
02941                       DB$Colon:
02942   07C8 CDD607                 CALL    DB$MSGI         ;Display in-line message
02943   07CB 203A2000               DB      ' : ',0
02944   07CF C9                     RET
02945                       ;
02946                       ;       DB$Blank
02947                       ;       Simple routine to display ' '.
02948                       ;
02949                       DB$Blank:
02950   07D0 CDD607                 CALL    DB$MSGI         ;Display in-line message
02951   07D3 2000                   DB      ' ',0
02952   07D5 C9                     RET
03100                       ;#
03101                       ;
03102                       ;       Message processing subroutines
```

Figure 10-2. (Continued)

```
03103                           ;
03104                           ;       DB$MSGI (message in-line)
03105                           ;       Output null-byte terminated message that follows the
03106                           ;       CALL to MSGOUTI
03107                           ;
03108                           ;       Calling sequence
03109                           ;
03110                           ;               CALL    DB$MSGI
03111                           ;               DB      'Message',0
03112                           ;               ... next instruction
03113                           ;
03114                           ;       Exit parameters
03115                           ;               HL -> instruction following message
03116                           ;
03117                           ;
03118                           DB$MSGI:
03119                                                           ;Get return address of stack, save
03120                                                           ;  user's HL on top of stack
03121   07D6 E3                         XTHL                    ;HL -> message
03122
03123   07D7 F5                         PUSH    PSW             ;Save all user's registers
03124   07D8 C5                         PUSH    B
03125   07D9 D5                         PUSH    D
03126                           DB$MSGI$Next:
03127   07DA 7E                         MOV     A,M             ;Get next data byte
03128   07DB 23                         INX     H               ;Update message pointer
03129   07DC B7                         ORA     A               ;Check if null byte
03130   07DD C2E507                     JNZ     DB$MSGIC        ;No, continue
03131
03132   07E0 D1                         POP     D               ;Recover user's registers
03133   07E1 C1                         POP     B
03134   07E2 F1                         POP     PSW
03135   07E3 E3                         XTHL                    ;Recover user's HL from stack, replacing
03136                                                           ;  it with updated return address
03137   07E4 C9                         RET                     ;Return to address after 00-byte
03138                                                           ;  after in-line message
03139                           DB$MSGIC:
03140   07E5 E5                         PUSH    H               ;Save message pointer
03141   07E6 4F                         MOV     C,A             ;Ready for output
03142   07E7 CD0A05                     CALL    DB$CONOUT
03143   07EA E1                         POP     H               ;Recover message pointer
03144   07EB C3DA07                     JMP     DB$MSGI$Next    ;Go back for next char.
03145
03146                           ;
03147                           ;       DB$MSG
03148                           ;       Output null-byte terminated message
03149                           ;
03150                           ;       Calling sequence
03151                           ;
03152                           ;       MESSAGE:        DB      'Message',0
03153                           ;               :
03154                           ;               LXI     H,MESSAGE
03155                           ;               CALL    DB$MSG
03156                           ;
03157                           ;       Exit parameters
03158                           ;               HL -> null byte terminator
03159                           ;
03160                           ;
03161                           DB$MSG:
03162   07EE F5                         PUSH    PSW             ;Save user's registers
03163   07EF C5                         PUSH    B
03164   07F0 D5                         PUSH    D
03165                           DB$MSG$Next:
03166   07F1 7E                         MOV     A,M             ;Get next byte for output
03167   07F2 B7                         ORA     A               ;Check if 00-byte terminator
03168   07F3 CA0008                     JZ      DB$MSG$X        ;Exit
03169   07F6 23                         INX     H               ;Update message pointer
03170   07F7 E5                         PUSH    H               ;Save updated pointer
03171   07F8 4F                         MOV     C,A             ;Ready for output
03172   07F9 CD0A05                     CALL    DB$CONOUT
03173   07FC E1                         POP     H               ;Recover message pointer
03174   07FD C3F107                     JMP     DB$MSG$Next     ;Go back for next character
03175                           ;
03176                           DB$MSG$X:
03177   0800 D1                         POP     D               ;Recover user's registers
03178   0801 C1                         POP     B
03179   0802 F1                         POP     PSW
```

Figure 10-2. (Continued)

```
03180    0803 C9                     RET
03300                       ;#
03301                       ;
03302                       ;        Debug input routine
03303                       ;
03304                       ;        This routine helps debug code in which input instructions
03305                       ;        would normally occur. The opcode of the IN instruction
03306                       ;        must be replaced by a value of 0E7H (RST 4).
03307                       ;
03308                       ;        This routine picks up the port number contained in the byte
03309                       ;        following the RST 4, converts it to hexadecimal, and
03310                       ;        displays the message:
03311                       ;
03312                       ;                Input from port XX :
03313                       ;
03314                       ;        It then accepts two characters (in hex.) from the keyboard,
03315                       ;        converts these to binary in A, and then returns control
03316                       ;        to the byte following the port number
03317                       ;
03318                       ;        *******
03319                       ;        WARNING - This routine uses both DB$CONOUT and BDOS calls
03320                       ;        *******
03321                       ;
03322    0804 496E707574DBIN$Message:   DB      'Input from Port '
03323    0814 5858203A20DBIN$Port:      DB      'XX : ',0
03324                       ;
03325                       ;
03326                       DB$Input:
03327    081A 220904             SHLD    DB$Save$HL      ;Save user's HL
03328    081D E1                 POP     H               ;Recover address of port number
03329    081E 2B                 DCX     H               ;Backup to point to RST
03330    081F 220F04             SHLD    DB$Call$Address ;Save for later display
03331    0822 23                 INX     H               ;Restore to point to port number
03332                                                    ;Note: A need not be preserved
03333    0823 7E                 MOV     A,M             ;Get port number
03334    0824 23                 INX     H               ;Update return address to bypass port number
03335    0825 220D04             SHLD    DB$Save$RA      ;Save return address
03336    0828 C5                 PUSH    B               ;Save remaining registers
03337    0829 D5                 PUSH    D
03338    082A F5                 PUSH    PSW             ;Save port number for later
03339
03340
03341    082B CDB108             CALL    DB$Flag$Save$On ;Save current state of debug flag
03342                                                    ;  and enable debug output
03343
03344    082E CDC107             CALL    DB$CRLF         ;Display carriage return, line feed
03345    0831 CD7C07             CALL    DB$Display$CALLA;Display call address
03346    0834 F1                 POP     PSW             ;Recover port number
03347    0835 211408             LXI     H,DBIN$Port
03348    0838 CDA307             CALL    DB$CAH          ;Convert to hex. and store in message
03349    083B 210408             LXI     H,DBIN$Message  ;Output prompting message
03350    083E CDEE07             CALL    DB$MSG
03351    0841 0E02               MVI     C,2             ;Get 2 digit hex. value
03352    0843 CDCF08             CALL    DB$GHV          ;Returns value in HL
03353    0846 7D                 MOV     A,L             ;Get just single byte
03354
03355    0847 CDBF08             CALL    DB$Flag$Restore ;Restore debug output to previous state
03356
03357    084A D1                 POP     D               ;Recover registers
03358    084B C1                 POP     B
03359    084C 2A0904             LHLD    DB$Save$HL      ;Get previous HL
03360    084F E5                 PUSH    H               ;Put on top of stack
03361    0850 2A0D04             LHLD    DB$Save$RA      ;Get return address
03362    0853 E3                 XTHL                    ;TOS = return address, HL = previous value
03363    0854 C9                 RET
03500                       ;#
03501                       ;
03502                       ;        Debug output routine
03503                       ;
03504                       ;        This routine helps debug code in which output instructions
03505                       ;        would normally occur. The opcode of the OUT instruction
03506                       ;        must be replaced by a value of 0EFH (RST 5).
03507                       ;
03508                       ;        This routine picks up the port number contained in the byte
03509                       ;        following the RST 5, converts it to hexadecimal, and
03510                       ;        displays the message:
03511                       ;
```

Figure 10-2. (Continued)

