M T U - 1 3 0

CODOS

Channel-Oriented Disk Operating System

Release 2.0

U S E R M A N U A L

January 1982

REV. B

TABLE OF CONTENTS

APPENDICES

CHAPTER 1.

INTRODUCTION TO CODOS

CODOS (Channel-Oriented Disk Operating System) is an extremely powerful and versatile disk operating system for the MTU-130 microcomputer. It is a program which provides all the necessary functions for managing the resources of the MTU-130 computer, and includes a Console Monitor for direct control of the system from the keyboard. CODOS provides a single-user operating system with exceptional levels of performance and reliability. The system is designed from the ground up for integration with the MTU-130's Disk Controller and takes full advantage of its many engineering achievements.

CODOS is loaded from disk into memory automatically on power-up by the on-board bootstrap loader PROM (Programmable Read-Only Memory). Loading the system takes about one second, after which CODOS enables the hardware write-protect on the "system" 8K of memory in order to prevent an errant user program from inadvertently "crashing" the system by accidently writing into system memory. Once loaded into memory, the CODOS Monitor assumes control. The Monitor first reads a file of commands from a special "STARTUP" file on disk. This STARTUP file allows any desired list of programs or Monitor commands to be executed automatically without operator intervention when the system is powered-up, facilitating turnkey applications. When all the commands on the STARTUP file have been completed, the CODOS Monitor accepts additional commands from the operator in an interactive mode. These commands allow you to execute programs, store or retrieve files from disk, examine or modify various system attributes, etc. For example, to execute your inventory-management program, you might type "INVENTORY", and the file named INVENTORY would be located on the disk by CODOS, loaded into memory, and executed. This type of program is called a "User-defined command".

The CODOS Monitor provides 36 built-in commands with free-format input. These commands are quite versatile. For example, the command "TYPE MYTEXT" will display the contents of the file called MYTEXT on the Console, "TYPE MYFILE P" will print the file on a line printer, and "TYPE MYFILE YOURFILE:1" will create a duplicate copy of the file with a new name on disk drive 1. If you make a mistake, English-language error messages such as "FILE NEEDED WAS NOT FOUND" help pinpoint your errors quickly. An almost unlimited number of User-defined commands can be added to the built-in commands. For example, typing "SAVE FFT 2000 2340" creates a new User-Command called "FFT" which executes using memory locations $2000 through $2340. Once defined, this command can be executed by merely typing its name as input to the Monitor. Arguments may also be passed to user-defined Commands. For example, "FFT NEWDATA" executes the User-command FFT with the argument NEWDATA. Any kind of commands can also be read from disk files or other devices, allowing for BATCH processing. For example, typing "DO TUESDAYJOB.J" would tell CODOS to read and execute all the Monitor commands found on the file called "TUESDAYJOB.J".

The convenience of CODOS has not been gained at the expense of efficiency. Disk operations are much faster than on competetive systems. For example, CODOS can typically locate, load, and begin execution of a 32K byte program in 3 seconds. CODOS uses relatively little memory and address space because it is written entirely in optimized machine language and uses over 15 overlays. These overlays are automatically loaded into memory only when needed. This loading operation occurs so fast that it is generally imperceptible to the operator, and provides the functionality of a much larger system in less than 12K bytes.

CODOS provides true device-independent I-O over logical "channels", as found in many mainframe computers. A program can output to a printer, display, or disk file with equal ease. Since I-O channels can be assigned by a Monitor command, programs can access different devices or files without modification. Disk I-O is completely transparent to application programs, which do not need to provide buffers, "File Control Blocks", or other artifices in order to do disk I-O. A disk file can be randomly accessed at any position in the file with one disk access or less. Disk files may be as large as the remaining space on the disk, and can be increased in size at will. Unlike many other systems, no "compaction" is needed.

Both single and double-sided 8 inch disk drives may be used. CODOS will automatically use both sides of a double-sided disk, and only one side of a single-sided disk in a double-sided drive, allowing the double-sided owner complete flexibility in reading or writing both single and double-sided disks. Single and double-sided drives may be freely mixed on the same system. Up to four drives may be used, allowing up to a whopping four megabytes of formatted online storage using double-sided drives.

Interfacing for user-written Assembly-language programs is provided in the form of "Supervisor Call" Pseudo-instructions (SVCs), which simplify program development and enhance portability among different CODOS-based systems. Programs using SVCs may usually be exchanged between the MTU-130 and KIM, SYM, AIM-65, and PET computers using CODOS without modification. Many examples of SVC usage are given in sections 5 and 6, and a complete demonstration program is provided in Appendix D which illustrates CODOS's ability to perform high-speed, direct-to-disk data aquisition using interrupts.

In addition to the built-in Monitor commands, several Utility programs are provided. A FORMAT Utility initializes new disks and erases old disks. It can be used to convert any soft-sectored disk to CODOS format. If desired, the FORMAT Utility can thoroughly test disks for defective sectors. If bad sectors are found, they are bypassed automatically by the system when allocating disk space so that the disk can still be used without errors. File copying utilities are provided for both single and multiple-drive systems. The COPYF Utility can copy files individually, copy in groups, or copy an entire disk. For example, typing "COPYF STUFF" copies only the file called "STUFF" from drive 0 to drive 1, "COPYF OLD*.?" copies all file names which begin with "OLD", and "COPYF" will simply copy all files which do not already exist on the destination disk.

Extensive provisions have been made for "customizing" CODOS to your particular needs. An interactive utility program called SYSGENDEVICE is provided for adding additional devices to your system (such as a line printer). Once devices have been added to the system, they can be assigned to any CODOS I-O channels at will. A second Utility program is used for optimizing the performance of your disk drives. Tables are provided detailing the system memory map and the location of key system variables.

Be sure to fill out the system registration card that came with your MTU-130 and return it to the factory. This card assures that you will receive announcements of future CODOS upgrades, changes, and new software offerings as they become available.

IMPORTANT NOTE

If this is the first time your MTU-130 has been used please refer to the First Time Power-Up section of the Setup and Installation manual at the front of your manual binder for first time use procedures.

CODOS 2.0 SPECIFICATIONS AND SUMMARY OF FEATURES

DISK TYPE: 8 inch Shugart-compatible floppy disk, single or double-sided.

NUMBER OF DRIVES: One to four. Single and double-sided drives may be mixed.

CONTROLLER REQUIRED: MTU K-1013 Floppy Disk Controller/16-K RAM Board.

DATA RECORDING TECHNIQUE: MFM double density; IBM-compatible soft-sectors, 256 bytes per sector, 26 sectors per track, 77 tracks per side.

MAXIMUM FORMATTED CAPACITY: 4 Megabytes (4 double-sided drives).

MAXIMUM NUMBER OF FILES: 247 per disk.

MAXIMUM FILE SIZE: 500K bytes, single-sided; 1M byte, double sided disk.

FILE NAMES: 2 to 12 characters with optional 1-character extension denoting type.

NUMBER OF I-O DATA CHANNELS: 10. Up to 8 may be assigned simultaneously to active disk files. All channels are bi-directional.

USER RAM REQUIRED FOR OPEN FILE: None.

MINIMUM SYSTEM MEMORY REQUIREMENTS: Normally uses the top 16K of memory system residence, user buffers, and standard I/O drivers. Uses locations $0200-$06FF on the CPU board for system parameters. Can be reduced to 10K or less if necessary.

DISK OVERHEAD: For single-sided drives, all of track 0 and 12 plus sectors 0 through 5 of track 13 are used for the system and directory. For two-sided drives, sectors 0 though 25 of track 0 and sectors 0 through 31 of track 12 are used.

FILE SPACE ALLOCATION: Files are dynamically allocated using a unique inverted list allocation method which requires no "compaction".

FILE ORGANIZATION: Files consist of an arbitrarily large array of bytes, accessible at any position. Files may be appended at any time.

RESERVED CODES ON FILES: None.

FILE ACCESS METHODS: Sequential or true random access. Any byte in the file can be selectively accessed with one disk access.

RECORD SIZE FOR FILE ACCESS: Any size desired, 1 to 65,534 bytes, variable within the same file.

TRANSFER METHOD: Direct Memory Access (DMA).

CYCLES LOST DURING DMA TRANSFER: None.

AVERAGE CONTINUOUS THROUGHPUT RATE: 19,600 bytes per second, typical.

INTERRUPT SUPPORT: IRQ and NMI are both fully available to user at all times. Interrupts are permitted during disk access without harm.

BOOTSTRAP LOADER: 256 byte PROM.

SYSTEM MONITOR: 36 built-in commands. User Commands may be added at will.

MONITOR DIALOG: Commands may be entered directly from the Console keyboard or from any I-O device including disk files, giving a BATCH capability. STARTUP file is executed automatically on power-up.

COMMAND FORMAT: Free-format; command verb can be abbreviated. Arguments spearated by blanks and specified by position; many have defaults.

NUMERIC COMMAND ARGUMENTS: Free-format arithmetic expressions using hexadecimal or decimal values and addition, subtraction, multiplication, division, and remainder operators.

ERROR MESSAGES: 50 different error messages specified by number and English explanation. Provision for User-defined error processing.

SUPERVISOR CALL FACILITY: 31 SVCs provided for address-independent input-output and utility functions. Includes 16 bit arithmetic pseudo-processor with multiply and divide operators.

I-O METHOD: True device-independent Input-Output over data channels. Channels may be assigned to any device or file on disk.

UTILTIY PROGRAMS: Automated file copy for single and multiple-drive systems; disk initializer (FORMAT); System generation programs; File attributes list; more than 15 other programs.

EXTENDED MEMORY SUPPORT: Full support of programs and data in any of the MTU-130's four memory banks (256K byte address space).

RELIABILITY ENHANCEMENTS: Hardware Cyclic Redundancy Check (CRC); hardware write-protect for operating system nucleus and directory; multiple automatic retry with reset on read/write errors and seek errors; FORMAT utiltiy tests and bypasses any defective sectors on disk; all critical directory information is redundantly recorded; utility programs for generating backup files and backup disks; files are individually write-protectable in software; disks are individually write-protectable in hardware.

DISK RELIABILITY - ITS REALLY UP TO YOU!

Floppy disks provide an excellent low-cost storage media for programs and data, with very high reliability. When used with the high-quality MTU-130 Controller and CODOS Software, the incidence of data read-write failures should be virtually nil, provided a few simple handling precautions are observed. The way floppy disks are handled and stored will materially affect their lifetime and reliability. To insure that you receive the reliability and performance the MTU-130 system is capable of, follow the rules below religiously:

1. Always keep the diskette in its protective envelope. Get in the habit of removing a disk from the drive directly to the paper envelope. Dust particles look like a boulder to a recorded bit!

2. Do not touch the exposed recording surface of the disk. Fingerprints are a killer, too.

3. Do not bend the disk. It's called a flexible disk, but you may damage it if you try to prove it!

4. Do not write on the disk directly with pen or pencil. Use only a soft-tip marker, and write only in the label area, or you may damage the magnetic surface underneath.

5. Avoid exposure to harsh environments such as extreme heat or cold. Storage in a locked car on a hot day is a killer!

6. Keep the diskette away from strong magnetic fields such as speakers and magnetic note hangers.

7. Cigarette smoke is bad for disks as well as people.

If you follow these simple procedures, don't be surprised if you never get a read/write error. Other disk systems don't often work like that, but we're sure you won't mind doing without the disk errors!

WHAT KIND OF DISKS SHOULD BE USED?

Any quality soft-sectored 8 inch floppy disk may be used. We recommend Dysan double density disks for maximum data integrity. However, satisfactory results can usually be obtained even with diskettes rated only for single density if they are of good quality, due to the exceptionally high-quality data separator used in the MTU-130 double-density controller, and the automatic error recovery software built into CODOS. The FORMAT Utility will automatically record the proper double density timing marks on any soft-sectored disk.

Floppy disks are normally sold in boxes of 10, and can be purchased from almost any computer or business supply house. The disk supplied by MTU is called the Distribution disk (Note: your MTU-130 may have been supplied with two diskettes. The one labelled CODOS 2.0 is the Distribution disk.) Figure 1-1 illustrates the important parts of the floppy disk, which are:

1. Manufacturer's permanent label.

2. User's label and MTU Copyright notice. These blank labels are provided in several colors with the box of disks. Fill out the label before affixing it to the disk, and be sure to include the CODOS Copyright notice on all disks which are to contain a copy of the system.

3. Index hole. There is a hole in the disk surface and a hole in the jacket. While the disk rotates in its jacket inside the drive, a beam of light shines through the hole. Once each revolution the holes line up and the light passes through, triggering timing circuits in the drive. Double-sided disks have their index hole slightly further off center than single-sided disks, as shown in Figure 2-2. Soft-sector diskettes have only one index hole in the disk itself, but hard sectored disks (which cannot be used with the MTU-130) have 33.

4. Drive spindle hole. When the disk is in the drive, the drive inserts the spindle into this hole and clamps on the exposed disk surface, spinning the disk inside its jacket.

5. Head slot. This portion of the disk is exposed for access by the read/-write head.

6. Write-protect notch. This notch should be covered with the small gummed label provided in order to write on the disk and should be removed to prevent writing on the disk. WARNING: User-supplied disk drives may not "recognize" write -protect labels and can therefore write on a protected disk. Disk drives supplied by MTU will honor write-protected disks.

Disks should be inserted into the drive in the direction indicated by the arrow, with the label side towards the movable part of the door.

FIGURE 1-1: FLOPPY DISK

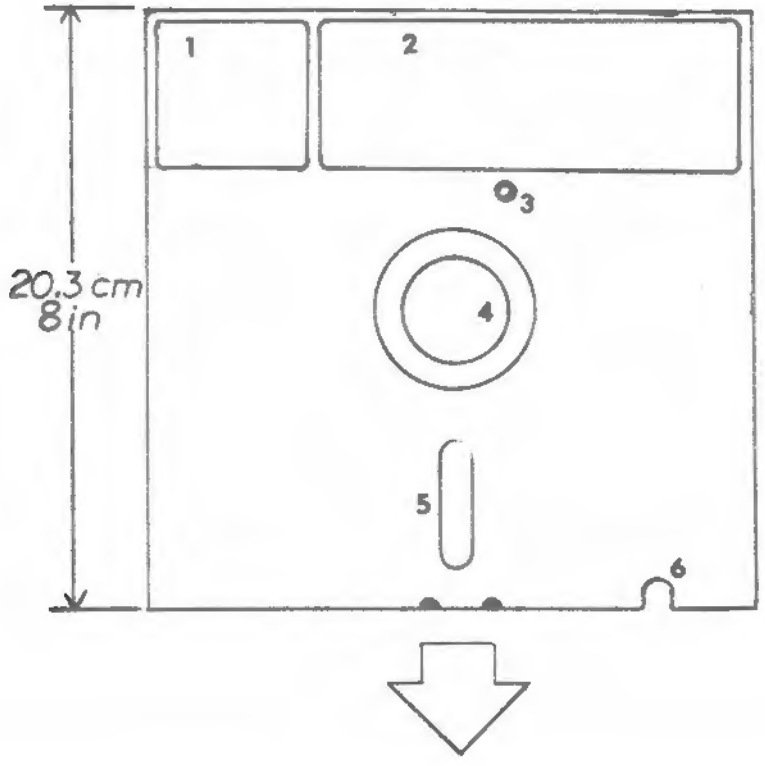
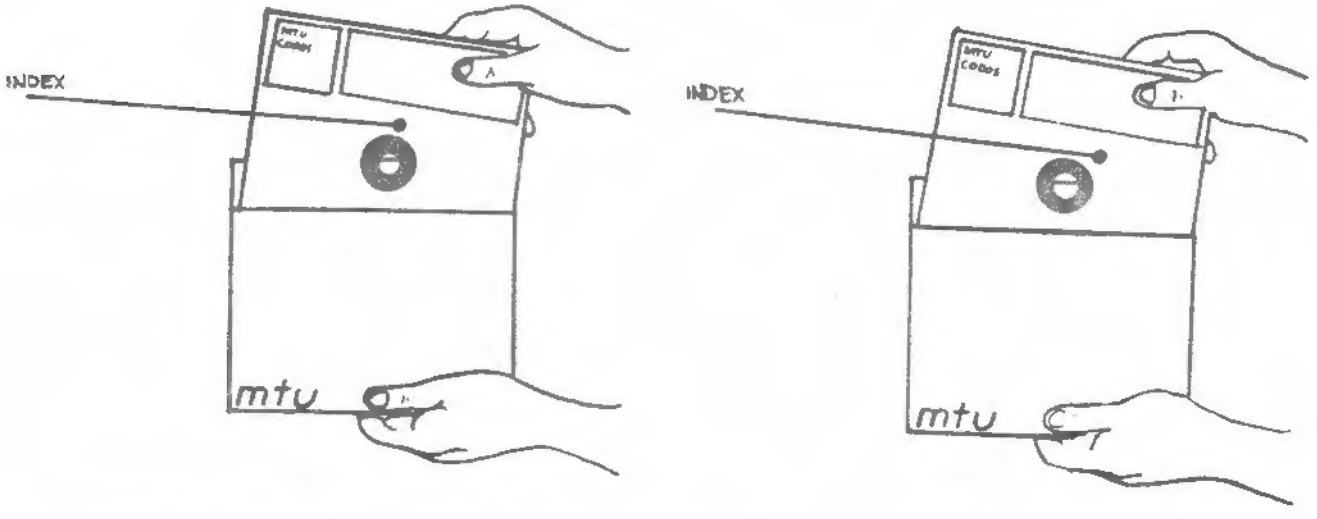


FIGURE 1-2: DISTINGUISHING DOUBLE-SIDED DISKS



ONE-SIDED DISKETTE

TWO-SIDED DISKETTE

CHAPTER 2.

CODOS SYSTEM CONCEPTS

The CODOS Operating System is a powerful computer program for managing the resources of the MTU-130 computer. In particular, it provides a convenient method for storing and retrieving programs and data on floppy disk storage. The user will normally interact with CODOS principally through three built-in facilities:

1. The CODOS System Monitor for direct operator control.
3. The CODOS SVC Processor for assembly-language programs.
3. The CODOS Interface Library (CIL) for BASIC programs.

The System Monitor provides a simple method for you to interact directly with CODOS by typing commands on the MTU-130 keyboard (hereafter called the Console). These Commands are most often used to initiate execution of other programs, examine the status of various system attributes (such as the names of files present on floppy disk), or to alter the status of the system (for example, adding a new program to floppy disk). The CODOS System Monitor is initiated automatically when the system is "booted" up. A prompting message is issued on the console display, and the system awaits your commands. These commands may be either Built-in commands, Utilities or User-defined commands. All three types of commands are described in detail later.

All users of the CODOS system will use the functions of the System Monitor to some degree. In addition, however, programmers will also want programs to interact with the operating system. For example, assembly language programmers will wish to be able to display messages on the Console and input characters from the keyboard. In most conventional microcomputer systems, support for this type of activity is provided in a limited sense by making available to the programmer a list of addresses of system subroutines which perform the basic input/output functions essential to programming. The programmer can use these functions by writing a Call (JSR) to the appropriate system subroutine from within the application program. CODOS provides a different, higher-level method of support for user-written assembly language programs called the SUPERVISOR CALL (SVC). Although not normally found on microcomputers, SVC's are used extensively on the mainframe computers. Instead of a JSR instruction to a system routine, the SVC consists of a BRK ($00) instruction followed by a byte which identifies the function desired. There are several advantages to this method, the most important of which is that SVCs are address-independent. This means that a program using SVCs will run without modification regardless of the location or version of the operating system. SVCs are discussed in detail in a later Sections 5, 6, and 7.

The MTU-130 BASIC Interpreter provides a limited interface to CODOS at all times for saving and loading BASIC programs. Use of the CODOS-BASIC Interface Library (CIL) gives the BASIC programmer full access to the many features of CODOS including sequential and random access files, file creation and deletion, and even provision for executing any CODOS command from a running BASIC program. These powerful extensions to BASIC are described in the MTU-130 BASIC Reference Manual and in the CIL manual.

CHANNELS

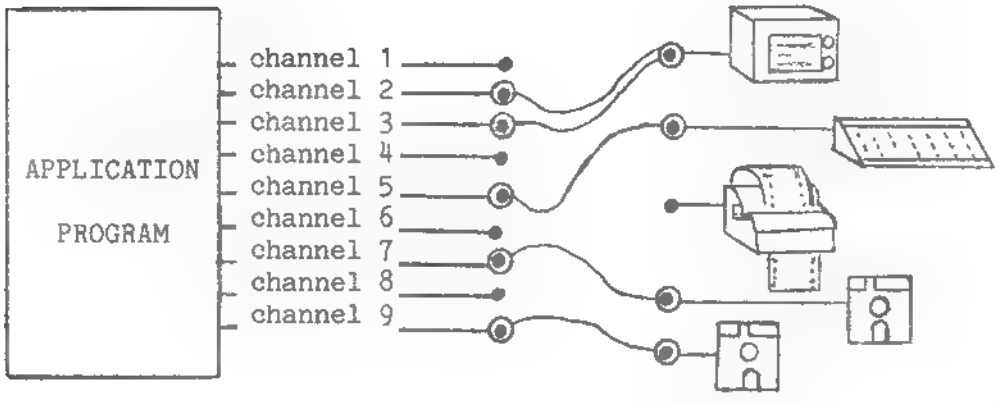
CODOS provides a capability not normally found on micros called device-independent I-O. Device-independence means that a program (or System Monitor command) can perform input or output to or from a variety of devices or disk files without modification. For example, a program which normally displays its output on the system Console device can be run with the output directed instead to a printer, without any modification whatsoever to the program. Input or output can also be re-directed to a file on disk. This feature lends great flexibility to programs. The devices to be used can be selected by a simple Monitor command, or by the executing program itself.

