# Section 1

## THE SYSTEM 88 SYSTEM PROGRAMMER'S GUIDE

1.0

This manual is the preliminary release of the System Programmer's Guide for the System 88. The complete manual will be released with the second release of System 88 software. Sections 5 and 8, referenced in the text, are not present in this preliminary edition, nor are the appendicies and index; these will be present in the released manual. This preliminary manual also reflects a system in transition; some of the features described are based on material added with the second release.

### 1.1 PURPOSE OF THIS MANUAL

The purpose of this manual is to assist the systems developer in building and tailoring products based on the System 88. To make use of the facilities of the system that are provided at the machine language level, a detailed functional description is required. The user of this manual is expected to be experienced with assembly language program development, and in particular with the 8080 microprocessor. Experience with, and knowledge of such topics as general system organization and data structures are assumed. This is not a manual for the novice.

This manual was written by the designer of the System 88 software. In preparing this manual, a difficult decision had to be made between the desire to protect the proprietary nature of the system, and the wish to provide the systems programmer with the information needed to be able to make use of the extensive facilities provided by the system. The writer of this manual hopes that the proprietary nature of this manual will be respected; the only effective recourse against promiscuous duplication and release of this material is to stop making it available at all.

### 1.2 THE DISK ACCOMPANYING THIS MANUAL

The disk included with this manual is a complete System Disk and includes a number of programs for use by the systems programmer. These include Emedit, the error message editor; Szap, a disk utility; and Auth, an authorization program. The utilities Szap and Emedit should be used with care, and by the experienced programmer.

### 1.3 SECTIONS OF THIS MANUAL

Section 1 of the preliminary release of the System Programmer's Guide is this introduction. Section 2 begins the technical body of the manual, and is a description of the System 88 file system. It describes in detail the structure and allocation of files and disk directory space. The first part of Section 3 describes the data areas of the system. The second part of Section 3 describes the utilities and primitives available to the assembly language programmer. Of particular interest is section 3.4, on overlays. Section 4 describes the utility programs provided for the system programmer, Emedit, Szap, and Auth. Section 5 discusses the requirements for a user-written Exec. (This section is not present in the preliminary manual). Section 6 is a listing of the System 88 error messages. Section 7 gives a listing of a sample system overlay. Section 8 discusses various algorithms used in the system, such as those for packing the disk, and system startup. (This section is not present in the preliminary edition of this manual). Section 9 describes the format of the HELP file used in conjunction with the system HELP command.

Many of the discussions in this manual are, of necessity, closely interrelated. Frequent reference is made to items defined in Section 3. If an item is unclear, look through the manual for discussions involving it, or similar topics.

## 1.4 ACKNOWLEDGEMENTS

The System 88 operating system was designed and written by R. T. Martin. Processors and utilities, as well as able assistance, were provided by Robin Soto, Larry Deran, and Glenn McComb. Credit is also due to many people at Scientific Data Systems/Xerox, especially Ed Bryan, Richard Hustvedt, John Collins, and Mike Macfarlane. Many of the design philosophies embedded in the System 88 come from the BPM/BTM - UTS - CPV lineage of systems innovated at SDS/Xerox.

# Section 2

# THE SYSTEM 88 FILE SYSTEM

## 2.0

The System 88 file system provides a versatile structure and at the same time maintains the internal simplicity required for reliability and ease of use. The file system is composed of file names and extensions, which together with other file data form file directory entries (FDE's). The FDE's are gathered with directory management information to form the disk directory. Each disk has its own directory. This is initially provided by the system INITIAL command, which writes zeros on the disk to perform a simple surface test, and then writes out the empty disk directory. The system LIST command displays the disk directory to the user. Modification of the directory and its FDE's may be performed by the DELETE, UNDELETE, RENAME, SAVE, and PACK commands, and the Gfid overlay service (see Section 3.4.2). In addition, the experienced systems programmer may use the system utility program Szap to manipulate the disk and its contents (see Section 4.2).

## 2.1 DISKS

Each disk drive in the system is treated as a sequential collection of 256-byte sectors of data. Data is transferred to and from the disk through use of the Dio utility (see Section 3) by specifying the device number, disk address, and memory address. The first four sectors of each disk, sectors 0, 1, 2, and 3, contain the file directory for the disk. Note that the system overall deals with a generalized disk address; Dio breaks this down into the proper track and sector information required for dealing with the device.

## 2.2 FILES

A file is a contiguous group of physical sectors on a disk, accessible through, and defined by a file directory entry (FDE) in the disk directory for the device. A file must be totally contained on a single disk, and files may not overlap or contain the same sector or sectors. The internal format of the file is determined entirely by the program or programs that read and write that file.

## 2.3 FILE DIRECTORY ENTRIES (FDE's)

The File Directory Entry (FDE) in the directory defines a file on the particular disk. The FDE format is also used by the system Gfid utility (see Section 3) for looking up and entering file names into the directory. The FDE consists of the following information, in this order:

2.3.1) Flag byte (8 bits)
2.3.2) File name (variable length)

2.3.3) File extension (16 bits)
2.3.4) FDA - Starting disk address (16 bits)
2.3.5) DNS - File length in sectors (16 bits)
2.3.6) LA - File load address (16 bits)
2.3.7) SA - File start address (16 bits)

### 2.3.1 FDE Flag byte

The first byte of the FDE contains three one-bit flags, and the five-bit file name length:

```
+-+-+-+-+-+-+-+-+
:D:S:N:...L.....: -> Length of file name
+-+-+-+-+-+-+-+-+
 : : +------------> New file (20H)
 : +--------------> System file (40H)
 +----------------> Deleted file (80H)
```

The 80H bit, if set, indicates the file has been deleted. If this bit is set in a disk directory FDE, that FDE will not be examined in the file lookup procedure, and will not be displayed by the system LIST command. FDE's marked deleted are returned to normal status by the UNDELETE command. The space taken up by deleted files, both in the directory area and on the disk is reclaimed by the system PACK command.

The 40H bit, if set, denotes a "System" file. This bit is checked by system commands such as DELETE, RENAME, TYPE, and PRINT. A file marked by the System bit may not be deleted, renamed, or displayed by PRINT or TYPE.

The 20H bit denotes a "new" file. When a file is created or modified, its corresponding FDE is marked with the new bit to mark it eligible for saving by the system file maintenance processor. This bit is cleared when the file is backed up to another media by the file maintenance processor.

Note that any combination of the above three bits is allowed.

The last five bits of the flag byte give the length of the file name that follows the flag byte. This restricts the file name to 31 characters or less; a file name must be at least one character long. Note that the file name length DOES NOT include the two character extension.

### 2.3.2 FDE File Name

The file name follows the FDE flag byte and is the only variable length entry in the FDE. The number of bytes used by the file name is contained in the lower five bits of the FDE flag byte. File names usually consist of 7-bit ASCII characters, although programs may generate file names consisting of arbitrary 8-bit quantities that cannot be entered from the keyboard. When a file name is displayed on the screen, control characters (ASCII 00 to 1FH) display as greek characters.

### 2.3.3 FDE Extension

The FDE file extension is a 16-bit, or two character field that follows the file name. The extension is used by the system in identifying the contents of the file. The bytes appear in the extension in the same order in which they would be typed, rather than the "standard 8080" byte-reversed form. For example, the extension "GO" would appear in memory in the FDE as the two characters "G", followed by "O". The extension is not restricted to any set quantity; any 16-bit value may be used. A number of extensions are defined by, and recognized by the system. As the system expands, this list may also expand:

| Extension | Use |
|---|---|
| GO | Runnable machine code file |
| OV | System overlay (see Section 3.4) |
| BS | BASIC source program |
| DT | BASIC data file |
| TB | Tiny BASIC source program |
| TX | Text, assembly language source file |
| SY | Symbol table file |

### 2.3.4 FDE FDA - First Disk Address

The FDA is a 16-bit field in the FDE containing the starting disk address for the file.

### 2.3.5 FDA DNS - File Size in Sectors

The DNS is a 16-bit field in the FDE that contains the size of the file in sectors.

### 2.3.6 FDA LA - File Load Address

For runnable machine code files (extensions .GO or .OV), LA contains the 16-bit load address for the program. When the file is loaded into memory by the system Runr service (see Section 3), it is read into memory starting at the address contained in LA. For non-runnable files, the LA field in the FDE contains zero; since there is read-only-memory at location 00 in the System 88, it is not possible to load machine code files starting at this address for execution.

### 2.3.7 FDA SA - File Start Address

For runnable machine code files (see Section 2.3.6), SA gives the 16-bit starting execution address. If the FDA LA field is zero, indicating a non-runnable file, this field may be used for other purposes.

### 2.3.8 FDE Summary

The FDE defines the file, and its status in the system. It contains all the information required to locate, access, and delimit the file data on the disk. Since the file name in the FDE is of variable length, the FDE itself is also of variable length.

## 2.4 THE DISK DIRECTORY

The disk directory is the collection of FDE's and control data required for allocation and retrieval of files. The directory always appears in sectors 0, 1, 2, and 3 of each initialized disk. The system INIT command is used to set up the initial directory structure on the disk. As the directory is a fixed 1024 bytes in length, the number of FDE's it may contain is limited, and depends on the length of the file names in the individual FDE's. The disk directory consists of the following fixed fields, followed by a list of file directory entries (FDE's):

| Displacement | Section | Name | Description |
|---|---|---|---|
| 0 | 2.4.1 | Dck | 8 bit directory checksum |
| 1 | 2.4.2 | Dname | 8 byte disk name |
| 9 | 2.4.3 | Nf | Number of files on disk |
| 0BH | 2,4.4 | Nfa | Next free directory address |
| 0DH | 2.4.5 | Nda | Next free disk address |
| 0FH | ..... | ... | Start of FDE list |

Since the directory resides in memory in the SBUF1 area (see Section 3), the offsets given above are in hexadecimal from SBUF1.

### 2.4.1 Dck - Directory checksum

Byte 0 of the directory contains an 8-bit checksum computed by the Cksm service (see Section 3). This checksum is the 8-bit sum of the remaining 1023 bytes of the directory, and provides more security in handling the disk directory. When a directory is read into memory, the directory checksum is calculated by the Cksm routine, and compared to byte 00 of the directory. If the directory checksum does not match, the directory is considered destroyed, and a 03FFH error results. Whenever the directory is updated in memory, the directory checksum is also updated.

### 2.4.2 Dname - Disk Name

The disk name is an 8-character field following the disk directory checksum. The name is stored into the directory by the system INIT command, and displayed with directory listings produced by the system LIST command.

### 2.4.3 Nf - Number of files on the disk

Nf is a sixteen bit field containing the total number of files on the disk. This count includes deleted and undeleted files. It is used as a secondary sanity check of the directory structure, and is displayed by the system LIST command in directory listings.

### 2.4.4 Nfa - Next FDE Address

Nfa is a 16-bit pointer to the first free byte after the FDE list in the directory. Note that this pointer assumes the directory is residing in SBUF1. When the disk is initialized, Nfa is set to SBUF1+0FH. When a file is entered into the directory, it is entered at the address pointed to by Nfa, and then Nfa updated to point past the newly entered FDE. Nfa is also used to check for space remaining in the directory.

### 2.4.5 Nda - Next Disk Address

Nda contains the 16-bit disk address of the first free sector on the disk. Since files are allocated sequentially, it is also the number of sectors in use on the disk. When the disk is initialized, Nda is set to 4, pointing right after the directory on the disk. Thus, the LIST command given on an empty disk will show 4 sectors in use.

### 2.4.6 The Initialized Disk

Before a disk can be used by the system, it must be initialized. The initialization process fills the disk with sectors of zeroes, to perform a simple surface-check. Then the clean directory is written to sectors 0, 1, 2, and 3 of the disk. The directory has the name specified by the user in the INIT process. Nf, the number of files on the disk, is set to sixteen bits of 00. Nfa, the next FDE address, is set to SBUF1+0FH, for entering FDE's. Nda is set to 4, the first free sector on the disk. The remainder of the directory area is set to zero, the checksum computed by calling Cksm, stored in Dck, and the directory written to the disk.

### 2.4.7 Allocating File and Directory Space

Space for files and FDE's is allocated sequentially on the disk. Nda always points to the first sector past the used area of the disk; Nfa always points past the end of the last FDE in the directory. When a file is written to a disk, the data is written starting at the disk address contained in Nda. When the FDE is entered into the directory, it is stored at Nfa, and Nfa updated to point past the new entry. Nda is updated by the size of the file just entered, from the Dns field of the FDE. This means that space allocation in both the directory and the disk is sequential and contiguous in nature. Files may not overlap, and the ordering of FDE's in the directory is replicated in the ordering of the file data on the disk. When files are deleted, the corresponding FDE is marked deleted, but the space in the directory, and in the data area of the disk is not reclaimed until the PACK command is given.

### 2.4.8 Updating the Disk Directory

NOTE:

The Gfid system service (see Section 3.4) has been provided to update the disk directory. The user is STRONGLY encouraged to make use of this service and NOT write programs that update the directory unless absolutely necessary. An improperly updated directory may cause an immediate catastrophe, or the disaster may be postponed until the disk is PACK'ed, or new files are entered on it.

Updating the disk directory in memory (in the SBUF1 area) involves the system cells NFCK and NFDIR, described in Section 3, and the system routine Cksm, also described in Section 3. Cksm computes the checksum of the directory in the SBUF1 area. NFCK is a copy of that checksum. NFDIR is the drive number of the directory currently in SBUF1. If the user MUST update the directory rather than using the Gfid services described in Section 3.4 of this manual, the following procedure may be used.

1) Disable interrupts and compare the drive number desired with the contents of NFDIR. If the proper directory is in memory, go to step 3.

2) Force the directory into SBUF1 by calling the Look service described in Section 3 to look up a file that does not exist, such as the file with the single byte name 00H. If any error code other than 0300H is returned by Look, the directory is unreadable.

3) With the interrupts disabled, call Cksm to compute the directory checksum. This returned checksum must match the contents of NFCK and of byte 00 of the directory. If it does not match, load 03FFH into DE and jump to the system Error routine; the directory is destroyed.

4) Update the directory with the interrupts disabled, and do it carefully. Why not use Gfid??

5) Call Cksm to recompute the directory checksum. Store the checksum in NFCK and in byte 00 of the directory (SBUF1).

6) Call Dio to write 4 sectors to disk address 00, memory address SBUF1, to the device number in NFDIR. If any errors are returned by Dio, store 0FFH into NFDIR and NFCK to prevent the damaged directory from being used, and jump to Error to process the error.