```
03512                   ;               Output to port XX : AA
03513                   ;
03514                   ;       where AA is the contents of the A register prior to the
03515                   ;       RST 5 being executed.
03516                   ;       Control is then returned to the byte following the port number.
03517                   ;
03518                   ;       *******
03519                   ;       WARNING - This routine uses both DB$CONOUT and BDOS calls
03520                   ;       *******
03521                   ;
03522                   ;
03523   0855 4F75747075DBO$Message:     DB      'Output to Port '
03524   0864 5858203A20DBO$Port:        DB      'XX : '
03525   0869 414100     DBO$Value:      DB      'AA',0
03526                   ;
03527                   ;
03528                   DB$Output:
03529   086C 220904             SHLD    DB$Save$HL      ;Save user's HL
03530   086F E1                 POP     H               ;Recover address of port number
03531   0870 2B                 DCX     H               ;Backup to point to RST
03532   0871 220F04             SHLD    DB$Call$Address ;Save for later display
03533   0874 23                 INX     H               ;Restore to point at port number
03534   0875 324B04             STA     DB$Save$A       ;Preserve value to be output
03535   0878 7E                 MOV     A,M             ;Get port number
03536   0879 23                 INX     H               ;Update return address to bypass port number
03537   087A 220D04             SHLD    DB$Save$RA      ;Save return address
03538   087D C5                 PUSH    B               ;Save remaining registers
03539   087E D5                 PUSH    D
03540   087F F5                 PUSH    PSW             ;Save port number for later
03541
03542   0880 CDB108             CALL    DB$Flag$Save$On ;Save current state of debug flag
03543                                                   ;  and enable debug output
03544
03545   0883 CDC107             CALL    DB$CRLF         ;Display carriage return, line feed
03546   0886 CD7C07             CALL    DB$Display$CALLA;Display call address
03547   0889 F1                 POP     PSW             ;Recover port number
03548   088A 216408             LXI     H,DBO$Port
03549   088D CDA307             CALL    DB$CAH          ;Convert to hex.and store in message
03550
03551   0890 3A4B04             LDA     DB$Save$A
03552   0893 216908             LXI     H,DBO$Value     ;Convert value to be output
03553   0896 CDA307             CALL    DB$CAH          ;Convert to hex.and store in message
03554
03555   0899 215508             LXI     H,DBO$Message   ;Output prompting message
03556   089C CDEE07             CALL    DB$MSG
03557
03558   089F CDBF08             CALL    DB$Flag$Restore ;Restore debug flag to previous state
03559
03560   08A2 D1                 POP     D               ;Recover registers
03561   08A3 C1                 POP     B
03562   08A4 2A0904             LHLD    DB$Save$HL      ;Get previous HL
03563   08A7 E5                 PUSH    H               ;Put on top of stack
03564   08A8 2A0D04             LHLD    DB$Save$RA      ;Get return address
03565   08AB E3                 XTHL                    ;TOS = return address, HL = previous value
03566   08AC 3A4B04             LDA     DB$Save$A       ;Recover A (NOTE: FLAG NOT RESTORED)
03567   08AF C9                 RET
03700                   ;#
03701                   ;
03702                   ;       DB$Flag$Save$On
03703                   ;       This routine is only used for DB$IN/OUT.
03704                   ;       It saves the current state of the debug control flag,
03705                   ;       D$Flag, and then enables it to make sure that
03706                   ;       DB$IN/OUT output always goes out.
03707                   ;
03708   08B0 00         DB$Flag$Previous:       DB      0       ;Previous flag value
03709                   ;
03710                   DB$Flag$Save$On:
03711   08B1 F5                 PUSH    PSW                     ;Save caller's registers
03712   08B2 3A0604             LDA     DB$Flag                 ;Get current value
03713   08B5 32B008             STA     DB$Flag$Previous        ;Save it
03714   08B8 3EFF               MVI     A,0FFH                  ;Set flag
03715   08BA 320604             STA     DB$Flag
03716   08BD F1                 POP     PSW
03717   08BE C9                 RET
03800                   ;#
03801                   ;
```

Figure 10-2. (Continued)

```
03802                   ;       DB$Flag$Restore
03803                   ;       This routine is only used for DB$IN/OUT.
03804                   ;       It restores the debug control flag, DB$Flag, to
03805                   ;       its former state.
03806                   ;
03807                   DB$Flag$Restore:
03808   08BF F5                 PUSH    PSW
03809   08C0 3AB008             LDA     DB$Flag$Previous        ;Get previous setting
03810   08C3 320604             STA     DB$Flag                 ;Set debug control flag
03811   08C6 F1                 POP     PSW
03812   08C7 C9                 RET
03813
03814                   ;
03900                   ;#
03901                   ;
03902                   ;       Get hex. value
03903                   ;
03904                   ;       This subroutine outputs a prompting message, and then reads
03905                   ;       the keyboard in order to get a hexadecimal value.
03906                   ;       It is somewhat simplistic in that the first non-hex value
03907                   ;       terminates the input. The maximum number of digits to be
03908                   ;       converted is specified as an input parameter. If more than the
03909                   ;       maximum number is entered, only the last four are significant.
03910                   ;
03911                   ;*************************************************************
03912                   ;                               W A R N I N G
03913                   ;       DB$GHV will always use the BDOS to perform a read console
03914                   ;       function (#10). Be careful if you use this routine from
03915                   ;       within an executing BIOS.
03916                   ;*************************************************************
03917                   ;
03918                   ;       Entry parameters
03919                   ;
03920                   ;               HL -> 00-byte terminated message to be output
03921                   ;               C = number of hexadecimal digits to be input
03922                   ;
03923                   ;
03924                   DB$GHV$Buffer:                          ;Input buffer for console characters
03925                   DB$GHV$Max$Count:
03926   08C8 00                 DB      0                       ;Set to the maximum number of chars.
03927                                                           ;  to be input
03928                   DB$GHV$Input$Count:
03929   08C9 00                 DB      0                       ;Set by the BDOS to the actual number
03930                                                           ;  of chars. entered
03931                   DB$GHV$Data$Bytes
03932   08CA                    DS      5                       ;Buffer space for the characters
03933                   ;
03934                   ;
03935                   DB$GHV:
03936   08CF 79                 MOV     A,C                     ;Get maximum characters to be input
03937   08D0 FE05               CPI     5                       ;Check against maximum count
03938   08D2 DAD708             JC      DB$GHV$Count$OK         ;Carry set if A < 5
03939   08D5 3E04               MVI     A,4                     ;Force to only four characters
03940                   DB$GHV$Count$OK:
03941   08D7 32C808             STA     DB$GHV$Max$Count        ;Set up maximum count in input buffer
03942   08DA CDEE07             CALL    DB$MSG                  ;Output prompting message
03943   08DD 11C808             LXI     D,DB$GHV$Buffer         ;Accept characters from console
03944   08E0 0E0A               MVI     C,B$READCONS            ;Function code
03945   08E2 CD0500             CALL    BDOS
03946
03947   08E5 0E02               MVI     C,B$CONOUT              ;Output a line feed
03948   08E7 1E0A               MVI     E,0AH
03949   08E9 CD0500             CALL    BDOS
03950
03951   08EC 210000             LXI     H,0                     ;Initial value
03952   08EF 11CA08             LXI     D,DB$GHV$Data$Bytes     ;DE -> data characters
03953   08F2 3AC908             LDA     DB$GHV$Input$Count      ;Get count of characters input
03954   08F5 4F                 MOV     C,A                     ;Keep count in C
03955                   DB$GHV$Loop:
03956   08F6 0D                 DCR     C                       ;Downdate count
03957   08F7 F8                 RM                              ;Return when all done (HL has value)
03958   08F8 1A                 LDAX    D                       ;Get next character from buffer
03959   08F9 13                 INX     D                       ;Update buffer pointer
03960   08FA CD1B09             CALL    DB$A$To$Upper           ;Convert A to uppercase if need be
03961   08FD FE30               CPI     '0'                     ;Check if less than 0
03962   08FF D8                 RC                              ;Yes, terminate
03963   0900 FE3A               CPI     '9' + 1                 ;Check if > 9
03964   0902 DA1009             JC      DB$GHV$Hex$Digit        ;No, it must be numeric
```

Figure 10-2. (Continued)