The key to device-independence in CODOS is the use of software I-O Channels. The System Monitor and programs communicate with the outside world over channels. At any time, these channels may be associated with a given device or file. The standard CODOS system has ten channels, numbered 0 through 9. Each of these channels may be used to send or receive data, or both. For example, a channel assigned to a a printer would be used as an output-only channel, but if it was assigned to the system Console it could both send and receive data.

Certain channels have pre-defined uses, and other channels have been given suggested standard uses in the interest of uniformity among applications. These channel definitions are given in Table 2-1.

The way in which channels are used will become clearer in following sections which introduce the CODOS Monitor Commands. The section on interfacing to user programs, Section 5, describes the use of channels from a programmer's point of view.

Figure 2-1: Conceptual View of CODOS I-O Channels



CODOS provides "software patch cords" which let you select what device or file is to be accessed without modifying the program.

TABLE 2-1: STANDARD CHANNELS

Channel 0: Reserved for internal CODOS operation.
Channel 1: Input commands to CODOS Monitor.
Channel 2: Output from CODOS Monitor.
Channel 3: Available. (Input preferable).
Channel 4: Available. (Input preferable).
Channel 5: Standard input for programs.
Channel 6: Standard output for programs.
Channel 7: Available.
Channel 8: Available. (Output preferable).
Channel 9: Available. (Output preferable).

NOTES FOR TABLE 2-1:

1. Channel 1 and Channel 2 are normally assigned to the console by default.

2. The notation "input preferable" or "output preferable" simply means that if it is convenient to do so, input should be assigned to the lower numbered channels and output to the higher channels. This is merely a convention and is not enforced in any way. All channels can be used in either direction or bi-directionally.

DEVICES

As we have already seen, CODOS communicates with the outside world over numbered channels. These channels can be associated with either physical devices or with files. The devices available on any given system are defined during system generation by using the SYSGENDEVICE Utility, and are identified by a single letter. Every system has at least two devices: the system Console and the Null device. The system Console is the MTU-130 keyboard (for input) and the MTU-130 CRT display (for output), and is given the device name "C".

The null device is given the name "N" and is predefined to mean a device that does nothing. This may seem of dubious merit, but is actually very useful. For example, if you wish to run a program which normally generates diagnostic messages on channel 9, you can suppress the diagnostics by merely assigning channel 9 to the null device.

Additional devices, such as a printer, may be available on any given system, and may be named as desired during system generation. In the interest of uniformity among systems, the recommended device names are given in Table 2-2 for selected devices.

Remember that all devices have a single letter name. The use of device names will be illustrated shortly in the section describing Monitor commands.

TABLE 2-2: DEVICES

Device Name	Description
C	Console. Input-output terminal device. (Required)
N	Null device. (Required)
P	Printer.
R	Paper tape reader and/or punch.
T	Terminal (e.g., a Teletype)
M	Memory.

NOTES FOR TABLE 2-2:

1. Other devices may be named as desired during system generation, using a single letter for each. See Chapter 10 and the SYSGENDEVICE Utility program described in Chapter 4.

FILES

Programs, text, and data of any type can be stored and retrieved from floppy disk for permanent storage using CODOS. A File is a collection of related information stored as a logical entity on disk. Each file on disk has a unique name, designated by the creator of the file. The name consists of from two to twelve characters, optionally followed by a "." and a one-character file extension. The first character of the name must be alphabetic. The remaining characters may be alphabetic, numeric, or the special character "_" (underline), which is used to improve readability of composite names and to help search for related files using "wildcards", as will be discussed in Chapter 4. The single-character file extension may be alphabetic or numeric. If the optional file extension is omitted, a default file extension of ".C" is assumed by the system. Thus some examples of legal file names include:

```
A2
YANK
MY3RDFILE.A
HIS_STUFF.T
OLD_X_Y_DATA.8
```

The first two file names above will have a default extension of ".C" appended by the system. The single character file extension is intended to provide the user with an indication of the kind of file. Altnough CODOS does not enforce any particular convention, Table 2-3 lists the standard file extensions which are strongly suggested for use. Unlisted extensions may be freely used to cover special kinds of files not included in the list. Note that the extension must be exactly one character long if given. Also remember that all file names must have at least two characters. This enables CODOS to distinguish between device names (which always have one character) and file names.

TABLE 2-3: FILE EXTENSIONS

Extension	Meaning
.A	Assembly language source program.
.B	BASIC Program (tokenized memory image format)
.C	Command (User-defined command programs and System Utility Programs).
.D	Data.
.E	BASIC program (ASCII format, use BASIC ENTER command to load).
.G	Graphic data or Display memory image.
.H	Hex file (i.e., paper-tape-type format).
.J	Job file (i.e., a text file of CODOS Monitor commands)
.L	Listing.
.T	Text.
.X	Executable code other than a command (e.g., subroutine package).
.Z	CODOS reserved system file.

NOTES FOR TABLE 2-3:

1. If the extension is not given, ".C" will be assumed. While running BASIC, the default file extension is ".B".

2. Other extensions may be devised by the user as needed.

3. The extensions given are recommended but not required. Any kind of file can have any kind of extension, so long as it is one alphanumeric character.

CODOS SYSTEM MONITOR

The CODOS Monitor is an interactive program which allows the user to enter commands to the system. The Monitor is part of CODOS and is entered automatically during startup of the system. When the system is "booted" up, the CODOS memory image is loaded into memory from the disk in drive 0. A special file called STARTUP.J is then read by the Monitor and all commands on that file are executed. At the completion of the startup procedure, a prompting message will be issued indicating the version of CODOS which is active, and the prompt, "CODOS) " will appear. At this time, a valid command can be entered from the Console keyboard.

Every command typed must be terminated by a carriage return, which signals the Monitor to execute the command. Certain characters may be used for correcting typing errors or editing the command line during entry; these are summarized in Table 2-4. In particular note that you can use the BACKSPACE, RUBOUT, cursor left and cursor right keys to make corrections to your commands. If the line has been edited, the cursor may be left in the middle of the line when the carriage return is entered and CODOS will see everything that you see on the displayed command line. You can also add a comment to your command line if you wish. Any characters after the ";" character will be ignored by CODOS. To use editing characters in Table 2-4 which start with CTRL, you must hold down the CTRL key and then depress the other character indicated.

There are two main types of commands in CODOS: User-defined Commands and Built-in Commands. Built-in commands are pre-defined by the system. User commands may be added easily at will by writing an assembly-language program and defining it as a Command using the built-in SAVE command. In the following discussion, only built-in commands will be discussed, so the term "command" will be understood to mean "built-in command".

In order to improve readability and ease the learning process, CODOS commands ususally consist of full English words which suggest the function to be performed. However, any built-in command (not user command) can be abbreviated using the "!" character. Thus, for example,

```
ASSIGN
ASSI!
AS!
```

are all equivalents for the ASSIGN command. It is only necessary to type enough characters before the "!" to uniquely identify the command desired. All built-in commands must be spelled using uppercase letters.

Most commands require one or more arguments following the command keyword. These arguments tell the system what entities the command is to operate on. For example, the command,

```
ASSIGN 6 MYFILE.T
```

has two arguments. The first argument in this case is a channel number, and the second argument is a file name. The command tells CODOS to associate channel 6 with the file called MYFILE.T.

Arguments must be separated from the command keyword and from each other by one or more BLANKS (not commas!). A few commands use other special delimiters such as "=" or ":" in certain places in the command; these will be clearly defined.

Sometimes arguments are optional, in which case the user may elect to specify the argument or else accept the default argument which will be assumed by the system. In other cases, the user has a choice of several different kinds of arguments. If you type in more arguments than CODOS expects, the extra arguments are treated as comments and ignored. Often an arbitrary number of arguments may be given. In order for this manual to have a uniform method for describing the syntax of various commands and arguments, the following notation is adopted:

1. Angle brackets, "<" and ">", are used to enclose words describing the kind of entry required.

2. Square brackets, "[" and "]", are used to enclose optional arguments or symbols, which may be included or omitted as desired.

3. Ellipsis, "...", are used to indicate an arbitrary number of repetitions of the previous argument(s).

4. Symbols not enclosed in angle brackets are literal symbols which must be typed exactly as shown.

5. Curly brackets, "{" and "}", are used to enclose each of several mutually-exclusive choices, only one of which may be selected.

For example, we could use this meta-language (a meta-language is a language used to describe another language) to describe several BASIC statements as follows:

GOTO <line #>
FOR <variable> = <value> TO <value>[STEP <value>]

In the following section, each of the Built-in commands will be defined and illustrated. Some of the commands require numeric values for arguments. In this case, either decimal or hexadecimal values may be used. Unless otherwise indicated, all numeric arguments are assumed to be in hexadecimal. To specify a decimal argument, use the prefix ".". If desired, the "$" prefix can be used to clarify hex values. An arithmetic expression can be used anywhere a numeric value is called for, except for disk drive numbers. Arithmetic expressions may be formed using the usual operators, "+", "-", "*", "/" and "\". "\" is the remainder operator. All expressions are evaluated left-to-right without any operator precedence. The value entered may not exceed 65535 decimal or be less than -32768 decimal (including any intermediate point in the computation). The following examples illustrate the evaluation of numeric expressions:

```
100   evaluates as 256 decimal (100 hex).
.100   evaluates as 100 decimal (64 hex).
B+ 10   evaluates as 27 decimal (1B hex). (Blanks are ignored in expressions)
1+.10*3   evaluates as 33 decimal (21 hex).
$1498/.256   evaluates as 20 decimal (14 hex).
40BC \ 100+1   evaluates as 177 decimal (BD hex).
```

Arguments which specify memory addresses may also specify a memory bank by appending a :<bank> at the end of the argument where <bank> is the digit 0, 1, 2, or 3. If a bank is not specified, memory bank 0 is assumed. Where two arguments specify a range of memory addresses, only the first argument should specify a bank since the end of the range is assumed to be in the same bank as the beginning. The following illustrates the use of bank notation:

```
C000:1   Specifies address $C000 in bank 1 (the first display memory location).
0:2 3FFF Specifies a range of addresses from $0000 through $3FFF in bank 2
         (applicable to commands that use address ranges).
```

TABLE 2-4: COMMAND EDITING CHARACTERS

Character	Meaning
BACKSPACE	Backspace 1 character
←	Backspace 1 character (see Note 3).
RUBOUT	Backspace 1 character then erase character at new cursor position.
→	Forward space 1 character without erasing (see Note 3).
blank	Erase character under cursor then forward space 1 character
CTRL-X	Delete entire line (start line over).
;	Comment. Any characters after ";" are ignored.
!	Command abbreviation character. See text.
RETURN	End-of-command.
HOME	Place cursor in upper left screen corner.
INSERT	Enter insert mode. Characters will be inserted before the character the cursor is on, pushing remaining text over to the right. Any editing character except rubout clears insert mode.
DELETE	Delete character under cursor and "close-up" remaing part of line.
SHIFT-HOME	Erase screen and place cursor in upper left screen corner.
or	Does not discard line (even though erased).
CTRL-L	
CTRL-S	Temporarily suspend output display (see Note 1).
CTRL-Q	Resume suspended output display (see Note 1).
CTRL-C	Command abort (during display) (see Note 1).
CTRL-Z	End-of-File (for keyboard entry only).
CTRL-R	Re-display entire present line starting at cursor position.
CTRL-W	Delete from present cursor position to end-of-line.
CTRL-E	Turn off/on echo of keyboard characters to CRT (see Note 4).
CTRL-B	Recall a previously typed line (see note 2).
SHIFT- →	Jump cursor to last character of existing line.
SHIFT- ←	Jump cursor to first character of existing line.

NOTES FOR TABLE 2-4:

1. CTRL-S and CTRL-C are active only while a command or program is actually printing text on the console.

2. Each CTRL-B recalls one line, beginning with the most recently entered line. Recalled lines may then be edited using other keys in this table. The number of lines which can be recalled depends on the length of the lines; about 8 to 15 average lines are usually retained. As new lines are entered, the oldest lines are discarded. After the last recallable line has been displayed, the next CTRL-B will "wrap around" to display the most recent line again. In this way if you "overshoot" the line you wanted, you can just continue pressing CTRL-B till it comes around again. You will find CTRL-B very useful!

3. Cursor-left and cursor-right are normally confined to the limits of the existing line. You can "escape" from the current line position by using cursor-up or cusor-down when the cursor is at end-of-line. However, once you escape from the present line, you cannot backup to change it unless you first do a CTRL-R.

4. CTRL-E is useful for entering commands when you do not want to "clutter" the screen. After CTRL-E, keys pressed do not show up on the screen. Depressing CTRL-E a second time will re-enable echo. You can completely eliminate normal CODOS screen activity and still enter commands by assigning channel 2 to N (the nul' device) and using CTRL-E to eliminate keyboard echo.

TABLE 2-5: BUILT-IN COMMANDS

Command	Page	Purpose
ASSIGN	3-1	Display or alter channel assignments for I-O.
BEGINOF	3-2	Position channel to beginning-of-data.
BOOT	3-3	Boot-up CODOS (cold start).
BP	3-3	Define breakpoint address for machine language debugging.
CLOSE	3-4	Close-out operations on disk specified.
COMPARE	3-5	Compare two blocks of memory.
COPY	3-5	Copy memory block.
DATE	3-6	Set date.
DELETE	3-7	Delete file from disk directory.
DISK	3-7	Display attributes of disks.
DO	3-8	Execute a list of Monitor commands on a"batch" job file.
DRIVE	3-9	Designate the default drive.
DUMP	3-9	Display contents of memory.
ENDOF	3-10	Position channel to end-of-file.
FILES	3-11	List names of files on disk.
FILL	3-12	Fill block of memory with a constant.
FREE	3-12	Release channel if assigned.
GET	3-13	Load program into memory from disk.
GETLOC	3-14	Display load addresses of loadable file.
GO	3-16	Begin execution of program in memory.
HUNT	3-17	Search for string of bytes in memory.
LOCK	3-18	Enable write-protect on disk file.
MSG	3-19	Print a message on a device or file.
NEXT	3-19	Resume execution of suspended program in memory.
ONKEY	3-20	Define function key legend and action to be taken.
OPEN	3-21	Open-up operations on a disk.
PROTECT	3-22	Enable hardware write-protect on system memory.
REG	3-23	Display or alter contents of 6502 registers.
RENAME	3-24	Change the name of a file.
RESAVE	3-25	Same as SAVE command except updates existing file.
SAVE	3-26	Save program, command, or memory image on a file.
SET	3-27	Set memory to value(s).
SVC	3-28	Enable or disable Supervisor Call Processor (SVCs).
TYPE	3-28	Display or print contents of file.
UNLOCK	3-30	Disable write-protect on file.
UNPROTECT	3-30	Disable hardware write-protect on system memory.

NOTES FOR TABLE 2-5:

1. The underlined portion indicates the minimum allowable abbreviation for the command (using "!").

2. Utility programs, which are very much like built-in commands, are discussed in Chapter 4.

3. See Chapter 3 and Appendix G for detailed descriptions of the built-in commands.

CHAPTER 3.

BUILT-IN COMMANDS

See Appendix G for Syntax diagrams for all built-in CODOS commands.

COMMAND NAME: ASSIGN. A!

PURPOSE: To assign an input-output channel to a file or device, or to display all current channel assignments.

SYNTAX: ASSIGN [<channel> {<device> / <file> [:<drive>]} ...]

ARGUMENTS:

<channel> = desired channel number, 0 to 9.
<device> = single character device name.
<file> = file name desired.
<drive> = disk drive number, 0 to 3. Defaults to current default drive, usually 0.

EXAMPLES:

```
ASSIGN
```

displays the current channel assignments. A typical display might be:

```
CHAN. 1 C
CHAN. 2 C
CHAN. 6 MYTEXT.T:0
```

which indicates that channel 1 and 2 are assigned to the Console, and channel 6 is assigned to a file called MYTEXT.T on drive 0.

```
ASSIGN 6 C  ; OUTPUT TO CONSOLE PLEASE.
```

assigns channel 6 to the system console device. Everything after the ";" character is a comment.

```
ASSIGN 5 MYTEXT.T
```

assigns channel 5 to the disk file called MYTEXT.T on the default drive (usually drive 0). The system responds to file assignments with either "NEW FILE" or "OLD FILE" depending on whether or not the given file already exists. If you get "NEW FILE" when you were expecting "OLD FILE", it probably means you misspelled the file name. You can correct this by merely doing the assignment over, since assigning a channel which is already assigned automatically frees the old assignment first.

CAUTION: CHANNELS 0, 1 AND 2 ARE USED INTERNALLY BY THE SYSTEM AND SHOULD NOT BE REASSIGNED UNTIL YOU HAVE A THOROUGH UNDERSTANDING OF THE SYSTEM OPERATION!

```
ASSIGN 4 C  7 YOURS.A : 1
```

assigns channel 4 to the Console and assigns channel 7 to the file called YOURS.A on drive 1. If YOURS.A does not exist, it will be created automatically and will initially contain nothing. Files which contain nothing disappear automatically when they are FREEd from their channel assignments.

NOTES:

1. Assigning a channel to a file always positions the file to beginning of data, even if the file is already assigned to another channel and is not at beginning of data.

2. More than one channel can be assigned to the same file or device.

3. The CODOS Monitor reads its input from channel 1 and outputs to channel 2. These channels are both normally assigned to the Console. You can, however, reassign these channels. If you have a sequence of Monitor commands that you execute often, you can TYPE these commands onto a file, and then ASSIGN channel 1 to the file. CODOS will execute every command on the file and then automatically reassign the console when End-of-File is encountered. The Console is also automatically assigned if an error is detected. This kind of file is called a "Job file" and has the extension ".J". The file called STARTUP.J is a special job file which is assigned to channel 1 by the system when CODOS is booted up. Chapter 10 discusses STARTUP.J Jobs in some detail. The "DO" command also assigns channel 1 to a job file.

COMMAND NAME: BEGINOF. B!

PURPOSE: To position a file associated with a given channel to beginning-of-data.

SYNTAX: BEGINOF ⟨channel⟩ ...

ARGUMENTS:

⟨channel⟩ = desired channel number, previously assigned to a file.

EXAMPLES:

BEGINOF 5

positions the file presently assigned to channel 5 to beginning of data.

BEG! 7 8 9

repositions the files assigned to channels 7, 8, and 9 to beginning of data. You will recall that the "!" character can be used to abbreviate any built-in command.

NOTES:

1. It is permissible to use the BEGINOF command on a channel which is assigned to a device instead of a file. In this case, the command is ignored.

COMMAND NAME: BOOT. *BO!*

PURPOSE: To re-boot the CODOS operating sytem from the disk in drive 0.

SYNTAX: BOOT

ARGUMENTS: None.

EXAMPLES:

```
BOOT
```

will cause CODOS to be reloaded and re-initialized from the disk in drive 0.

NOTES:

1. The BOOT command performs a jump to the ROM bootstrap loader. It does not close any disks or perform any other action before doing so.

COMMAND NAME: BP.

PURPOSE: To set a program breakpoint for debugging purposes.

SYNTAX: BP [<addr>]

ARGUMENTS: <addr> = address of first byte of instruction at which breakpoint is desired.

EXAMPLES:

```
BP 420A
```

will set a breakpoint at address 420A. Following a GO or NEXT command, when program execution reaches 420A, control will be returned to CODOS which clears the breakpoint and then prints the contents of all registers (see REG command description for the format of the register printout). Up to 3 breakpoints may be set simultaneously.

```
BP
```

clears all breakpoints.

NOTES:

1. BP works by temporarily replacing the op-code at the indicated location with a BRK instruction. The original op-code is replaced when the breakpoint is cleared.

2. Breakpoints may be placed anywhere, even at BRK instructions or Supervisor Calls (SVCs). They should only be placed at the op-code byte of an instruction.

3. Breakpoints may be set in any memory bank by following the address with a ":" and the bank number.

4. Breapoints should not be set in system memory or in I/O drivers.

5. The register printout is preceeded by the keyword "BP" to idicate that entry into CODOS was through a breakpoint.

COMMAND NAME: CLOSE. C!

PURPOSE: To conclude operations on a disk in preparation for removing it from the drive or powering-down the system.

SYNTAX: CLOSE [<drive> ...]

ARGUMENTS:

<drive> = desired disk drive number, 0 to 3. Defaults to drive 0.

EXAMPLES:

CLOSE

closes drive 0. The default for the close command is always drive 0.

CLOSE 0 1

closes drives 0 and 1. The disks may then be removed.

CAUTION: YOU SHOULD ALWAYS CLOSE EVERY DISK BEFORE REMOVING THE DISK FROM THE DRIVE OR POWERING DOWN.

While CODOS is running, it maintains certain tables and buffers in memory which may need to be copied back to the disk before the disk is removed. CLOSEing the disk assures that this operation is done. This updating is needed only when writing to disk, not when just reading it. Normally, if you forget to enter the CLOSE command before removing a disk, it will not matter, since all system programs update the disk automatically when they terminate. However, if a program which wrote to a disk file was aborted, terminated abnormally, or did not FREE the channel assigned to the file, then the file on disk may not be complete unless the disk is CLOSEd before removing the disk. Therefore it is a good practice to always CLOSE a disk before changing disks or powering down. Using reset to interrupt CODOS during disk operations is not recommended since it may leave the system in an undefined state. Programmers should note that it is considered good practice for programs to FREE channels assigned to files before terminating, so that successful operation will not depend on the user remembering to CLOSE the disk.

NOTES:

1. It is permissable to close a file which is already closed. In this case, no action takes place.

COMMAND NAME: COMPARE. com!

PURPOSE: To determine if two blocks of memory are identical.