# Section 3

## MEMORY LAYOUT OF THE SYSTEM 88

3.0

The basic 8080A central processor is capable of addressing 64K (K=1024) bytes of memory. This address space on the System 88 disk system is segmented into the following regions:

Locations 0000H-0BFFH: System ROM

The first 3K of the address space is occupied by read-only-memory (ROM). The first 1K block of this ROM is the 4.0 Monitor ROM. This is the same ROM used on the POLY 88 cassette-based systems. The second and third ROMs contain the portion of the disk system software that must be resident at system boot time, as well as certain utility functions. Beginning at location 400H is a series of "jump vectors" that provide access to system functions in the ROMs. These vectors are described in section 3.3.

Locations 0C00H-0DFFH: System Stack and Wormholes
Locations 0E00H-0FFFH: System Stack and Wormholes

Locations 0C00H through 0DFFH also appear as locations 0E00H through 0FFFH of CPU board RAM. This memory is used for the system input and output wormholes (described in Section 3.3), and the system stack area.

Locations 1000H-17FFH: Unused and reserved for expansion

Locations 1800H-1BFFH: Video board RAM

The video display memory in the System 88 occupies locations 1800H through 1BFFH. In the POLY 88 cassette-based systems the video display is addressed at F800H; it has been moved to 1800H in the disk system to allow full expansion of system memory.

Locations 1C00H-1FFFH: Unused and reserved for expansion

Locations 2000H-2FFFH: Disk system RAM

The 4K byte region from 2000H to 2FFFH is used entirely by the disk system. This area is described in Section 3.2 of this manual.

Locations 3000H-33FFH: Printer and File handler

The 1K byte region from 3000H to 33FFH is used for the resident system printer driver, and for resident file channel processing.
Locations 3400H-FFFFH: User RAM

The memory space from location 3400H to the end of memory is available for user programs. The system is delivered with a minimum of 16K bytes of memory starting at location 2000H. This leaves 11K bytes from 3400H to 5FFFH for your use.

## 3.1 SYSTEM ROM

The 3K of System 88 ROM are separated based on address and function in the following way:

### 3.1.1 Locations 0000H-03FFH: 4.0 Monitor ROM

This ROM is the same ROM used in the POLY 88 cassette-based systems. It provides several services: basic startup and interrupt vectoring; the front-panel mode used in assembly language debugging; real-time clock service; and management of the video display. A listing of the 4.0 Monitor ROM code is included as Appendix M of this manual.

### 3.1.2 Locations 0400H-0BFFH: Disk system ROM

These two ROMs provide services for the disk system (basic initialization, disk handlers, and utility and system management functions). Beginning at location 400H is a series of "jump vectors" that provide access to the various utilities in these ROMs. These vectors are described in Section 3.3 of this manual.

## 3.2 SYSTEM RAM

Locations 2000H to 33FFH are used by the disk system for disk buffers, system transients (overlays), and tables. This 5K system area is described in general below, followed by a detailed description of selected areas. The entire 5K region is used by the disk system. Beware of changing locations in this area that are not documented in this manual; catastrophic system failure may result. Some portions of this 5K area are not used at present, but they will be used in future versions of the disk system.

### 3.2.1 Locations 2000H-27FFH: Overlay area

This 2K byte area holds system transients (overlays). Overlays are described in Section 3.4 of this manual.

### 3.2.2 Locations 2800H-2BFFH: Directory area

The 1K byte disk directory resides in this area. Selected system functions also use it as a buffer area.

### 3.2.3 Locations 2C00H-33FFH: Tables and miscellaneous items

Various other material resides in the remaining area.

Many of the regions of system RAM are documented in the data base descriptions that follow. Caution!: work with care when changing both documented and undocumented areas. A system failure can occur as a result of damaged tables or pointers.

The descriptions give the internal system label or name (and the address or address range) for each item in the data base. We also give a brief (or not so brief) description of the area (usually including information on what parts of the system examine and/or modify the named area).

Each name is given as it appears in the system symbol table file, SYSTEM.SY. In using system routines or data areas in assembly language programs, it is a good idea to use REF statements to define symbol values from the symbol file SYSTEM.SY rather than using an EQU with the value given in this manual. If system symbols change from version to version, those programs using REFs only require re-assembly, where those using EQU's require a great deal of editing. The use of REF also forces commonality in naming of system routines and data areas.

#### 3.2.3.1 OVRLY

Item Name: OVRLY
Address: 2000H-27FFH
Description:

OVRLY is the system overlay area. The user uses the system routines Ovrto and Gover to bring transient processing routines into OVRLY. (Ovrto and Gover are described in Section 3.3 of this manual; overlays themselves are described in Section 3.4.)

#### 3.2.3.2 SBUF1

Item Name: SBUF1
Address: 2800H-2BFFH
Description:

SBUF is the 1K byte buffer that holds disk directories when they are read into memory from a disk. See Section 2.6 for information on the structure of the disk directory.

#### 3.2.3.3 CMND

Item Name: CMND
Address: 2D40H-2D7FH
Description:

CMND is the 64-character Exec command buffer. The system routine RLWE (see Section 3.3) places the user input line into CMND. This buffer holds the last command read by Exec. Various utilities (such as directory LIST, COPY, RENAME, etc.) examine CMND for arguments. When you use these utilities, you can place a command string in CMND (terminated by a carriage return), and then invoke an overlay service routine using Ovrto or Gover (described in Section 3.3).

#### 3.2.3.4 MEMTOP

Item Name: MEMTOP
Address: 2D80H-2D81H
Description:

When the system starts up Exec scans memory starting at 3000H to find the end of user RAM (the last usable memory location on the system). This 16-bit integer is stored in MEMTOP and is used by various processors as the upper limit of memory on the system. The user who wishes to write a custom Exec should note that Exec must do this memory scan and setup of MEMTOP at the time of system start up.

3.2.3.5 KBEX

Item Name: KBEX
Address: 2D86H-2D87H
Description:

KBEX contains an address of a routine that is called after a character has been returned from the system input wormhole (WHO). The address contained in KBEX differs depending upon whether the Exec commands FOLD, FULL or flip are used (for information on FOLD, FULL and flip, see Section 3.3). At the time of system start up, the disk system ROMs set up KBEX to point at a RET instruction in ROM (setting FULL mode).

3.2.3.6 CMDF

Item Name: CMDF
Address: 2D88H
Description:

CMDF is a single-byte flag. When the flag is non-zero, it indicates that command file input is in progress. If the flag is zero, command file mode is not in use.

3.2.3.7 DONT

Item Name: DONT
Address: 2D90H
Description:

DONT is a one-byte flag. It controls the processing of interrupt handling for Control-Y, Control-Z, and the like. DONT is non-zero during disk I/O, and is examined by the processing routines that handle user and system break characters. When DONT is zero and the system is in enabled mode, for example, a Control-Z will cause entry to he system front panel display. If DONT is non-zero (indicating that a critical process is in progress), Control-Z will not cause front panel entry. Also see PVEC, UVEC, SCHR, and UCHR below.

3.2.3.8 SBRK

Item name: SBRK
Address: 2D91H

Description:

SBRK is a single byte flag that is set non-zero when a control-Y is received from the keyboard. SBRK is set non-zero at the interrupt level. In processing the control-Y request from the user (see also PVEC, DONT, and EFLG1 in this section), EFLG1 is checked to see if the EIC (Exec In Control) bit is set. If EIC is set, the control-Y request is ignored. If EIC is not set in EFLG1, SBRK is set non-zero to note that a control-Y request has been made. Next, the DONT flag is checked, and if zero, Killi is called, and then control passed to the routine pointed to by PVEC. If DONT is non-zero, indicating disk I/O in progress, the request is ignored, but SBRK has been set non-zero. SBRK, then, is useful as a flag indicating that a control-Y request has been made, and possibly ignored because of I/O in progress. SBRK is used by BASIC in interrupting program execution. The system only sets SBRK non-zero; it is the responsibility of the program using SBRK to set it to zero again.

3.2.3.9 PVEC

Item Name: PVEC
Address: 2D93H-2D94H
Description:

PVEC holds the 16-bit address of the routine to be entered at the interrupt level when a Control-Y is typed. Exec sets PVEC to point to the routine IEXEC (described in Section 3.3). PVEC is also set by processors such as BASIC to point to a routine for handling the Control-Y interrupt. Return control through Ioret in the 4.0 Monitor ROM (or equivalent code) to restore the user environment when writing a program to handle the Control-Y interrupt. The Control-Y program interrupt is disabled when the EIC bit is set in EFLG1 (indicating that Exec is in control); KILLI will be called to flush input and to kill command file use, but the routine pointed to by PVEC will not be entered. KILLI will be called if DONT is non-zero, but the routine will not be invoked. Whenever Exec is invoked, PVEC is set to point to IEXEC in the root; therefore, processors that invoke Exec must re-establish the contents of PVEC when Exec returns.

3.2.3.10 UBRK

Item Name: UBRK
Address: 2D97H
Description:

UBRK is a single byte flag similar to SBRK, described above. UBRK is set non-zero at the interrupt level when the character set in UCHR is received from the keyboard. UBRK is set non-zero even if the dispatching through UVEC is inhibited by the DONT flag. As in the case of SBRK, it is the reponsibility of the programmer to set UBRK to zero after using it.

3.2.3.11 UVEC

Item Name: UVEC
Address: 2D95H-2D96H
Description:

UVEC holds a 16-bit address of a routine to be entered at the interrupt level. This routine will be entered when the system receives the character contained in UCHR (described below) from the keyboard. This vector is initialized by the disk system ROM at start up time to point to Ioret in the 4.0 Monitor ROM. See descriptions of PVEC and UCHR for additional information. The contents of the DONT flag determine whether or not the routine addressed by UVEC will be used. The EIC flag in EFLG1 does not affect the use of this routine.

3.2.3.12 SCHR

Item Name: SCHR
Address: 2D98H
Description:

When the system receives a character from the keyboard at the interrupt level, it compares that character against the contents of SCHR. If a match is found, the front panel mode is entered. SCHR is initialized to zero (00H) at the time of system start-up, and changed to 1AH (Control-Z) by the Exec ENABLE command. The Exec DISABLE command resets SCHR to zero again. SCHR is used as the flag that determines whether the system is in enabled or disabled mode. Access to various system commands and features (e.g., INIT, IMAGE, front panel display, etc.) is allowed or prohibited depending upon whether or not the system is operating in enabled mode. Although the customary character for front panel entry is Control-Z, any valid ASCII character except for Control-Y (19H) may be placed in SCHR. Note that SCHR is detected at the interrupt level only; it will not be detected if it appears in a command file.

3.2.3.13 UCHR

Item Name: UCHR
Address: 2D99H
Description:

UCHR contains a user-defined interrupt character. When this character is received from the keyboard at the interrupt level, the routine addressed by UVEC (described above) is called. UCHR can contain any 7-bit ASCII code other than Control-Y (19H) or the character code in SCHR (usually Control-Z; 1AH): checks for a Control-Y and for the contents of SCHR are made before the character in UCHR is examined. Note that UCHR is detected at the interrupt level only; it will not be detected if it appears in a command file.

### 3.2.3.14 ERROR

Item Name: ERROR
Address: 2D9AH-2D9BH
Description:

ERROR contains the error code and subcode of the last error reported by the system. It is updated by Exec and/or the system error-message writer, Emsg.

### 3.2.3.15 LERR

Item Name: LERR
Address: 2D9CH-2D9DH
Description:

LERR contains the error code and subcode of the previous error. Before ERROR is updated by Exec or Emsg, its contents are moved to LERR. This keeps a record of the last error issued by the system before the current error.

### 3.2.3.16 NFDIR

Item Name: NFDIR
Address: 2DA0H
Description:

NFDIR contains the number of the drive holding the disk whose directory is in the SBUF area. The values 00 or FFH indicate that the current contents of SBUF are not valid, and that the directory must be read in from the required drive. NFDIR is set by Look (see Section 3.3, and also Section 2.6), and cleared by various error recovery routines that force re-reading of the directory from the disk.

### 3.2.3.17 NFCK

Item Name: NFCK
Address: 2DA1H
Description:

NFCK is a copy of the SBUF directory checksum value. If the directory in SBUF is valid, the checksum stored in the first byte of SBUF will match that contained in NFCK. Thus NFCK provides additional error-checking for directory management in the system. See Section 2.6 on directory structure.

### 3.2.3.18 RAW

Item Name: RAW
Address: 2DB1H
Description:

RAW (Read-After-Write) is the verify flag. It is set non-zero by system initialization to force the verification of every disk write operation. The verify mode may be cleared by using the Exec command DONT VERIFY; this sets the RAW flag to zero. Several system processes turn the verify mode back on (PACKing a disk, INITializing a disk, IMAGEing a disk, etc.).

3.2.3.19 TRIES

Item Name: TRIES
Address: 2DB2H
Description:

TRIES contains the number of disk operation retries to perform before declaring a hard error. System initialization sets TRIES to 10; more than about 20 retries are not recommended, since after 10 or so further retries are futile.

3.2.3.20 ONCE

Item Name: ONCE
Address: 2DC5H
Description:

ONCE is the cold boot flag. It is set to zero by the disk system ROM cold-start routine, and provides the flag to Exec and others indicating that this is system start-up. Exec uses this flag in its boot process (see Section 8 on system start-up). For user-written Execs, if ONCE is zero, a scan for the top of memory must be done, and the last good address stored as MEMTOP. A search for the INITIAL file is then made.

3.2.3.21 EFLG1

Item Name: EFLG1
Address: 2DC9H
Description:

EFLG1 is one of two single-byte flags kept by Exec for its own use. The bits of interest to the systems programmers are:

80H EIC: When set, this bit indicates that Exec is in control. Control-Y is disabled while EIC is set.

40H EERR: Set by Err in the root (see Section 3.3) to tell Exec that it has an error code/sub-code in ERROR to process before reading a command.

When the Exec enters the INITIAL program at system boot time, (see Section 8), the program is entered with EIC set in EFLG1. This prevents the user from interrupting the program by typing control-Y while the INITIAL program is performing initialization. Note that if INITIAL is an assembly language program, it must clear the EIC bit in EFLG1, or do an overlay call (Gover or Ovrto, see Section 3.3) before the control-Y interrupt will be enabled. If the program does not clear EIC in one of these ways, it cannot be interrupted.

The EERR bit is cleared by the system error message handler, Emsg. If the user supplies his own Emsg, this bit must be cleared each time Emsg is entered. Failure to clear EERR will result in a system panic halt the next time Err is called to report an error (see Err, section 3.3).