```
03965   0905 FE41          CPI     'A'                   ;Check if < 'A'
03966   0907 D8            RC                            ;Yes, terminate
03967   0908 FE47          CPI     'F' + 1               ;Check if > 'F'
03968   090A D0            RNC                           ;Yes, terminate
03969   090B D637          SUI     'A' - 10              ;Convert A through F to numeric
03970   090D C31209        JMP     DB$GHV$Shift$Left$4   ;Combine with current result
03971               ;
03972               DB$GHV$Hex$Digit:
03973   0910 D630          SUI     '0'                   ;Convert to binary
03974               DB$GHV$Shift$Left$4:
03975   0912 29            DAD     H                     ;Shift HL left four bits
03976   0913 29            DAD     H
03977   0914 29            DAD     H
03978   0915 29            DAD     H
03979   0916 85            ADD     L                     ;Add binary value in LS 4 bits of A
03980   0917 6F            MOV     L,A                   ;Put back into HL total
03981   0918 C3F608        JMP     DB$GHV$Loop           ;Loop back for next character
04100               ;#
04101               ;
04102               ;       A to upper
04103               ;       Converts the contents of the A register to an uppercase
04104               ;       letter if it is currently a lowercase letter
04105               ;
04106               ;       Entry parameters
04107               ;
04108               ;               A = character to be converted
04109               ;
04110               ;       Exit parameters
04111               ;
04112               ;               A = converted character
04113               ;
04114               DB$A$To$Upper:
04115   091B FE61          CPI     'a'             ;Compare to lower limit
04116   091D D8            RC                      ;No need to convert
04117   091E FE7B          CPI     'z' + 1         ;Compare to upper limit
04118   0920 D0            RNC                     ;No need to convert
04119   0921 E65F          ANI     5FH             ;Convert to uppercase
04120   0923 C9            RET
```

Figure 10-2. Debug subroutines (continued)

```
B>ddt fig10-2.hex<cr>
DDT VERS 2.0
NEXT  PC
0924 0000
-g100<cr>

0116 : Flags : Flags = C1Z0M1E1I0
0120 : A Register : A = AA
012F : B Register : B = BB
013E : C Register : C = CC
014D : D Register : D = DD
015C : E Register : E = EE
016B : H Register : H = FF
017A : L Register : L = 11
0189 : Memory Dump #1 : Start, End Address  : 0108, 0128
  0108 :  05 3E AA 01 CC BB 11 EE : .>*. L;.n
  0110 :  DD 21 11 FF B7 37 CD 52 05 00 46 6C 61 67 73 00 : ]!.. 77MR ..Fl ags.
  0120 :  CD 52 05 02 41 20 52 65 67 : MR.. A Re g

01A0 : Memory Dump #2 : Start, End Address  : 0100, 011F
  0100 :  31 6B 03 CD EA 04 CD 15 05 3E AA 01 CC BB 11 EE : 1k.M j.M. .>*. L;.n
  0110 :  DD 21 11 FF B7 37 CD 52 05 00 46 6C 61 67 73 00 : ]!.. 77MR ..Fl ags.

01B7 : Memory Dump #3 : Start, End Address  : 0101, 0100

** ERROR - Start Address > End **
01CE : Memory Dump #4 : Start, End Address  : 0100, 0100
  0100 :  31 : 1
```

Figure 10-3. Console output from debug testbed run

```
01E5 : BC Register : BC = BBCC
01F5 : DE Register : DE = DDEE
0205 : HL Register : HL = FF11
0215 : SP Register : SP = 0369
022E : Byte at (BC) : (BC) = BC
023F : Byte at (DE) : (DE) = DE
0250 : Byte at (HL) : (HL) = F1
026A : Word at (BC) : (BC+1),(BC) = 0B0C
027B : Word at (DE) : (DE+1),(DE) = 0D0E
028C : Word at (HL) : (HL+1),(HL) = 0F01
Debug output has been re-enabled.
This message should display 5 times
This message should display 5 times
This message should display 5 times
This message should display 5 times
This message should display 5 times
032B : Input from Port 11 : aa

032D : Output to Port 22 : AA
```

Figure 10-3. Console output from debug tested run (continued)

containing all of the symbols in your program, along with their respective addresses. Once the program has been loaded by SID, you can refer to the memory image of your program not by address, but by the actual symbol name from your source code. SID also supports the "pass count" concept when using breakpoints.

ZSID (Z80 Symbolic Debug)

This is the Z80 CPU's version of SID. The mini-assembler/disassembler uses Zilog instruction mnemonics rather than those used by Intel.

Bringing Up CP/M for the First Time

It is much harder to bring up CP/M on a new computer system than to debug an enhanced version on a system already running CP/M. You will often find yourself staring at a programmatic "brick wall" with no adequate debugging tools to assist you.

For example, you install the CP/M system on a diskette (using another CP/M-based computer system), put the diskette into the new computer, and press the RESET button. The disk head loads on the disk, and then — nothing! You cannot use any programs such as DDT or SID because you do not yet have CP/M up and running on the new computer. Or can you?

The answer is, wherever possible, debug the code for the new machine on an existing CP/M system. You may have to "fake" some aspects of the new bootstrap or BIOS so that the act of testing it on the host machine does not interact with the CP/M already running on it.

This scheme permits you to be fairly sure of your program logic before loading the diskette into the new machine. It will help pin down problems caused by hardware problems on the new computer.

The hardest situation of all is if you have only the new computer and the release diskettes from Digital Research. Your only option is to find a way of reading the CP/M image on the release diskette into memory, hand patch in new console and disk drivers (not a trivial task), write the patched image back onto a diskette, and resort to Orville Wright testing.

If you value your time, it is always more cost-effective to use another system with CP/M already installed. This is true even if the two systems do not have the same diskette format. You can still do the bootstrap and build the CP/M image on the host machine. Then download the image directly into the memory of the new machine and write it out to a diskette.

This *downloading* process does require, however, that the new computer have a read-only memory (ROM) monitor program. Depending on the capability of this ROM monitor program, you may have to hand patch into the new machine's memory a primitive "download" program that reads 8-bit characters from a serial port, stacking them up in memory and returning control to the monitor program when you press a keyboard character on the new machine's console. In fact, some ROM monitor programs have a downloading program built in.

Debugging the CP/M Bootstrap Loader

The CP/M bootstrap loader, as you may recall, is written on one of the outermost tracks on a diskette or hard disk. On a standard 8-inch single-sided, single-density diskette, CP/M's bootstrap loader is stored on the first sector of the first track. The loader is brought into memory by firmware that gets control of the CPU when you turn your machine on or press the RESET button.

The bootstrap has to be compact, as the diskette space on which it is stored is limited: no more than 128 bytes for standard 8-inch diskettes. This tends to rule out the use of the debug subroutines already described, so you have to fall back to more primitive techniques.

Testing the Bootstrap Under CP/M

A bootstrap is best developed on a CP/M-based system. The task is easiest of all if you already have CP/M running on your new machine and are simply preparing an enhanced version of the bootstrap loader. In this case, you can test most of the code as though it were a user program running in the transient program area (TPA).

Most bootstraps get loaded into memory at location 0000H, so at the front of the code to be debugged you must put a temporary origin line that reads

```
ORG         100H
```

If you omit this and ask DDT to load the HEX file output by the assembler, it will load at the true origin, 0000H, and wipe out the contents of the base page for the version of CP/M that you are running. This will cause a system crash; you will have to press the RESET button and reload CP/M. When this happens, DDT does not tell you directly that anything is amiss; it just displays a "?" after your request to load the HEX file. You will discover that the system has "gone away" only when you try to do something else.

You also will need to adjust the addresses into which the bootstrap tries to load the CP/M image. If you do not, you will overwrite the version of CP/M presently running.

With these adjustments made, you can load the bootstrap under DDT and watch it execute, confirming that it does load the correct image into the correct addresses for debugging and transfer control to the BIOS jump vector. When everything appears to be functioning correctly, use the IF instruction to disable the debug code, reassemble the bootstrap, and write it onto a diskette. Then put the diskette into drive A and press RESET.

Was the Bootstrap Loaded?

At this point you must establish whether the bootstrap is being loaded into memory when the machine is turned on or RESET is pressed. The best way of doing this, and one that you can leave in place permanently, is to output a sign-on message as soon as the loader gets control. This requires hardware set up to prepare the USART (Universal Synchronous/Asynchronous Receive/Transmit) chip to output data, although some manufacturers write this initialization code into the firmware that loads the bootstrap. A suitable sign-on message would be the following:

```
CP/M Bootstrap Loader : Vn 1.0 11/18/82
```

If you do not see this message, assume that control is *not* being transferred to the bootstrap loader. This will be useful in the future if someone should call you with a complaint that CP/M cannot be loaded. If this message does not appear, they probably do not have CP/M on the disk.

Did the Bootstrap Load CP/M?

This is a harder question to answer than whether the bootstrap itself has been loaded, especially if the bootstrap loader sign-on is displayed and then the system crashes. A sign-on message early in the BIOS cold boot processing can confirm the correct transfer of control into the BIOS.

If the problems with the bootstrap program are severe, you may have to adapt the memory-dump debugging subroutine, dumping the contents of memory to the console in order to see what information the bootstrap loader is placing in memory. Display 100H bytes starting from the front of the BIOS jump vector. This

table has an immediately recognizable pattern of 0C3H values every three bytes.

You should also check to see that the bootstrap is loading the correct number of sectors from the disk into memory. If it loads too few, CP/M may sign on only to crash a few moments later because it attempts either to execute code or access a constant at the end of the BIOS. If the bootstrap loads too many sectors from the disk, the excess may "wrap around" the top of memory and overwrite the bootstrap itself, down at location 0000H, before it has completed its task. In this case, you would see only the sign-on for the bootstrap, not for the BIOS.

Debugging the BIOS

Rather than try to debug the BIOS as a single piece of code, debug it as a series of separate functional modules.

Notwithstanding current "top-down" philosophies of dealing with overall structure first, it can be quicker to debug the low-level subroutines in a device driver first. This gives you a solid base on which to build.

The BIOS can be divided up into its constituent modules as follows:

- Character input
  - Interrupt service
  - Non-interrupt service
- Character output
- Interrupt routines
  - Real time clock
  - Watchdog timers
- Disk drivers
  - High-level (deblocking)
  - Low-level (physical I/O)

Plan to write a *testbed* program for each of these modules. This testbed code serves two purposes; first, it provides a means of transferring control into the module under test in a controlled way. Second, it includes the necessary modules or dummy modules to "fool" the module under test into responding as if it were running in a complete BIOS under CP/M.

Using the testbed, you can check every part of the module's logic except the part that may be time-critical. Problems caused by timing, such as interrupts disabled for too long or code that is too slow or too fast for a particular peripheral controller chip, tend to show up only when you are testing on the final hardware and when you are running your new BIOS under CP/M.

What You Should Test for in the BIOS

Describing fully how to debug each module in the BIOS ould fill several books. Remember that you are trying to establish the *absence* of errors using a technique that, by its very nature, tends to show only their *presence*.

There are two basic approaches to debugging. One is the plodding method, checking every aspect of the code to ensure that every feature really does work. The second is to try to do something useful with the code.

Plan to use both. Start with the plodding method, testing each feature under control of the testbed until you are sure that it is working *in vitro*. When all of the BIOS modules have been tested individually, build a CP/M system and try to do some useful work with it. Trying to use the system for actual work testing *in vitro* can be a good test.

Feature Checklist

Make a list of the specific features included in the various BIOS modules. Then devise specific test sequences that will show that each of the features is working correctly.

The same testbed code can often test all of the features of a driver module. If it cannot, create a new testbed for the more exotic features.

Keep the testbed routines. Experience shows that they are most often needed shortly after you have erased them. Even after you have tested the BIOS, the testbed routines will come in handy if you decide to enhance a particular driver later on. You can extract the driver code from the BIOS, glue it together with the testbed, and test the new feature code in isolation from the BIOS.

The following sections show example testbeds for the various drivers, along with example checklists. These checklists were used to test the example BIOS routines shown in earlier chapters.

Character Drivers

Figure 10-4 shows the code for an example testbed routine for character I/O drivers in the BIOS. This code would be followed by the actual character I/O drivers, exactly as they would appear in the BIOS except that all IN and OUT instructions would be replaced with RST 4's and 5's respectively (see Figure 10-2) so that you could enter input values and inspect output values on the console.

This example contains the initialization code for the debug package shown in Figure 10-2 and the code setting up an RST 6 used to "fake" incoming character interrupts.

The main testbed loop consists of a faked incoming character interrupt followed by optional calls to CONIN or CONOUT, the return of control to DDT, or a loop back to fake another character interrupt. You can only return control to DDT if you used DDT to load the testbed and driver programs in the first place.

```
                ;       Testbed for character I/O drivers in the BIOS
                ;
                ;       The complete source file consists of three components:
                ;
                ;               1. The testbed code shown here
                ;               2. The character I/O drivers destined for the BIOS
                ;               3. The debug package shown in Figure 10-2.
                ;
FFFF =          TRUE    EQU     OFFFFH
0000 =          FALSE   EQU     NOT TRUE

FFFF =          DEBUG   EQU     TRUE            ;For conditional assembly of RST
                                                ;  instructions in place of IN and
                                                ;  OUT instructions in the drivers
0030 =          RST6    EQU     30H             ;Use RST 6 for fake incoming character
                                                ;  interrupt
0100                    ORG     100H
                START:
0100 31D101             LXI     SP,Test$Stack   ;Use a local stack
0103 CDD101             CALL    DB$Init         ;Initialize the debug package
0106 3EC3               MVI     A,JMP           ;Set up RST 6 with JMP opcode
0108 323000             STA     RST6
010B 21D101             LXI     H,Character$Interrupt   ;Set up RST 6 JMP address
010E 223100             SHLD    RST6 + 1
                ;
                ;       Make repeated entry to character interrupt routine
                ;       to ensure that characters can be captured and stored in
                ;       an input buffer
                ;
                Testbed$Loop:
0111 3EAA               MVI     A,0AAH          ;Set registers to known pattern
0113 01CCBB             LXI     B,0BBCCH
0116 11EEDD             LXI     D,0DDEEH
0119 2111FF             LXI     H,0FF11H
011C F7                 RST     6               ;Fake interrupt for incoming character

011D CDD101             CALL    DB$MSGI         ;Display in-line message
0120 0D0A456E74         DB      0DH,0AH,'Enter I to Input Char., O to Output, D to enter
0152 444454203A         DB      'DDT : ',0

0159 CDD101             CALL    DB$CONINU       ;Get uppercase character
015C FE49               CPI     'I'             ;CONIN?
015E CA7201             JZ      Go$CONIN
0161 FE44               CPI     'D'             ;DDT?
0163 CA6E01             JZ      Go$DDT
0166 FE4F               CPI     'O'             ;CONOUT?
0168 CA9101             JZ      Go$CONOUT
016B C31101             JMP     Testbed$Loop    ;Loop back to interrupt again
                Go$DDT:
016E FF                 RST     7               ;Enter DDT (RST 7 set up by DDT)
016F C31101             JMP     Testbed$Loop
                Go$CONIN:
0172 CDD101             CALL    CONST           ;Get console status
0175 CA1101             JZ      Testbed$Loop    ;No data waiting
0178 CDD101             CALL    CONIN           ;Get data from buffer

017B CDD101             CALL    DB$Display      ;Display character returned
017E 02                 DB      DB$A            ;  in A register
017F 434F4E494E         DB      'CONIN returned',0

018E C37201             JMP     Go$CONIN        ;Repeat CONIN loop until no chars.
                                                ;  waiting
                ;
                Go$CONOUT:
0191 CDD101             CALL    CONST           ;Get console status
0194 CA1101             JZ      Testbed$Loop    ;No data waiting
0197 CDD101             CALL    CONIN
019A 4F                 MOV     C,A             ;Ready for output
019B CDD101             CALL    CONOUT          ;Output to console
019E C39101             JMP     Go$CONOUT       ;Repeat while there is still data
                ;
01A1 9999999999         DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
01B1 9999999999         DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
01C1 9999999999         DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
```

Figure 10-4. Testbed for character I/O drivers in the BIOS