SYNTAX: COMPARE <from> <to> <dest.>

ARGUMENTS:

<from> = starting address for first block.
<to> = ending address for first block.
<dest.> = starting address for second block.

EXAMPLE:

COMPARE 2000 2FFF 4000

will compare every byte of the block of memory from $2000 to $2FFF to the corresponding bytes in the block from $4000 to $4FFF. If the blocks are identical, CODOS will display:

SAME.

If the blocks differ, the address and content of the first byte which differs will be displayed and the comparison will terminate. For the example command above, a possible result might be:

2006=30, 4006=90

which indicates that the first 6 bytes of the blocks match, but the seventh bytes differ as shown.

NOTES:

1. Only the first differing byte is displayed.

2. The values displayed are in hex.

3. The comparison may be between different memory banks. The default bank for the <dest.> argument is the same bank as was specified for <from>.

COMMAND NAME: COPY. co!

PURPOSE: To copy a block of memory to another memory location.

SYNTAX: COPY <from> <to> <dest.>

ARGUMENTS:

<from> = starting address of block to be copied.
<to> = ending address of block to be copied.
<dest.> = desired starting address of destination of copy.

EXAMPLES:

COPY 100 2FF 2000

copies $0100 through $02FF to $2000 (through $21FF).

COPY 2000 2000+.80 2002

copies $2000 through $2050 to $2002 (through $2052).

NOTES:

1. The block may be any size.

2. The destination for the copy can overlap the block being copied. This fact can be used to advantage to "open up" or "close up" space in memory.

3. Copying can be performed in either direction (higher address to lower address or lower address to higher address).

4. The content of one memory bank may be copied into a different memory bank. The default bank number for the dest. argument is the same as was specified for the from argument.

5. CAUTION: Unlike the SET or FILL commands, COPY does not check for reserved memory violations, nor does it verify the bytes as they are deposited. The reason for this is explained in the description of the SET command.

COMMAND NAME: DATE. OA!

PURPOSE: To set the creation date for any new files generated.

SYNTAX: DATE [<dd-mmm-yy>]

ARGUMENTS:

<dd-mmm-yy> = desired date.

EXAMPLE:

DATE 08-AUG-80

sets the date field to "08-AUG-80". Any files created thereafter before powering down the system, re-booting, or issuing another DATE command, will be dated accordingly. The date field for files is displayed by the DIR Utility.

NOTES:

1. The first 9 characters (after any leading blanks) are used for the date. No format checking is provided, so you may freely use other forms such as "1/24/81" if you wish.

2. The date is assigned to a file at its initial creation. It is not altered by any changes to the file, including writing, truncating, or renaming it. However, since COPYF and TYPE (with a file name for a second argument) actually create a new file, these new files will have the current date, not the original. Therefore you can effectively change the date on any file by using the date command, copying the file and deleting the original.

3. The DIR Utiltiy can be used to ascertain the creation date of a file.

4. When CODOS is booted up, it will prompt for the initial date entry by the user. If the user replies with a carriage Return, the default date field, "*UNDATED*" will be used.

COMMAND NAME: DELETE. DE!

PURPOSE: To remove a file from the disk.

SYNTAX: DELETE <file>[:<drive>] ...

ARGUMENTS:

<file> = file name to be removed.
<drive>= disk drive number, 0 to 3. Defaults to the current default drive, usually 0.

****

CAUTION: USE THE DELETE COMMAND WITH CARE; THERE IS NO PROMPT FOR A "VETO" BEFORE THE FILE IS REMOVED, SO TYPE CAREFULLY! ALL IMPORTANT FILES SHOULD BE LOCKED IMMEDIATELY AFTER THEIR CREATION TO PREVENT INADVERTANT DELETION BY AN ERRONEOUS DELETE COMMAND! FOR GENERAL FILE DELETION, YOU SHOULD CONSIDER USING THE "KILL" UTILITY PROGRAM DESCRIBED IN SECTION 4 INSTEAD, WHICH REQUIRES VERIFICATION OF EACH FILE TO BE DELETED.

****

EXAMPLES:

```
DELETE MYDATA
```

deletes the file MYDATA.C from the default disk (usually drive 0).

```
DELETE PROG_1A:1   Y3  HIS_STUFF.T
```

deletes three files, one from drive 1 and two from drive 0.

NOTES:

1. It is recommended that the KILL Utility be used in lieu of the DELETE command in an interactive environment. The DELETE command is more convenient to use in a batch job, however, and does not use the memory area reserved for utilities ($B400-BDFF) to operate.

2. Once a file is deleted, it cannot be recovered.

3. Backup copies of important files should always be maintained on any disk system.

COMMAND NAME: DISK. DI!

PURPOSE: To display the number of files, remaining space, and volume serial number on all open disk drives.

SYNTAX: DISK

ARGUMENTS: none.

EXAMPLE:

```
DISK
```

will display the number of files, volume serial number, and free space on all open drives. A typical display might be:

```
 11 FILES:0 (VSN=2001), 890K FREE
108 FILES:1 (VSN=0003), 72K FREE
```

which indicates that drives 0 and 1 are open, with 11 files on drive 0 and 108 files on drive 1. There is about 890K free (1K = 1024 bytes; therefore about 911,360 bytes remain available) on drive 0, and the serial number specified when that diskette was last formatted is 2001. The number of "K" free is a decimal number.

NOTES:

1. Disk space is allocated and displayed in blocks of 2K bytes on single-sided drives and 4K bytes on double-sided drives.

COMMAND NAME: DO.

PURPOSE: To execute a list of CODOS Monitor commands stored on a "Job" file.

SYNTAX: DO <file> :<drive>

ARGUMENTS:

<file> = desired text file of commands to be executed by CODOS.
<drive> = optional disk drive number, 0 to 3, which defaults to the current default drive, normally drive 0.

EXAMPLES:

```
DO MAILINGLIST.J
```

will cause CODOS to read and execute every line of the file MAILINGLIST.J on drive 0 as a Monitor command. When end-of-file is reached, CODOS will resume reading input from the Console.

NOTES:

1. If an error is detected by CODOS while executing a command from the "job" file, an error message will be issued in the ususal manner showing the offending command from the file, and CODOS will accept input from the Console. The remainder of the commands on the job file will be ignored.

2. Normally the Editor is used to create a new Job file (see the MTU Screen Editor manual for details). Suppose you used "EDIT ODDJOB.J:1" to create the following file:

```
GET ODDSUBS  ;LOAD MY ORIGINAL ODD SUBROUTINES
SET 2245=4C 68 2B ; MAKE PATCH FOR STRANGE STUFF
ASSIGN 7 OLDDATA.D  ;DATA FILE NEEDED DURING PROCESSING
ASSIGN 8 P ;PRINTER
ODDERPROG ;EXECUTE MAIN PROGRAM
FREE 7 ;DONT NEED DATA FILE ANYMORE
```

It is considered good practice to place comments on your command lines, so you will remember what the job does when you later look at the file. After exiting from the Editor,

DO ODDJOB.J:1

will cause CODOS to execute the six commands stored on ODDJOB. This is called "batch" execution of a "Job" file.

3. The DO command produces the same effect as assigning channel 1 to the "job" file.

4. DO commands cannot be nested. You may place a DO command in a command file, and control will transfer to the indicated file. However, when End-of-File is reached on the new file, control will revert to the Console and will not resume with the next command after the DO on the first file.

COMMAND NAME: DRIVE. DR!

PURPOSE: To designate the default disk drive number to be used when files are referenced without a drive number being explicitly given.

SYNTAX: DRIVE <drive>

ARGUMENTS:

<drive> = desired drive number, 0 to 3.

EXAMPLE:

DRIVE 1

sets the default drive to drive 1.

NOTES:

1. The default drive is 0 when the system is booted up or Reset.

2. The DRIVE command only affects the drive for file name references. It does not affect the default drive for OPEN, CLOSE, FILES, etc.

3. The drive number selected using the DRIVE command is referred to as the "default drive".

COMMAND NAME: DUMP. D!

PURPOSE: To display the contents of a block of memory in hexadecimal and as ASCII characters.

SYNTAX: DUMP <from> [<to>[|<device>|<channel>|]]

ARGUMENTS:

⟨from⟩ = desired starting address.
⟨to⟩ = desired ending address (see note 1 below). Default is from +15.
⟨device⟩ = desired device on which to display the output. Defaults to the console.
⟨channel⟩ = desired channel on which to display the output.

EXAMPLES:

DUMP 1000

displays 16 bytes of memory starting at $1000.

DUMP 1000 101A

displays memory starting at $1000 and will include memory through $101A. The resulting display might look similar to:

```
      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
1000  34 77 D7 4B 20 00 56 78  4F 4B 20 46 45 4C 4C 41  4w.H..Vx OK.FELLA
1010  55 67 09 42 59 45 00 03  03 03 20 00 00 00 10 20  Ug.BYE.. ........
```

Of course, the actual values displayed will depend on the contents of memory. The sixteen rightmost characters of each line are the ASCII characters for the line, with each non-displayable character converted to ".", including blanks.

DUMP 1000 1000+.500 P

dumps 500 (decimal) bytes starting at $1000. The display will be output to the the printer.

NOTES:

1. A complete line is always displayed even if the to address is not an even multiple of 16 bytes. Sufficient complete lines will be displayed to ensure that the to address is included in the display.

2. As with any command, CTRL-S can be used to temporarily suspend the console display and CTRL-Q to restart it. CTRL-C can be used to abort the DUMP.

3. The righthand portion of each line of the dump displays "." in place of each non-printable character, including blanks. Characters considered printable are any of the 96 printable ASCII characters except blank, provided bit 7 is 0.

4. If desired, the number of bytes displayed per line can be altered to accommodate narrower or wider devices. See Appendix E.

5. Each memory location dumped is actually read twice so be careful when dumping I/O register contents that may be affected by the very act of reading.

COMMAND NAME: ENDOF.

PURPOSE: To position a file associated with a given channel to End-of-File.

SYNTAX: ENDOF ⟨channel⟩ ...

ARGUMENTS:

⟨channel⟩ = desired channel to position.

EXAMPLES:

```
ENDOF 5
```

positions the file assigned to channel 5 to End-of-File.

```
END! 6   4
```

positions the files assigned to channel 6 and channel 4 to End-of-File.

NOTES:

1. If the channel specified is assigned to a device and not a file, the command is ignored.

2. The ENDOF command can be used (with caution) to concatenate files or extend files. See the TYPE command for details.

3. Don't forget that ASSIGN always re-positions a file to beginning of data; therefore assigning another channel to the file after using ENDOF will negate the effect of the ENDOF command.

COMMAND NAME: FILES. FI!

PURPOSE: To display the name of every file on a disk.

SYNTAX: FILES [<drive>]..

ARGUMENTS:

<drive> = selected disk drive number, 0 to 3. Default is always drive 0.

EXAMPLES:

```
FILES
```

displays the names of all the files on drive 0, five names per line.

```
FILES 1
```

displays the names of all the files on drive 1.

NOTES:

1. The DIR utility program can be used to display more information about selected files. See Chapter 4.

2. As with any command, CTRL-S can be used to temporarily suspend the Console display and CTRL/Q to restart it. CTRL-C can be used to abort the command.

3. Built-in Monitor commands are not listed by the FILES command, because they are a part of the operating system, CODOS.Z.

COMMAND NAME: FILL.

PURPOSE: To fill a block of memory with a constant.

SYNTAX: FILL <from> <to> [=] { "<character>" | <value> | '<character>' }

ARGUMENTS:

<from> = desired starting address to be filled.
<to> = desired ending address for fill operation.
<value> = numeric constant to be deposited into each byte of the memory block.
<character> = single ASCII character to be deposited into each byte of the memory block.

EXAMPLES:

```
FILL 1200 12FF 0
```

fills every byte between $1200 and $12FF inclusive with $00.

```
FILL C000:1 FBFF 73
```

fills the entire MTU-130 display memory with $73 bytes (displays vertical bars).

```
FILL 1000 1000+.100 '"'
```

fills $1000 though $1064 with $22 (an ASCII ").

NOTES:

1. As each byte is deposited in memory, the result is verified by the system. An attempt to fill ROM, reserved-memory, defective memory, or non-existent memory will abort the command at the point where the error occurred.

2. The FILL command may be used to fill memory locations reserved for CODOS if an UNPROTECT command has been issued. Indiscriminant FILLing can lead to system crashes.

3. Either single or double quote marks may be used to delimit the character, but must be the same on both sides.

COMMAND NAME: FREE. F!

PURPOSE: To disassociate an Input-Output channel from a device or file.

SYNTAX: FREE <channel> ...

ARGUMENTS...

<channel> = desired channel number to free, 0 to 9.

EXAMPLES:

FREE 6

frees channel 6 from its prior assignment.

FREE 8 4

frees both channel 8 and channel 4.

NOTES:

1. It is permissable to free an unassigned channel.

2. FREE is the inverse operation of ASSIGN.

COMMAND NAME: GET. G!

PURPOSE: To load a memory image from a disk file.

SYNTAX: GET <file> [: <drive>] [= <dest.> ..]

ARGUMENTS:

<file> = desired file name to be loaded into memory. See note 1 below.
<drive> = drive number, 0 to 3. Defaults to the default drive, normally 0.
<dest.> = destination starting address for load to be used in lieu of the from address which was specified when the file was saved.

EXAMPLES:

GET MYPROG

Loads the file called MYPROG into memory. It will be loaded at the address which was specified at the time the file was created using the SAVE command. The Program Counter will be set to the entry point address which was saved with the file.

GET OLD_PROG.X:1=700 =1B00

Will load the file OLD_PROC.X from drive 1 into memory. The first block (which is whatever size was SAVEd on the file) will be loaded starting at address $0700, regardless of what load address was specified when the file was created. The second block (if it exists) will be loaded starting at address $1B00. Any additional blocks (should they exist) will be loaded at the addresses specified during the creation of the file.

NOTES:

1. A file may be loaded into a different memory bank from which it was saved.

2. The file to be loaded must be a loadable format file such as is generated by the SAVE command. An attempt to load a text file or other type file will result in an error. The format of a loadable file is described in Appendix B.

3. The file may consist of several non-contiguous blocks of memory, all of which will be loaded. See the SAVE command description for details.

4. If fewer <dest.> arguments are supplied than there are blocks in the file to be loaded, the remaining blocks are loaded starting at the addresses given when they were saved.

5. If more ⟨dest.⟩ arguments are supplied than there are blocks in the file to be loaded, the extra arguments are ignored.

6. GET always sets the Program Counter, P, to the value of the Entry point which was specified when the file was saved.

7. Specifying ⟨dest.⟩ does not affect the value used for the Entry point. The Program Counter will still be set to the value specified as the Entry point when the file was saved.

8. Naturally, the GET command with ⟨dest.⟩ specified does not relocate any machine language code; it merely loads the memory image at a different location. Therefore most programs will not run properly if loaded at a different address than was intended.

9. The GETLOC command can be used to ascertain the values of the ⟨entry⟩, ⟨from⟩, and ⟨to⟩ arguments which were used when the file was saved.

10. The GET command will not load a file into areas of memory reserved for CODOS unless an UNPROTECT command has been given. In addition, it will not load directly into memory below address $0200, unless the system has been UNPROTECTed. This encourages the good programming practice of reserving page 0 for scratch storage and page 1 for the stack. Be aware however that locations $0200-$06FF are used for system parameter storage and should not be overwritten by a GET command unless it is specifically desired to change these parameters.

COMMAND NAME: GETLOC. GETL!

PURPOSE: To display the Entry point, Starting load address, and Final load address for a file previously generated by the SAVE command.

SYNTAX: GETLOC <file> [:<drive>]

ARGUMENTS:

<file> = desired file name.
<drive> = disk drive number, 0 to 3. Defaults to the current default drive, usually 0.

EXAMPLES:

```
GETLOC VMT
```

will display the memory block and entry point used by the program VMT.C on drive 0. A typical display might be:

```
VMT.C=5014 5000 587C
```

which indicates that VMT loads into addresses $5000 through 587C inclusive, and execution starts at $5014.

```
GETLOC SEGS.X : 1
```

will display the load attributes of SEGS.X on drive 1. Assuming that SEGS.X was saved with three distinct blocks of memory (see SAVE command), the display might typically be:

```
SEGS.X=2000 2000 342D
            1300 13DD
            1780 17A8
```

which indicates that issuing a GET SEGS.X:1 command (or executing SEGS.X:1) would result in memory images being loaded into $2000 through $342D, $1300 through $13DD, and $1780 through $17A8. If SEGS.X:1 is executed, the program will be entered at $2000.

NOTES:

1. If the file specified was not generated by the SAVE command or other program generating loadable-format files, an error will result.

2. When using GETLOC to determine memory usage by a program, remember that programs loaded may use additional scratch RAM other than that actually loaded.

COMMAND NAME: GO.

PURPOSE: To begin execution of a machine-language program in memory.

SYNTAX: GO [<from>]

ARGUMENTS:

<from> = desired starting address. Defaults to current value of the Program counter (as displayed by the REG command).

EXAMPLES:

GO

begins execution at the current address of P. The current value of P can be displayed using the REG command.

GO 1200

begins execution of a machine language program at $1200.

GO 101A:3

executes a program starting at address $101A in bank 3 (assuming expansion memory is present in bank 3).

NOTES:

1. Upon entry to the program, the registers will be set as displayed (or defined) by the REG command, except the stack will be discarded (that is, S=FF).

2. The program is actually entered by a JSR instruction, so that a corresponding RTS will return control to the system. If a program re-enters CODOS in this manner, a subsequent REG command will display the status of all registers except P at the time of the RTS. This is useful for debugging subroutines since the GO command can be used to enter the subroutine, and the routine will return to CODOS on completion with the contents of the registers displayable. An RTS will return control to CODOS even from another memory bank other than bank 0.

3. The difference between the NEXT command and the GO command is that the NEXT command preserves the stack and enters the program via a jump (thus effectively continuing execution), whereas the GO command discards any stack (sets stack pointer to FF) and enters the program via a JSR. EXCEPTION: A GO command issued using SVC number 13 (see Section 6) will not discard the stack.

4. If the ¼from½ argument specifies a memory bank other than 0, the program is entered with the program bank and the data bank set to the specified bank number.

COMMAND NAME: HUNT. H!

PURPOSE: To search a block of memory for a string of bytes.

SYNTAX: HUNT <from><to> { " <char>..." / <value> ... / ' <char>...' }

ARGUMENTS:

<from>= starting address for the search.
<to>= final address for the search.
<char>= an ASCII character.
<value>= a numeric value, 0 to $FF. In a string of values, one (and only one) value can be replaced by the wildcard, "?", which matches any single byte.

EXAMPLES:

HUNT 2000 2400 'CODOS'

will search memory from $2000 though $2400 inclusive and list the address of all occurrances of the ASCII character string, "CODOS". If a match is made for all 5 bytes, the starting address of the matching string is displayed, and the search resumes at the next byte.

HU! 200:2 200+.100 4C

will search from $0200 to $0264 in memory bank 2 for the byte $4C.

HUNT 2320 4FFF 20 ? 03

will search from $2320 to $4FFF for a $20 followed by any byte followed by $03. This might be useful for searching for a JSR ($20) opcode to a subroutine which you know is somewhere in page 3 but you don't know exactly where.

HUNT 3000 3300 0D 'NOW IS THE TIME'

searches for a $0D byte (a carriage return) followed by the ASCII string "NOW IS THE TIME", between 3000 and 3300.

HUNT 200 4480 'GO' ? 'NEW'

Searches from 200 to 2260 for "GO" followed by any single byte followed by "NEW". This would match such strings as "GO3NEW", "GO NEW", "GO,NEW" and "GOTNEW".

NOTES:

1. HUNT reports all occurrences of the target byte-string in the region. A CNTRL-S can be used to temporarily suspend the display and CNTRL-C can be used to abort the command.

2. Only one "?" wildcard can be used, and it cannot be the first byte of the target byte string (that would be meaningless).

3. A "?" inside an ASCII string enclosed in quotes is not a wildcard and will only match a "?" in memory.

4. The largest byte-string permitted is 11 bytes.

5. When searching for a character string, if the region being searched includes the CODOS line-input buffer, you will always get a match at that location, because you are matching the string you typed in. However, if you DUMP the specified address, you will not see the desired string, because typing the DUMP command altered the content of the line-input buffer.

COMMAND NAME: LOCK. L!

PURPOSE: To enable the software write-protect for a file.

SYNTAX: LOCK <file>[:<drive>] ...

ARGUMENTS:

<file> = desired file name.
<drive> = disk drive desired. Defaults to the current default drive, usually drive 0.

EXAMPLES:

LOCK INVENTORY.T

sets the write-protect for the file called INVENTORY.T on drive 0. This will not affect other files on the disk.

Notes:

1. The LOCK command is used to protect files against INADVERTENT destruction. It is not intended to provide any kind of file security. For floppy disk systems, the most appropriate method of securing information is physical security of the disk.

2. The LOCK command will protect files from DELETE, SAVE, and RENAME commands, and from SVCs and languages which write or truncate the file. It will NOT protect files from the FORMAT utility program, nor from other software using the disk controller directly.

3. A backup disk should always be maintained for all important files on any floppy disk system.

COMMAND NAME: MSG. A1!

PURPOSE: To print a message over a specified channel.

SYNTAX - Single line form: MSG <chan> <text> CR
Multi-line form: MSG <chan> ^CR
<text> CR ...
<text> ^CR

ARGUMENTS:

<chan> = channel number from 0 to 9
<text> = any printable ASCII text except the "^" (caret) character.
CR = ASCII Carriage Return control character (not the letters "CR")

EXAMPLES:

MSG 2 Starting assembly.

will print "Starting assembly." followed by a carriage return over channel 2, which is usually assigned to the Console display. This would be useful, say, in a batch job file to keep the operator informed of the progress of the job.