3.2.3.22 BUGS

Item Name: BUGS
Address: 2DFC-2DFE

BUGS is a three byte region used to count the incidence of disk I/O error codes 102, 103, and 104. These three errors indicate data transmission problems between a disk and the controller. Each time one of these errors occurs on any drive in the system, the counter associated with it is incremented. If the contents of the counter is 0FFH, denoting 255 reported errors, it is not incremented. The location BUGS counts 102 errors; these are errors involving the sector preamble on the disk, and are caused by things such as uninitialized disks, media errors, and disk head alignment problems. BUGS+1 counts 103 errors; this error code indicates a checksum error when reading a sector. The usual cause of 103 errors is faulty media, although incorrect head alignment may cause 103 errors on a disk when read by a drive other than the one on which the data was written. BUGS+2 counts 104 errors. This error code idicates a write validation failure, caused by media or drive problems, or possibly faulty memory, or the contents of memory being changed during the verify operation. These counters are incremented each time one of these errors occurs; in the vast majority of cases, the automatic re-trying of the operation (see TRIES) will result in a successful data transfer. As such, these counters provide an indication of the "soft" error rate in disk data transmission, and may be used to identify media, disk drive, and compatibility problems. The Exec DISPLAY and SQUEAL commands may be used to display these cells.

## 3.3 SYSTEM SERVICE VECTORS

The System 88 provides an extensive set of services to the assembly language programmer through three vehicles: RAM vectors (the Wormholes), ROM vectors, and overlay services. This section gives details on the RAM and ROM vector services. Overlay services are described in Section 3.4. The RAM services described are the standard "wormhole" service routines also used in the POLY 88 cassette-based system for character output to the screen (WH1) and for keyboard input (WHO). The ROM vectors starting at location 400H provide mainly disk-oriented functions. In each of the following descriptions the routine name is given along with its calling address. Register contents at the time of the call, and on return from the call are discussed where appropriate. A text description is given for each routine along with error conditions resulting from, or reported by the routine. The routine names are given as they appear in the system symbol table file SYSTEM.SY for use by the Assembler. The calling address is given for reference purposes; it is a good practice when using a system symbol or routine to use the assembler REF statement, which defines the symbol from the SYSTEM.SY file, rather than using an EQU with the given reference address. If the symbol value were to change in future systems, those programs coded using REFs for system items would only need to be re-assembled, where those programs that used EQUs would have to be edited as well.

### 3.3.1 WHO

Routine Name: WHO
Vector Address: 0C20H
Purpose: Return input character

Registers on entry:

| | |
|---|---|
| HL: | unused |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | unchanged |
| DE: | " |
| BC: | " |
| A: | ASCII character |
| PSW: | Junk |

Description:

WHO is called to get a character input from the keyboard. It is an extension of the POLY 88 cassette-based system's standard input wormhole. It is connected to resident code in the disk system ROMs, rather than the standard 4.0 Monitor keyboard code so that type-ahead and command files may be supported. Type-ahead allows up to 64 characters to be entered into an internal system buffer and stored for later retrieval by calls to WHO. When the disk system is operating in command file mode, calls to WHO will return characters from the command file rather than from the keyboard buffer until command file use is ended. The systems programmer who redirects this wormhole should take care to maintain these functions.

Error conditions:

No error conditions are detected or reported by the processing done by WHO. Note that ASCII 00 characters (eight bits of zero) are suppressed, and that the processing of Control-Z, Control-Y, and user interrupt characters (see UVEC, PVEC, SCHR, and UCHR in Section 3.2) depend on the disk system keyboard support code. The driver and interrupt handler for the keyboard are different from those used in the POLY 88 cassette-based systems. The standard 4.0 Monitor keyboard handling routine is not used, and will not work if reconnected; the video board (and therefore the keyboard port address) is not where the 4.0 Monitor ROM code expects it to be.

### 3.3.2 WH1

Routine Name: WH1
Vector Address: 0C24H
Purpose: Output character to video screen

Registers on entry:

| | |
|---|---|
| HL: | unused |
| DE: | " |
| BC: | " |
| A: | ASCII character |
| PSW: | unused |

Registers on exit:

| | |
|---|---|
| HL: | unchanged |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Description:

WH1 is called to place the character in A on the video screen. This wormhole is initially connected to the screen driver in the 4.0 Monitor ROM. The driver code processes the following special ASCII control characters:

| | |
|---|---|
| FF (0CH) | Form Feed. Clear the screen and place cursor in top left corner. |
| VT (0BH) | Vertical tab. Place cursor in top left of screen. |
| CR (0DH) | Carriage return. Move cursor to start of next line, may scroll screen image. |

| | |
|---|---|
| TAB (09H) | Horizontal tab. Tab stops are simulated every 8 positions on the screen. |
| DEL (7FH) | Delete. The cursor is backed up one place. |

All other ASCII control characters (00H-1FH) are ignored and cause no cursor movement or screen change. Characters sent to the screen will wrap around to the next line after 64 characters are sent, possibly scrolling the screen. ASCII control characters (00-1FH) will be displayed as special symbols if the 80H bit is set (see Appendix C).

### 3.3.3 WH7

Routine Name: WH7
Vector Address: 0C3CH
Purpose: Send character in A to printer

Registers on entry:

| | |
|---|---|
| HL: | unused |
| DE: | " |
| BC: | " |
| A: | Character |
| PSW: | unused |

Registers on exit:

| | |
|---|---|
| HL: | unchanged |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | junk |

Description:

WH7 is connected to the system printer driver. Characters passed to WH7 in the accumulator (A) are placed in a ring buffer and sent to the serial printer. The printer driver recognizes the special ASCII character codes CR (carriage return), LF (line feed), TAB (horizontal tab) and FF (form feed). More on the printer driver later

### 3.3.4 Ioret

Routine Name: Ioret
Vector Address: 64H
Purpose: Return from interrupt level

Registers on entry:

| | |
|---|---|
| HL: | unused |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | From stacked interrupt environment |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Description:

Any routine that is entered at the interrupt level can jump to Ioret (Note: jump to; NOT call). When an interrupt occurs, the PC is automatically pushed onto the stack by the processor. The interrupt handling code in the 4.0 Monitor ROM then pushes the contents of the remaining registers (called the interrupt environment) onto the stack before jumping to the specified interrupt handler. After the handler has performed the necessary tasks, it jumps to IORET to restore the interrupt environment and to continue the interrupted process. For further information on Ioret and the format of the interrupt environment saved on the stack, see the 4.0 Monitor listing in Appendix M.

3.3.5 Begin

Routine Name: Begin
Vector Address: 400H
Purpose: Cold-start the disk system

Registers on entry:

| | |
|---|---|
| HL: | unused |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | never returns! |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Description:

On system start-up, the 4.0 Monitor ROM sets up initial areas needed by itself. The Monitor then calls Begin which completes the initialization of the system data area and cold-starts the system by booting the Exec. For a complete explanation of the boot process, see Section 8 of this manual.

Error conditions:

On system boot, any error conditions arising from disk errors will cause an error message of the following form to be displayed on the screen:

(Error 106)

The number displayed is the error code being reported, and represents a fatal error in the boot process. The most common error codes are:

| | |
|---|---|
| 106 | No disk in drive #1, or the door is open. |
| 300 | The disk in drive #1 is not a system disk (i.e., no Exec on it). |
| 306 | Same as 106. |

### 3.3.6 Warm

Routine Name: Warm
Vector Address: 403H
Purpose: Warm-start the system

Registers on entry:

| | |
|---|---|
| HL: | unused |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | never returns |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Description:

Warm is called to warm-start the system. The stack pointer is reset and the Exec is invoked from the disk. The actual code that appears at Warm is:

```
Warm    DI
        LXI     SP,STACK        ; reset the stack
        EI                      ; allow intrusions
        CALL    Gover           ; invoke the Exec
        DB      'Exec'
        JMP     Warm            ; loop if it returns
```

The contents of user memory are unchanged, as are the system tables and flags. Warm may be used to ensure that the stack pointer is valid after an error occurs, or it may be used during error bailout.

Error conditions:

In general, the same error conditions that occur when cold starting the system may appear during a warm-start. It is possible, however, to see spurious error reporting because internal system tables have been scrambled.

### 3.3.7 Dio

```
Routine Name:    Dio
Vector Address:  406H
Purpose:         Do disk I/O

Registers on entry:
        HL:      Disk address, 0 <= da <= 349 (decimal)
        DE:      Memory address
        BC:      B: Command: 0=write, 1=read, 2=verify
                 C: Unit number: 1, 2, or 3
        A:       Number of sectors, 1 <= # <= 255 (decimal)
        PSW:     Unused

Registers on exit:
        HL:      Junk
        DE:      If carry bit set in PSW, error code
                 If carry bit not set, Junk
        BC:      Junk
        A:       Junk
        PSW:     Carry bit set if error, clear otherwise
                 all other flags unknown
```

Description:

Dio is the central system service routine for transferring data to and from disk, and verifying those transfers. Call Dio with the register contents outlined above. The specified transfer will then be done by Dio. Think of the disk as a set of sectors; Dio worries about tracks and track position. Each sector contains 256 (decimal) bytes. Note that if the system flag RAW (verify flag) is non-zero, each write operation will automatically be verified.

Error conditions:

The error codes reported by Dio all have an error code of 1, and are listed below:

| Code | Description |
|---|---|
| 0101 | Bad parameters passed to Dio: A=0; invalid command in B; invalid drive # in C; disk address in HL, or disk address in HL plus number of sectors to be transferred, is greater than 349 (the number of sectors on the disk). |
| 0102 | The sector preamble is bad. This indicates a non-initialized disk, or a serious error with the hardware or with the contents of the disk. |
| 0103 | An incorrect sector checksum on data read from the disk. |
| 0104 | A verify operation finds that the contents of |

memory and the contents of the specified sector (or sectors) do not match.

0105 An attempt was made to write on a write-protected disk. No data will be transferred to the disk.

0106 This error occurs when the system does not receive sector interrupts from the selected drive. Several conditions may cause this: no drive on the system with the specified drive number (e.g., you tried to access drive #3 on a one-drive system); there is no disk in the drive specified; the door on the drive specified is open; the disk is inserted wrong.

All errors are reported in the DE register pair, with the Carry bit set in PSW. Error codes 102, 103, and 104 are tallied in locations BUGS through BUGS+2 to provide an indication of the soft error rate.

### 3.3.8 Dhalt

Routine Name: Dhalt
Vector Address: 409H
Purpose: Halt disk drives

Registers on entry:

| | |
|---|---|
| HL: | Unused |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | Unchanged |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Description:

Dhalt is called to halt the disk drive motors. It should only be called for reasons such as system panic stops; any disk transfers or operations that may be in progress will be aborted in an unclean and non-recoverable manner. Calling Dhalt while a disk write is in progress will result in the loss of data on one or more sectors of the disk. Dhalt is used by the internal system memory test facility to shut down disk I/O before beginning the memory test.

### 3.3.9 Msg

Routine Name: Msg
Vector Address: 40CH

Purpose: Display a message on the video screen

Registers on entry:

| | |
|---|---|
| HL: | Points to the text to be displayed; the text is terminated by a 00 byte. |
| DE: | Unused |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | Points at the 00 byte in the message |
| DE: | Unchanged |
| BC: | Unchanged |
| A: | 00 |
| PSW: | Zero flag set, from ORA A / RZ pair |

Description:

Msg displays text on the video screen by using the system output wormhole, WH1. The text is pointed to by HL, and on return from Msg, HL points at the 00 delimiter byte. Interrupts are enabled on return from Msg and remain enabled during the execution of Msg. Since Msg calls WH1 for output, the text will be displayed by whatever routines are connected to that wormhole.

Error conditions:

No error conditions are reported by this routine. If the 00 delimiter byte is left off the text, the contents of memory up to the first 00 byte will be displayed on the screen.

### 3.3.10 Err

Routine Name: Err
Vector Address: 40FH
Purpose: Abort process, display error, and warm-start system

Registers on entry:

| | |
|---|---|
| HL: | Unused |
| DE: | Error code and subcode |
| BC: | Unused |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | Never returns |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Description:

Err is called with an error code/subcode pair in DE. The type-ahead buffer is flushed, and if the command file mode is active it is aborted. (This is done by calling routine KILLI described below). If the flag bit EERR in EFLG1 is set, we have an error condition arising from an attempt to report a previously reported error: we are in serious trouble, because that flag should have been cleared. The code/subcode currently in ERROR is displayed on the screen in the form:

(Error xxyy)

where xx is the code in D, and yy is the subcode in E. After displaying the error code the system HALTS. The Emsg overlay is responsible for clearing the EERR bit in EFLG1. User-written Emsg handlers must remember to clear this bit, or a system panic halt may result.

If EERR in EFLG1 is not set, Err sets it now so that when Exec begins execution it knows that it has an error to process. Err then store the code/subcode in the system cell ERROR. We then jump to WARM to warm-start the system. The Exec, after doing its cleanup will see the EERR flag set in EFLG1, and invoke the system error message handler, Emsg, to process the error code. If a message is present in the error writer, the message will be displayed; if no message is present, the text

?No message found for error xxyy

will be displayed, where xx and yy are the error code and the error subcode contained in DE.

Error conditions:

Possible error conditions are the same as for the service routine Warm.

### 3.3.11 Ovrto

Routine Name: Ovrto
Vector Address: 412H
Purpose: Invoke an overlay

Registers on entry:

| | |
|---|---|
| HL: | Defined by the overlay |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | Defined by the overlay |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Description:

Ovrto and Gover provide the mechanisms for invoking system functions by name, and for extending the available system services in a powerful manner. These facilities represent the cornerstones on which the System 88 disk operating system is built. Use Ovrto or Gover to invoke a function that is in an overlay. (See below for the differences between Ovrto and Gover.) The overlay desired may or may not be in memory before you invoke it. Both the entering and exiting register contents are defined by the overlay invoked. Common system conventions for overlays that process more than one function suggest that the function code be passed in A. The invocation of an overlay takes the form of the example below (assuming that the registers have already been set up to hold the proper contents):

```
CALL    Ovrto
DB      'Dfnl'
;
; Return to here from the overlay
;
```

Overlay names are defined to be four bytes long, and these four bytes of the overlay name must follow the call to Gover or Ovrto. If the overlay desired is not currently in memory, it is transferred into memory from the System Disk. We enter at the overlay start address. (See Section 3.4 for a description of overlay formats and conventions). We will return from the function to the byte following the text of the overlay name in the Ovrto or Gover call.

In looking for an overlay, the system calls Runr to find a file on the system disk with the name specified after the call to Ovrto or Gover, with the extension OV. If the file is not found, or is not runnable (see description of Runr in this section), Errwt is called to process the error. If the file is found, Runr reads it into memory at the load address specified in the file; we do NOT check to see that this is 2000H! The overlay, when loaded, is entered at 2004H. The overlay name in locations 2000H-2003H is used by the system to "remember" what overlay is in memory for the Ovrto service.