```
                    Test$Stack:
                    ;
                    ;       Dummy routines for those shown in other figures
                    ;
                    ;       BIOS routines (Figure 8-10)
                    ;
                    CONST:                  ;BIOS console status
                    CONIN:                  ;BIOS console input
                    CONOUT:                 ;BIOS console output;
                    Character$Interrupt:    ;Interrupt service routine for incoming chars.
                    ;
                    ;       Debug routines (Figure 10-2)
                    ;
                    DB$Init:                ;Debug initialization
                    DB$MSGI:                ;Display message in-line
                    DB$CONINU:              ;Get uppercase character from keyboard
                    DB$Display:             ;Main debug display routine
0002 =              DB$A    EQU     02      ;Display code for DB$Display
```

Figure 10-4. Testbed for character I/O drivers in the BIOS (continued)

Executing an RST 7 without using DDT will cause a system crash, as DDT sets up the necessary JMP instruction at location 0038H in the base page.

The faked incoming character interrupt transfers control directly to the interrupt service routine in the BIOS (see the example in Figure 8-10, line 04902, label Character$Interrupt). This reads the status ports of each of the character devices; you can enter the specific status byte values that you want. If you enter a value that indicates that a data character is "incoming," you will be prompted for the actual 8-bit data value to be "input." You can make the interrupt service routine appear to be inputting characters and stacking characters up in the input buffer. For debugging purposes, reduce the size of the input buffer to eight bytes. Making it larger means you will have to input more characters to test the buffer threshold logic. To check the interrupt service routine, you will pass through the main testbed loop doing nothing but faking incoming character interrupts and entering status and data values. The data characters will then be stacked up in the input buffer.

To check the correct functioning of the interrupt service routines, you can stay in control with DDT from the outset. Alternatively, you can just use DDT to load the testbed/driver HEX file, loop around inputting several characters, and then request that the testbed return control to DDT. Then you can use DDT to inspect the contents of the device table(s) and input buffers.

Another possibility is to create debugging routines that display the contents of the device table in a meaningful way, with each field captioned like this:

```
DEVICE TABLE 0
     Status Port     81   Data Port       80
     Output Ready    01   Input Ready     02
     DTR high        40
     Reset Int. Prt D8    Reset Int. Val. 20
     :
     :
     Status Byte 1
          Output Suspended
          Output Xon Enabled
     :
```

```
:
Buffer Base OE8C
Put Offset      05      Get Offset      01
Char. Count     04      Control Count   00
Data Buffer
41 42 43 44 45 00 00 00
```

This display device table routine will require a fair amount of effort to code and debug — but it will pay dividends. You can obtain a complete "snapshot" of the device table without having to decode hexadecimal memory dumps and individual bits. Constant values in the device tables are also displayed, so that if a bug in your code corrupts the table, you will know about it immediately.

The next section shows examples of the specific tests you need to make, along with a description of the strategy you can use.

Interrupt Service Routine Checklist In a functioning BIOS, control is transferred to the interrupt service module whenever an incoming character causes an interrupt. In the example BIOS in Figure 8-10 (line 4900), the code scans each character device in turn to determine which one is causing the interrupt.

When you are debugging the interrupt service routines using the "fake" input/output instructions, you will have to enter specific status byte values. Refer to the device table declarations in Figure 8-10, line 1500, to determine what values you must enter to make the service routine think that an incoming character is arriving or that data terminal ready (DTR) is high or low.

Start the debugging process using the first device table. Then repeat the tests on the other device tables.

The following is a checklist of features that should be checked in debugging the interrupt service routine:

Are all registers restored correctly on exit from the interrupt servicing?

Using DDT, start execution from the beginning of the testbed. Set a breakpoint (with the G100,nnnn command) to get control back immediately before the CALL Character$Interrupt. Use the X command to display all of the registers, and then, by using the G,nnnn command, you set a breakpoint at the instruction that immediately follows the CALL Character$Interrupt. The character drivers will prompt you for the status values. Enter 00 (which indicates that no character is incoming). Display the registers again — their values should be the same. Remember to check the value of the stack pointer and the amount of the stack area that has been used.

NOTE: Do not be too surprised if you lose control of the machine when you first try this test. You may have some fundamental logic errors initially. If the system crashes, reset it, reload CP/M, and then start the test again. This time, rather than setting the second breakpoint at the instruction following the CALL Character$Interrupt, venture down into the Character$Interrupt code and go through the code a few instructions

at a time, setting breakpoints before any instructions that could cause a transfer of control. Find out how far you are getting into the driver before it either jumps off into space or settles into a loop.

Does the service routine push a significant number of bytes onto the stack after an interrupt has occurred?

When you get control back after the CALL Character$Interrupt, use the D (dump) command to dump the stack area's memory on the console. Check how far down the stack came by looking for the point where the constants that used to fill the stack area are overwritten by other data.

The example BIOS in Figure 8-10 saves only the contents of the HL register pair on the pre-interrupt stack. It then switches over to a private BIOS stack to save the contents of the rest of the registers and service the interrupt.

Are data characters added to the input buffer correctly?

"Input" a noncontrol character via the Character$Interrupt routine. Then check the contents of the appropriate device table. The character count and the put offset should both be set to one. Then check the contents of the input buffer itself; does it contain the character that you "input?"

Are control characters added to the input buffer correctly?

"Input" a control character such as 01H. Do not use ETX, ACK, XON, or XOFF (03H, 06H, 11H, and 13H, respectively); these may cause side effects if you have errors in the protocol handling logic. Check that the character is stored in the next byte of the input buffer and that the character and control counts are set to two and one, respectively. The put offset should also be set to two.

When the input buffer full threshold is reached, does the driver output the correct protocol character?

Set the first status byte in the first device table to enable input XON or RTS protocol, or both. Then go round the main testbed loop putting characters into the input buffer. Check the console display to see if the drivers output the correct values when the buffer is almost full (the default threshold is when five bytes remain). The driver should then drop the RTS line or output an XOFF character or both, according to the input protocol that you enabled.

When the input buffer is completely full, does the driver respond correctly?

This is an extension of the test above. Input one more character than can fit into the buffer. Check to see that the drivers do not stack the character into the input buffer and that a BELL character (07H) is output to the data port.

Are protocol characters XON/XOFF recognized and the necessary control flags set or reset?

Reload the testbed and drivers. Set the status byte to enable the output XON/XOFF protocol. Then use the Character$Interrupt routine to input an XOFF character (13H). Check to see that the XOFF character has not been put into the input buffer. Instead, the status byte should be set to indicate that output has indeed been suspended.

Input an XON and check to see that the output suspended flag has been reset.

Does the driver detect and reset hardware errors correctly?

Proceed as though you were going to input a character into the input buffer, but instead enter a status byte value that indicates that a hardware error has occurred (enter the value given in the device table for DT$Detect$Error$Value).

Check that the driver detects the error status and outputs the correct error-reset value to the appropriate control port.

Non-interrupt Service Routine Checklist In a "live" BIOS, non-interrupt service routines are accessed via the CONIN and CONST entry points in the BIOS jump vector. During debugging, the testbed can call the CONIN and CONST code directly.

Is input redirection functioning? Does control arrive in the driver with the correct device table selected?

This is best tested directly with DDT. Use the Gnnnn,bbbb command to transfer control into the CONIN code with a breakpoint at the RET instruction at the end of the Select$Device$Table routine (see Figure 8-10, line 04400). Check that the DE register pair is pointing at device table 0. If it is not, you will have to restart the test. Use the Tn command to make DDT trace through the Select$Device$Table subroutine to find the bug.

Are characters returned correctly from the buffer?

Use the testbed to "input" a character or two. Then use the testbed to make several entries into CONIN. Check the characters returned from the buffer.

Are the data character and control character counts correctly decremented?

After each character has been removed from the buffer by CONIN, use DDT to examine the device table and check that the data character and control character counts have been decremented correctly. Also check that the get pointer has moved up the input buffer.

When the buffer "almost empty" threshold is reached, does the driver emit the correct protocol character or manipulate the request to send (RTS) line correctly?

Use DDT to enable the input RTS or XON protocol or both. Then input characters into the input buffer until it reaches the buffer full threshold (the

default is when only five spare bytes remain in the buffer). Confirm that "buffer almost full" processing occurs. Then make repetitive calls to CONIN to flush data out of the buffer. Check that the "buffer emptying" processing occurs when the correct threshold is reached. For RTS protocol, the driver should output a raise RTS value to the specified RTS control port. For XON, the driver should output an XON character to the data port (after first having read the status port to ensure that the hardware can output the character).

Does the driver handle buffer "wraparound" correctly?

Input characters to the input buffer until it becomes completely full. Then make a single CONIN call to remove the first character from the buffer. Follow this by inputting one more character to the buffer. Check that the get pointer is set to one and the put pointer set to zero.

Next, make successive CONIN calls to empty the buffer. Then input one more character to the buffer. Check that this last character is put into the first byte of the input buffer.

Can the driver handle "forced input" correctly?

Using DDT, set the forced input pointer to point to a 00-byte-terminated string; for example, use one of the function key decode default strings. (In Figure 8-10, the forced input pointer is initialized to point to a "startup string"—this is declared at the beginning of the configuration block at line 00400.)

Using DDT, call the CONST routine and check that it returns with A= 0FFH (indicating that there appears to be input data waiting).

Make successive calls to CONIN and confirm that the data bytes in the forced input string are returned. Check that the forcing of input ends when the 00H-byte is detected.

Does the console status routine operate correctly when it checks for data characters in the buffer, control characters in the buffer, and forced input?

Input a single noncontrol character, such as 41H, into the input buffer. Using DDT, check that the second status byte in the device table has the fake type-ahead flag set to zero. Call the CONST routine—it should return with A= 0FFH (meaning that there is data in the buffer). Then set the fake type-ahead bit in the second status byte and call CONST again. It should return with A = 00H (meaning that there is now "no data" in the buffer). Input a single control character into the buffer. Now CONST should return with A = 0FFH because there is a control character in the buffer.

Does the driver recognize escape sequences incoming from keyboard function keys?

This is a difficult feature to test when the real time clock routine is not running. The driver uses the watchdog timer to wait until all characters in

the escape sequence have arrived. You will therefore have to modify the code in CONIN so that the watchdog timer appears to time out immediately, rather than waiting for the real time clock to tick. To make this change, refer to Figure 8-10, line 2200; this is the start of the CONIN routine. Look for the label CONIN$Wait$For$Delay. A few instructions later there is a JNZ CONIN$Wait$For$Delay. Using DDT, set all three bytes of this JNZ to 00H.

Then, using the testbed, input the complete escape sequence into the input buffer. For example, input hexadecimal values 1B, 4F, 51 (ESCAPE, O, P), which correspond to the characters emitted on a VT-100 terminal when FUNCTION KEY 1 (PF1) is pressed.

Next, use the testbed to make successive calls to CONIN. You should see the text associated with the function key (FUNCTION KEY 1, LINE FEED) being returned by CONIN.

Repeat this test using different function key sequences, including a sequence that does not correspond to any of the preset function keys. Check that the escape sequence itself is returned by CONIN without being changed into another string.

Can the driver differentiate between a function key and the same escape sequence generated by discrete key strokes?

This is almost the same test as above. Make the same patch to the CONIN code, only this time do not enter the complete escape sequence into the buffer. Enter only the hex characters 1B and 4F. Make sure that the CONIN routine does not substitute another string in place of this quasi-escape sequence.

This test only mimics the results of manually entering an escape sequence. You could not press the keys on a terminal fast enough to get all three characters into the input buffer within the time allowed by the watchdog timer.

Character Output Checklist

Can the driver output a character?

The CONOUT option in the testbed calls CONIN first to get a character. To start with, you may want to use DDT to set the C register to some graphic ASCII character such as 41H (A), and transfer control into CONOUT directly. Check that CONOUT reads the USART's status, waits for the output ready value, and then outputs the data to the data port. Note that the testbed will output all characters waiting in the input buffer (or forced input) when you select its CONOUT option. This is a convenience for advanced testing of the drivers—for initial testing you may want to modify the testbed to make only one call to CONIN and CONOUT and then return to the top of the testbed loop.

Does the driver suspend output when a protocol control flag indicates that output is to be suspended?

Using DDT, set the status byte in the device table to enable output XON/XOFF protocol. Then input an XOFF character and confirm that the output suspended bit in the status byte is set. Output a single character, and using DDT, confirm that the driver will remain in a status loop waiting for the output suspended bit to be cleared. Clear the bit using DDT and check that the character is output correctly.

When using ETX/ACK protocol, does the driver output an ETX after the specified number of characters have been output, then indicate that output is suspended?

For debugging purposes, alter the ETX message count value in the device table to three bytes. Then output three bytes of data via CONOUT. Check that the driver sends an ETX character (03H) after the three bytes have been output and that the output suspended flag in the status byte has been set.

Then input an ACK character (06H). Check that this character is not stored in the input buffer and that the output suspended flag is cleared.

Does the driver recognize and output escape sequences?

Input an ESCAPE, "t" (1BH, 74H) into the input buffer. Then output them via CONOUT. Using DDT, check that the CONOUT routine recognizes that an escape sequence is being output and selects the correct processing routine. In this case, the forced input pointer should be set to point at the ASCII time of day in the configuration block.

Does each of the escape sequence processors function correctly? Can the time and date be set to specified values using escape sequences?

Repeat the test above using all of the other escape sequences to make sure that they can be recognized and that they function correctly.

Real Time Clock Routines

A separate testbed program, shown in Figure 10-5, is used to check these routines. It calls the interrupt service routine directly to simulate a real time clock "tick," and then displays the time of day in ASCII on the console.

As you can see, the testbed makes a call into the debug package's initialization routine, DB$Init, and then uses an RST 6 to generate fake clock "ticks."

There is a JMP instruction in the testbed that bypasses a call to Set$Watchdog. Remove this JMP, either by editing it out or by using DDT to change it to NO OPERATIONs (NOP, 00H) when you are ready to test the watchdog routines.

Real Time Clock Test Checklist

Is the clock running at all?

Using DDT, trace through the interrupt service routine logic. Check that the seconds are being updated.