MSG 6^
This represents the results of applying
a smoothing function to the data.^

will print the two line message above as two lines on whatever is currently assigned to channel 6. This might be useful for identifying the output from a program that does not identify it itself for some reason.

NOTES:

1. Be sure to remember the final "^" character in the multi-line form or the message will never terminate.

COMMAND NAME: NEXT. N!

PURPOSE: To resume execution after a break or interrupt or to initiate execution of a machine language program in memory.

SYNTAX: NEXT [<from>]

ARGUMENTS:

<from> = starting address. Defaults to current value of the Program Counter (P), as displayed by the REG command.

EXAMPLES:

NEXT

will begin execution at the address currently stored in the P register.

NEXT 223B

will begin execution at $223B.

NOTES:

1. The values of all registers upon entry to the program will correspond to the values shown or set by the REG command. This includes the stack pointer.

2. The program is actually entered via a JMP instruction, so that an RTS instruction will return to the address on the top of the stack, not to the CODOS monitor.

3. The program will be entered with the same program and data bank setting as was in effect the last time CODOS was entered.

4. The difference between GO and NEXT is that GO enters the program with a JSR after discarding any stack (i.e., sets S=FF), whereas NEXT enters via a JMP with the stack preserved. The primary advantage of the NEXT command is it enables a user to continue execution after a breakpoint has been encountered.

COMMAND NAME: ONKEY. ON!

PURPOSE: Define a function key legend and associated substitution string.

SYNTAX: ONKEY [<key #> [<legend> [<string> [<term>]]]]

ARGUMENTS:

<key #> = a function key number between 1 and 8 inclusive.
<legend> = a string of 8 or fewer characters enclosed in quotes which is to be displayed in the specified legend box.
<string> = a string of 31 or fewer characters in quotes which is to be entered into the input line buffer when the specified function key is pressed.
<term> = the numeric value of the termination character to be entered into the input line buffer following the ¼string½. If omitted, a carriage return will be entered. If bit 7 of the character is set, the ¼string½ will not be echoed to the console.

EXAMPLES:

```
ONKEY 3 "PAYROLL" "PAYROLL LASTWEEK.D THISWEEK.D"
```

will display the legend PAYROLL in the box at the bottom of the screen associated with function key 3. Following this, any time the operator presses the f3 key, the string PAYROLL LASTWEEK.D THISWEEK.D will be entered into the input line buffer just as if it was typed in by the operator. Since no termination was specified, the default value of $0D (ASCII carriage return) is entered next which causes the line to be executed immediately as a CODOS command. The string is also displayed (echoed) on the console display.

```
ONKEY 8 '  ASM' 'ASM DEVELOPMENT:1 L=' 0
```

will display the legend ASM in the rightmost function key box approximately entered (because of the two leading blanks). When the operator presses f8, the string "ASM DEVELOPMENT:1 L=" will be entered into the input buffer and also displayed on the console. Since a termination of $00 (ASCII NUL) was specified, the cursor will be positioned just beyond the last "=". The operator would presumably type in a file name and a carriage return to execute the complete command or use editing keys to alter the command.

```
ONKEY 1 'PRINTER' 'ASSIGN 6 P' 8D
```

will display the legend PRINTER in the leftmost function key box. Pressing the f1 key would execute the CODOS command "ASSIGN 6 P". Since bit 7 of the termination character is set, the substitution string will not be shown on the console output which essentially "hides" what the f1 key does in terms of CODOS commands.

```
ONKEY 3
```

will clear the legend for function key 3.

```
ONKEY
```

will clear all of the legends and substitution strings.

NOTES:

1. If the <legend> is longer than 8 characters, the excess is ignored.

2. If the <string> is longer than 31 characters, the excess is ignored.

3. Either single quotes or double quotes may enclose the legend and substitution string but must be the same on both sides.

4. The substitution string may contain only one line. If multiple lines are desired, you can prepare a Job file with the multiple lines and then specify a DO command as the substitution string.

COMMAND NAME: OPEN. O!

PURPOSE: To declare a disk ready for access by the system.

SYNTAX: OPEN [<drive>] ...

ARGUMENTS:

<drive> = disk drive number to be opened, 0 to 3. Defaults to drive 0.

EXAMPLES:

```
OPEN
```

opens the disk in drive 0 for operations.

```
O!1
```

opens drive 1 for subsequent operations.

NOTES:

1. Every disk must be OPENed prior to performing any command or operation on it (except FORMAT). The disk must be in the drive and the door closed before typing OPEN. Failure to open a disk before accessing it will result in an error message; if drive 0 is not open, an error number will be displayed without a message, since the system gets the error messages from a disk file (consult Appendix A).

2. The system requires that an OPEN disk be present in drive 0 at all times with a valid copy of the operating system on it. In addition, any user programs or data may also be on the disk in drive 0. Most Monitor commands are overlays which are loaded into memory from disk as needed; therefore an open disk in drive 0 is essential. Generally, the disk in drive 0 should only be closed when exchanging it for another disk or powering down the system. Certain Utility programs such as the single-drive copy utility open and close drive 0 automatically.

3. Unlike many other systems, it is not necessary to open or close individual files when using CODOS. It is only necessary to OPEN each disk as it is inserted, and CLOSE each disk before it is removed from the drive, or before powering down.

4. See the description of the CLOSE command for more details on OPEN/CLOSE considerations.

5. The disk in drive 0 is automatically OPENed by the system when it is "booted" up.

6. OPENing a disk which is already OPEN is permissible.

COMMAND NAME: PROTECT. P1

PURPOSE: To enable the memory-protect hardware on the upper 8k block of memory on the disk controller board (addresses $E000-$FFFF in bank 0) and enable the reserved-memory checking for SET and FILL commands.

SYNTAX: PROTECT

ARGUMENTS: none.

EXAMPLE:

```
PROTECT
```

NOTES:

1. The CODOS system normally "comes up" in protected mode.

2. In protected mode, the system will not allow any SET or FILL command into the portion of page 0 reserved for CODOS, nor into the stack nor into addresses $E000-$FFFF on the disk controller.

3. The effects of PROTECT are nullified by an UNPROTECT command.

4. PROTECT and UNPROTECT do not affect the disk or the effect of LOCK and UNLOCK commands.

COMMAND NAME: REG. R!,

PURPOSE: To display or alter the contents of the user's 6502 registers.

SYNTAX: REG [⟨reg. desig.⟩ [=] {"⟨character⟩" | ⟨value⟩ | '⟨character⟩'}]

ARGUMENTS:

⟨reg. desig.⟩ = register name to be altered, A, X, Y, F, S, or P.
⟨value⟩ = desired numeric value or numeric expression.
⟨character⟩ = desired ASCII character.

EXAMPLES:

```
REG
```

will display the contents of the registers.

```
REG  A=0
```

sets the A register to $00.

```
REG  X .65 Y="B" A = 10
```

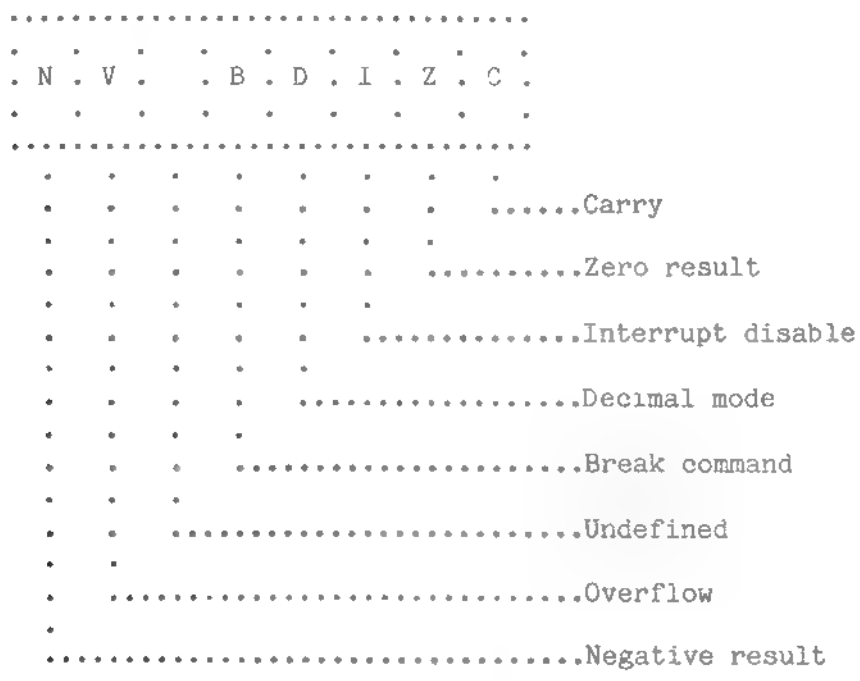
sets the X register to $41, the Y register to $42, and the A register to $10.

NOTES:

1. The REG command without arguments displays the user's registers in the format illustrated below:

```
      .....Current Program Counter (P)
      .
      .    .....Current Program Bank
      .    .
      .    . ......Current Data Bank
      .    . .
      .    . .  .......Contents of memory at P through P+2 in hex
      .    . .  .
      .    . .  .
      .    . .  .
      .    . .  .
   P=1B1F:0/0 (201A17)  A=2A X=05 Y=00 F=32 S=FD
                            .    .    .    .    .
                            .    .    .    .    .
Contents of accumulator (A).    .    .    .    .
                            .    .    .    .    .
      Contents of X reg.....    .    .    .    .
                                .    .    .    .
          Contents of Y reg.....    .    .    .
                                    .    .    .
               Contents of Flags(F).....    .
                                            .
                   Current Stack pointer(S)....
```

The individual bits in the Flags (F) register display are the same as the hardware Processor Status Word, as described below:



2. Either single or double quotes may be used to enclose the character when setting a register to an ASCII character, but the same type of quote must be used on both sides of the character.

3. The "=" between the register designator and the value is optional and in no way affects the meaning of the command.

COMMAND NAME: RENAME. REN!

PURPOSE: To change the name of an existing file.

SYNTAX: RENAME ⟨file⟩ [:⟨drive⟩] ⟨newfile⟩

ARGUMENTS:

⟨file⟩ = the existing file name.
⟨drive⟩ = disk drive number for the existing file. Defaults to the current default drive, usually 0.
⟨newfile⟩ = desired new file name.

EXAMPLES:

```
RENAME JUNK GARBAGE
```

changes the name of file JUNK.C on drive 0 to GARBAGE.C.

```
RENAME MYNEWTEXT.T :1  MYOLDTEXT
```

changes MYNEWTEXT.T on drive 1 to MYOLDTEXT.C. Since no extension was given for the new file name, ".C" was assumed.

COMMAND NAME: RESAVE. RES!

PURPOSE: To replace an existing file with a program or memory image(s).

SYNTAX: RESAVE ⟨file⟩ [:⟨drive⟩ [=⟨entry⟩ ,⟨from⟩ [=⟨dest.⟩]] ⟨to⟩ ...

ARGUMENTS:

⟨file⟩ = desired file name.
⟨drive⟩ = desired disk drive, 0 to 3. Defaults to the default drive, usually 0.
⟨entry⟩ = entry point desired. Defaults to from .
⟨from⟩ = starting address for the block of memory.
⟨dest.⟩ = address at which the block is to be loaded into memory on subsequent GET commands. Defaults to ⟨from⟩.
⟨to⟩ = final address of the memory block.

EXAMPLE:

```
RESAVE DOIT 200 2E3
```

saves the contents of memory locations $0200 through $02E# inclusive on the file named DOIT, replacing whatever was in file DOIT previously. If the new memory image is larger than the existing contents of the file, the file size will be increased automatically. If the new memory image is smaller, the file size will be reduced. Please refer to the description of SAVE below for further explanation.

NOTES:

1. The RESAVE command performs exactly the same function as the SAVE command except that if the specified file already exists, it will be replaced. The SAVE command would give an error under that condition.

2. If the specified file does not exist, there is no difference between RESAVE and SAVE.

3. The RESAVEd file will be rewritten in the same blocks on the disk as the original. This can be useful if it is desired to patch certain position sensitive system files such as CODOS.Z itself.

4. You may resave a file which is larger than the file being replaced without harm. CODOS will allocate additional space as needed.

COMMAND NAME: SAVE. 5!

PURPOSE: To save one or more blocks of memory on a file.

SYNTAX: SAVE <file>[:<drive>] [=<entry>]<from>[=<dest.>] <to> ...

ARGUMENTS:

<file> = desired file name.
<drive> = desired disk drive, 0 to 3. Defaults to the default drive, usually 0.
<entry> = entry point desired. Defaults to <from>.
<from> = starting address for the block of memory.
<dest.> = address at which the block is to be loaded into memory on subsequent GET commands. Defaults to <from>.
<to> = final address of the memory block.

EXAMPLES:

SAVE DOIT 200 2DF

saves the contents of memory locations $0200 though $02DF inclusive on a file called DOIT.C on drive 0 (by default). Since no optional arguments were specified, the entry point will be saved on the file as $0200, the same as the starting address of the block. DOIT is now a User-Command, so subsequently typing DOIT will cause the block to be loaded from disk into memory at $0200, and execution begun at $0200.

SAVE RALPH_PROG.C:1 = 2424 2000 20FE 340 3A0

saves a file called RALPH_PROG.C on drive 1. The file contains two memory blocks, the first from $2000 to $20FE, and the second from $0340 to $03A0. The entry point is $2424. Subsequently typing a RALPH_PROG:1 command will cause the two blocks of memory to be re-loaded from disk, and program execution begun at $2424.

SAVE SUBPKG.X 400=2000 400+.100

saves 100 decimal bytes of memory on a file called SUBPKG.X, starting at $0400. Since a dest. address was specified, a subsequent GET SUBPKG.X command will cause the memory block to be loaded into address $2000 and up instead of the $0400 address at which it was saved.

SAVE DISP_IMAGE.G C000:1 FBFF

saves the entire current screen image of the MTU-130 display on a file.

NOTES:

1. The existence of the "=" in the command indicates the existence of one of the optional arguments <entry> or <dest.>. Pay careful attention to the position the arguments.

2. When using <dest>, note that no relocation of any possible address references is made; the memory block is still exactly as saved. Therefore specifying <dest.> is not generally a satisfactory method of relocating machine language programs.

3. The <entry> point does not have to reside inside any of the saved blocks.

4. The number of blocks saved on a single file is limited only by the number of <from> <to> arguments you can fit on the command line.

5. Bank notation may be used on the <from> and <dest.> arguments. The <entry> must not specify a bank number, but will be understood to always reside in the same bank as the first block SAVEd.

6. The value FFFF or .65535 may not be used as the <to> argument.

COMMAND NAME: SET. SET.

PURPOSE: To set the value of memory locations.

SYNTAX: SET <from> [=] { "<character> ... " / <value> / '<character> ...' } ...

ARGUMENTS:

<from> = address at which to deposit the first value.
<value> = numeric value to be deposited.
<character> = an ASCII character to be deposited.

EXAMPLES:

```
SET 2000= 1B
```

sets address $2000 to $1B.

```
SET 2006 "ABC"
```

sets $2006 to $41 (ASCII "A"), $2007 to $42 (ASCII "B"), and $2008 to $43.

```
SET 1200 80-.10 "   " 80-.20 '"'
```

sets $1200 to $76, $1201 through $1203 to $20 (ASCII blank), $1204 to $6C, and $1205 to $22 (an ASCII double-quote character).

Notes:

1. The "=" is optional and has no effect on the meaning of the command.

2. As each byte is deposited in memory, it is verified by CODOS. If reading the byte back from memory results in a bad compare to the value deposited, an error message is issued and the command aborted.

3. Addresses are checked for validity before depositing each value. If an attempt is made to set Reserved memory, an error message will be issued, unless an UNPROTECT command was issued previously.

4. Occasionally it may be desired to set values into several adjacent I-O ports in a single command. The SET command generally can't do this since the verify may fail. One way to solve this is to SET the desired values elsewhere in memory and use the COPY command to actually install the values into the port addresses, since COPY has no validation or range checking.

COMMAND NAME: SVC. SV!

PURPOSE: To enable or disable SVCs (upon subsequent entry to user program).

SYNTAX: SVC [<off>]

ARGUMENTS:

<off> = any non-blank argument. Defaults to no argument.

EXAMPLES:

SVC

will cause SVCs to be enabled upon subsequent entry into any user program.

SVC OFF

will cause SVCs to be disabled upon subsequent entry into any user program.

NOTES:

1. The status of the SVC enable determines what action takes place when a BRK ($00) instruction is encountered in a user program. If disabled, control returns to the operating system Monitor and the register contents are displayed. If enabled, control is passed to the SVC processor, as discussed in Chapter 5.

2. Dumping memory location $EE will not necessarily show the current SVC status since it is not set until a user program is entered.

3. The SVC command does not load the SVC processor into memory. If the STARTUP.J file has not loaded the SVC processor automatically, then it must be loaded by using a GET SVCPROC.Z command. Using SVCs without having the SVC processor loaded results in unpredictable behavior.

COMMAND NAME: TYPE. T!

PURPOSE: To display, print, or create a text file.

SYNTAX: TYPE <device> / <file> :<drive> / <channel> [<dest.device> / <dest.file> :<drive> / <dest.channel>]

ARGUMENTS:

<device>= single character source device name.
<file> = desired file name to type.
<drive>= desired disk drive, 0 to 3. Defaults to the current default drive, ususally 0.
<channel> = desired pre-assigned source channel number, 0 to 9.
<dest.device> = desired output device name. Defaults to Console ("C").
<dest.file> = desired file to receive output from TYPE.
<dest.channel>= desired pre-assigned channel to recieve output from TYPE.

EXAMPLES:

```
TYPE MYSOURCE.A
```

will display the file MYSOURCE.A on drive 0 on the Console.

```
TYPE MYPROG.L P
```

will type the contents of the file MYPROG.L on the printer.

```
TYPE C NEW.T
```

will accept input from the console keyboard and put it on a file called NEW.T. This is one way to create a text file.

```
TYPE 5 STUFF.T:1
```

will accept input from the file or device assigned to channel 5 and output it to the STUFF.T file on drive 1.

NOTES:

1. The first argument specifies the source for the TYPE command; the second argument is optional and specifies the destination.

2. The second argument defaults to the Console ("C") device.

3. When the source for the TYPE command is the console keyboard, CNTRL-Z is used to enter End-of-File and therefore terminate the TYPE command.

4. If a file name is given for either argument, the file will be automatically positioned to Beginning-of-Data before typing starts. However, if a channel is used for the argument, no positioning takes place. This fact can be used to advantage to copy parts of a file or concatenate files. For example:

```
ASSIGN 6 OLDTEXT.T
ENDOF 6
TYPE C 6
```

can be used to append lines onto the existing file OLDTEXT.T from the Console. However,

```
TYPE C OLDTEXT.T
```

would overwrite and replace the existing file, so be careful!

5. The TYPE command always frees the channels used when it terminates, unless the command was aborted.

6. If the TYPE command is aborted using CNTRL-C, the channels it uses will remain assigned. You may use the ASSIGN command to check this.

7. The TYPE command assumes that the file to be typed will consist of ASCII characters. If you attempt to TYPE an executable file, you will see garbage displayed.

COMMAND NAME: UNLOCK. U!

PURPOSE: To disable the software write-protect for a file.

SYNTAX: UNLOCK <file>[:<drive>] ...

ARGUMENTS:

<file> = desired file name.
<drive>= desired disk drive, 0 to 3. Defaults to the current default drive, usually 0.

EXAMPLES:

```
UNLOCK VALUABLES
```

removes the write-protect from the file called VALUABLES.C on drive 0.

```
UNLOCK GOODIES.T:1 GOODIES.A:1
```

removes the write-protect from both files specified.

NOTES:

1. It is permissable to UNLOCK a file which is not LOCKed.

COMMAND NAME: UNPROTECT. UN!

PURPOSE: To disable the hardware write-protect on the top 8K of RAM on the disk controller board (addresses $E000-$FFFF in bank 0) and disable the system reserved-memory checking for SET and FILL commands.

SYNTAX: UNPROTECT

ARGUMENTS: none.

EXAMPLE:

```
UNPROTECT
```

NOTES:

1. Once UNPROTECTED, the SET and FILL commands will be able to freely over-write normally-reserved areas of memory including the part of page 0 used by CODOS, page 1, and the System RAM on the disk controller. Naturally, casual abuse of this facility is likely to cause strange and invariably unpleasant results.

2. The GET command can load into System RAM or page 0 or 1 after an UNPROTECT command. It is the user's responsibility to ensure that blocks loaded into these areas will not conflict with CODOS memory usage (see Appendix F).

CHAPTER 4.

CODOS UTILITY PROGRAMS

Utility Programs differ very little from built-in commands from the user's viewpoint. Utilities are invoked from the Monitor by merely typing the name of the desired Utility followed by any required or optional arguments, just as is the case for the built-in commands. However, the Utilities have the following distinctions:

1. The names of the Utility programs appear in the disk directory just like any user command, and can be deleted or renamed if desired.

2. The Utility programs execute in the System Utility area of RAM from $B400-BFFF. Some utilities also use a large buffer for copying disk files (called the "Large Transient buffer"). The standard location of this buffer is A000-B3FF in bank 0, but it may be altered. Please refer to Section 10 and Appendix E for the location of the system parameter that governs the buffer location and size. *

3. Utility names cannot be abbreviated using "!".

The standard Utilities that would be used by nearly all CODOS users are listed in Table 4-1, and are described in the following section. Many additional utilities with more specialized functions are listed in a separate utilities manual. Several other utility programs for system generation are described in section 10.