Differences between Ovrto and Gover:

Both Ovrto and Gover invoke a function in an overlay, which may not be in memory at the time, and both return control to the program just after the overlay name following the call to Ovrto or Gover. The only difference between Ovrto and Gover is that Ovrto "remembers" the overlay currently in the overlay area, and restores that overlay before returning to the caller, and Gover does not. Both Ovrto and Gover are "super-subroutine" calls; they can call subroutines that do not have to be in memory at the time. Ovrto can be used from WITHIN one overlay to call a function in another overlay, since the original overlay is restored after the called overlay completes its processing. Gover does not "remember" or restore the overlay currently in the overlay area, and so it can only be used from programs outside the overlay area.

Error conditions:

If an error occurs in invoking an overlay, the appropriate error code/subcode is passed to Err, which reports the error warm-starts the system. Any errors that are reported within the overlay are handled by that overlay.

### 3.3.12 Gover

Routine Name: Gover
Vector Address: 415H
Purpose: Invoke overlay

Registers on entry:

| | |
|---|---|
| HL: | Defined by overlay |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | Defined by overlay |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Description:

See Ovrto for a description of this system service and how it differs from Ovrto. Also see section 3.4 on overlays.

### 3.3.13 Killi

Routine Name: Killi
Vector Address: 41BH
Purpose: Kill type-ahead and command file mode

Registers on entry:

| | |
|---|---|
| HL: | Unused |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | Unchanged |
| DE: | " |
| BC: | " |
| A: | Junk |
| PSW: | Junk |

Description:

Killi is called by the error reporting service routine and by any service routine that wants to flush the keyboard type-ahead buffer and end command file mode. Killi first checks to see if the command file mode is set, and if so, aborts that mode. The following message is displayed on the screen:

(Cmdf abort)

to alert the user to the fact that the use of the command file has been ended. Flush is then called to flush the keyboard type-ahead buffer. NOTE THAT Killi RETURNS TO THE CALLER WITH THE INTERRUPTS DISABLED.

Error conditions:

No errors are reported by Killi. If command file mode was in progress, that mode is aborted and the message displayed on the screen: (Cmdf abort). Note that this error message is NOT contained in Emsg, the system error message handler; it is contained in the disk system ROMs, so it cannot be changed.

### 3.3.14 Flush

Routine Name: Flush
Vector Address: 41EH
Purpose: Flush keyboard type-ahead

Registers on entry:
- HL: Unused
- DE: "
- BC: "
- A: "
- PSW: "

Registers on exit:
- HL: Unchanged
- DE: "
- BC: "
- A: Junk
- PSW: Junk

Description:

Call Flush to reset the keyboard ring buffer pointers (which essentially clears the keyboard type-ahead buffer). Flush returns with INTERRUPTS DISABLED, and A and PSW modified.

### 3.3.15 Look

Routine Name: Look
Vector Address: 421H
Purpose: Look up file

Registers on entry:
- HL: Address of lookup block. HL points to a byte containing the length of the file name (from

```
        1 to 31 bytes) followed by the text for the
        name, and the two byte extension (if present).
DE:     Unused
BC:       "
A:      Drive number of disk to search for the file
        (1,2 or 3).  If the 80H bit is set, then the
        extension is not checked and a match will occur
        on equal names.
PSW:    Unused
```

Registers on exit:

```
HL:     Unchanged
DE:     if carry is set in PSW, DE contains error code
        resulting from Look; If carry not set, register
        contains FDE directory address.
BC:     Junk
A:      Junk
PSW:    Carry is set on error; clear otherwise.
```

Description:

Look looks up files on a disk. It is called with HL pointing to a "lookup block," which consists of the length of the file name (1 <= length <= 31), the text of the name, and the extension (if present). A contains the number of the drive to search (1, 2, or 3), and the 80H bit of A is used to indicate whether or not the extension has to match. If the file is found in the directory, Look returns the address of the FDE (File Directory Entry) in DE. If for some reason the file is not found, or an error occurs when reading the directory, the error code is passed back to the caller with the carry bit in the PSW set. An example of a lookup block and coding to look up file GRONK.BC on disk 2 would be:

```
;
Txt     DB      5,'GRONKBC'
;
        LXI     H,Txt
        MVI     A,2
        CALL    Look
        JC      Oops
;
```

Description of the Look process:

Look first checks to see if the directory to be searched is resident in the SBUF1 area; system cell NFDIR contains the drive number of the disk whose directory is in the directory area of memory. If the proper directory is not in memory, Dio is called to read the directory (sectors 0-3) from the specified disk into SBUF1; errors reported from Dio are passed back to the caller with the code in D changed to 03 (error 0103 becomes 0303, etc.). When the proper directory is in SBUF1, its checksum is computed, stored in cell NFCK, and compared to the first byte of the directory. If this checksum does not match that first byte, we consider the directory destroyed, and return to the user reporting an 03FF error. If the directory checksum is good, we mark NFDIR with the directory number and scan the directory for the specified file, skipping those files marked deleted. If we come to the end of the directory before finding a match, we report an 0300 error. If the 80H bit was passed in A, noting not to check the extension, Look will return a match on the first file in the directory with the specified name.

Error conditions:

See the description of Look above for errors generated and reported.

### 3.3.16 Runr

Routine Name: Runr
Vector Address: 424H
Purpose: Run a file

Registers on entry:

| | |
|---|---|
| HL: | Lookup block (see Look for description) |
| DE: | Unused |
| BC: | " |
| A: | Drive number to search; 80H bit set if extension does not have to match. |
| PSW: | Unused |

Registers on exit:

| | |
|---|---|
| HL: | If carry is clear, register holds start address from FDE. |
| DE: | If carry is set, register holds error code; else it holds junk. |
| BC: | Junk |
| A: | Junk |
| PSW: | Carry is set if error; clear otherwise. |

Description:

Runr is called pointing to a "lookup block" (see Look for description) and a drive number. Runr attempts to find the program identified and load it into memory. If it is sucessful, it returns the start address of the file in HL. If unsucessful, Runr returns an error code/subcode in DE.

Runr first calls Look with register contents the same as on entry to Runr. Runr returns if Look returns with the carry set, thus passing any Look errors to the caller of Runr. If the file asked for exists, the FDE (File Directory Entry) is examined for a load address (LA) and a start address (SA). If the start address is 0000 we return reporting a 201H error, since the file is not runable. If the start address is non-zero, we call Dio to read the file into the memory address given as LA in the FDE. Any Dio errors are passed to the caller. If no errors occur during the read, Runr returns with the start address from the FDE in HL, and the carry flag in the PSW is clear. Note that although calling Runr does not automatically execute the desired program, it is loaded into memory, possibly over-writing the routine calling Runr. If no extension was given on the file passed in A to Runr, Look will match on the first file on the specified disk with the given name- which may not be a "runnable" file. For example, if disk 2 has files "Flange.TX" and "Flange.GO" appearing in that order, telling Runr to run file "Flange" without specifying an extension will return a 201H error, as Look will find file Flange.TX, rather than Flange.GO.

Error conditions:

See the above description of Runr.

### 3.3.17 Rlwe

```
Routine Name:    Rlwe
Vector Address:  427H
Purpose:         Read line with editing

Registers on entry:
        HL:      Address of user buffer to read into.
        DE:      Prompt string terminated by 00 byte.
        BC:      C: Maximum # of characters to read.
                 B: If 0, echo termination character.
                    If 1, do not echo termination character.
        A:       Unused
        PSW:       "

Registers on exit:
        HL:      Points to last character in buffer.
        DE:      Junk
        BC:      B: Length of line read
                 C: Junk
        A:       Termination character
        PSW:     Junk
```

Description:

Rlwe is used to read an input line. It provides an input prompt by using Msg (see Msg in this section) to output to the screen the string pointed to by DE. Rlwe then reads in characters (allowing editing of those characters) into the user buffer pointed to be HL. C contains the maximum buffer size, and B contains a flag that controls echoing of the termination character. Characters are read into the user buffer until one of the following conditions is met: 1) the buffer is full; or, 2) the user enters a carriage return (CR). Rlwe returns with HL pointing at the termination character in the buffer, the termination character in A, and the line length in B.

Editing functions supported by Rlwe:

Single character deletion in Rlwe is accomplished by use of the DEL key (DELETE). Delete words by using Control-W. A word is defined as a contiguous sequence of the characters a-z, A-Z, 0-9. Delete an entire line by using Control-X.

### 3.3.18 Fold

Routine Name: Fold
Vector Address: 42AH
Purpose: Fold lower to upper case

Registers on entry:

| | |
|---|---|
| HL: | Unused |
| DE: | " |
| BC: | " |
| A: | ASCII character (7 bits) |
| PSW: | Unused |

Registers on exit:

| | |
|---|---|
| HL: | Unchanged |
| DE: | " |
| BC: | " |
| A: | ASCII character: lower case a-z folded to A-Z |
| PSW: | Junk |

Description:

Fold is a WHO post-processor routine. (WHO is the system character input routine.) Fold "folds" (i.e., converts) lower case letters to upper case. It is usually installed in the character input path by Exec as a response to the user FOLD command. Exec stores the address of Fold into the system cell KBEX. When the system WHO routine exits, it returns through the address in KBEX. Fold can also be called as a subroutine to fold lower case to upper case.

### 3.3.19 Flip

Routine Name: Flip
Vector Address: 42DH
Purpose: Flip upper and lower case alphabetics

Registers on entry:

| | |
|---|---|
| HL: | Unused |
| DE: | " |
| BC: | " |
| A: | ASCII character |
| PSW: | Unused |

Registers on exit:

| | |
|---|---|
| HL: | Unchanged |
| DE: | " |
| BC: | " |
| A: | ASCII character: a-z and A-Z interchanged |
| PSW: | Junk |

Description:

Flip is another post processor for the WHO character input path. It "flips" (i.e., switches) upper case A-Z and lower case a-z (e.g., "a" becomes "A" and "E" becomed "e"). It is usually installed by the Exec as a response to the command "flip". Exec stores the Flip vector address in system cell KBEX (see also Fold).

### 3.3.20 Ckdr

Routine Name: Ckdr
Vector Address: 433H
Purpose: Compute directory checksum

Registers on entry:

| | |
|---|---|
| HL: | Unused |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | Unchanged |
| DE: | " |
| BC: | " |
| A: | 8-bit checksum |
| PSW: | Junk |

Description:

Ckdr is called to compute the 8-bit checksum of the directory area of memory (SBUF1+1 to SBUF1+3FFH). The checksum is returned in A. While Ckdr does not itself disable interrupts, it is recommended that you disable them before calling Ckdr to prevent any interruptions of this process.

### 3.3.21 Iexec

Routine Name: Iexec
Vector Address: 436H
Purpose: Interrupt level Exec entry

Registers on entry:

| | |
|---|---|
| HL: | Unused |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Registers on exit:

| | |
|---|---|
| HL: | Returns through Ioret |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Description:

Iexec provides entry to the Exec in response to a user-typed

Control-Y. When a Control-Y is detected at the keyboard interrupt level, the contents of all registers have been pushed onto the stack, and we jump to the following code to invoke Exec.

```
;
Iexec   CALL    Ovrto   ; invoke Exec, save current
        DB      'Exec'
        JMP     Ioret   ; return to whatever
;
```

The Exec may thus be invoked in the middle of a process, used, and then the interrupted process resumed. All register contents and the previous overlay are restored. Iexec is usually connected to the Control-Y interrupt vector PVEC; every time Iexec is invoked by Exec, the address of Iexec is stored into PVEC.

Error conditions:

See Ovrto for possible error conditions.

## 3.4 OVERLAYS

The internal structure and flexibility of the System 88 disk system is based on the overlay mechanism. This section describes the internal structure of overlays on the System 88, and the facilities provided to the assembly language programmer by the various system overlays. It is assumed that you have perused the descriptions of Gover and Ovrto (the overlay linkage facilities) in section 3.3 of this manual.

The overlay area in System 88 memory is from 2000H to 27FFH; overlays are therefore assembled for this area of memory, and do not exceed 2K bytes in size. Overlay names are four characters long, and may not contain blanks, tabs, or other control characters. The first four bytes of the overlay (2000H-2003H) must contain the overlay name, which must match the file name. The file name and the internal name must match so that Ovrto can "remember" the current overlay. When an overlay is brought in by Ovrto or Gover, it is entered at location 2004H. The contents of the registers are unchanged from the call to Gover or Ovrto.

When writing overlays, the 2K byte space reserved for overlays may be used in any manner you choose. It is assumed that overlays are "pure" code; that is, code that does not modify itself. Portions of the overlay area may be used by the overlay itself for data or buffers. Remember that any such data will be lost if another overlay is invoked. Arguments may not be passed to other overlays through the overlay area itself.

When you enter the overlay area at location 2004H, the interrupts are disabled and the EIC (Exec in Control) bit in EFLG1 is not set. If the overlay wishes to process Control-Y interrupts from the user, PVEC should be set accordingly. Note that if the user types a Control-Y, the EIC bit is not set, and your program did not set PVEC, the overlay area will be overwritten by the Exec overlay (brought in as a response to the Control-Y). If the user then types the Exec CONTINUE command, the previous overlay will be restored from disk and re-entered at the point where it was interrupted. The overlay will be re-entered with the appropriate register contents, but without any data that it might have stored within the overlay area.

The overlays provided as part of the standard disk system are "protected" from abuse (deletion, renaming, etc.) by having the "system" bit set in the FDE for each (see section 2.4). You may set the "system" bit in the FDE (or clear it) using the SuperZap (Szap) utility described in Section 4 of this manual. No other facility for setting or clearing the "system" bit in FDEs is, or will be, provided. You must use either Szap, changing the bit by hand, or you may use the Gfid/replace function described later in this section. This degree of difficulty encourages careful action and discourages thoughtless experimenting.

As an example of overlay format, a listing of the system error message writer overlay, Emsg, is given in Section 7.

### 3.4.1 Overlay services: Emsg

Emsg is the main error message writer for the system. It is invoked with an error code/subcode in registers D and E, and prints the error text associated with that code/subcode. If no such error code/subcode pair is found in the text embedded in the overlay, a message of the form:

```
?No message for error XXYY
```

is displayed, where XX and YY represent the error code and subcode passed in D and E. Emsg also updates ERROR and LERR (see Section 3.2). Emsg returns to its caller after printing the appropriate text. Note that the standard texts in Emsg do not end with carriage returns: you must print a carriage return if one is desired. The text output by Emsg is sent through the standard system output wormhole, WH1 (see Section 3.3). Emsg also clears the error flag set by Err, EERR, in EFLG1.