```
                ;       Testbed for real time clock driver in the BIOS.
                ;
                ;       The complete source file consists of three components:
                ;
                ;               1. The testbed code shown here
                ;               2. The real time clock driver destined for the BIOS.
                ;               3. The debug package shown in Figure 10-2.
                ;
FFFF =          TRUE    EQU     0FFFFH
0000 =          FALSE   EQU     NOT TRUE

FFFF =          DEBUG   EQU     TRUE            ;For conditional assembly of RST
                                                ;  instructions in place of IN and
                                                ;  OUT instructions in the drivers.
0030 =          RST6    EQU     30H             ;Use RST 6 for fake clock tick.

0100                    ORG     100H
                START:
0100 318B01             LXI     SP,Test$Stack   ;Use local stack
0103 CD8B01             CALL    DB$Init         ;Initialize the debug package
0106 3EC3               MVI     A,JMP           ;Set up RST 6 with JMP opcode
0108 323000             STA     RST6
010B 218B01             LXI     H,RTC$Interrupt ;Set up RST 6 JMP address
010E 223100             SHLD    RST6 + 1

0111 C31D01             JMP     Testbed$Loop    ;<=== REMOVE THIS JMP WHEN READY TO
                                                ;     TEST WATCHDOG ROUTINES

0114 013200             LXI     B,50            ;50 ticks before timeout
0117 214201             LXI     H,WD$Timeout    ;Address to transfer to
011A CD8B01             CALL    Set$Watchdog    ;Set the watchdog timer
                ;
                ;
                ;       Make repeated entry to RTC interrupt routine
                ;       to ensure that clock is correctly updated
                ;
                Testbed$Loop:
011D 3EAA               MVI     A,0AAH          ;Set registers to known pattern
011F 01CCBB             LXI     B,0BBCCH
0122 11EEDD             LXI     D,0DDEEH
0125 2111FF             LXI     H,0FF11H
0128 F7                 RST     6               ;Fake interrupt clock

0129 CD8B01             CALL    DB$MSGI         ;Display in-line message
012C 436C6F636B         DB      'Clock =',0

0134 218B01             LXI     H,Time$In$ASCII ;Get address of clock in driver
0137 CD8B01             CALL    DB$MSG          ;Display current clock value
                                                ; (Note: Time$In$ASCII already has
                                                ;  a line feed character in it)
013A CD8B01             CALL    DB$MSGI         ;Display in-line message
013D 0D00               DB      0DH,0           ;Carriage return