TABLE 4-1. UTILITY PROGRAMS

Name	Function
DIR	Display file attributes for selected file(s), using "wildcard" name matching.
COPYF	Copy file(s) on a multiple-drive system.
COPYF1DRIVE	Copy file(s) on a single-drive system.
FORMAT	Initialize a new or existing disk; test and bypass any defective sectors; copy the operating system if desired.
KILL	Deletes files on disk using "wildcard" name-matching.

UTILITY NAME: DIR.

PURPOSE: To display the attributes of selected files.

SYNTAX: DIR <pattern> ...

ARGUMENTS:

<pattern> = desired file name, optionally using "wildcard" characters as described below:

* matches any string of characters terminated by (but not including) ".".

? matches any single character.

- (dash) matches any string of characters terminated by and including "_" (underline). See note 2 below.

The default pattern is "*.?" on the current default drive, ususally 0.

EXAMPLES:

DIR

will list the attributes of all the files on drive 0. A typical display might be:

```
CODOS.Z              :0 L  24-JAN-81  $0018C0
AIMEXT.Z             :0 L  24-JAN-81  $0001BF
SVCPROC.Z            :0 L  24-JAN-81  $00018F
COPYF.C              :0 L  24-JAN-81  $000082
MYTEXT.T             :0 -  *UNDATED*  $0010CD
```

The first column is the file name and extension. The ":0" indicates the drive. The next column either contains "-" or "L". The "L" indicates that the file is locked. The next column is the creation date for the file. The final column is the file size in hexadecimal bytes.

DIR *.T

will display the names of all files on drive 0 with a ".T" extension.

DIR *.?:1

will display all files on drive 1.

DIR INVENTORY.? ORDERS.?

will display all files on drive 0 named INVENTORY or ORDERS with any extension.

DIR DATA_-VS_Z.D

would display the attributes of file names such as DATA_X_VS_Z.D or DATA_Y_VS_Z.D, but not DATA_X_VS_Y.D

DIR OLD*.A

will display the attributes of any file starting with "OLD" with an ".A" extension.

NOTES:

1. In order to display the attributes of files on drives other than 0, the pattern argument must be given. For example, typing "DIR :1" will cause CODOS to attempt to execute the program called DIR on drive 1. If the DIR Utility exists on drive 1, then it will be executed, and since no arguments are given, it will display the attributes of all files on drive 0, which is probably not what was intended. To display all files on drive 1, the correct command is "DIR *.?:1".

2. A pattern of the form: *JAN.? will not match any file names. This is because the "*" is defined as matching any string terminated by a "." and that is clearly not possible within the confines of legal file names. If you wish to adopt a naming convention that uses a suffix to specify a class and a prefix to distinguish within the class (as the above pattern implies), use an "_" (underline) character in the file name to separate the prefix from the suffix and then use the "-" (minus) character in the pattern specification to represent any prefix. For example, -JAN.? will find SALES_JAN.D and PROJECT_JAN.D.

3. DIR will not correctly show the length of a file which has just been written until the channel assigned to it has been FREEd or the disk CLOSEd. If you see a file with a length of 0, check to see if it is still assigned.

UTILITY NAME: COPYF.

PURPOSE: To copy files on a multi-drive system.

SPECIAL HARDWARE: At least 2 disk drives are required by COPYF. See COPYF1DRIVE if you have only 1 disk drive.

SYNTAX: COPYF <pattern> [:<drive>] [<newdrive>]

or

COPYF <file> [:<drive>] [<newfile> [:<newdrive>]]

ARGUMENTS:

<pattern>= file name with "wildcard(s)". "Wildcards" recognized are:

* matches any string terminated by (but not including) ".".
? matches any single character except ".".
- matches any string terminated by (and including) "_". See note 7.

The default pattern is "*.?" which matches all files.

<file> = desired file name to copy.

<drive>= disk drive where file is to be found. Defaults to the current default drive, ususally 0.

<newfile> = desired new file name desired. Defaults to <file>.

<newdrive> = desired destination disk drive, 0 to 3. Defaults as follows:

if drive =...	then default newdrive =...
0	1
1	0
2	3
3	2

EXAMPLES:

```
COPYF
```

copies all files on the disk in drive 0 to drive 1, except files which already exist on drive 1. The system will display the names of the files, for example:

```
CODOS.Z:1 ALREADY EXISTS.
SYSERRMSG.Z:1 ALREADY EXISTS.
MYDATA.D COPIED.
GO_INVENTORY.C COPIED.
FORMAT.C COPIED.
AD_DRIVER:1 ALREADY EXISTS.
```

indicating that three of the files were not copied because they already existed on drive 1, and that the remaining three files were copied.

```
COPYF TURKEY
```

copies the file TURKEY.C from drive 0 onto drive 1. The new file on drive 1 will also be named TURKEY.C. No other files will be copied.

```
COPYF *.G:1
```

copies all files ending in ".G" on drive 1 to drive 0, unless they already exist on drive 0.

```
COPYF NEW*.? 2
```

copies all files which have names starting with "NEW" on drive 0 to drive 2, unless they already exist on drive 2.

```
COPYF DATA.D:1
```

copies file DATA.D from drive 1 to drive 0.

```
COPYF CLONE NEWCLONE:0
```

duplicates file CLONE on drive 0, with a name change. After the command, both CLONE and NEWCLONE will be on drive 0. Except for the name and creation date, they will otherwise be identical.

```
COPYF STUFF.T:3 OLDSTUFF.T:1
```

copies file STUFF.T from drive 3 to drive 1 and changes the file name on the drive 1 file to OLDSTUFF.T.

```
COPYF ????.C
```

copies all files with exactly four characters in the name and a ".C" extension from drive 0 to drive 1, except files which already exist on drive 1.

NOTES:

1. If an attempt is made to copy a specific file name (that is, without any wildcards), and that file already exists on the destination disk, then an error message will be given (unless the system "re-save OK" flag has been set as described in Appendix E). However, if a pattern is given (that is, with one or more wildcards), then file names which match but already exist on the destination disk are simply ignored, and the Utility execution continues. This allows users to easily copy all files which do not already exist on the destination disk by simply typing COPYF without arguments.

3. You may not use wildcards when changing the file name during a COPYF.

4. COPYF should not be used to copy the operating system, CODOS.Z. Copying the operating system is accomplished using the FORMAT program.

5. COPYF uses the Large Transient Buffer during the copy operation (see Appendix F for location).

6. Files created by COPYF are always unlocked, regardless of whether the original file was locked or not.

7. Please see note 2 in the DIR command description for additional information about wildcards.

UTILTIY NAME: COPYF1DRIVE.

PURPOSE: To copy files onto another disk in a one-drive system.

SYNTAX: COPYF1DRIVE

ARGUMENTS: none.

EXAMPLE:

    COPYF1DIRVE

executes the single-drive file copier. The Utility is completely interactive, and will prompt:

    PUT SOURCE DISK IN.
    FILE (OR CR IF DONE)?=

Type in the name of the file to be copied. The Utility will prompt:

    PUT DEST. DISK IN,
    CR WHEN READY.?=

Remove the source disk from the drive and insert the desired disk to receive the copy of the file. This disk must have been previously formatted. When the new disk is in and the door is closed, depress carriage Return. Usually at this point the system will prompt:

    PUT SOURCE DISK IN.
    FILE (OR CR IF DONE)?=

which indicates your file has been copied and you may now copy another file. If you do not want to copy anotner file, put whichever disk you want to use (old or new) into the drive and hit carriage Return. If you wish to continue copying other files, insert the desired source disk and type the file name.

Occasionally some files will be too long for the COPYF1DRIVE Utiltiy to copy in a single pass. In this case, the Utility will prompt:

    PUT SOURCE DISK IN.
    CR WHEN READY.?=

when you depress Carriage return, it will copy the remainder of the file. Several passes may be needed for files much larger than the Large Transient Buffer.

NOTES:

1. COPYF1DRIVE should never be executed from a job file.

2. Increasing the size of the Large Transient Buffer, as described in Section 10, will increase the file size which can be copied in a single pass. This is strongly recommended since the standard system uses a relatively small buffer suitable for dual-drive systems.

3. "Wildcards" are not available for COPYF1DRIVE.

4. Do not use COPYF1DRIVE to attempt to copy the operating system onto a disk. Only the FORMAT Utility can correctly copy the operating system.

5. Files created by COPYF1DRIVE are always UNLOCKed initially.

Blocks 4K
K = 1024

UTILITY NAME: FORMAT.

PURPOSE: To erase and re-format a disk for CODOS use, test and bypass defective disk sectors, and copy the operating system files to the disk.

SYNTAX: FORMAT [S [= ⟨interleave⟩]] [T = ⟨skew⟩]

ARGUMENTS:

⟨interleave⟩ = optional sector-to-sector interleave factor. See note 2 below. If "S" is specified without a numeric argument, S=3 is assumed. Defaults to S=2 if ommitted completely.

⟨skew⟩ = optional track-to-track skew factor, See note ^ below. Defaults to T=$C.

EXAMPLE:

FORMAT

initiates the interactive FORMAT Utility. The Utility will display different prompts, depending on whether you have a single-drive or multiple-drive system. On a Multiple Drive system, the program prompts:

```
WARNING: FORMAT WILL IRREVOCABLY
ERASE EVERYTHING ON DISK IN DRIVE 1.
ARE YOU READY (Y/N)?=
```

Any reply starting with "Y" or a carriage return will be interpreted as a "YES" reply. Anything else is a "NO" reply and aborts the command. Before replying make sure the disk you want to format is in drive 1. A "YES" reply will cause FORMAT to erase all tracks on the disk, write new timing information on the tracks, and test the directory track for bad sectors. All this takes about a half a minute (about one minute for double-sided disks). It will then prompt:

```
WANT TO TEST FOR BAD SECTORS (Y/N)?=
```

If you want to test every sector on the disk, type "YES". Testing a single-sided disk takes about 3 minutes to complete. Normally you will probably not want to test diskettes unless you have doubts about the integrity of the diskette. The test procedure consists of writing random data into every byte of every sector on the disk, reading it all back and comparing to the data written. If any errors occur, the sector will be bypassed automatically during file allocation by the system and not used. A message will indicate what track and sector was bad and bypassed. If the error occurs in the directory or system overlay portion of the disk, the Utility aborts with the message "DISK UNUSABLE", since directory sectors cannot be bypassed. See Note 3.

The next prompt issued by FORMAT is:

```
DISK VOLUME SERIAL NO. (VSN)?=
```

Enter any hexadecimal number desired between 0 and FFFF. This Volume Serial Number is written in the directory area of the disk and is intended to uniquely identify each disk. Therefore you will normally want to give every disk a different number. You may also want to write the VSN on the label portion of the disk using a soft magic marker for visual identification purposes. The DISK command displays the VSN. You may assign any VSN you wish. The next prompting message is:

WANT TO COPY DRIVE 0 SYSTEM (Y/N)?=

If you want to have a copy of the operating system on the newly-formatted disk, reply "YES". Normally you will want to copy the system onto all new disks. On multiple drive systems, it is only necessary for the disk in drive 0 to have an operating system image on the disk. Therefore if you only plan to use the new disk in another drive, you can reply "NO". The advantage of this is that you gain about 20K of additional free space on the disk. Normally this small saving in space does not justify the added potential inconvenience of being unable to "boot up" or run the disk in drive 0.

When the copy operation is complete, the Utility issues the message:

NEW DISK IS NOW OPEN.

The FORMAT Utility is completed. To ascertain which files were copied by the FORMAT program, type:

FILES 1

You may want to use the COPYF Utility to copy additional files. In particular, you will probably want to copy the COPYF Utility and the FORMAT Utility, and any device drivers (such as PRINTDRIVER.Z) needed by the STARTUP.J file. These are not copied by FORMAT. At this point, the disk in drive 1 can be used to "boot-up" the operating system at any future time by inserting it in drive 0 and executing the boot loader.

For single-drive systems, a similar dialog will be initiated by FORMAT, except that you will be prompted to change disks for copying the system. Be sure to remember to remove the old diskette and insert the new one (the "DEST." diskette) when prompted to do so, before replying "YES" to "ARE YOU READY?". You will not be given the option of not copying the system, since every disk must have it in a one-drive system. Use COPYF1DRIVE to copy the additional files desired upon completion of the Format utility.

NOTES:

1. FORMAT uses the Large Transient Buffer in memory. See the Memory Map, Appendix F, for its default location. The buffer location can be altered as described in Chapter 10.

2. The optional "S" option on the FORMAT command alters the "skew" of the sectors on the disk from the standard alternate-sector skew to a specified value. The meaning and purpose of this option is discussed in Appendix D. This argument should normally be ommitted. The "T" option specifies the number of physical sectors which will intervene between logically adjacent sectors when stepping from one track to the next. Normally this argument should not be specified.

3. When FORMAT discovers a defective sector during testing, it is normal for it to report the same defective sector twice and possibly three times.

4. If defective sectors are reported anywhere on track 0, the disk cannot be used to boot-up the system. It still can be used as a data-disk (no system present) on a multi-drive system however.

UTILITY NAME: KILL.

PURPOSE: To selectively delete files matching a given name with "wildcard" character matching.

SYNTAX: KILL ⟨pattern⟩

ARGUMENTS:

⟨pattern⟩ = desired file name, optionally containing the following "wildcard" characters:

* matches any string of characters terminated by (but not including) "."

? matches any single character.

- (dash) matches any string of characters terminated by and including "_" (underline). See note 1 below.

EXAMPLE:

```
KILL MYFILE
```

executes the KILL Utility to delete file MYFILE.C on drive 0. If the file is not found, no action takes place; otherwise, the KILL Utility displays:

```
ENTER CR OR Y TO DELETE, N TO KEEP FILE:
MYFILE.C:0 ?=
```

This prompt affords you an opportunity to make sure you got the file you really wanted. If you wish to delete the file, enter either a carriage return or a word starting with Y followed by a carriage return. Any other response will not delete the file.

```
KILL *.G :1
```

will display the names of all the file names with ".G" extensions on drive 1 and let you approve or veto the deletion for each file individually.

```
KILL  ???OLD*.?
```

will display the names of all the files with "OLD" for the fourth through sixth letters of the name, and let you kill or keep each file.

NOTES:

1. A LOCKed file cannot be KILLed.

2. Once a file is KILLed, it cannot be recovered. Therefore exercise caution and be certain you have the right file and drive.

3. Please see note 2 in the DIR command description for additional information about wildcards.

CHAPTER 5.

INTERFACING USER-WRITTEN ASSEMBLY-LANGUAGE PROGRAMS TO CODOS

INTRODUCTION

This section introduces methods by which user-written assembly-language programs may communicate with the outside world through the CODOS operating system, and take advantage of various utility functions provided by the system. Using the functions described here can greatly reduce program development time and effort.

Most operating systems provide a degree of support for assembly-language programming by making available the addresses of certain system subroutines which the user can call to perform I-O or other functions. For example, to output a character to the console, you might put the ASCII character into the A register and call the driver subroutine for the console display device. CODOS does not use this method, but instead provides a more powerful tool called the Supervisor Call Instruction (SVC). The SVC concept is not new; SVCs are found in various forms on many large mainframe computers.

The following discussion assumes a knowledge of 6502 assembly language programming on the part of the reader.

HOW SVC'S WORK

The CODOS implementation of the Supervisor Call capability consists of a BRK instruction ($00) followed by a one-byte numeric code which tells the system what function is required. The code numbers are listed in Table 5-1. Effectively, the SVC is a lot like a JSR (Call Subroutine) instruction, except that it is two bytes long instead of three, and the second byte is not an address, but a code which tells what pre-defined system subroutine is to be called. Individual SVCs are explained in detail in Chapter 6.

Why are SVC's better than a straightforward JSR? There are several reasons:

1. SVCs are address-independent. This is by far the most important advantage of SVCs. It means that future system upgrades which may alter the addresses of actual system routines will not affect the SVC numbers, and therefore will not adversely affect programs using SVCs. It also means that, for example, a program on an AIM-65 computer with CODOS at $8000 can be transported to an MTU-130 system and run without modification. If subroutine calls were used instead, it would be necessary to patch all the JSRs to the system routines before execution.

2. SVCs use less memory. Two bytes are cheaper than three.

3. SVCs preserve the values in registers. All registers are restored to their condition upon entry to the SVC when returning to the calling program, except when returning values to the calling program. This saves the programmer a lot of unnecessary saving and restoring registers.

4. SVC's are easier to debug. If an error is detected by the system while processing an SVC, the program will abort and CODOS will display the exact address of the offending Supervisor Call, the values of all the registers at the time of the SVC, and an error message explaining the difficulty. Illegal or unimplemented SVCs are also trapped in the same manner.

TABLE 5-1: CODOS SVC NUMBERS

SVC#	Description	Pass Regs.	Returns Regs.
0	Show registers, Enter CODOS Monitor.	-	-
1	Enter CODOS Monitor.	-	-
2	Output inline message	(See text)	-
3	Input byte from channel.	X	A, F
4	Output byte to channel.	A,X	-
5	Input line from channel.	X,U5	A,Y,F
6	Output line to channel.	X,Y,U6	-
7	Output string to channel.	X,Y,U6	-
8	Decode ASCII hex to value.	Y,U5	A,Y,F,U0
9	Decode ASCII decimal to value.	Y,U5	A,Y,F,U0
10	Encode value to ASCII hex.	Y,U0,U6	Y
11	Encode value to ASCII decimal.	Y,U0,U6	Y
12	Query buffer address & passed argument.	-	U5,U6,Y
13	Execute a CODOS Monitor command.	U5	(See Chapter 6)
14	Query channel assignment.	X	A,F
15	Read record from channel.	X,U1,U2	F,U1,U2
16	Write record to channel.	X,U1,U2	F
17	Position file to beginning.	X	-
18	Position file to end-of-file.	X	-
19	Position file.	X,U7	U7
20	Query file position.	X	X,U7
21	Assign channel to file or device.	X,A,U3	A,F
22	Free Channel.	X	-
23	Truncate file at present position.	X	-
24	Define interrupt vector.	U0	-
25	Define error-recovery vector.	U0	-
26	Restore default error recovery.	-	-
27	Enter 16-bit Pseudo-processor.	-	F
28	Query CODOS Version.	-	A,X,Y
29	Query file status.	Y,U5	A,Y,U3,F
30	Query date.	Y,U6	Y,U6

INITIALIZATION AND PARAMETER PASSING

In order to use SVCs, the user program must first enable the Supervisor by setting the SVC Enable flag, SVCENB (address $00EE), to $80 (bit 7 must be set to 1). If SVCs are not enabled, any BRK instruction will simply return to the Monitor with a display of the location of the BRK and register contents. Note that the SVCENB flag must be set to $80 by the user program, or by the SVC command. Setting $EE to $80 from the Monitor using the SET command will not work. The recommended procedure is to have the program set the SVC enable flag.

Usually, some type of argument needs to be passed to the Supervisor and/or returned to the user program from the Supervisor. The method for passing arguments

is defined for each SVC individually, and may be done in three possible ways:

1. Arguments may be passed or returned in 6502 registers.

2. Arguments may be passed in one or more "Pseudo-Registers" in page zero.

3. Arguments may be passed "in-line", immediately following the SVC.

Before proceeding further, an example program will illustrate SVC usage.

Example Program 1: Displaying a text message.

The first SVC we shall examine in an example is SVC 2, which outputs a message over a channel. This is a very unusual SVC in that the argument is passed in-line. However, it is so frequently needed in programming that it deserves our first attention.

PROBLEM: Write a program to display the message "HELLO THERE." on the console.

SOLUTION:

```
SVCENB    =       $EE     ;SVC ENABLE FLAG LOCATION
;
          *=      $2000   ;PROGRAM ORIGIN
GREET     LDA     #$80
          STA     SVCENB  ;ENABLE SVCS
          BRK             ;SVC...
          .BYTE   2       ;...#2 = OUTPUT INLINE MESSAGE...
          .BYTE   2       ;...OVER CHANNEL 2...
          .BYTE   'HELLO THERE.'
          .BYTE   0       ;0 TERMINATES MESSAGE TEXT
          RTS             ;RETURN TO MONITOR OR CALLING PROGRAM
          .END
```

EXPLANATION:

The program begins by enabling SVCs (note: once enabled, SVCs remain enabled until disabled by writing $00 into SVCENB; it is advisable to disable SVCs when not needed). The BRK instruction together with the first .BYTE 2 pseudo-instruction comprise the SVC, and Table 5-1 tells us that an SVC 2 is used to display an inline message. The second .BYTE 2 tells the System what channel to output the message on. Channel 2 was selected for our example because it is assigned to the console display by default. Of course, it could be re-assigned to any device or file. Following the channel is the text of the message, which can consist of up to 254 bytes and is terminated by a $00. The $00 also is the last argument of inline code. The System will output the message over channel 2 and then return control to the instruction following the $00 byte; in this case, the RTS which terminates the program.

Remember that SVCs do not alter any registers except to return values to the calling program; since SVC 2 does not need to return values, no registers are altered. This is a big benefit, since it means that you can put inline messages anywhere you please in your program for debugging purposes without having to worry about side effects to the registers. Note that SVC 2 does not output any carriage return automatically; if you want to output control characters, you may include them explicitly in the message, as illustrated below.

Example Program 2: Display message on a new line.

PROBLEM: Repeat Problem 1, above, but start the message on a new line.