Emsg consists of a small section of code that searches for the proper error message, and a large body of text that lists the error messages. Locations 2007H-2008H contain a pointer to the beginning of the error message entries. This pointer is used by the error message editor, Emedit (described in Section 4). Each error message entry is composed of an error code in binary, a subcode in binary, and an error text (followed by a zero byte). For example, the following is the format for the error message for code 0105:

```
DB 1,5,'DIO says: Write protected!',0
```

The end of the series of error message entries is denoted by an FFH byte. From this it is apparent that error codes may not start with FFH.

### 3.4.2 Overlay services: Gfid

The Gfid (Get-file-identifier) overlay provides the assembly language user with the following functions:

1) Parsing user input into a file name specification, and looking up the appropriate file.
2) Entering a new FDE (see Section 2.3) in a directory.
3) Replacing an existing FDE in a directory.

The function performed depends upon the parameter byte passed to Gfid in the A register. Text descriptions of these functions along with a discussion of register contents on entry to and exit from the function code may be found below.

If the bottom bit (bit-1) of A is a zero, the Get-file-identifier function is selected; if the bottom bit of A is a one, the enter/replace FDE function is chosen. When the Enter/replace function is chosen, the function performs a replace FDE operation rather than an enter FDE operation if the 80H bit of A contains a one. All functions return any error codes in DE with the carry bit in the PSW set.

3.4.2.1 Get-file-identifier Function

Function Name: Get-file-identifier;
Overlay Name: Gfid

Registers on entry:

- HL: If the 80H bit is set in A, HL points to a prompt string to be used by Rlwe in prompting the user (see RLWE in Section 3.3).
  If the 80H bit is not set in A, HL points to the text buffer to be examined in parsing the file descriptor. NOTE: this address MUST NOT be in the overlay area (2000H-27FFH).
- DE: Points to the 44-byte area used to build the file descriptor (described below). Note that this area is first set to zero by Gfid.
- BC: If the 20H bit in A is set, BC contains the default extension to use if the user does not specify an extension.
- A: Flag bits, as follows:
  - 80H: If set, read from user (via Rlwe) into an internal buffer, using the string pointed to by HL as a prompt string.
    If clear, use HL as a pointer to the text to parse into the file identifier.
  - 40H: If set, Look up the resulting file. If the file exists, the FDE will be copied from the directory into the buffer pointed to by DE (see below description). If an 0300 error (file does not exist) is returned by Look, return NFA from the directory (see Section 2); copy it into the FDA slot of the FDE.
  - 20H: If set, use the contents of BC as the default extension if the user does not specify one.
  - 1FH: These bits MUST be zero for the get file identifier function.

Registers on exit:

- HL: Points to the ending delimiter symbol in the text buffer.
- DE: If carry bit set in PSW, DE contains an error code/subcode; if carry is not set, and Look was requested (i.e., 40H bit set in A on entry), then DE points to the FDE address in the directory. IMPORTANT NOTE: because of the overlay mechanism, the directory in the SBUF1 area on return from Gfid MAY NOT BE FROM THE DISK

CONTAINING THE DESIRED FILE (see discussion).

| | |
|---|---|
| BC: | Junk |
| A: | Junk |
| PSW: | Carry bit set if errors detected; clear if not. |

Description:

The Get-file-identifier function of Gfid relieves the assembly language programmer of the burden of parsing a generalized file identifier. This function is used extensively within the disk system itself by commands such as SAVE, DELETE, RENAME, COPY, PRINT, and TYPE.

You can either pass Gfid a text buffer to scan (useful in cases where more than one file identifier may appear on a single line, as in the case of DELETE or RENAME), or request that Gfid read a specification from the user of the system. If you direct Gfid to read a file specification directly from a user, you must supply Gfid with the address of a prompt string (as in the case of the system SAVE code). In either case, Gfid scans the appropriate text and attempts to parse it into a valid file identifier. Gfid will then Look up the file identified if directed to do so by a set 40H bit in the PSW.

Notes on Using the Get-file-identifier Service:

As mentioned in the register contents descriptions above, you MUST NOT pass addresses in the HL and DE register pairs that are within the overlay area (2000H-27FFH). Doing so will cause anomalous (and deserved) behavior. You should be particularly careful when using the FDE address passed back from Look: Before using the FDE address in DE you should verify that the proper directory is in memory (either by checking the contents of NFDIR--see Section 3.2--or by calling Look to force in the directory). For example, suppose that the specified file does not reside on the disk in drive #1, and that Gfid was invoked by Ovrto (see Section 3.3). When Gfid returns, the directory in the SBUF area will be from the disk in drive #1, and not from the disk containing the specified file.

File Descriptor Block Built by Gfid:

The file descriptor block built by Gfid consists of one byte (containing the specified drive number and an extension presence tag) followed by a normal FDE. You pass to Gfid the address where this block is to be built. The block is assumed to be 44 bytes long and to contain maximum length file names. The block is initially zeroed by Gfid. If no extension was given by the user in the input to Gfid, the 80H bit of this initial byte will be set (pointed to by DE on entry to Gfid). If you requested that Gfid Look up the file (and the file exists), the buffer specified by DE will contain the drive number of the file plus an extension presence flag followed by the FDE copied from the directory. If the file did not exist, Look will report a 0300 error. Then the FDA slot in the FDE will contain the first free disk address on the specified device. Thus, if the user wants to create a new output file, the returned 0300 error code indicates that the file specified does not currently exist. In that case, the block contains the first disk address to write to. The file descriptor block built by Gfid is designed to be easily read by the Gfid/enter function (the function that creates file directory entries (FDEs)) and Look, the file look up function.

### Looking up the file, processing <?> as drive select

Gfid processes the wild card device selection, "<?>", in a file name. If Look was not specified in the call to Gfid, a 0509 error is generated. Note that if the error is returned, the filename has NOT been scanned into the lookup block. When <?> is recognized as the device selector, the disk drives in the system are searched for the file in the following manner:

1) Set drive # to 0
2) Increment drive #, Look up file. If no errors are returned by Look, go to 4).
3) If drive # <4 then go to 2, else set drive # to 1
4) Set drive # into lookup block, convert drive # to Ascii character and store into string where ? was found. Look up file and process accordingly.

Note carefully that the input string is modified. If the user gives Gfid the string <?>Bessel, and file Bessel.BS is found on drive 2, the string <2>Bessel will be in the user's buffer, and the FDE for file <2>Bessel will be returned in the lookup block. If the file is not found on any drive, the string <1>Bessel will be left in the buffer, and the procedure for handling a 0300 error will be followed. Note that ANY error returned by Look causes Gfid to examine the next drive, or stop the process.

### Interaction of default extension and user extension

When Gfid goes to Look up the file, the following procedure is used:

1) If no extension was passed to Gfid, and no default extension was given in BC, the Look is done with 80H+drive # passed to Look, allowing a match on any file with the same name. The 80H bit is returned in byte 0 of the lookup block indicating that no extension was given by the user.
2) If no extension was passed to Gfid, but a default extension was passed in BC, the Look is done with the default extension, passing only the drive # to Look in A, requiring an exact match. The 80H bit is returned in byte 0 of the lookup block indicating that no extension was given by the user.
3) If an extension was passed to Gfid, and no default extension was passed in BC, the Look is done requiring an exact match, using the user-supplied extension. The 80H bit is not set in byte 0 of the returned lookup block, indicating the user supplied an extension.

4) If the user supplies an extension, and a default extension was passed in BC, the user-supplied extension is used in the Look, which follows the procedure given in (3) above.

Note that the 80H bit in byte 0 of the returned lookup block indicates, if set, that the USER did not specify an extension. If the file was looked up by Gfid, an extension is present in the returned lookup block. If the 80H bit is returned set, this extension will match the default, if one was passed to Gfid. If no default extension was passed in BC, and the 80H bit is returned set, then the extension returned from the Look is from the first matching file on the drive.

Note: The following discussion is quite detailed!

To reduce the possible number of times that the system switches drives, you may wish to call Look from within your own code, rather than having Gfid do it. Let us assume that you want to invoke Gfid to get an input file specification from the user. You then want to read that input file. Assume also that the file is on a disk in a drive other than drive #1, and that Gfid is invoked by a call to Ovrto. If you request Gfid to Look up the file (40H bit set in A), then the system will access drive #1 to get the Gfid overlay. Gfid will then access the target drive to read its directory for Look, and then access drive #1 again to restore the previous overlay. You then access the target drive for data. For this scenario a total of 3 drive switches are involved (assuming drive #1 was selected at first). If your code does the Look rather than Gfid, then only 1 drive switch occurs. This type of optimization may be desirable in some cases.

Termination Characters and Character Scanning:

When Gfid parses the text buffer it will skip leading spaces and tabs. A file specification is delimited by a comma, plus sign, space, tab, or carriage return. The filename extension is delimited from the file name by a dot. If no drive specification is given by the user, drive #1 is assumed.

If you are invoking Gfid to scan multiple file specifications on a single line, note that the scan pointer passed in HL must be incremented past a comma or plus sign delimiter, since Gfid will not skip these characters.

Error Codes Returned by Get-file-identifier:

| | |
|---|---|
| 0500 | Invalid disk number specified |
| 0501 | Name longer than 31 characters |
| 0502 | Extension longer than 2 characters |
| 0503 | Zero length name given |
| 0509 | "<?>" specified, but Look not requested |

If Gfid is invoked requesting Look, then 03XX errors may be returned by Look and Dio (see Section 3.3).

Examples of Get-file-identifier Use:

```
; Sample coding showing use of Gfid to get a file
; identifier.  We want to get an input file spec
; from the user using .TX as a default extension,
; and have Gfid look it up for us.  OOPS is our error
; bailout point.
;
        REF     Ovrto    ; invoke overlay service
;
BUF     DS      43       ; where to put the body
Prompt  DB      'Input file is:',0
;
Doit    LXI     H,Prompt          ; the prompt to use
        LXI     D,BUF             ; put stuff here, please
        LXI     B,'TX'            ; default extension
        MVI     A,0E0H            ; read, look, ext.
        CALL    Ovrto             ; go get it
        DB      'Gfid'
        JC      OOPS              ; no good. Complain.
;
; We now have drive # in BUF, FDE starting at BUF+1
; We need to pick up FDA and NSCTR, and start reading.
;
```

3.4.2.2 Enter/Replace FDE Function

Function Name: Enter/replace FDE
Overlay Name: Gfid

Registers on entry:

| | |
|---|---|
| HL: | Points to file block built by Gfid (Get-file-identifier function) or file block in same format as a block built by Gfid. First byte of block contains disk drive number; this byte is followed by the FDE that is to be entered in the directory. |
| DE: | Unused |
| BC: | " |
| A: | 1H to enter new file into directory; 81H to replace existing FDE (File Directory Entry). |
| PSW: | Unused |

Registers on exit:

| | |
|---|---|
| HL: | Junk |
| DE: | If carry set in PSW, error code/subcode in register; else junk. |
| BC: | Junk |
| A: | Junk |
| PSW: | If carry set, DE contains error code/subcode. |

Description:

The Gfid Enter/replace function allows you to enter or replace FDEs (File Directory Entries) in disk directories. The file block passed to the Gfid/Enter or Gfid/Replace functions is the same block returned by the Gfid/Get-file identifier function, or is in the same format as a file block built by Gfid/Get-file-identifier. The Gfid Enter/replace functions are selected by a set 1H-bit in A. The replace function is chosen over the enter function if the 80H-bit is set in A.

The enter function creates a new file directory entry (a new FDE) in a specified directory. No undeleted files with the same name and extension as the new file can exist on the disk, or a 0505 error code (file already exists) will be returned. The replace function replaces the FDE for an existing file with a new FDE for that file. If the file that you specify does not exist a 0300 error (file does not exist) will be returned. The replace function CANNOT be used to change file names or extensions, but all other attributes within the FDE may be modified (such as deleted or system status, load and start addresses, etc.). Caution!: Do not change the starting disk address (FDA) in the FDE. The PACK command assumes that the sequential ordering of FDEs in the directory corresponds to the sequential ordering of disk sectors in the files on the disk.

Error Codes Returned by Gfid/Enter or Gfid/Replace:

Enter:

| | |
|---|---|
| 0505H | File already exists |
| 0504H | Directory full (file not entered) |

Replace:

| | |
|---|---|
| 0300H | File does not exist |

Since both enter and replace functions work with the directory, 03XX or 01XX errors may be reported as the result of data transfer errors.

Example of Replace Function Use:

The following routine demonstrates the use of the Gfid/Get-file-identifier and Gfid/Replace FDE functions for setting the "system" bit for specified files.

```
; Example showing Gfid use to tweak system bit
; in FDE's....
;
        REF     Ovrto   ; overlay service
        REF     Msg     ; display a message
;
BUF     DS      44      ; file buffer
Prompt  DB      'File name:',0
Ison    DB      'Its on!',0
Isoff   DB      'Its off!',0
;
; The code.
;
Start   LXI     H,Prompt
```

```
        LXI     D,BUF           ; prompt and buffer area
        MVI     A,0A0H          ; read and look
        CALL    Ovrto
        DB      'Gfid'
        JC      OOPS            ; nope, something wrong.
        LXI     H,BUF+1
        MOV     A,M             ; get tags byte
        XRI     40H             ; toggle sys bit
        MOV     M,A
        DCX     H               ; point at block start
        MVI     A,81H           ; tell 'em to replace
        CALL    Ovrto
        DB      'Gfid'
        JC      OOPS            ; nope....
        LXI     H,Ison
        LDA     BUF+1
        ANI     40H             ; see what we tell ya.
        JNZ     Doit            ; the right one.
        LXI     H,Isoff
Doit    CALL    Msg             ; print the message
;
; All done.
;
```

### 3.4.3 Other Overlay Services

This section describes functions available to the assembly language programmer made available by other system overlays. These functions are normally invoked by the Exec, and expect to have a command line in the system command buffer CMND (see section 3.2.3). To use these functions, then, you must put a command line into CMND similar to what you would type to the Exec, and terminated by a carriage return. The overlay function is then invoked by Ovrto or Gover, with the function code passed in A. Note that this command line in CMND must include characters for the command name, if the command takes arguments. In the argument scanning process used by these functions, characters beginning at CMND are skipped until a delimiter (Tab, space, carriage return, comma, etc,) is found, and then an argument scan is done (if the delimiter was not a carriage return). This procedure is used because most command names may be abbreviated (and because it is esthetically more pleasing). This means that if you wish to invoke overlay Dfn2 to list a disk, putting any one of the strings "XX 2", "list 2", "teafor 2" into CMND, followed by a carriage return, and invoking the overlay, will cause the directory of drive 2 to be listed. Note that only ONE line is put into CMND; those commands requiring further information read from the user through WHO, and not from CMND (such as SAVE).