013F C31D01             JMP     Testbed$Loop
                ;
                ;       Control arrives here when the watchdog timer times
                ;       out
                WD$Timeout:
0142 CD8B01             CALL    DB$MSGI
0145 0D0A576174         DB      0DH,0AH,'Watchdog timed out',0
015A C9                 RET                     ;Return to watchdog routine
                ;
015B 9999999999         DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
016B 9999999999         DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
017B 9999999999         DW      9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
                Test$Stack:
                ;
                ;       Dummy routines for those shown in other figures
                ;
                ;       BIOS routines (Figure 8-10)
                ;
                RTC$Interrupt:          ;Interrupt service routine for clock tick
                Set$Watchdog:           ;Set watchdog timer
                Time$In$ASCII:          ;ASCII string of HH:MM:SS, LF, 0
                ;
                ;       Debug routines (Figure 10-2)
                ;
                DB$Init:                ;Debug initialization
                DB$MSGI:                ;Display message in-line
                DB$MSG:                 ;Display message
```

Figure 10-5. Testbed for real-time-clock driver in the BIOS

Are the hours, minutes, and seconds carrying over correctly?

Let the testbed code run at full speed. You should see the time being updated on the console display—although it will be updated much more rapidly than real time.

Use DDT to set the minutes to 58 and then let the clock run again. Does it correctly show the hour and reset the minutes to 00? Then set the hours to 11 and the minutes to 58 and let the clock run. Do minutes carry over into hours and are hours reset to 0?

Repeat these tests with the clock update constants set for 24-hour format.

Is the clock interrupt service routine restoring the registers correctly?

Using DDT, check that the registers are still set correctly on return from the clock interrupt service routine.

How much of a load on the pre-interrupt stack is the service routine imposing?

Check the "low water mark" of the preset values remaining in the testbed stack area to see how much of a load the interrupt service routine is imposing on the stack.

Can the watchdog timer be set to a nonzero value? Can it be set back to zero?

Using the second part of the testbed, call the Set$Watchdog routine, and then monitor the testbed's execution as the watchdog timer times out. Check that the registers and stack pointer are set correctly when control is transferred to the timeout routine. Also check that control is returned properly from this routine, and thence from the interrupt service routine.

Disk Drivers

It is only feasible to check the low-level disk drivers in isolation from a real BIOS, as the BDOS interface to the deblocking code is very difficult to simulate. The testbed shown in Figure 10-6 serves only as a time-saver. It does not test the interface to the subroutines. Use DDT to set up the disk, track, and sector numbers, and then monitor the calls into SELDSK, SETTRK, SETSEC, SETDMA, and the read/write routines.

Unless you have the same disk controller on the host system as you do on the target machine, you will have to use the fake input/output system described earlier in this chapter, rather than attempt to read and write on real disks.

You can see that the testbed, after initializing the debugging package, makes calls to SELDSK, SETTRK, SETSEC, and SETDMA. It then calls a low-level read or write routine. The low-level routine called depends on which driver you wish to debug. For the standard floppy diskette driver shown in Figure 8-10, use ReadNoDeblock and WriteNoDeblock. For the 5 1/4-inch diskettes, use Read$Physical and Write$Physical. You will have to use DDT to set up some of the variables required by the low-level drivers that would normally be set up by the deblocking code.

```
                ;       Testbed for disk I/O drivers in the BIOS
                ;
                ;       The complete source file consists of three components:
                ;
                ;               1. The testbed code shown here
                ;               2. The Disk I/O drivers destined for the BIOS
                ;               3. The debug package shown in Figure 10-2.
                ;
FFFF =          TRUE    EQU     0FFFFH
0000 =          FALSE   EQU     NOT TRUE

FFFF =          DEBUG   EQU     TRUE            ;For conditional assembly of RST
                                                ;  instructions in place of IN and
                                                ;  OUT instructions in the drivers.
0100                    ORG     100H
                START:
0100 314704             LXI     SP,Test$Stack   ;Use a local stack
0103 CD4704             CALL    DB$Init         ;Initialize the debug package
                ;
                ;       Make calls to SELDSK, SETTRK, SETSEC and SETDMA,
                ;       then either a read or write routine.
                ;
                Testbed$Loop:
0106 314704             LXI     SP,Test$Stack   ;Use local stack

0109 3A1202             LDA     Logical$Disk    ;Set up for SELDSK call
010C 4F                 MOV     C,A
010D CD4704             CALL    SELDSK

0110 CD4704             CALL    DB$Display      ;Display return value in HL
0113 14                 DB      DB$HL
0114 53454C4453         DB      'SELDSK returned',0

0124 223201             SHLD    DPH$Start       ;Set up to display disk parameter header
0127 111000             LXI     D,16            ;Compute end address
012A 19                 DAD     D
012B 223401             SHLD    DPH$End         ;Store into debug call

012E CD4704             CALL    DB$Display      ;Display DPH
0131 18                 DB      DB$M            ;Memory
                DPH$Start:
0132 0000               DW      0
                DPH$End:
0134 0000               DW      0
0136 53656C6563         DB      'Selected DPH',0

0143 2A1302             LHLD    Track           ;Call SETTRK
0146 E5                 PUSH    H
0147 C1                 POP     B               ;SETTRK needs track in BC
0148 CD4704             CALL    SETTRK

014B 3A1502             LDA     Sector          ;Call SETSEC
014E 4F                 MOV     C,A             ;SETSEC need sector in C
014F CD4704             CALL    SETSEC

0152 011702             LXI     B,Test$Buffer   ;Set DMA address
0155 CD4704             CALL    SETDMA
0158 3A1602             LDA     Write$Disk      ;Check if reading or writing
015B B7                 ORA     A
015C C2D101             JNZ     Test$Write

015F CD4704             CALL    Read$No$Deblock ;*** or Read$Physical depending on which
                                                ;*** drivers you are testing
0162 CD4704             CALL    DB$Display      ;Display return code
0165 02                 DB      DB$A
0166 5465737420         DB      'Test Read returned',0

0179 CD0102             CALL    Check$Ripple    ;Check if ripple pattern in buffer
017C CA0601             JZ      Testbed$Loop    ;Yes, it is correct

017F CD4704             CALL    DB$MSGI         ;Indicate problem
0182 14                 DB      DB$HL           ;Display HL (points to offending byte)
0183 526970706C         DB      'Ripple pattern incorrect. HL -> failure.',0

01AC CD4704             CALL    DB$Display      ;Display test buffer
01AF CD1800             CALL    DB$M            ;Memory
01B2 1702               DW      Test$Buffer
```

Figure 10-6. Testbed for disk I/O drivers in the BIOS

```
01B4 0002                 DW     Test$Buffer$Size
01B6 436F6E7465           DB     'Contents of Test$Buffer',0

01CE C30601               JMP    Testbed$Loop

               Test$Write:
01D1 CDF201               CALL   Fill$Ripple       ;Fill the test buffer with ripple pattern
01D4 CD4704               CALL   Write$No$Deblock;*** or Write$Physical depending on which
                                                   ;*** drivers you are testing

01D7 CD4704               CALL   DB$Display        ;Display return code
01DA 02                   DB     DB$A
01DB 5465737420           DB     'Test Write returned',0

01EF C30601               JMP    Testbed$Loop

               Fill$Ripple:                        ;Fills the Test$Buffer with a pattern
                                                   ;  formed by putting into each byte, the
                                                   ;  least significant 8-bits of the byte's
                                                   ;  address.
01F2 010002               LXI    B,Test$Buffer$Size
01F5 211702               LXI    H,Test$Buffer
               FR$Loop:
01F8 75                   MOV    M,L               ;Set pattern value into buffer
01F9 23                   INX    H                 ;Update buffer pointer
01FA 0B                   DCX    B                 ;Down date count
01FB 79                   MOV    A,C               ;Check if count zero
01FC B0                   ORA    B
01FD C2F801               JNZ    FR$Loop           ;Repeat until zero
0200 C9                   RET
               ;
               Check$Ripple:                       ;Check that the buffer is filled with the
                                                   ;  correct ripple pattern
                                                   ;  Returns with zero status if this is true,
                                                   ;  nonzero status if the ripple is not
                                                   ;  correct. HL point to the offending byte
                                                   ;  (which should  = L)
0201 010002               LXI    B,Test$Buffer$Size
0204 211702               LXI    H,Test$Buffer
               CR$Loop:
0207 7D                   MOV    A,L               ;Get correct value
0208 BE                   CMP    M                 ;Compare to that in the buffer
0209 C0                   RNZ                      ;Mismatch, nonzero already indicated
020A 23                   INX    H                 ;Update buffer pointer
020B 0B                   DCX    B                 ;Downdate count
020C 79                   MOV    A,C               ;Check count zero
020D B0                   ORA    B
020E C20702               JNZ    CR$Loop           ;Repeat until zero
0211 C9                   RET                      ;Zero flag will already be set
               ;
               ;         Testbed variables
               ;
0212 00        Logical$Disk:     DB     0          ;A = 0, B = 1,...
0213 0000      Track:            DW     0          ;Disk track number
0215 00        Sector:           DB     0          ;Disk sector number
0216 00        Write$Disk:       DB     0          ;NZ to write to disk
               ;
0200 =         Test$Buffer$Size          EQU    512      ;<=== Alter as required
0217           Test$Buffer:      DS     Test$Buffer$Size
               ;
0417 9999999999           DW     9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
0427 9999999999           DW     9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
0437 9999999999           DW     9999H,9999H,9999H,9999H,9999H,9999H,9999H,9999H
               Test$Stack:
               ;
               ;         Dummy routines for those shown in other figures
               ;
               ;         BIOS routines (Figure 8-10)
               ;
               SELDSK:                   ;Select logical disk
               SETTRK:                   ;Set track number
               SETSEC:                   ;Set sector number
               SETDMA:                   ;Set DMA address
               Read$No$Deblock:          ;Driver read routines
               Read$Physical:
               Write$No$Deblock:         ;Driver write routines
               Write$Physical:
```

Figure 10-6. (Continued)