SOLUTION:

```
SVCENB    =       $EE
;
          *=      $2000
GREET     LDA     #$80
          STA     SVCENB  ;ENABLE SVCS
          BRK
          .BYTE   2       ;SVC 2 = INLINE MESSAGE
          .BYTE   2       ;...ON CHANNEL 2
          .BYTE   13      ;13=$0D=ASCII CARRIAGE RETURN
          .BYTE   'HELLO THERE.'
          .BYTE   0       ;TERMINATOR
          RTS
```

EXPLANATION:

The only change to this program from Example Program 1 is the addition of the ".BYTE 13" at the start of the message, which produces a carriage return. Any control characters desired can be embedded in the message in this manner, except ASCII NUL (because NUL = $00, the message terminator.).

There are three common programming errors which you should avoid when using SVC 2 to generate messages:

1. Forgetting to enable SVC's (in which case the program will simply return to the Monitor with a display of the registers when the first BRK instruction is encountered).

2. Forgetting the CHANNEL argument (which usually results in an error message of "ILLEGAL CHANNEL NUMBER" or "CHANNEL NEEDED IS UNASSIGNED").

3. Forgetting the zero-byte terminator for the message, (which often results in your program going into "hyperspace" after displaying the message).

PASSING ARGUMENTS TO THE SVC PROCESSOR IN 6502 REGISTERS

The example programs above passed their arguments to the Supervisor in-line. A much more common method of parameter-passing is the use of the 6502 registers. The following example illustrates register parameter passing.

Example Program 3: Character Input-Output.

PROBLEM: Write a program which reads a stream of bytes from channel 5 until a "." character is encountered, or end-of-file is reached. Display a message indicating which of these two events occurred. Assume channel 5 has been previously assigned to a valid file or input device.

SOLUTION:

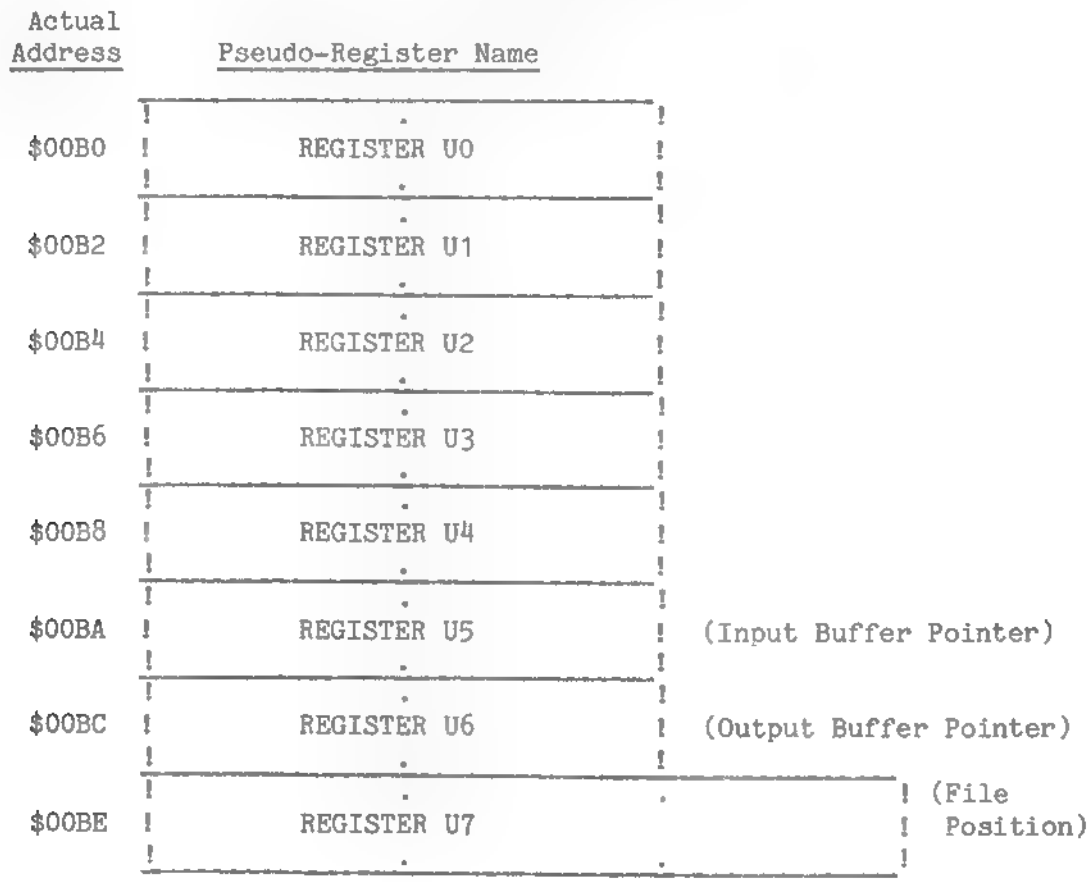
```
SVCENB    =       $EE
;
STRMIN    LDA     #$80
          STA     SVCENB
NEXTCH    LDX     #5        ;CHANNEL 5 FOR INPUT STREAM
          BRK
          .BYTE   3         ;SVC #3 = INPUT CHARACTER FROM CHAN (X)
          BCS     EOFENC    :BRANCH IF END-OF-FILE ENCOUNTERED
          CMP     #'.       ;ELSE EXAMINE CHARACTER INPUT
          BNE     NEXTCH    :IF NOT ".", READ MORE
          BRK
          .BYTE   2         ;ELSE DISPLAY INLINE MESSAGE
          .BYTE   2         :...ON CHANNEL 2
          .BYTE   13,'"." ENCOUNTERED.',0 ;GIVE MESSAGE
          RTS
EOFENC    BRK
          .BYTE   2         ;SVC 2= INLINE MESSAGE
          .BYTE   2         ;...ON CHANNEL 2
          .BYTE   13,'E-O-F ENCOUNTERED.',0 ;GIVE MESSAGE
          RTS
```

EXPLANATION:

This program illustrates a number of aspects of SVC usage. The line labelled NEXTCH is used to load the channel number desired into X. The Supervisor expects to find the channel number in register X when the SVC is processed, as is detailed in Chapter 6. SVC 3 returns the character read in the A register, and sets the carry flag only if End-of-File was encountered. End-of-File is an important concept. The End-of-File flag (the carry flag) is set by the SVC processor only if no more characters can be read from the selected channel. If the input channel is the console keyboard, this means that CNTRL-Z was entered (the CNTRL-Z character is not returned in A). If channel 5 was assigned instead to a file, it simply means that the previous character was the last character in the file. The programmer should always check for End-of-File when doing any kind of input operation, so that programs are device-independent. No error will occur if you attempt to read beyond end-of-file; the result in A is just not meaningful. It is the Programmer's responsibility to test the carry on every input operation and take appropriate action if it is set.

In our example, once we have ascertained that E-O-F was not encountered, the character received from channel 5 is checked to see if it is a ".". If not, another character is read. Once one of the two terminal conditions is met, an SVC 2 is used to issue a message to the console (channel 2) indicating which event occurred.

FIGURE 5-2 PSEUDO-REGISTERS



NOTES FOR FIGURE 5-2:

1. All values are passed in the usual 6502 fashion with low byte first.

2. The memory locations shown are not used by the system for any purpose whatsoever except processing user SVCs. This memory can therefore be freely used by the user.

3. The SVC enable flag is at address $EE.

PASSING ARGUMENTS IN CODOS PSEUDO-REGISTERS

Sometimes it is necessary to pass addresses or other 16-bit information to the SVC processor. The 8-bit A, X, and Y registers of the 6502 are inadequate for this purpose, so a set of eight Pseudo Registers (hereafter called P-registers or simply P-regs) are provided in zero-page, as shown in figure 5-2 . P-regs U0 through U6 are each 16 bits wide; U7 is 24 bits wide, and is used for file positioning, as we shall see later. Note that if SVCs are not enabled, these P-regs are not used for any purpose whatsoever by the system, and may be freely used as ordinary program memory by application programs. Values to be passed to the SVC processor are installed in these P-registers in the usual manner for memory. The SVC processor

expects to find certain addresses or values in specific P-registers, depending on the SVC. For example, most I-O functions (except single character I-O) use U5 to hold a pointer to an input buffer and U6 to hold a pointer to an output buffer. Each SVC description tells what P-registers are used, if any. Certain SVCs return information to the application program in P-regs. For example, SVC 12 ($0C) does not pass any P-regs to the SVC processor, but the system returns U5 and U6 to the application. The addresses returned are pointers to the system input and output line buffers, respectively.

Example Program 4: Line-Oriented I-O.

Most programs need to deal with input and output of strings or lines of characters. Several SVCs are provided for support. Applications programs will make heavy use of the 6502 (Indirect),Y addressing mode in these applications. In general, P-register U5 (for input) or U6 (for output) must be initialized to point to the start of a buffer containing the current line of interest. The Y register is used to index the particular character of interest within the line. Normally, the System Input and Output buffers are the most convenient to use, since an SVC 12 will automatically setup the proper addresses in U5 and U6, but the programmer may select any location for the buffers. The System buffers are sufficiently large for lines of up to 192 characters. The following problem illustrates line-processing.

PROBLEM: Write a program to copy lines of input text from channel 5 to channel 6 until an End-of-File is encountered. Assume Channel 5 and 6 have been given appropriate assignments.

SOLUTION:

```
SVCENB    =       $EE
U5        =       $BA       ;P-REG U5
U6        =       $BC       ;P-REG U6
;
NCHARS    *=*+1             ;TEMP SAVE FOR COUNT OF CHARACTERS
;
COPY56    LDA     #$80
          STA     SVCENB    ;ENABLE SVCS
          BRK
          .BYTE   12        ;SVC 12 = QUERRY SYS. BUFFER ADDRESSES
NEXT      LDX     #5        ;CHANNEL 5 FOR INPUT
          BRK
          .BYTE   5         ;SVC #5 = INPUT LINE TO BUF. AT (U5)
          BCS     EOFENC    ;BRANCH IF END--OF-FILE ENCOUNTERED
          STA     NCHARS    ;ELSE SAVE CHARACTER COUNT
LOOP      LDA     (U5),Y    ;COPY CONTENT OF INPUT BUFFER...
          STA     (U6),Y    ;...TO OUTPUT BUFFER
          INY
          CPY     NCHARS
          BNE     LOOP      ;...UNTIL WHOLE LINE COPIED
          LDX     #6        ;CHANNEL 6 FOR OUTPUT
          BRK
          .BYTE   6         ;SVC #6 = OUTPUT LINE AT (U6)
          JMP     NEXT      ;REPEAT FOR NEXT LINE
EOFENC    RTS               ;END
```

EXPLANATION:

You may have wondered why byte-oriented I-O was not used to copy the file since this would be substantially simpler. One reason is that the line-input SVC (SVC #5) supports the line editing characters such as BACKSPACE and RUBOUT from the Console, but the byte-input SVC (SVC #3) does not. Thus using line input gives more flexibility when the input channel is assigned to the keyboard (Console). SVC number 3 (byte input) returns control to the application program immediately when a key is depressed; SVC number 5 does not return until an entire line terminated by a carriage return is entered. The edited line is returned to the user program in the buffer pointed to by U5, and the number of characters in the line is returned in the 6502 A register. This count does not include the carriage return delimiter.

The Example program starts by enabling SVCs and setting U5 and U6 to the addresses of the system line buffers, using the SVC 12 function. An SVC 5 is then used to input the source input line into the buffer addressed by U5, and End-ofFile is tested as before. Note that the SVC 5 function returns the character count in A, and Y is set to 0 (therefore ready to index the first character of the line). The character count of the line is saved in a temporary variable. The line is then copied from the input buffer to the output buffer. The output buffer is then output over channel 6 with the character count in the Y register.

An alternative to copying the input buffer's contents to the output buffer would simply be to copy the pointer in U5 to U6. Normally, however, you will want to use separate input and output buffers since you will be performing other operations on the line anyway.

Example Program 5: Read Hexadecimal Input Value.

Looking in Table 5-1, you may be surprised to find no direct way to input or output numeric values. Instead, a combination of two SVCs must be used to perform this function. This turns out to be a great deal more versatile. A pair of definitions are needed to get us started:

Decoding is the operation of scanning a string of ASCII characters and returning the numeric value they represent.

Encoding is the inverse operation; encoding accepts a (binary) value and returns the string of ASCII characters representing its value.

For example the ASCII string " 010B " when decoded returns the binary value 0000000100001011 ($010B), assuming that hexadecimal decoding was selected. The following problem illustrates how to input and decode a hex value.

PROBLEM: Write a subroutine which reads a hexadecimal number from channel 5 and returns its value in P-register U0.

SOLUTION:

```
SVCENB    =       $EE
;
HEXIN     LDA     #$80
          STA     SVCENB  ;MAKE SURE SVCS ARE ENABLED
          SVC     12      ;SVC 12 = GET BUFFER ADDRESSES
          LDX     #5      ;CHANNEL 5 FOR INPUT
          SVC     5       ;INPUT LINE
          SVC     8       ;DECODE INPUT LINE
          RTS
```

EXPLANATION:

The first thing you may notice about the program above is the "SVC" mnemonic. The MTU assembler has a built-in mnemonic for handling SVCs in this manner. If you are using a different assembler, use BRK and .BYTE mnemonics instead as shown for previous examples.

The enabling of SVCs and selection of the System buffers should be familiar by now. In practice, these functions would probably be performed only once during program initialization, and would not be included in this subroutine, thus reducing the subroutine to six lines. The SVC 5 operation inputs a line into the buffer addressed by P-register U5, as previously seen. The SVC 8 function searches the buffer (starting with the character indexed by Y, which was 0 in our case since SVC 5 always returns Y=0) for a character string representing a hex value. Note that any number of leading blanks may preceed the number, and the number may have any number of characters, so long as the represented value does not exceed $FFFF. For example, "00D7 ", " 0D7 " and "D7" will all be acceptable. SVC 8 keeps scanning until a non-hex character is encountered. Thus, for example, " 2B7,2 " will return U0 = $02B7, because the comma will terminate the scan. When control is returned to the calling program, the Y register points to the delimiter (the comma in the example immediately above), and the A register holds the delimiter encountered. This is very useful when scanning a line containing multiple values. In addition, the carry flag is returned to the calling program as a "Valid Data Enocuntered" flag. Although the example program above did not do so, it is easy for the application program to check the status of the carry upon completion of SVC 8; if it is not set, then no valid hex digits were encountered prior to the delimiter (or end-of-line). The end-of-line delimiter is $0D.

MISCELLANEOUS CONSIDERATIONS WHEN USING SVCS

The example programs presented have used the system input and output line buffers. In practice, during program generation and debugging, it is advisable to use other buffers, because any interaction with the system will cause your buffers to be "wiped out" (for instance, any command you enter goes into the system input buffer). To define your own buffers merely install pointers to the buffers into U5 and U6, instead of using SVC 12. Naturally if you wish to process arguments passed on the command line, you will need to use the system buffers for that.

SVCs may only be issued by a program running in bank 0. If an SVC is attempted from any of the other banks, it will be treated as a regular BRK and control will be returned to CODOS. Programs which must run in banks other than bank 0 should arrange to have their I/O and and scanning routines reside somewhere in bank 0. There are no restrictions on the use of data banks however, and all SVCs will preserve the data bank selection. The input and output line buffers are always assumed to be in bank 0 so be careful when (Indirect),Y addressing is used to access these buffers. See Appendix I in this manual and section 4.6 in the Monomeg Single Board Computer hardware manual for more information on bank switching.

The example programs presented above should provide you with an understanding of how SVCs work. In the following section, the SVCs available are described individually. Appendix D contains a complete program using SVCs which you may want to study. You may also wish to study some of the source programs provided on the MTU-130 distribution disk, most of which use SVCs extensively. The MTU-130 Assembler manual also contains an example program using SVCs.

CHAPTER 6.

SVC DESCRIPTIONS

Note: a summary of SVCs is provided in Table 5-1, page 5-2.

SVC #0 ($00)

PURPOSE: Display register contents and return to CODOS Monitor.

ARGUMENTS: None.

ARGUMENTS RETURNED: None.

DESCRIPTION:

SVC #0 returns control to the CODOS Monitor with a display of the register contents. It is functionally equivalent to the normal BRK to the CODOS monitor with SVCs disabled.

EXAMPLE:

```
SVCENB    =       $EE
          ...
          LDA     #$80
          STA     SVCENB
          ...
          BRK
          .BYTE   0       ;DISPLAY REGS, RETURN TO MONITOR.
          ...
```

SVC #1 ($01)

PURPOSE: Return to CODOS Monitor.

ARGUMENTS: None.

ARGUMENTS RETURNED: None.

DESCRIPTION:

SVC #1 returns control to the CODOS Monitor. It has two possible advantages over simply using an RTS to return to the Monitor. First, it can be executed anywhere, even in a subroutine, provided that SVCs are enabled. Second, the value of the Program Counter (P) shown by the REG command after returning will show the address of the SVC 0; using a RTS to return to Monitor will not update the P register value shown. On the other hand, using an RTS to terminate a program has the advantage that it can then be called as a subroutine or from within a job file using SVC 13. You will probably want to use an RTS for normal program terminations and SVC 0 or SVC 1 for abnormal terminations.

EXAMPLE:

```
SVCENB    =       $EE
          ...
          LDA     $80
          STA     SVCENB
          ...
          BRK
          .BYTE   1       ;RETURN TO MONITOR.
          ...
```

NOTES:

1. The difference between SVC #0 and SVC #1 is that SVC #0 displays the register contents at the BRK, and SVC #1 does not.

SVC #2 ($02)

PURPOSE: Output inline message over channel.

ARGUMENTS:

First Byte after SVC 2 = desired channel number.

Second through Nth byte = desired ASCII message text, terminated by a zero byte ($00).

DESCRIPTION:

SVC 2 can be used to display a message at any point in a program (provided SVCs are enabled). It does not affect any registers. The message may be any length up to 254 bytes, and can contain any byte including unprintable characters, except NUL ($00), which is the message terminator. Control will be returned to the instruction immediately following the 0-byte terminator. The channel specified must be assigned to a valid device or file.

EXAMPLE:

```
SVCENB    =       $EE     ;LOCATION OF SVC ENABLE FLAG
CR        =       13      ;ASCII CARRIAGE RETURN
          ...
          LDA     #$80
          STA     SVCENB
          ...
          JSR     DOIT7
          BRK
          .BYTE   2       ;SVC #2 = OUTPUT MESSAGE...
          .BYTE   6       ;...ON CHANNEL 6
          .BYTE   CR,'SUB. DOIT7 DONE, CALLING DOIT8.',0
          JSR     DOIT8
          ...
```

This program segment will output this message to channel 6:

SUB. DOIT7 DONE, CALLING DOIT8.

NOTES:

1. Attempting to output more than 254 bytes will cause the program to "hang".

2. The message will always be displayed starting at the present position. If the message should start on a new line, then the carriage return should be explicitly included, as in the example above.

3. Be careful to check that you have not forgotten the CHANNEL NUMBER argument before the message, or the 0-BYTE TERMINATOR after the message!

SVC #3 ($03)

PURPOSE: Input byte from channel.

ARGUMENTS:

X = desired channel number.

ARGUMENTS RETURNED:

A = byte received from channel.

Flags: CY (carry) = 1 means End-of-File was encountered.

DESCRIPTION:

SVC 3 inputs a single byte from a selected channel, which must be assigned to a valid device or file. The value of the byte returned can be anything, including control characters ($00 to $FF), if the selected channel is assigned to a file. If assigned to a normal, character-oriented input device, such as the keyboard, then a CNTRL-Z (ASCII SUB, $1A) will be interpreted as End-of-File. For files, End-of-File is true only when no more bytes can be read from the file. It is the programmer's responsibility to check the status of the Carry after every SVC 3 to ensure that End-of-File was not reached. The A register is not meaningfully returned if the Carry is set.

EXAMPLE:

```
SVCENB  =       $EE     ;ADDRESS OF SVC-ENABLE FLAG
        ...
        LDA     #$80
        STA     SVCENB  ;ENABLE SVCS.
        ...
        LDX     #5      ;SELECT CHANNEL 5
        BRK
        .BYTE   3       ;SVC #3 = INPUT BYTE ON CHANNEL (X)
        BCS     EOFHI   ;BRANCH IF END-OF-FILE
        CMP     #'C'    ;WAS INPUT CHARACTER 'C'?
        ...
```

This program segment inputs a character from the file or device assigned to channel 5 and checks to see if it was an ASCII "C".

NOTES:

1. The remaining flags (other than CY) are not meaningfully returned; in any case, the decimal mode flag will not be set.

2. Any value byte can be input including $00, $08, $7F, $FF, etc. No editing characters are recognized.

3. For applications requiring high-speed disk input of large amounts of data, SVC 15 is preferred to SVC 3. Since the SVC processor is called for every byte of input using SVC 3, the overhead involved limits throughput to less than 1,000 bytes per second. SVC 15 is capable of throughput in excess of 15,000 bytes per second.

SVC #4 ($04)

PURPOSE: Output byte over channel.

ARGUMENTS:

X = Channel desired.

A = Byte to be output.

ARGUMENTS RETURNED:

FLAGS: CY = 1 if at End-of-File after output operation.

DESCRIPTION:

SVC 4 outputs the byte in the accumulator over the channel specified in the X register. The channel must be assigned to a valid file or device. Although there is no need to do so, application programs may wish to test the Carry flag after SVC 4 to distinguish whether the character written was the last character of the file or was re-written over some other part of the file. If the channel is assigned to a device instead of a file, the Carry will always be returned set, since End-of-File has no meaning in this context.

EXAMPLE:

```
SVCENB    =       $EE      ;ADDRESS OF SVC ENABLE FLAG FOR SYSTEM
          ...
          LDA     #$80
          STA     SVCENB   ;ENABLE SVCS
          ...
          LDA     #$07     ;BYTE DESIRED TO OUTPUT
          LDX     #2       ;CHANNEL 2
          BRK
          .BYTE   4        ;SVC 4 = OUTPUT BYTE
          JMP     THERE
          ...
```

This program segment outputs $07 over channel 2. Note that $07 is not the character "7" but simply a byte with value 7. If channel 2 is assigned to the MTU-130 console display, this will sound a short beep thrcugh the speaker since $07 is the ASCII BEL control character.