In processing the command in CMND, each overlay takes control of PVEC to intercept control-Y. Each overlay processes its own errors, calling Emsg to display an error message, and returns control to the caller with no indication of success or failure in processing the command.

#### 3.4.3.1 Functions provided by Dfn1

The following commands are processed by the overlay Dfn1. The command line is expected to be in CMND, terminated by a carriage return.

| Code in A | Function Name | User's Guide Reference Section |
|---|---|---|
| 0 | SAVE | 5.2.9 |
| 1 | IMAGE | 5.2.13 (see Note 1) |
| 2 | INIT | 5.2.14 (see Note 2) |
| 3 | HELP | See Note 3! |
| 4 | RENAME | 5.2.8 |
| 5 | SetSys | See Note 4! |

Note 1: When the IMAGE process is completed, the system is warm-started by jumping to Warm (see section 3.3.5).

Note 2: INIT first calls Killi to flush type-ahead, and abort command files, and then asks the user for the number of the drive to initialize. Invoking this command from another program, then, is of marginal utility.

Note 3: The HELP command is not fully implemented at this time. It will be fully implemented in a later release of the system.

Note 4: The SetSys command is not documented in the User's Manual. This command prompts the user for a disk number, and then sets the system bit on each file on that drive, even files marked deleted. It is documented here, as it may be of use to systems/applications builders as a method of easily protecting all the files on a disk.

3.4.3.2 Functions provided by Dfn2 overlay

The following functions are provided by Dfn2. The command line is expected to be in CMND, terminated by a carriage return. The function code is passed to the overlay via A.

| Code in A | Function Name | User's Manual Reference Section |
|---|---|---|
| 0 | LIST | 5.2.2 |
| 1 | DELETE | 5.2.3 |
| 2 | UNDELETE | 5.2.4 |
| 3 | PACK | 5.2.5 |
| 4 | TYPE | 5.2.6 |
| 5 | Sniff | See note 1! |
| 6 | PRINT | 5.X.X |

Note 1: Sniff is another command not documented in the User's Manual. When called with a drive number, in the same command format as LIST (e.g. Sniff 2), Sniff reads each sector of the disk sequentially, and will report any errors it finds by displaying the offending sector number, and error code on the screen in hexadecimal. For example, the notation 00BF/0103 displayed by Sniff indicates that in reading sector BF of the disk, a 0103 error, checksum bad, was reported by Dio.

3.4.3.3 Functions provided by the Dfn3 overlay

The following functions are provided by the Dfn3 overlay. As with Dfn1 and Dfn2, the command line is expected to be in CMND, and the function code passed in A.

| Code in A | Function Name | User's Manual Reference Section |
|---|---|---|
| 0 | MEMTEST | 5.2.16 |
| 1 | Dump | Note 1! |

Note 1:
The Dump utility is provided as a convenience to the systems programmer, and is also used by the system. The Dump utility is called in Dfn3 specifying a memory range to be dumped to the printer, and a comment line to

be placed at the beginning of that memory dump. The specified area of memory is dumped in hexadecimal form to the printer (via WH7) and the screen (via WH1). Duplicate lines of output are supressed.

Function Name: Dump
Overlay Name: Dfn3

Registers on entry:

| | |
|---|---|
| HL: | Gives the starting address of the memory area to dump |
| DE: | Ending address of area to dump |
| BC: | The address of a string to be printed at the top of the memory dump, terminated by a CR (carriage return) and a 00 byte |
| A: | 1 (to select the Dump function in Dfn3) |
| PSW: | unused |

Registers on exit:

| | |
|---|---|
| HL: | junk |
| DE: | " |
| BC: | " |
| A: | " |
| PSW: | " |

Description:

Dump dumps the selected memory area specified by the contents of HL and DE to the printer and the screen. The string pointed to by BC is printed along with the memory limits as the first line of the memory dump. This string should be terminated by a carriage return and a zero (00) byte. Dump first outputs the selected memory limits and the title string to the printer and the screen, and then begins dumping memory in hexadecimal form, sixteen bytes per line. After printing a line, the next sixteen bytes of memory are examined to see if they are identical to the previous 16 bytes. If they are identical, this 16 byte area is not printed, as it is a duplication of the preceeding area.

Example of Dump function use:

The following code dumps the area from 4400H to 47FFH to the printer and the screen, with the title line shown as Title.

```
; Dump 4400H to 47FFH to the printer...
;
Title   DB      ' The sector and its preamble',0DH,0
;
Snap    MVI     A,1      ; function code for Dfn3
        LXI     H,4400H  ; start address
        LXI     D,47FFH  ; end address
        LXI     B,Title  ; string
        CALL    Ovrto
        DB      'Dfn3'   ; go dump me some crud.
        RET              ; that's all
```