```
                ;
                ;       Debug routines (Figure 10-2)
                ;
                DB$Init:                ;Debug initialization
                DB$MSGI:                ;Display message in-line
                DB$Display:             ;Main debug display routine
0002 =          DB$A    EQU     02      ;Display codes for DB$Display
0014 =          DB$HL   EQU     20
0018 =          DB$M    EQU     24
```

Figure 10-6. Testbed for disk I/O drivers in the BIOS (continued)

Before issuing the write call, the testbed fills the disk buffer with a known pattern. This pattern is checked on return from a read operation.

For both reading and writing, the testbed shows the contents of the A register. If you have added the enhanced disk error handling described in the previous chapter, the return value in A must *always* be zero.

Disk Driver Checklist *Does SELDSK return the correct address and set up the required system variables?*

Check that the correct disk parameter header address is returned for legitimate logical disks. Check, too, that it returns an address of 0000H for illegal disks.

Check that any custom processing, such as setting the disk type and deblocking requirements from extra bytes on the disk parameter blocks, is performed correctly.

Does the SETTRK and SETSEC processing function correctly?

Using DDT, check that the correct variables are set to the specified values.

Does the driver read in the spare-sector directory correctly?

Set up to execute a physical read and, using DDT, trace the logic of the READ entry point. Check that the spare-sector directory would be loaded into the correct buffer. If you are using fake input/output, use DDT to patch in a typical spare-sector directory with two or three "spared-out" sectors.

Does the driver produce the correct spare sector in place of a bad one?

Continuing with the physical read operation, check that, for "good" track/sectors, the sector-sparing logic returns the original track and sector number, and for "bad" track/ sectors, it substitutes the correct spare track and sector. If you are using sector skipping, check that the correct number of sectors is skipped.

Can a sector be read in from the disk?

Continuing further with the physical read, check that the correct sector is read from the specified disk and track. If you are using real I/O (as

opposed to faking it), the "ripple pattern" set by the testbed can be used, or you can fill the disk buffer area with some known pattern (using DDT's F command) so you can tell if any data gets read in.

Make sure you do not have any disks or diskettes in the computer system that are not write-protected — you may inadvertently write on a disk rather than read it during the early stages of testing.

Can a sector be written to the disk?

Using DDT, set up to write to a particular disk, track, and sector. Remove any write protection that you put on the target disk during earlier testing. You can either use the testbed's ripple pattern or fill the disk buffer area with a distinctive pattern. Write this data onto the disk, fill the buffer area with a *different* pattern, and read in the sector that you wrote. Check that the disk buffer gets changed back to the pattern written to the disk.

Does the driver display error messages correctly?

Rather than deliberately damaging a diskette to create errors, use DDT to temporarily sabotage the disk driver's logic. Make it return each of the possible error codes in turn, checking each time that the correct error message is displayed.

For each error condition in turn, check that the disk driver performs the correct recovery action, including interacting with the user and offering the choice of retrying, ignoring the error, or aborting the program.

Live Testing a New BIOS

Given that the drivers have passed all of the testing outlined above, you are ready to pull all of the BIOS pieces together and build a CP/M image.

For your initial testing, disable the real time clock, and use simple, polled I/O for the console driver if you can. It is important to get *something* up and running as soon as possible, and it is easier to do this without possible side effects from interrupts.

Prepare a complete listing of the BIOS and plan to spend at least an hour checking through it. Take a dry run through the console and disk driver — if there are any serious bugs left in these two drivers, CP/M may not start up. Remember that once the BIOS cold boot code has been executed and control is handed over to the CCP, the BDOS will be requested to log in the system disk, and this involves reading in the disk's directory.

Pay special attention to checking some of the major data structures. Make certain that everything is at a reasonable place in memory; for example, if the last address used by the BIOS is greater than 0FFFFH, you will need to move the entire CP/M image down in memory.

Then build a system disk, load it into the machine, and press the RESET button. You should see the bootstrap sign on, then the BIOS, and after a pause of about one second, the A> prompt (or 0A> if you have included the special feature that patches the CCP).

If you see both sign-on messages but do not get an A> prompt, a likely cause of the problem is in the disk drivers. Alternatively, the directory area on the disk may be full of random data rather than 0E5H's.

If you cannot see what is wrong with the system, you might try faking the disk drivers to return a 128-byte block of 0E5H's for each read operation. The CCP should then sign on.

Once you do have the A> prompt, you can proceed with the system checkout. Start by checking that the warm boot logic works. Type a CONTROL-C. There should be a slight pause, and the A> prompt should be output again.

Next, check that you can read the disk directory by using the DIR command. If you have an empty directory, you should get a NO FILE response. If you get strange characters instead, you either forgot to initialize the directory area or the disk parameter block is directing CP/M to the wrong part of the disk for the file directory. If the system crashes, there is a problem with the disk driver.

Check that you can write on the disk by entering the command SAVE 1 TEST. Then use the DIR command to confirm that file TEST shows up in the file directory. If it does, use the ERA command ERA TEST and do another DIR command to confirm that TEST has indeed been erased.

If TEST either does not show up on the disk or cannot be erased, then you have a problem with the disk driver WRITE routine.

Put a standard CP/M release diskette into drive B and use the DIR command to check that you can access the drive and display a disk directory. If you do, then load the DDT utility and exit from it by using a G0 (G, zero) command. This further tests if the disk drivers are functioning correctly.

To test the deblocking logic (if you are using disks that require deblocking), use the command:

```
PIP A:=B:*.*[V]
```

This copies all files from drive B to drive A using the verify option. It is a particularly good test of the system, and if you have any problems with the high-level disk drivers and deblocking code, you will get a Verify Error message from PIP. You can also get this message if you have hardware problems with the computer's memory, so run a memory test if you cannot find anything obviously wrong with the deblocking algorithm.

To completely test the deblocking code, you need to use PIP to copy a file of text larger than the amount of memory available. Thus, you may have to create a large text file using a text editor just to provide PIP with test data.

With the disk driver functioning correctly, rebuild the system with the real time clock enabled. Bring up the new system and check that the ASCII time of day is

being updated in the configuration block; use DDT to inspect this in memory. Set the clock to the current time, let it run for five minutes, and see if it is still accurate. You may have to adjust one of the initialization time constants for the device that is providing the periodic interrupts for the clock.

Rebuild the system yet again, this time with the real interrupt-driven console input and the real console output routines. Check that the system comes up properly and that the initial forced-input startup string appears on the console.

Check that when you type characters on the keyboard they are displayed as you type them. If not, there could be a problem with either the CONIN or CONOUT routines. Experimentally type in enough characters to fill the input buffer. If the terminal's bell starts to sound, the interrupt service routine is probably not the culprit. Check the CONOUT routine again.

Check that the function key decode logic is working correctly. With the A> prompt displayed, press a function key. The CONIN driver should inject the correct function key string and it should appear on the terminal. For example, with the BIOS in Figure 8-10, pressing PF1 on the VT-100 terminal should produce this on the display:

```
A>Function Key1
Function?
A>
```

The CCP does not recognize "Function" as a legitimate command name, nor is there such a COM file—hence the question mark.

Using DDT, write a small program that outputs ESCAPE, "t" to the console, and check that the ASCII time of day string appears on the console. This checks that the escape sequence has been recognized.