NOTES:

1. The value $00 (NUL) can be output using SVC 4, as can any other possible 8-bit code.

2. For applications requiring high-speed disk output of large amounts of data, SVC 16 is preferred to SVC 4. Since the SVC processor is called for every byte of output using SVC 4, the overhead involved limits throughput to less than 1,000 bytes per second. SVC 16 is capable of throughput in excess of 15,000 bytes per second.

3. After you have finished writing to a file, it is a good practice to FREE the channel. This ensures that the system buffer for that file will be "flushed" to disk and that the directory will be updated. Otherwise, the actual disk contents will not be updated until you CLOSE the disk or change the file position.

SVC #5 ($05)

PURPOSE: Input line of text from channel.

ARGUMENTS:

X = Channel number to read from.

U5 = Pointer to desired input buffer for line.

ARGUMENTS RETURNED:

A = Count of characters in line.

Y = 0.

Flags: CY = 1 if End-of-File was encountered immediately.

DESCRIPTION:

SVC 5 inputs a line of text from the file or device. The text will be deposited in a buffer whose address is specified in U5. The line of text will be terminated by a CR ($0D). After the SVC is processed, the Carry will be set only if no characters could be read from the channel because End-of- File was encountered. The A register will contain a character count for the input line. This count does not include the $0D terminator. The Y register is always returned as 0 to facilitate user processing of the line using Indirect, Y addressing. End-of-Line is defined as the first carriage return ($0D) encountered.

EXAMPLE:

```
SVCENB  =       $EE     ;LOCATION OF SVC ENABLE FLAG
U5      =       $BA     ;P-REGISTER U5 LOCATION
        ...
        LDA     #$80
        STA     SVCENB  ;ENABLE SVCS
        LDA     #$00
        STA     U5
        LDA     #$10
        STA     U5+1    ;DEFINE BUFFER ADDRESS AS $1000.
        ...
        LDX     #5      ;CHANNEL 5
        BRK
        .BYTE   5       ;SVC 5 = INPUT LINE FROM CHAN. (X).
        BCS     EOFHI   ;BRANCH IF END-OF-FILE
        STA     NCHLN   ;ELSE SAVE COUNT OF CHARACTERS IN LINE
        ...
```

This program segment inputs a line of text from channel 5 and places it in a buffer starting at address $1000.

NOTES:

1. The system maintains a "Maximum Input Record Length" parameter for text input, which has a default value of 192 ($C0) characters. If an SVC 5 attempts to input a line with more than 192 characters, then the system will automatically add an end-of-line character after 192 characters are read. This prevents SVC 5 from wiping out all of memory if the channel is inadvertently assigned to a non-text file which does not contain end-of-line terminators. The value of the Maximum Record Length parameter can be altered if it is necessary to read lines of greater than 192 characters (see Appendix E). The system buffers are only 192 characters long, however, so the user will have to provide a buffer elsewhere and not use SVC 12 to define the buffer address.

2. When SVC 5 is used to read a channel assigned to the Console, all normal system editing characters (such as RUBOUT, CTRL-B, etc.) will be in effect. If an attempt is made to input more than 192 characters from the Console, a short beep will be sounded and no more characters will be accepted for insertion. This affords the user the chance to backup and change the line, perhaps to make it fit in the 192-character limit.

3. No editing characters are recognized when reading from any other device or file other than the Console. Therefore if you copy lines with embedded control characters using SVC 5 and SVC 6, these characters will not be corrupted.

4. When maximum throughput is essential for reading disk files, you may wish to use SVC 15 instead of SVC 5. If you use SVC 15 to read a large block from disk and then remove lines from the buffer individually as needed, throughput will normally be substantially enhanced compared with using individual calls to SVC 5 to read every line.

SVC #6 ($06)

PURPOSE: Output line of text on channel.

ARGUMENTS:

X = Channel desired.

Y = Number of characters in line.

U6 = Starting address of line of text.

ARGUMENTS RETURNED: None.

DESCRIPTION:

SVC 6 outputs a line of text over a channel which is assigned to a valid file or device. U6 must contain a pointer to a buffer containing the text to be sent. The Y register must hold the number of characters to be sent, not including the line terminator.

EXAMPLE:

```
SVCENB    =       $EE      ;LOCATION OF SVC-ENABLE FLAG
U6        =       $BC      ;LOCATION OF P-REGISTER U6
          ...
          LDA     #$80
          STA     SVCENB   ;ENABLE SVCS
          ...
          LDA     PROD
          STA     U6       ;DEFINE ADDRESS OF TEXT TO BE SENT
          LDA     PROD/256
          STA     U6+1
          LDX     #6       ;CHANNEL 6
          LDY     #11      ;11 CHARACTERS IN LINE
          BRK
          .BYTE   6        ;SVC 6 = OUTPUT LINE
          ...
PROD      .BYTE   'DISK SYSTEM'
          ...
```

This program segment will output "DISK SYSTEM" followed by an end-of-line character (CR) on channel 6.

NOTES:

1. The line to be output cannot exceed 254 characters. If the system output buffer is used, the programmer must not fill the buffer with more than 192 characters. Failure to limit the amount put into the system output buffer will cause memory above the system buffer to be wiped out.

2. The character count must be passed in Y. This is normally convenient since if you advance Y after each character is installed in the buffer, it will automatically contain the character count. Also SVCs which perform encoding of numeric values automatically return Y as the character count.

SVC #7 ($07)

PURPOSE: Output string of text on channel.

ARGUMENTS:

X = Channel desired.

Y = Number of characters in string.

U6 = Starting address of string of text.

ARGUMENTS RETURNED: None.

DESCRIPTION:

SVC 7 outputs a string of text over a channel which is assigned to a valid file or device. U6 must contain a pointer to a buffer containing the text to be

sent. The Y register must hold the number of characters to be sent.

EXAMPLE:

```
SVCENB    =       $EE      ;LOCATION OF SVC-ENABLE FLAG
U6        =       $BC      ;LOCATION OF P-REGISTER U6
          ...
          LDA     #$80
          STA     SVCENB   ;ENABLE SVCS
          ...
          LDA     PROD
          STA     U6       ;DEFINE ADDRESS OF TEXT TO BE SENT
          LDA     PROD/256
          STA     U6+1
          LDX     #6       ;CHANNEL 6
          LDY     #11      ;11 CHARACTERS IN LINE
          BRK
          .BYTE   7        ;SVC 7 = OUTPUT STRING
          ...
PROD      .BYTE   'DISK SYSTEM'
          ...
```

This program segment will output "DISK SYSTEM". NO End-of-line character will be added by the system.

NOTES:

1. The text to be output cannot exceed 254 characters.

2. The difference between SVC #6 and SVC #7 is that SVC #6 outputs a carriage return at the end of the string, and SVC #7 does not. Naturally carriage returns can be embedded in the string itself if desired.

SVC #8 ($08)

PURPOSE: Decode hexadecimal ASCII string to 16-bit value.

ARGUMENTS:

Y = Index to first character of string to be decoded.

U5 = Pointer to the string of ASCII characters.

ARGUMENTS RETURNED:

A = Delimiting character encountered.

Y = Index to delimiting character.

Flags: CY = 1 if at least one valid hex digit was encountered prior to the delimiter.

U0 = Value returned (in normal low-byte, high-byte order).

DESCRIPTION:

SVC 8 scans a string of characters in memory and returns the numeric value represented by the string. The string must represent a hexadecimal number.

EXAMPLE:

```
SVCENB  =       $EE             ;LOCATION OF SVC ENABLE FLAG
U0      =       $B0             ;PSEUDO-REG. U0
U5      =       $BA             ;PSEUDO-REG. U5
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        ...
        LDA     #$00
        STA     U5              ;SET U5 TO ADDRESS OF START OF STRING
        LDA     #$10
        STA     U5+1
        LDY     #0              ;START AT 1ST CHARACTER IN STRING
        BRK
        .BYTE   8               ;SVC 8 = DECODE HEX
        BCC     ERROR           ;BRANCH IF NO LEGAL HEX NUMBER FOUND
        ...
```

This program segment decodes a string of characters starting at $1000 and returns the value in U0. If the contents of memory starting at $1000 was "02B " ($30, $32, $42, $20), then at the end of the program segment, A = $20, Y = $03, and the carry flag is set. Memory location $00B0 (U0) = $2B, and $00B1 = $00.

NOTES:

1. The string to be decoded may contain any number of leading blanks or zeroes.

2. The hex number must be unsigned.

3. The decoding process halts as soon as a delimiter is encountered. Any non-hex character is a delimiter, including a blank.

4. If no valid hex characters are encountered prior to the delimiter, the carry is cleared and U0 = 0. This provides the user with the option of either accepting a blank field as a zero entry or rejecting it as invalid.

5. See Chapter 5 Example Program 5 for a complete example of hexadecimal input.

SVC #9 ($09)

PURPOSE: Decode decimal ASCII string to 16-bit value.

ARGUMENTS:

Y = Index to first character of string to be decoded.

U5 = Starting address of string of ASCII characters.

ARGUMENTS RETURNED:

A = Delimiting character encountered.

Y = Index to delimiting character.

Flags: CY = 1 if at least one valid digit was encountered prior to the delimiter.

U0 = Value returned (in normal low-byte, high-byte order).

DESCRIPTION:

SVC 9 scans a string of characters in memory and returns the numeric value represented by the string. The string must represent a decimal integer.

EXAMPLE:

```
SVCENB  =       $EE             ;LOCATION OF SVC ENABLE FLAG
U0      =       $B0             ;PSEUDO-REG. U0
U5      =       $BA             ;PSEUDO-REG. U5
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        ...
        LDA     #$00
        STA     U5              ;SET U5 TO ADDRESS OF START OF STRING
        LDA     #$10
        STA     U5+1
        LDY     #2              ;START AT 3RD CHARACTER IN STRING
        BRK
        .BYTE   9               ;SVC 9 = DECODE DECIMAL
        BCC     ERROR           ;BRANCH IF NO LEGAL DECIMAL NUMBER FOUND
        ...
```

This program segment decodes a string of characters starting at the third character in a string located at address $1000 in memory. If the string at $1000 was "YZ 300,23 ", then at the end of the program segment, A = $2C (","), Y = 6, and the carry is set. Memory location $00B0 (U0) contains $2C and $00B1 contains $01, since 300 decimal is 012C hex.

NOTES:

1. The string being decoded may contain any number of leading blanks or zeroes.

2. The decimal number must be unsigned.

3. The decoding process halts as soon as a delimiter is encountered. Any non-digit character is a delimiter, including a blank.

4. If no valid digits are encountered prior to the delimiter, the carry is cleared and U0 = 0. This provides the user with the option of either accepting a blank field as a zero entry or rejecting it as invalid.

SVC #10 ($0A)

PURPOSE: Encode 16-bit value to hexadecimal ASCII string.

ARGUMENTS:

Y = Index to byte in buffer to receive first character encoded.

U0 = Value to be encoded.

U6 = Pointer to buffer.

ARGUMENTS RETURNED:

Y = Index to byte after last character of hex number (Y returned = Y passed+4).

DESCRIPTION:

SVC 10 encodes the unsigned value in U0 into four hex characters starting at the memory location addressed by (U6),Y.

EXAMPLE:

```
SVCENB   =        $EE           ;ADDRESS OF SVC ENABLE FLAG
U0       =        $B0           ;ADDRESS OF USER P-REG U0 (VALUE)
         ...
ANSWER   *=*+2                  ;16-BIT VALUE TO BE OUTPUT
         ...
HEXOUT   LDA      #$80
         STA      SVCENB        ;ENABLE SVCS
         LDA      ANSWER
         STA      U0            ;COPY ANSWER TO U0 (LOW BYTE)...
         LDA      ANSWER+1
         STA      U0+1          ;...AND HI BYTE.
         BRK
         .BYTE    12            ;SVC 12 = GET LOCATION OF SYSTEM BUFFERS
         LDY      #0            ;START AT 1ST CHARACTER OF BUFFER
         BRK
         .BYTE    10            ;SVC 10 ($0A) = ENCODE U0 TO 4 ASCII CHARS.
         LDX      #6            ;CHANNEL 6 FOR OUTPUT
         BRK
         .BYTE    6             ;SVC 6 = OUTPUT LINE TO CHANNEL
         ...
```

This program segment displays the hexadecimal value of the contents of ANSWER on channel 6. If ANSWER contained $3B and ANSWER+1 contained $0A, and channel 6 was assigned to the Console, then "0A3B" would be displayed, followed by a carriage return.

SVC #11 ($0B)

PURPOSE: Encode 16-bit value to decimal ASCII string.

ARGUMENTS:

Y = Index to byte in buffer to receive first character encoded.

U0 = Value to be encoded.

U6 = Pointer to buffer.

ARGUMENTS RETURNED:

Y = Index to byte after last character of decimal number.

DESCRIPTION:

SVC 11 encodes the unsigned value in U0 into a decimal ASCII string starting at the memory location addressed by (U6),Y.

EXAMPLE:

```
SVCENB   =        $EE           ;ADDRESS OF SVC ENABLE FLAG
U0       =        $B0           ;ADDRESS OF USER P-REG U0 (VALUE)
         ...
ANSWER   *=*+2                  ;16-BIT VALUE TO BE OUTPUT
         ...
HEXOUT   LDA      #$80
         STA      SVCENB        ;ENABLE SVCS
         LDA      ANSWER
         STA      U0            ;COPY ANSWER TO U0 (LOW BYTE)...
         LDA      ANSWER+1
         STA      U0+1          ;...AND HI BYTE.
         BRK
         .BYTE    12            ;SVC 12 = GET LOCATION OF SYSTEM BUFFERS
         LDY      #8            ;START AT 9TH CHARACTER OF BUFFER
         BRK
         .BYTE    11            ;SVC 11 ($0B) = ENCODE U0 TO ASCII CHARS.
         LDX      #6            ;CHANNEL 6 FOR OUTPUT
         BRK
         .BYTE    6             ;SVC 6 = OUTPUT LINE TO CHANNEL
         ...
```

This program segment displays the first 8 characters of the system output buffer and the decimal value of the contents of ANSWER on channel 6. Suppose ANSWER contained $3B and ANSWER+1 contained $0A, channel 6 was assigned to the Console, and the first 8 bytes of the system output buffer contained "ANSWER =". The Console would then display, "ANSWER =2619", followed by a carriage return.

NOTES:

1. The encoded string will have from 1 to 5 characters, depending on the magnitude of the value in U0.

SVC #12 ($0C)

PURPOSE: Obtain location of system input line buffer, output line buffer, and arguments passed to user-defined command.

ARGUMENTS: None.

ARGUMENTS RETURNED:

Y = Index to first argument passed.

U5 = Pointer to System Input-line buffer.

U6 = Pointer to System Output-line buffer.

DESCRIPTION:

User-defined programs may process arguments passed in the same manner as for CODOS built-in commands, by using SVC 12. In addition, SVC 12 returns the starting address of the CODOS text input buffer and output buffer, which may be used by the program for input-output.

EXAMPLE:

```
SVCENB   =       $EE          ;ADDRESS OF SVC-ENABLE FLAG
U5       =       $BA          ;P-REG. U5
         ...
         *=      $2000        ;SAMPLE PROGRAM ORIGIN
MYCOMD   LDA     #$80
         STA     SVCENB       ;ENABLE SVCS
         ...
         BRK
         .BYTE   12           ;SVC 12 ($0C) = GET BUFFERS, ARGUMENT POINTER
         LDA     (U5),Y       ;FETCH FIRST CHARACTER OF ARGUMENT
         CMP     #$0D         ;TEST IF CARRIAGE RETURN
         BEQ     NOARG        ;BRANCH IF END-OF-LINE (NO ARGUMENT)
         CMP     #';'
         BEQ     NOARG        ;BRANCH IF COMMENT (NO ARGUMENT)
         CMP     #'M'
         BEQ     MONDAY       ;BRANCH IF FIRST CHAR IS "M"
         CMP     #'T'
         BEQ     TUETHR       ;BRANCH IF FIRST CHAR IS "T"
         ...
```

This program segment sets U5 to the location of the CODOS input buffer and U6 to the location of the CODOS output buffer. The System Input Buffer on entry to a program always contains the CODOS command which initiated the program. SVC 12 returns Y as a pointer to the first non-blank character following the command. This allows a program to process arguments. For example, if the above program segment was initiated by the CODOS command:

```
MYCOMD  T
```

then (U5),Y addresses "T", and the program would branch to TUETHR (not shown). If the program was entered by

```
GO 2000  M
```

then the program would branch to MONDAY instead.

SVC #13 ($0D)

PURPOSE: To execute any CODOS Monitor command.

ARGUMENTS:

U5 = pointer to command in memory. Command must be terminated by CR ($0D).

ARGUMENTS RETURNED:

All registers and Pseudo-registers are returned as set by the Monitor command executed (see note 1 below). Registers not changed by the command are unaffected.

DESCRIPTION:

SVC #13 ($0D) is the most powerful of all SVCs provided. Creatively used, it can give tremendous leverage to an application program. Simply stated, SVC #13 calls the CODOS Monitor as a subroutine, with the command read from memory instead of from channel 1 (normally the Console). Each invocation of SVC #13 will execute one CODOS Monitor command and then return to the invoking program in the normal manner. Thus a program can easily OPEN or CLOSE drives, FILL or COPY memory, GET, SAVE or TYPE files, etc. Utilities and User-defined programs may also be executed in the usual manner. This provides a way to chain programs together, load overlays, selectively execute certain programs based on computed results, etc. To use SVC #13, the desired command must exist as an ASCII string in memory, terminated by a Carriage Return ($0D), and P-register U5 must contain the address of the start of this command string.

EXAMPLE:

```
SVCENB  =       $EE
U5      =       $BA
        ...
        *=      $2100           ;LOCATION OF COMMAND TO BE EXECUTED
        .BYTE   'PROGTWO',$0D   ;EXECUTE PROGRAM 2 COMMAND
        ...
OVRLA2  LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        ...
        LDA     #00
        STA     U5              ;DEFINE ADDRESS OF COMMAND IN U5
        LDA     #$21
        STA     U5+1
        BRK
        .BYTE   13              ;EXECUTE MONITOR COMMAND AT $2100 ("PROGTWO")
        BRK
        .BYTE   2               ;DISPLAY INLINE MESSAGE
        .BYTE   2               ;ON CHANNEL 2
        .BYTE   $D,'PROGRAM 2 EXECUTION COMPLETE.',0
        ...
```

This program segment executes the Monitor command "PROGTWO", which loads and executes the User-defined program called PROGTWO.C from the default drive. When PROGTWO returns (by executing an RTS), it will effectively return to the program segment above instead of to the CODOS Monitor. The message "PROGRAM 2 EXECUTION COMPLETE" is then displayed using SVC #2.

NOTES:

1. Since the return path to the invoking program is stored on the stack, the Monitor command executed must not redefine the stack pointer (except by normal usage of balanced JSR, RTS, pushes and pops, etc., of course). Therefore the REG command cannot be used with "S=n" as an argument.

2. When executing a Utility program or User-defined command, it is the programmer's responsibility to ensure that no memory conflicts occur with the invoking program. Naturally, if you execute a program which occupies the same memory as the invoking program, you will wipe it out. Of course you could SAVE any conflicting memory blocks using another SVC #13, and restore them with a GET.

3. When executing a program, registers and Pseudo-registers may be used for passing arguments in either direction (to and from the program being executed), if desired.

4. SVC #13's can be nested up to seven deep. That is, a program can execute another program which in turn uses SVC #13 to execute other commands or programs.

5. A program invoked using SVC #13 will return to the CODOS Monitor and not to the invoking program if an SVC #0 or SVC #1 is used. Only an RTS instruction can be used to return to the invoking program.

SVC #14 ($0E)

PURPOSE: To determine the channel assignment for a selected channel.

ARGUMENTS:

X = Channel number desired, 0 to 9.

ARGUMENTS RETURNED:

Flags: Carry is set if the channel is assigned. Other flags undefined.

A = disk drive number if returned as 0 to 3; otherwise returned as the single-character device name. Not meaningfully returned if CY is clear.

DESCRIPTION:

SVC 14 ($0E) enables a program to determine if a specified I-O channel is assigned or available. If it is assigned, then the device or drive assigned can also be determined.

EXAMPLE:

```
SVCENB  =       $EE             ;SVC ENABLE FLAG
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        ...
        LDX     #6              ;CHANNEL 6
        BRK
        .BYTE   14              ;SVC 14 ($0E) = QUERY CHANNEL STATUS
        BCC     ISAVAL          ;BRANCH IF CHANNEL 6 IS UNASSIGNED
```

```
        CMP     #4
        BCC     ISFILE          ;BRANCH IF ASSIGNED TO FILE
        CMP     #'N'            ;"N" = NULL DEVICE NAME
        BEQ     ISNULL          ;BRANCH IF ASSIGNED TO NULL DEVICE
        ...
```

This program segment tests the current assignment of channel 6. If it is unassigned, the program branches to ISAVAL (not shown). If the channel is assigned to a file on disk, it branches to ISFILE. Otherwise, the channel must be assigned to a device, in which case the device name is checked and a branch to ISNULL is made if the Null device is assigned.

SVC #15 ($0F)

PURPOSE: To read a record from a channel.

ARGUMENTS:

X = Channel number desired, 0 to 9. Must already be assigned to a valid file or input device.

U1 = Starting address to receive contents of record in the currently selected data bank.