The output produced on the printer and the screen is similar to the following:

```
4400 to 47FFH The sector and its preamble
4400 88 0C 80 00 DB 58 68 C4 18 12 E0 99 D2 43 60 0B
4410 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
47F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The skip of addresses from 4410H to 47F0H on lines two and three indicates that the region of memory from 4410H to 47FFH contained all zeroes.

# Section 4

# UTILITIES FOR THE SYSTEMS PROGRAMMER

## 4.0

Section 4 describes three utilities for the System 88. Each of these programs is on the system disk included with this manual. The first utility program described is Emedit, an editor for error message overlays. This program allows the systems builder to tailor system error messages to the end user, and to add new messages for use by applications systems. The second utility is Szap, a program used for examining and manipulating the contents of disks and memory. Szap is a powerful tool meant for use by experienced programmers. The third utility included is the Auth overlay, which requires users to give authorized names and passwords before using the system. With the Auth overlay present on the system disk, unauthorized use of the system is made difficult. Many users may be authorized for each sustem disk, each with their own password. Each user is able to change his or her own password, but not those of other users.

## 4.1 UTILITY PROGRAM Emedit - AN EDITOR FOR ERROR MESSAGE OVERLAYS

Emedit allows the system programmer to examine and modify error message overlays in the System 88. Using Emedit, the user may: view the messages in an error message overlay, delete messages, add new messages, list the error messages to the system printer, or replace messages.

### 4.1.1 Restrictions

Emedit will edit only system error message overlays. This means the name of the file to be edited must be exactly four characters long. The load and start addresses in the file must be 2000H. Location 2007H in the overlay is expected to contain a pointer to the body of messages within the overlay (see Section 7). In addition, Emedit must be invoked in the enabled mode. If invoked in disabled mode, Emedit returns to the Exec.

### 4.1.2 Using Emedit

After Emedit is invoked in the enabled mode, the version number and command list are displayed on the screen. Commands are given to Emedit by typing a command character, in either upper or lower case. The command characters are:

| Character | Section | Command |
|---|---|---|
| A | 4.1.3 | Add message |
| D | 4.1.4 | Delete message |
| E | 4.1.5 | Edit error file |
| R | 4.1.6 | Replace message |
| V | 4.1.7 | View messages |

| | | |
|---|---|---|
| L | 4.1.8 | List messages |
| X | 4.1.9 | Exit |

If the character typed is not a recognized command character, the command list is displayed on the screen.

4.1.3 The A Command - Adding messages to the text

In response to the A command, Emedit prompts the user for the error code to add by displaying the text:

    Add error code:

and accepts a hexadecimal number. Lower case letters are folded to upper case, and the conversion stops when a character not in the set [a-z,A-Z,0-9] is encountered. This number is the number under which the new message will be stored. Any existing messages in the file with that code will be deleted. Emedit then displays:

    Terminate new message with ESC-CR

on the screen, and prompts the user for lines of input with the prompt character '<'. Input is accepted until a line ending with an escape (ESC) character followed by a carriage return (CR) is detected. This terminates processing in the Add command. If no file has been opened for editing by using the E command, the error text

    No file open for editing- use E first

is displayed on the screen. If the added text would force the overlay over the maximum size of 2K bytes, the message

    Message truncated-Overlay is full!

is displayed, the message truncated, and the command terminated.

4.1.4 The D Command - Deleting a Message

The D command is used to delete messages from the file. The user is prompted with

    Delete error code:

and a hexidecimal number is input, for the message to delete. If the message is not found, the text

    I can't find that message

is displayed. If no file has been opened for editing, the text

    No file open for editing- use E first

is displayed, and the command terminated.

### 4.1.5 The E Command - Opening a File for Editing

After typing E to invoke the Edit command, Emedit prompts with

    Edit file name:

and waits for the user to enter the name of the error message overlay to edit. The file is validated, as described in the section on Restrictions, above. Any errors in looking up the file are reported to the user, and terminate the command. If the file does not look like an error message overlay, the text

    That's not an error message overlay!

is displayed and the command terminated. If another file was open for editing at the time the E command was given, that file is closed, and re-written to disk if modifications have been made.

### 4.1.6 The R Command - Replacing a Message

The R command is similar to the A command for adding a message, but assumes that there is a message with that code already in the file. The user is first prompted with the text

    Replace error code:

and the error code is read. If no message with that code is found within the overlay text, the message

    I can't find that message.

is displayed on the screen, and the command terminates. If the message exists, it is deleted, and the A (Add) command invoked to add the message.

### 4.1.7 The V Command - Viewing the Contents of the File

The V command displays the messages in the file on the screen. The display is stopped at the end of each page, and a dot (.) is displayed. The user may type either the single character 'x' or 'X' to abort the display at that point, or any other character to continue the display. The error codes and texts are displayed in their order of appearance in the file. Messages added with the A (Add) or R (Replace) commands will appear at the end of the file. If no file is open for input, the message

    No file open for editing- use E first

is displayed, and the response to the command aborted.

### 4.1.8 The L Command - Listing messages to the system printer

The L command lists all error messages in the currently open file to the system printer. An error message results if no file is currently open for editing. The error messages are listed in the same format as produced by the V (View) command.

### 4.1.9 The X Command - Exiting the program

If no file was open for input when the X command was given, or no modifications had been made to the file currently open, Emedit returns to the Exec. If the currently open file was modified by use of the D (Delete), A (Add), or R (Replace) commands, the new file will be written to disk. If the file has not increased in size, the new contents will be written over the old file on the disk. If the file has increased in size, the old copy of the file is deleted (even if it is a system file), and a new copy of the file is created on the disk.

### 4.1.10 BASIC Error Messages

The error messages for BASIC are in the error message overlay Berr. To save space in the overlay, if the last character in a message is the letter "e", this will be expanded to the word "error". This expansion turns the string "Syntax e" in the Berr file into the string "Syntax errror" on the screen when the error is reported from BASIC. The user should therefore beware of inserting messages into the Berr file that end in the single character "e".

### 4.1.11 Suggestions for using Emedit

When adding new error codes to the system, write them down and add them to the User's Guide, as well as to the System Programmer's Guide, and any applications documents. Make error messages clear, give as much information about what caused the error as possible, and use good grammar. Be cautious in inserting obscene error messages- if a disk containing vulgar error messages accidentaly gets released or sent to customers, it can cause a lot of trouble!

## 4.2 UTILITY PROGRAM Szap

Szap (SuperZap) is a utility program that allows the experienced systems programmer to examine and modify the contents of both system RAM and disk storage. Szap is a powerful tool when used correctly, and is capable of destroying the contents of disks and main memory when used incorrectly.

Szap allows the systems programmer to display a selected 256-byte page of main memory, or a selected disk sector. Szap displays the page in hexadecimal form, with an optional character display. The user may move an editing cursor through the selected page by use of the cursor controls and so display the previous or next page of memory or disk. To modify data present in the display, the programmer enters either hexadecimal bytes or character strings. Additionally, Szap can zero the contents of the page from the cursor to the end of the page in response to a single keystroke. Szap makes modifications to main memory pages as the user enters the new data. A modified disk page (sector) is written to the disk when a request is made to display another page or to exit the program. The programmer may disable error-checking and reporting when modifying disk sectors: this allows the systems programmer to attempt to reconstruct damaged disk directories and the like.

### 4.2.1 Running Szap

The user must be in the enabled mode to execute Szap. If the user tries to invoke Szap when in the disabled mode, Szap immediately returns control to Exec. When Szap begins execution, the control-Y vector is set to force exiting of the program (that is, a control-Y will cause you to exit from Szap). Szap clears the screen and a cursor is present in the upper left corner of the display. A command summary and the Szap version number also appear. Szap then waits for a command. A Szap command is either a single character or a hexadecimal number. (The single character commands appear below.) A number alone is an implicit command to Szap to place that byte at the cursor position in the page that is displayed on the screen. A hexadecimal number is terminated by a space whether it appears as the default data entry command, or as a command argument. The command characters recognized by version 2.1 of Szap (and their associated functions) are:

| Character | Function |
|---|---|
| Control-E | Exit |
| Control-Y | Exit |
| ' | Begin text entry (single quote) |
| ESC | Toggle text display mode |
| ESC | Terminate text entry (escape) |
| :n | Select device n for display |
| /n | Select page n for display |
| I | Display indirect |
| ! | Toggle error check on disk data transfers |
| Z | Zero data from cursor to end of page |
| RET | (Carriage return) Display next page |
| LF | (Line feed) Display previous page |

(n = a hexadecimal number)

Szap folds lower case letters to upper case when accepting commands or hexadecimal numbers as input.

The four cursor controls have the following functions:

| Cursor Control | Function |
|---|---|
| UP | Move to beginning of previous line |
| DOWN | Move to beginning of next line |
| LEFT | Move cursor left one byte |
| RIGHT | Move cursor right one byte |

Szap displays the preceding page if the user moves UP or LEFT from the top of the screen display; it displays the next page if the user moves DOWN or RIGHT from the bottom of the display.

The cursor appears in the upper left hand corner (byte 00) of a new page display.

### 4.2.2 Exiting Szap: Control-Y or Control-E

To exit from Szap the user types a control-Y or control-E. Any modified disk pages not yet written out will be written to the selected disk drive. Once the user exits Szap, Szap may not be restarted or reentered by way of the system commands START and REENTER. The user must re-invoke Szap to use it again.

### 4.2.3 Hexadecimal Data Entry

Entering a hexadecimal number is the implicit command to Szap to store the least significant eight bits of that number in the memory location pointed to by the cursor. The number is delimited by a space. Once the space has been entered, the selected byte is updated and the cursor moved to the right (the next location in the page). Typing errors are corrected by simply typing enough characters so that the least significant eight bits (the last two digits) of the number are correct. The strings "2 ," "9A02 ," and "3E002 " all store the eight-bit hexadecimal quantity "02."

### 4.2.4 Text Entry: The ' Command

A single quote symbol places Szap in the text entry mode. All characters typed from that point on, with the exception of control-Y, and ESC (escape) will be entered into successive locations in the displayed page. Note that this includes control characters such as carriage return, and the cursor control keys! ESC (escape) is used to terminate the text entry mode. To terminate text entry and exit from Szap the user types a control-Y.

### 4.2.5 Toggling the Text Display: The ESC Command

The user may display the page in text form on the right portion of the screen. The ESC command enables or disables this display. When the text display is enabled the frame address is not displayed, and those characters in the range 00 to 7F hexadecimal are displayed in their normal ASCII form; the values 80H through FFH display as blanks. NOTE: Szap will not display the contents of the screen properly if you try to display the video board itself!

### 4.2.6 Selecting the Device - The : Command

The command character ':' (colon) followed by a hexadecimal number selects the device to be displayed and edited. Device zero denotes main machine memory, device 1 is disk drive 1, and so on. If the page currently on display represents a disk page that has been modified, that page will be written out to the proper device before the : command is processed. When a disk is edited, the frame number displayed in the upper right corner of the screen consists of the device number and a four digit hexadecimal number representing the sector address. (Note: the frame address does not appear when the text display is enabled--see Section 4.2.5.) When a disk is selected, sector 00 is automatically displayed on the screen as the current page.

### 4.2.7 Selecting the Displayed Page: The / Command

To display a particular page of the device being edited, the user types /nnnn, where nnnn is a hexadecimal number. This number selects the desired page. When device 00 (main memory) is being edited, only the upper eight bits (the two most significant digits) of the last four digits of the number are used to select the page to be displayed. When the user is editing a disk (by using the : command--see Section 4.2.6), Szap uses the entire number as a sector address. In either case the number is terminated by a space. Typing errors are corrected by entering more digits since only the last four hexadecimal digits of the number are used. For example, suppose the user enters the number 12345678. If a disk is being edited, sector number 5678 will be displayed. If main memory is being edited, page number 5600 will be displayed.

### 4.2.8 Display Indirect: The I Command

The I command uses the 16-bit address pointed to by the cursor in the present frame as the new frame to display. This number is treated in standard 8080 fashion, least significant byte first. If the user executes the I command while the cursor is pointing to the two bytes containing the number "3D 01," sector 13DH is displayed (if a disk is being edited). If main memory is being edited, page 100H is displayed. If the page currently on display is a modified disk sector, that sector will be written out to the disk before the next sector is displayed.

### 4.2.9 Disabling Disk Error Reporting: The ! Command

WARNING: THIS IS DANGEROUS!

The ! command toggles a flag that enables or disables Szap disk error detection and reporting. When the user inputs the ! command, this flag is displayed on the screen following the frame address. A value of 0000 indicates that errors will be reported; a value of FFFF indicates that errors will be ignored. It is sometimes useful to disable error detection and reporting when attempting to recover destroyed or unreadable disk directories. Although useful, this feature is dangerous--use with extreme caution!

### 4.2.10 Szap Display of Error Conditions

When Szap encounters an error (such as a disk transfer error), it clears the text display flag, and displays the error code on the screen to the right of the frame address. The error code displayed is the one reported by the system. (See Section 5 for a listing of error codes and their associated messages.)

### 4.2.11 Zeroing the Page: The Z Command

The Z command zeroes the contents of the page on display from the cursor position to the end of the page. The previous contents of the page are lost. If the user accidentally gives this command while viewing a memory page, that page of memory is zeroed and the previous contents of that page are lost beyond recovery. If the user gives this command by accident while displaying a disk page, the ONLY way to prevent the (partially) zeroed sector from being written to the disk is to RESET THE SYSTEM (and be more careful from then on).

### 4.2.12 Displaying the Next Page: The RETURN Command

To display the next page, the user types a carriage return (CR or RETURN). If a disk is being edited, the next sector on the disk is displayed. If main memory is being edited, the next 256-byte page is displayed.

### 4.2.13 Displaying the Previous Page: The LINE FEED Command

The previous page will be displayed when a LINE FEED (LF) is typed.

### 4.2.14 Cursor Movement Using the CURSOR Keys

The four arrow keys at the right of the keyboard are used to move the cursor up, down, left, and right within the page on display. Their use may also cause the previous or next page to be displayed if they are used to move off the top or bottom of the frame being displayed. The left and right arrows move the cursor left or right one byte. The up arrow moves the cursor to either the beginning of the current line, or to the begining of the previous line. The down arrow moves the cursor to the begining of the next display line. The cursor keys in coordination with LINE FEED and RETURN allow the user to move the cursor forward or backward one byte, one line (16 bytes), or one page (256 bytes).

### 4.2.15 Attempting to Reconstruct Directories

NOTE: A complete understanding of Section 2 of this manual is necesary, but may not be sufficient, in attempting to reconstruct a damaged disk directory. Making back-up copies of important disks on a regular basis is much easier than trying to reconstruct a damaged directory.

When Szap is instructed to read disk sectors 0, 1, 2, or 3 of a disk device (the directory sectors) one sector is read into the internal editing buffer. When another sector is selected, or any other event takes place that would cause that updated sector to be written out to the disk as part of the directory, Szap follows the following procedure:

1) Each of the directory sectors 0, 1, 2, and 3 of the selected device are read into the system

directory one sector at a time. This means that four individual calls to Dio are made to read the directory, each requesting one sector, rather than one call to Dio requesting four sectors.

2) The sector updated by Szap is copied to its correct place in the directory area.

3) The directory checksum is recomputed and stored in both the directory header and NFCK (see Section 3).

4) The verify mode is turned on by setting RAW non-zero (see Section 3).

5) The four directory sectors are written out by one call to Dio.

If a disk directory is unreadable because of a checksum error on one of its sectors, or some similar error, the following procedure is suggested, BUT NOT GUARANTEED:

1) Try reading the disk directory on other drives in the system.

2) Image the disk onto a scratch disk, and try to read that disk on other drives.

3) If (1) and (2) have not succeeded, use Szap to examine the first four sectors of the disk to determine the type of problem, and which sector or sectors are affected. You can also use Sniff to check for hard errors.

4) If the system can read sectors 0 through 3, chances are some program has gone wild and hosed into the directory. In this case, the directory may be carefully reconstructed by hand, one sector at a time.

5) If a checksum or preamble error has occurred, making one or more sectors of the directory unreadable, the ! command may be used to disable error checking. You can then read the offending sector into memory, correct it by hand, and then write it back to disk. After this is done, use the ! command again to enable error checking. Re-examine the directory sectors to determine if there is a hard media error, or if the error has been covered up.

6) After the disk has been "fixed," by performing (4), (5), or other procedures, the important files on it should be INDIVIDUALLY copied to other disks, and then the offending disk should be re-initialized by using the INIT command. This is very important, especially if a directory was re-built by hand. Such a reconstructed directory may have subtle errors in

it that are not immediately apparent, but that will cause a catastrophe the first time a file is deleted, the disk is packed, or a new file is created on the disk.

### 4.2.16 Morals to Reconstructing Directories

The following suggestions are made in the hope you will never need Section 4.2.15, and the trauma that accompanies it:

1) Perform preventative maintenance on your system on a regularly scheduled basis. This should consist of running the memory test, cleaning the heads on cassette machines, etc.

2) Log hard disk errors, such as checksum errors and preamble errors, recording both the name of the disk and the offending drive. This information may help in tracking down a bad drive, or compatibility problems between drives.

3) If possible, write-protect System disks.

4) Keep write-protected backup disks. The more important the contents of a disk is, the more often it should be backed up. When making backup copies use a SET of disks for backup, and rotate the usage of the backup disks so that you write over the oldest backup copy each time. After making a backup copy, "Sniff" the disk; or use some other procedure to verify that the backup is good. Backup disks should be write-protected and stored away from other disks.

The general moral of this section is to treat your system like a "real computer." Regularly scheduled and performed preventative maintenance is capable of detecting problems before they cause trouble. Regular backup of the file system leaves you less vulnerable if disaster does strike. Preventative measures take time and use up disks, but can minimize losses.

## 4.3 UTILITY PROGRAM Auth

The Auth overlay is an optional component of the System 88 that requires users to give an authorized name and password before using the system. Systems containing Exec version number 52 (or later) perform this authorization process if the Auth overlay is present on the system disk. This authorization process is not meant to be "totally secure", or to totally prevent unauthorized use of the system; it IS meant to make unauthorized use of the system difficult.

### 4.3.1 Signing on to the System

The system makes a check for the Auth overlay during every system boot. If the overlay is present on the system disk, the system invokes it with a function code of 00, which it passes to Auth in the Accumulator. The function code of 00 tells Auth to ask for a user name and password. Auth prompts the user to enter his or her name, and Auth checks the name against an internal list of authorized names. If the name is present, Auth then prompts the user to enter a password. The password does not echo to the screen as the user enters it. If Auth either does not find the name on its authorization list, or the password is incorrect, it displays an error message to the user; the system then goes into a loop after disabling interrupts and zeroing part of memory. At that point the user must re-boot the system if he or she wishes to try again.

The user name may be up to 60 characters in length, and must be terminated by a carriage return. When processing the password, Auth reads up to 60 characters terminated by a carriage return; however, it uses only the first 16 in the validation process. If the password contains less than 16 characters, Auth automatically appends nulls to fill it out to that length. The initial greeting message, the password request message, and the failure message are in the system error message writer, Emsg. The systems programmer may use the Emedit utility described in Section 4.1 to tailor these messages.

### 4.3.2 The Exec Auth Command

With the authorization processor, a new command, Auth, is added. This command must be given in the enabled mode, and allows the system user to add, delete, and list authorized users, as well as changing passwords for users. The commands to Auth are single characters, as follows:

| Command Character | Auth Function |
|---|---|
| X | Exit Auth, warm-starting the system |
| A | Add user to authorization list |
| D | Delete user from list |
| C | Change password for user |
| L | List names of authorized users |

Those commands that modify the authorization list (Add, Delete, and Change) cause the system to re-write the overlay to the system disk; therefore this disk must not be write-protected when these commands are given.

### 4.3.3 The Auth X Command - Exiting

The X command causes Auth to exit, warm-starting the system.

### 4.3.4 The Auth A Command - Adding Users

The A command is used to add users to the authorization list. Auth first asks for the user name. If this name already appears in the user list, Auth gives an error message and aborts the A command. If the name does not appear on the list, Auth requests the password. The password echoes to the screen as a sequence of question marks (?). The name and password are entered onto the user list, and the overlay is written back to the disk.

### 4.3.5 The Auth D Command - Deleting Users

The D command is used to remove a user name from the list. The user is first asked for the name, which must be on the list, or an error message results. The user is then asked for the password. This must match the password for the user, or an error message is given, and the user name is not removed from the list. If the password matches, the user name is removed.

### 4.3.6 The Auth C Command - Changing Passwords

The C command is used to change a user's password. The user is first prompted for a name, which must appear on the authorization list or an error message is generated. The old password must then be entered, and must match that currently in the file. A new password is then asked for, and replaces the old password in the file, which is then re-written to disk.

### 4.3.7 The Auth L Command - Listing User Names

The L command lists the names of authorized users. Passwords are NOT listed. The listing is pagenated every 14 lines.

### 4.3.8 Auth Messages in Emsg

Most of the messages that the Auth processor uses reside in the system error message overlay, Emsg. Emedit may be used to tailor these messages. The codes and descriptions for the Auth messages are:

| Code | Message Description |
|---|---|
| 801H | Greeting message issued by Auth |
| 802H | Unauthorized user message |
| 803H | Prompt for password request |
| 804H | Can't find that user- D or C commands |
| 805H | User already authorized- A command |
| 806H | Incorrect password given- D or C commands |

The messages initially in Emsg for these codes are somewhat whimsical in nature.

### 4.3.9 Installing Auth on the System 88

To install the authorization checker, copy the file Auth.GO from the disk included with the System Programmer's Guide to the desired system disk as file Auth.OV. Note that for Auth to be used, the Exec on the system must be version 52 or later. Using the Exec Auth command, authorize one or more users. No users are authorized in the file as it is shipped. The Szap utility may be used to set the system bit (see Section 2) on the Auth.OV file to insure it is not deleted, or the SetSys command may be used (see Section 3) to make all files on the system disk system files. If the Exec is version 52 or later, and the Auth.OV file resides on the system disk, whenever the system is booted, the user must enter a name and password before the system may be used.

### 4.3.10 How Auth Connects to the Exec

In the initialization process, before the Exec looks for the INITIAL file, it checks to see if the file Auth.OV exists on the system disk. If this file exists, it is called by an Ovrto (see Section 3) with a function code of 00 in A. When the overlay is entered, MEMTOP HAS NOT BEEN SET, and therefore contains zero. The Auth overlay "disconnects" PVEC and UVEC, and sets the system in disabled mode by clearing SCHR, to prevent it from being interrupted by the user. If the user is authorized the Auth overlay returns. If the user is not authorized, the remainder of the overlay area is zeroed, and the system hangs.

### 4.3.11 User Written Auth Overlays

To provide for more security, or for other reasons, the systems user may wish to provide a custom Auth overlay. This overlay should be written to conform to the conventions described for overlays in this manual. As noted before, since Auth is called very early in the boot process, MEMTOP has not been set, so no sstem services that depend on this cell should be used. The user-written Auth overlay should recognize two function codes passed in the A register:

| Code in A | Auth Function |
|---|---|
| 00 | Verify user authorization |
| 01 | Exec Auth command given |

### 4.3.12 Storage of Names and Passwords in Auth

The list of user names and passwords authorized is stored as part of the Auth overlay. The password entry associated with each name is stored in an encrypted form. The encryption process used is simple-minded, and is present as a hinderance in obtaining the passwords of others, rather than as absolute security. In the validation process, or in validating a password for the C (change) or D (delete) commands, the password entered by the user is encrypted, and compared to the encrypted entry within Auth. This insures that the "clear text" of the password is not left in memory for very long.

### 4.3.13 "I forgot my password" - or - How to Break Auth

All that is required to "break" Auth is a system disk that does not have Auth connected, and a system with more than one disk drive. The "unprotected" system may be booted and used to delete the copy of Auth from the protected system disk. If the copy of Auth is marked a system file, protecting it from deletion and renaming, Szap or a similar program may be used to clear the system bit, and then delete Auth. A different method is to copy everything from the protected system disk except Auth.

Once users are authorized, those authorizations may not be changed or removed without knowing the associated passwords. A new, "clean" copy of Auth may be installed, without any authorizations, and then user names added. It should be possible for the persistent user to break the encryption process used on the passwords, but no details on the algorithm used will be given here.

### 4.3.14 Suggestions for Using Auth

User names may be as long as desired, up to 60 characters in length. The password selected should be easy for the user to remember. A password that is easy and quick to type is desirable in those cases where others are watching you type your password. Remember: if you forget your password, it is very difficult to recover. To be effective at a computer installation, every system disk should have Auth on them, including backup disks. The Auth processor is NOT meant to provide "absolute" security from unauthorized use of the system; it is meant to hinder unauthorized use.

# Section 6

# SYSTEM 88 ERROR MESSAGES

## 6.0

This section contains a listing of the System 88 Error messages in numerical order. Section 6.1 contains the messages from Emsg, the system error message writer. Section 6.2 contains the messages from Berr, the BASIC error message writer. As the system expands, these error message writers will undoubtably expand with it. When using the system error message editor, Emedit to modify messages, or add new messages to an error message writer, please update both the user and system documentation.

## 6.1 Emsg ERROR MESSAGES

The following messages are generated by the Emsg error message writer:

> The messages with error code 01 are generated by Dio as the result of either bad parameters passed for a disk transfer, or an error in attempting the disk transfer.

Error code 0101
DIO says: Bad parameters!

Error code 0102
DIO says: Hard error! Preamble bad!

Error code 0103
DIO says: Checksum error!

Error code 0104
DIO says: Verify error!

Error code 0105
That disk is write protected!

Error code 0106
The door is open, or no disk in the drive!

> Messages with code 02 are generated by the Exec in response to user requests.

Error code 0201
I can't run that file

Error code 0202
Nothing to run!

Error code 0203
Dont what?

Error code 0204
What?

> Messages with code 03 are generated by Look. Error code 0300 indicates a file was not found on a Look request. Code 03FF indicates a destroyed directory (see Section 2). The other 03 codes result from Dio errors in attempting to read the directory; the code of 01 from Dio is changed to 03 by Look.

Error code 0300
I can't find that file!

Error code 0302
Disk directory unreadable!

Error code 0303
Disk directory unreadable!

Error code 0306
I can't read the directory- no disk in the drive, the door is open, or no such drive

Error code 03FF
Disk directory destroyed!

> Message codes 05, 06, and 07 are generated by various system processors such as Gfid, the Editor, and the assembler.

Error code 0500
Gfid says: Bad disk identifier

Error code 0501
Gfid says: Name too long

Error code 0502
Gfid says: Illegal extension

Error code 0503
Gfid says: Name null or wierd!

Error code 0504
I can't: the directory is full

Error code 0505
I can't: the disk is full

Error code 0506
I can't rename across drives: use copy

Error code 0507
No new extension given

Error code 0508
I can't do that to a system file

Error code 0509
"<?>" is not allowed here

Error code 0600
That file already exists

Error code 0601
That file does not exist

Error code 0701
Output file not specified

Error code 0702
Output file already exists

Error code 0703
Input file not specified

Error code 0705
Input file does not exist

Message codes 08 are generated by the system authorization processor, Auth. Please see Section 4.3 for more details.

Error code 0801
I am the keeper of the Gates, Cerberus:
What is your name?

Error code 0802
Go away, kid, you bother me...

Error code 0803
What is the password?

Error code 0804
I can't find a user with that name.

Error code 0805
That user is already authorized. Use D or C.

Error code 0806
Incorrect password. Command ignored.

## 6.2 Berr - ERROR MESSAGES FOR BASIC

The following messages are generated by Berr, the BASIC error message writer. Remember that if a Berr message ends with "e", it will be expanded to "error" when displayed.

Error code 0400
Syntax e

Error code 0401
Syntax e

Error code 0402
Subscript e

Error code 0403
Bad argument e

Error code 0404
Dimension e

Error code 0405
Function definition e

Error code 0406
Out of bounds e

Error code 0407
Type e

Error code 0408
Format e

Error code 0409
I can't find that line

Error code 040A
FOR-NEXT e

Error code 040B
RETURN without GOSUB

Error code 040C
Division by zero

Error code 040D
Function definition e

Error code 040E
Missing matching NEXT

Error code 040F
Read e

Error code 0410
Oops...BASIC goofed!

Error code 0411
Oops...BASIC goofed!

Error code 0412
Input e

Error code 0413
Out of memory

Error code 0414
I cant do that directly

Error code 0415
Argument mismatch e

Error code 0416
That line was too long!

Error code 0417
Overflow e

Error code 0418
Tape checksum e

Error code 0419
Tape verify e

Error code 041A
Can't continue!

Error code 041B
That's not a BASIC file!

Error code 041C
Nothing to save!

Error code 041D
That channel not open!

Error code 041E
That channel not open for input

Error code 041F
That channel not open for output

Error code 0420
End of file on that channel

Error code 04FF

Can't do that to an OUT file

## Section 7

## A SAMPLE SYSTEM OVERLAY

7.0

The following assembly listing gives a sample of the form that a system overlay takes. The listing is of a previous version of the system error message writer, Emsg. Note the use of the REF statement to obtain values of system symbols. Using the REF statement makes the program easier to update if a system symbol changes; a re-assembly is all that is required. For error message overlays, note the pointer at OVRLY+7, which points to the start of the text. This pointer is used by the error message editor, Emedit, to access the text.

```
                    ;
                    ; The error message handler.
                    ; We are invoked with the error code expected in DE.
                    ; We puts it into ERROR, moving the previous contents
                    ; to LERR first, and then look for a message associat
                    ; with that error number, and spits it out. If we don
                    ; find the text, we display an "I don't know" and spl
                    ;
                            REFS    SYSTEM   ; definitions
        2000                REF     OVRLY    ; overlay area
                            IDNT    OVRLY,OVRLY
                    ;
        03D1        DEOUT   EQU     3D1H     ; print contents of DE
        041B                REF     Killi    ; kill cmd file, type ahead
        040C                REF     Msg      ; print a message
        2D9A                REF     ERROR    ; system error cell
        2D9C                REF     LERR     ; last error cell
        2DC9                REF     EFLG1    ; error to report flag in her
        0040                REF     EERR     ; error flag in EFLG1
                    ;
   2000 456D7367            DB      'Emsg'   ; our name.
   2004 C30920              JMP     GO
   2007 6420                DW      ETXT     ; pointer for error editor
                    ;
                    ; And away we go....
                    ;
   2009 CD1B04      GO      CALL    Killi    ; go tromp that stuff.
   200C 2A9A2D              LHLD    ERROR
   200F 229C2D              SHLD    LERR     ; move over, please
   2012 EB                  XCHG             ; new one
   2013 229A2D              SHLD    ERROR    ; plotz.
   2016 3AC92D              LDA     EFLG1
   2019 F640                ORI     EERR
   201B EE40                XRI     EERR     ; clear it.
   201D 32C92D              STA     EFLG1    ; for recovery.
   2020 116420              LXI     D,ETXT   ; start of the text
   2023 EB                  XCHG
```

```
                        ;
                        ; We now search the text. It is in the form code,sub
                        ; followed by the message, followed by zero. The end
                        ; of the list is an FF byte.
                        ;
2024 7E         EFND    MOV     A,M
2025 FEFF               CPI     0FFH    ; end hit?
2027 CA5B20             JZ      Nope    ; jmp/yup, no such msg.
202A BA                 CMP     D       ; is this the one, then?
202B C23520             JNZ     EFN1    ; jmp/nope.
202E 23                 INX     H
202F 7E                 MOV     A,M
2030 BB                 CMP     E       ; this the one?
2031 CA4020             JZ      Yup     ; jmp/yes, go print it
2034 2B                 DCX     H
2035 23         EFN1    INX     H
2036 23         EFN2    INX     H
2037 7E                 MOV     A,M
2038 B7                 ORA     A
2039 C23620             JNZ     EFN2
203C 23                 INX     H       ; point past the stinker.
203D C32420             JMP     EFND    ; find this one's end.
                        ;
                        ; Found it.
                        ;
2040 23         Yup     INX     H       ; point past subcode, dummy...
2041 C30C04             JMP     Msg     ; let Msg return for us...
                        ;
                        ; Didn't find it.
                        ;
2044 3F4E6F20   NT      DB      '?No message for error ',0
2048 6D657373
204C 61676520
2050 666F7220
2054 6572726F
2058 722000
205B 214420     Nope    LXI     H,NT
205E CD0C04             CALL    Msg
2061 C3D103             JMP     DEOUT   ; print the code on the way ou
                        ;
                        ; Now comes the text.  In no particular order, 'cause
                        ; Emedit to sort 'em.
                        ;
2064 02014920   ETXT    DB 2,1,'I can',27H,'t run that file',0
2068 63616E27
206C 74207275
2070 6E207468
2074 61742066
2078 696C6500
207C 03004920           DB 3,0,'I can',27H,'t find that file',0
2080 63616E27
2084 74206669
2088 6E642074
208C 68617420
2090 66696C65
2094 00
```

```
2095 03024469        DB 3,2,'Disk directory unreadable!',0
2099 736B2064
209D 69726563
20A1 746F7279
20A5 20756E72
20A9 65616461
20AD 626C6521
20B1 00
20B2 03FF4469        DB 3,0FFH,'Disk directory destroyed!',0
20B6 736B2064
20BA 69726563
20BE 746F7279
20C2 20646573
20C6 74726F79
20CA 65642100
20CE 03064920        DB 3,6,'I can',27H,'t read directory-no disk (
20D2 63616E27
20D6 74207265
20DA 61642064
20DE 69726563
20E2 746F7279
20E6 2D6E6F20
20EA 6469736B
20EE 206F7220
20F2 646F6F72
20F6 206F7065
20FA 6E2100
20FD 03034469        DB 3,3,'Disk directory unreadable!',0
2101 736B2064
2105 69726563
2109 746F7279
210D 20756E72
2111 65616461
2115 626C6521
2119 00
211A 01014449        DB 1,1,'DIO says: Bad parameters!',0
211E 4F207361
2122 79733A20
2126 42616420
212A 70617261
212E 6D657465
2132 72732100
2136 01064449        DB 1,6,'DIO says: No disk or door open!',0
213A 4F207361
213E 79733A20
2142 4E6F2064
2146 69736B20
214A 6F722064
214E 6F6F7220
2152 6F70656E
2156 2100
2158 01024449        DB 1,2,'DIO says: Hard error! Preamble bad!',(
215C 4F207361
2160 79733A20
2164 48617264
2168 20657272
```

```
216C 6F722120
2170 50726561
2174 6D626C65
2178 20626164
217C 2100
217E 01034449          DB 1,3,'DIO says: Checksum error!',0
2182 4F207361
2186 79733A20
218A 43686563
218E 6B73756D
2192 20657272
2196 6F722100
219A 01044449          DB 1,4,'DIO says: Verify error!',0
219E 4F207361
21A2 79733A20
21A6 56657269
21AA 66792065
21AE 72726F72
21B2 2100
21B4 01054449          DB 1,5,'DIO says: Write protected!',0
21B8 4F207361
21BC 79733A20
21C0 57726974
21C4 65207072
21C8 6F746563
21CC 74656421
21D0 00
21D1 0203446F          DB 2,3,'Dont what?',0
21D5 6E742077
21D9 6861743F
21DD 00
21DE 02024E6F          DB 2,2,'Nothing to run!',0
21E2 7468696E
21E6 6720746F
21EA 2072756E
21EE 2100
21F0 05004766          DB 5,0,'Gfid says: Bad disk identifier',0
21F4 69642073
21F8 6179733A
21FC 20426164
2200 20646973
2204 6B206964
2208 656E7469
220C 66696572
2210 00
2211 05014766          DB 5,1,'Gfid says: Name too long',0
2215 69642073
2219 6179733A
221D 204E616D
2221 6520746F
2225 6F206C6F
2229 6E6700
222C 05024766          DB 5,2,'Gfid says: Illegal extension',0
2230 69642073
2234 6179733A
2238 20496C6C
```

```
223C 6567616C
2240 20657874
2244 656E7369
2248 6F6E00
224B 05034766          DB 5,3,'Gfid says: Name null or wierd!',0
224F 69642073
2253 6179733A
2257 204E616D
225B 65206E75
225F 6C6C206F
2263 72207769
2267 65726421
226B 00
226C 02045768          DB 2,4,'What?',0
2270 61743F00
2274 06005468          DB 6,0,'That file already exists',0
2278 61742066
227C 696C6520
2280 616C7265
2284 61647920
2288 65786973
228C 747300
228F 06015468          DB 6,1,'That file does not exist',0
2293 61742066
2297 696C6520
229B 646F6573
229F 206E6F74
22A3 20657869
22A7 737400
22AA 05054920          DB 5,5,'I can',27H,'t: the disk is full',0
22AE 63616E27
22B2 743A2074
22B6 68652064
22BA 69736B20
22BE 69732066
22C2 756C6C00
22C6 05044920          DB 5,4,'I can',27H,'t: the directory is full'
22CA 63616E27
22CE 743A2074
22D2 68652064
22D6 69726563
22DA 746F7279
22DE 20697320
22E2 66756C6C
22E6 00
22E7 05064920          DB 5,6,'I can',27H,'t rename across drives: u
22EB 63616E27
22EF 74207265
22F3 6E616D65
22F7 20616372
22FB 6F737320
22FF 64726976
2303 65733A20
2307 75736520
230B 636F7079
230F 00
```

```
 2310 05074E6F          DB 5,7,'No new extension given',0
 2314 206E6577
 2318 20657874
 231C 656E7369
 2320 6F6E2067
 2324 6976656E
 2328 00
 2329 05084920          DB 5,8,'I can',27H,'t do that to a system file
 232D 63616E27
 2331 7420646F
 2335 20746861
 2339 7420746F
 233D 20612073
 2341 79737465
 2345 6D206669
 2349 6C6500
 234C 0509223C          DB 5,9,'"<?>" is not allowed here',0
 2350 3F3E2220
 2354 6973206E
 2358 6F742061
 235C 6C6C6F77
 2360 65642068
 2364 65726500
 2368 07014F75          DB 7,1,'Output file not specified',0
 236C 74707574
 2370 2066696C
 2374 65206E6F
 2378 74207370
 237C 65636966
 2380 69656400
 2384 07024F75          DB 7,2,'Output file already exists',0
 2388 74707574
 238C 2066696C
 2390 6520616C
 2394 72656164
 2398 79206578
 239C 69737473
 23A0 00
 23A1 0703496E          DB 7,3,'Input file not specified',0
 23A5 70757420
 23A9 66696C65
 23AD 206E6F74
 23B1 20737065
 23B5 63696669
 23B9 656400
 23BC 0705496E          DB 7,5,'Input file does not exist',0
 23C0 70757420
 23C4 66696C65
 23C8 20646F65
 23CC 73206E6F
 23D0 74206578
 23D4 69737400
                 ; End of stuff for now.
 23D8 FFFF              DW OFFFFH       ; insurance....
                        END
```