U2 = Size of record to be read, in bytes.

ARGUMENTS RETURNED:

U1 = Address of last byte read, plus one.

U2 = Actual number of bytes read.

Flags: If Carry is set then End-of-File was encountered before any bytes could be read. All other flags are not meaningfully returned.

DESCRIPTION:

SVC 15 ($0F) reads a block of bytes from a channel.

EXAMPLE:

```
SVCENB  =       $EE             ;SVC ENABLE FLAG ADDRESS
U1      =       $B2             ;P-REGISTER U1
U2      =       $B4             ;P-REGISTER U2
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        ...
        LDA     #$00
        STA     U1              ;DEFINE STARTING ADDRESS FOR RECORD = $2000...
        LDA     #$20
        STA     U1+1
        LDA     #$20            ;DEFINE RECORD SIZE AS 800 DECIMAL BYTES=$0320
        STA     U2
        LDA     #$03
        STA     U2+1
        LDX     #5              ;CHANNEL 5
```

```
        BRK
        .BYTE   15              ;SVC 15 (0F) = READ RECORD INTO MEMORY
        BCS     DONE            ;BRANCH IF END-OF-FILE
        ...
```

This program segment reads 800 bytes into memory from channel 5, starting at address $2000.

NOTES:

1. If the specified channel is assigned to a file, then reading begins at the current file position and continues until the specified number of bytes are read or End-of-File is encountered. Any type of data bytes may be read; there are no reserved End-of-Record or End-of-File characters.

2. If the channel is assigned to a device (not a file), then reading continues until the specified number of bytes are read or the End-of-File character (ASCII SUB = $1A = CNTRL-Z) is read. The CNTRL-Z is not returned as part of the record in memory.

3. If the Carry flag is returned set then no bytes at all could be read from the channel because End-of-File was encountered immediately.

4. If the Carry flag is returned clear then at least 1 byte was read before End-of- file. The actual number of bytes read is returned in U2. If U2 contains a smaller count than was specified but the Carry is returned clear, it indicates that End-of-File was encountered during the read operation.

5. Remember that the record is read into the memory bank that corresponds to the data bank selected when SVC #15 is executed. Normally this is bank 0.

6. When reading large amounts of data from a file, using large records will significantly improve the reading speed. For example, if a program is to read 1000 80-character records, the obvious way to do it is to use a loop which invokes SVC #15 1000 times with U2 set to 80. However, a significant speed improvement can be realized by instead using, say, 100 SVCs of 800 bytes each, if sufficient memory is available for an 800 byte buffer. By using large records it is possible to read in excess of 15,000 bytes per second continuous throughput from a file.

SVC #16 ($10)

PURPOSE: To write a record to a channel.

ARGUMENTS:

X = Channel number desired, 0 to 9. Must already be assigned to a valid file or input device.

U1 = Starting address of record in memory in currently selected data bank.

U2 = Size of record to be written, in bytes.

ARGUMENTS RETURNED:

Flags: If Carry is set then the channel was positioned at End-of-File after completing the write.

DESCRIPTION:

SVC 16 ($10) writes a block of bytes in memory to a channel.

EXAMPLE:

```
SVCENB  =       $EE             ;SVC ENABLE FLAG ADDRESS
U1      =       $B2             ;P-REGISTER U1
U2      =       $B4             ;P-REGISTER U2
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        ...
        LDA     #$00
        STA     U1              ;DEFINE STARTING ADDRESS FOR RECORD = $2000...
        LDA     #$20
        STA     U1+1
        LDA     #$20            ;DEFINE RECORD SIZE AS 800 DECIMAL BYTES=$0320
        STA     U2
        LDA     #$03
        STA     U2+1
        LDX     #6              ;CHANNEL 6
        BRK
        .BYTE   16              ;SVC 16 ($10) = WRITE RECORD
        ...
```

This program segment writes a record on channel 6. The record to be written is 800 decimal bytes long and starts at $2000 in memory.

NOTES:

1. If the selected channel is assgined to a disk file, then writing begins at the current file position. Any type of data bytes may be written; no special End-of-Record characters will be written by the system. If the CY is returned clear, it indicates that the the file was not positioned to End-of-File on the completion of the write operation (therefore part of the file must have been overwritten).

2. If the specified channel is assigned to a device, then writing continues to that device until the specified number of bytes has been output. The CY flag is always returned set when writing to a device.

3. Using large records will improve writing speed. For example, writing 100 records of 80 bytes each takes longer than writing 10 records of 800 bytes each. Continuous output to disk in excess of 15,000 bytes per second is possible by using large records.

4. After you have finished writing to a file, it is a good practice to FREE the channel. This insures that the system buffer for that file will be "flushed" to disk and that the directory will be updated. Otherwise, the actual disk contents will not be updated until you CLOSE the disk or change the file position.

5. Remember that the record is written from the memory bank that corresponds to the data bank selected when SVC #16 is executed.

SVC #17 ($11)

PURPOSE: To set the file position for a channel to Beginning-of-Data.

ARGUMENTS:

X = Channel number desired, 0 to 9. Must already be assigned to a valid file or device.

ARGUMENTS RETURNED: None.

DESCRIPTION:

After executing SVC 17 ($11), a subsequent read or write operation will access the first data byte of the file assigned to the specified channel.

EXAMPLE:

```
SVCENB  =       $EE             ;SVC ENABLE FLAG ADDRESS
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        LDX     #5              ;SELECT CHANNEL 5
        BRK
        .BYTE   17              ;SVC 17 ($11) = "REWIND" THE FILE
        ...
```

This program segment positions the file assigned to channel 5 to Beginning-of-Data.

NOTES:

1. If the selected channel is assigned to a device instead of a file, no action takes place.

2. A file is always initially positioned to Beginning-of-Data when it is assigned.

3. Executing SVC 17 will always result in a physical disk access, even if the file is already positioned at Beginning-of-Data.

SVC #18 ($12)

PURPOSE: To set the file position for a channel to End-of-File.

ARGUMENTS:

X = Channel number desired, 0 to 9. Must already be assigned to a valid file or device.

ARGUMENTS RETURNED: None.

DESCRIPTION:

SVC 18 ($12) positions the file assigned to the specified channel to End-of file. A subsequent write operation would therefore append the file.

EXAMPLE:

```
SVCENB  =       $EE             ;SVC ENABLE FLAG ADDRESS
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        LDX     #6              ;SELECT CHANNEL 6
        BRK
        .BYTE   18              ;SVC 18 ($12) =  MOVE TO END-OF-FILE
        ...
```

This program segment positions the file assigned to channel 6 to End-of-File.

NOTES:

1. If the selected channel is assigned to a device instead of a file, no action takes place.

2. A file is always initially positioned to Beginning-of-Data when it is assigned to a channel.

3. Executing SVC 18 will always result in a physical disk access, even if the file is already positioned at End-of-File.

SVC #19 ($13)

PURPOSE: To specify the file position for a channel.

ARGUMENTS:

X = Channel number desired, 0 to 9. Must already be assigned to a valid file or device.

U7 = Desired file position (3 bytes).

ARGUMENTS RETURNED: None.

DESCRIPTION:

SVC 19 ($13) positions the file assigned to the specified channel to the position specified by U7.

EXAMPLE:

```
SVCENB  =       $EE             ;SVC ENABLE FLAG ADDRESS
U7      =       $BE             ;P-REG U7 (3 BYTES LONG)
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        LDA     #72
        STA     U7              ;SELECT POSITION TO ACCESS 73RD BYTE OF FILE...
        LDA     #0
        STA     U7+1
        STA     U7+2
        LDX     #6              ;SELECT CHANNEL 6
        BRK
        .BYTE   19              ;SVC 19 ($13) = POSITION CHANNEL TO (U7)
        ...
```

This program segment positions the file assigned to channel 6 to $000048. A subsequent read or write will begin at this position.

NOTES:

1. If the selected channel is assigned to a device instead of a file, no action takes place.

2. A file is always initially positioned to Beginning-of-Data when it is assigned to a channel.

3. Executing SVC 19 will always result in a physical disk access, even if the file is already positioned at the location specified by U7.

4. If the position specified by U7 is beyond the current End-of-File, the file will be postioned to the current End-of-file.

5. The 16-bit arithmetic Pseudo-Processor (SVC #27) provides a very simple method of computing 24-bit file positions given a 16-bit record size and a 16-bit record number. See Chapter 7 for details.

SVC #20 ($14)

PURPOSE: To determine the position of a file assigned to a channel.

ARGUMENTS:

X = Channel number desired, 0 to 9. Must already be assigned to a valid file or device.

ARGUMENTS RETURNED:

U7 = File position (3 bytes)

DESCRIPTION:

SVC 20 ($14) returns the present file position for the specified channel in U7.

EXAMPLE:

```
SVCENB  =       $EE             ;SVC ENABLE FLAG ADDRESS
U7      =       $BE             ;P-REG U7 (3 BYTES LONG)
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        ...
        LDX     #8              ;CHANNEL 8
        BRK
        .BYTE   20              ;SVC 20 ($14) = QUERRY FILE POSITION
        ...
```

This program segment sets U7 to the present position of the file assigned to channel 8. For example, if 82,965 bytes had been read thus far starting at Beginning-of-Data, then $BE = $15, $BF = $44, and $C0 = $01.

NOTES:

1. If the specified channel is assigned to a device (not a disk file), then U7 is always returned as $000000.

SVC #21 ($15)

PURPOSE: To assign a channel to a device or file.

ARGUMENTS:

A = Disk drive number, 0 to 3, or single-character device name to be assigned.

X = Channel number desired, 0 to 9.

U3 = Pointer to file name in memory (applies to assignment to file only).

ARGUMENTS RETURNED:

A = status byte as follows:

Bit 6 = File flag. If not set, then channel assigned to device, not file.
Bit 7 = Old flag. If set, then file already exists.
Bit 5 = Locked flag. If set, then file is locked (read-only).

Flags: Sign flag and Overflow flag reflect value of bits 7 and 6 respectively of status byte as described above.

DESCRIPTION:

SVC 21 ($15) assigns the channel specified by X to the device or disk drive specified by A.

EXAMPLE:

```
SVCENB   = '     $EE          ;SVC ENABLE FLAG ADDRESS
U3       =       $B6          ;P-REG U6
         ...
FILENM   .BYTE   'MYDATA5.D'  ;DESIRED FILE NAME
         ...
         LDA     #$80
         STA     SVCENB       ;ENABLE SVCS
         ...
         LDA     #FILENM
         STA     U3           ;INSTALL POINTER TO FILE NAME IN U3
         LDA     #FILENM/256
         STA     U3+1
         LDX     #5           ;CHANNEL 5
         LDA     #1           ;DISK DRIVE 1
         BRK
         .BYTE   21           ;SVC 21 = ASSIGN CHANNEL
         BPL     NOFILE       ;BRANCH IF FILE DOES NOT ALREADY EXIST
         ...
```

This program segment assigns channel 5 to a file called MYDATA5.D on drive 1, and branches to NOFILE if it is not an old file.

NOTES:

1. U3 is not used if the channel is assigned to a device (not file).

2. The file name can be terminated by any character which is not legal in a file name. The extension may be included or omitted.

3. Assigning a channel to a file always positions the file to Beginning-of-Data.

4. Assigning a channel which is already assigned, automatically frees the old channel assignment before making the new assignment.

5. See SVC #29 for more information on file assignments.

SVC #22 ($16)

PURPOSE: To free a channel.

ARGUMENTS:

X = Channel number desired, 0 to 9.

ARGUMENTS RETURNED: None.

DESCRIPTION:

SVC 22 ($16) frees the channel specified by X.

EXAMPLE:

```
SVCENB  =       $EE             ;SVC ENABLE FLAG ADDRESS
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS
        ...
        LDX     #5              ;CHANNEL 5
        BRK
        .BYTE   22              ;SVC 22 = FREE CHANNEL
        ...
```

This program segment frees channel 5.

NOTES:

1. Freeing a channel which is unassigned results in no action.

2. It is important that programs which write to disk always free the channel when the file is completed. Otherwise, if the disk is removed from the drive by the operator without a CLOSE command, the file could be incomplete.

SVC #23 ($17)

PURPOSE: To truncate a file at the present file position.

ARGUMENTS:

X = channel number to truncate (already assigned to a file).

ARGUMENTS RETURNED: none.

DESCRIPTION:

SVC #23 makes the present file position End-of-File. It is normally used to discard the unwanted end part of a file, or to discard the unwanted residual when overwriting an exisitng file with a shorter file.

EXAMPLE:

```
SVCENB  =       $EE             ;SVC-ENABLE FLAG LOCATION
U7      =       $BE             ;P-REG U7 (3 BYTES LONG) FOR FILE POSITIONING
        ...
        LDA     $#80
        STA     SVCENB          ;ENABLE SVCS
        LDA     #0
        STA     U7              ;SET U7 TO $002000 (= 8K BYTES)
        LDA     #$20
        STA     U7+1
        LDA     #0
        STA     U7+2
        LDX     #6              ;CHANNEL 6 FILE
        BRK
        .BYTE   19              ;POSITION FILE TO 8K BYTES
        BRK
        .BYTE   23              ;TRUNCATE REST OF FILE, IF ANY
        ...
```

This program segment truncates the file to a maximum of 8K bytes of data. If the file contained less than 8K of data, it would not be changed. If it contained more than 8K of data, the rest of the file would be discarded.

NOTES:

1. If the channel is assigned to a device instead of a file, no action takes place.

SVC #24 ($18)

PURPOSE: To define the address of an interrupt service routine.

ARGUMENTS:

U0 = pointer to interrupt service routine for IRQ.

ARGUMENTS RETURNED: none.

DESCRIPTION:

SVC #24 allows a user program to use interrupts, without interfering with operation of SVCs, by defining the address of the desired interrupt-service subroutine. After executing SVC #24, any IRQ will cause control to be transferred to the user-defined service routine. BRKs will still be processed by the SVC processor in the normal manner. The following paragraph explains BRK and IRQ processing by CODOS in detail.

When CODOS is booted up, it sets the IRQ vector to point to CODOS's IRQ/BRK processor. Thereafter, when a BRK or IRQ occurs, control vectors to CODOS. CODOS tests the BRK bit in the processor status word to determine whether a BRK occurred

This modifies Location

or an IRQ occurred. If a BRK occurred, it checks the status of SVCENB, and either branches to the SVC processor or simply displays the registers and branches to the Monitor. If the processor status word indicates that an interrupt has occurred, CODOS jumps to the user's service routine. This check is very fast and only adds 24 machine cycles to the time otherwise needed to arrive at the service routine. All registers and stack are preserved exactly as they would normally be if vectoring directly to the service routine. If SVC #24 has never been used to define the address of the service routine, then the CODOS monitor will be re-entered with an "INTERRUPT (IRQ)" message.

EXAMPLE:

```
SVCENB  =       $EE             ;SVC ENABLE FLAG ADDRESS
U0      =       $B0             ;PSEUDO REGISTER U0 (2 BYTES)
        ...
        LDA     #$80
        STA     SVCENB          ;ENABLE SVCS...
        ...
        LDA     #$00
        STA     U0              ;DEFINE U0 = ADDRESS OF SERVICE ROUTINE = $2000
        LDA     #$20
        STA     U0+1
        BRK
        .BYTE   24              ;SVC 24 = DEFINE IRQ SERVICE ROUTINE ADDRESS
        CLI                     ;ENABLE INTERRUPTS
        ...
```

This program segment defines the IRQ service routine to be at $0800. Thereafter, any IRQ will cause control to vector to $2000. BRKs, however, will continue to be processed by the SVC processor in the normal manner.

NOTES:

1. A non-maskable interrupt normally causes re-entry into CODOS and a register printout. Typically, only the keyboard INT key will cause non-maskable interrupts. You may freely modify the NMI jump vector at $02FA - $02FC.

2. In rare cases, a program must respond to an interrupt so fast that it cannot tolerate even the extra 24 machine cycles of overhead used by CODOS. In this case, the program should not use SVC #24 to define the interrupt service routine location, but should modify the normal system IRQ vector directly. Naturally, once this is done, all BRKs and IRQs will go to the user's service routine, thus disabling the SVC facility. The IRQ jump vector is at $02FD-$02FF.

3. Memory location $00EC is used by CODOS for temporarily saving the A register before entering the user-defined service routine.

4. The User's interrupt service routine is entered with all registers in the same condition as they would be if the service routine was entered by direct vectoring, as described in note 2 above.

5. The interrupt service routine must reside in memory bank 0. The service routine is entered with both bank selection registers unchanged from the values they had when the interrupt was recognized. The Interrupt Mode flip-flop will be set however which cancels the effect of the program bank register. The service routine must return with an RTI instruction to properly reset this flip-flop.

6. Appendix D contains a complete program lisiting using interrupts for high-speed, direct-to-disk data aquisition.

SVC #25 ($19)

PURPOSE: To define the address of a user-defined error recovery routine.

ARGUMENTS:

U0 = pointer to desired error recovery procedure.

ARGUMENTS RETURNED: none.

DESCRIPTION:

SVC #25 Provides a method by which the advanced programmer can defeat the error handling procedure built into the CODOS Monitor. Normally, when an error in an SVC or command is detected, CODOS aborts the program, displays an error number and mesage, and returns control to the Monitor. In certain circumstances, the user may wish to temporarily bypass this error recovery. To do this, use SVC 25, with Pseudo-register U0 specifying the address of the User's machine-language error processing routine. Thereafter, any error detected by CODOS will exit to the user-defined error processor. On entry to the user-defined error processor, the registers are all undefined; memory location $00ED contains the error number which would normally be displayed by CODOS (see Appendix A). The top of the stack contains the address where the error was detected (not the address of the error!). The state of the system is undefined and usually unprotected. It is entirely the user's responsibility to take appropriate action. Usage of SVC #25 should be reserved for very special circumstances and should not be used indiscriminantly.

EXAMPLE:

```
SVCENB  =       $EE
U0      =       $B0
        ...
        LDA     #00
        STA     U0          ;SET USER ERROR PROCESSOR ADDRESS TO $8000
        LDA     #$80
        STA     U0+1
        BRK
        .BYTE   25          ;REDEFINE ERROR VECTOR
        ...
```

This program segment enables a user error-processor at $8000.

NOTES:

1. The User-defined error processor remains in effect until the system is RESET, or SVC #26 is executed.

3. Executing an RTS from the user error-processor is not an appropriate method to reenter the erring program and may crash the system.

4. One method of error recovery is:
   a. Execute SVC 25 to define your error-recovery routine.
   b. Save the stack pointer immediately before executing the SVC desired.
   c. In the error-processor, examine the error number in $00ED and then clear it to 0. If the error is not recoverable, execute SVC #26, issue your own error message, and abort to Monitor. If the error is recoverable, correct it, restore the stack, and re-execute the desired SVC.

5. The User-defined error processor must reside in bank 0.

SVC #26 ($1A)

PURPOSE: To reinstate normal error processing by CODOS.

ARGUMENTS: none.

ARGUMENTS RETURNED: none.

DESCRIPTION:

SVC #26 restores the normal error processing by CODOS after previous execution of SVC #25.

EXAMPLE:

```
SVCENB  =       $EE
        ...
        LDA     #$80
        STA     SVCENB
        BRK
        .BYTE   26          ;SVC 26 = RESTORE NORMAL CODOS ERROR-PROCESSING
        ...
```

This program segment cancels the effect of the previous SVC #25.

NOTES:

1. Executing SVC #26 without a previous SVC #25 is permissable.

SVC #27 ($1B)

PURPOSE: To enter the CODOS 16-bit Pseudo-processor.

ARGUMENTS: First through n-th bytes following the SVC are instructions for the 16-bit pseudo processor. A zero instruction (not zero byte) terminates the string.

ARGUMENTS RETURNED:

Flags are returned as described in section 7.

DESCRIPTION:

SVC #27 enters the built-in CODOS 16-bit arithmetic Pseudo-processor for performing double precision arithmetic including multiply and divide, and for computing 24-bit file positions. The operation of the Pseudo-processor is described in Chapter 7.

SVC #28 ($1C)

PURPOSE: To return information about the version of CODOS which is running.

ARGUMENTS: none.

ARGUMENTS RETURNED:

A = High address byte of the System 8K RAM on the K-1013 disk controller (Will always be $E0 on the MTU-130 system).
X = Release level of CODOS, expressed as two hex digits.
Y = Code number indicating the kind of system on which CODOS is running. Presently defined codes are:

1 = MTU-130 with 8-inch floppy disk(s)
2 = KIM-1 with 8-inch floppy disk(s)
3 = AIM-65 with 8-inch floppy disk(s)
4 = PET or CBM with 8-inch floppy disk(s)
5 = SYM-1 with 8-inch floppy disk(s)

DESCRIPTION:

SVC #28 ($1C) returns information about the particular version of the CODOS system which is running. The A register is the first page of System RAM for CODOS. For example an AIM system with System RAM at $8000-$9FFF will return A=$80. The X register will contain the Release level of the operating system, expressed as two hex digits. For example, Release 2.0 will return X=$20. The Y register returns a numeric code indicating which target machine CODOS should be running on. The purpose of this SVC is to provide a mechanism for a program to make decisions based on the hardware and operating system environment.

EXAMPLE:

```
SVCENB  =       $EE
        ...
        LDA     #80
        STA     SVCENB      ;MAKE SURE SVCS ARE ON
        BRK
        .BYTE   28          ;GET INFO ABOUT SYSTEM
        CPY     #3
        BEQ     ITSAIM      ;BRANCH IF ITS AN AIM
        ...
```

This program segment tests the version of CODOS and branches to ITSAIM (not shown) if CODOS is running on an AIM.

NOTES:

1. Additional codes for new systems will be defined as necessary.

SVC #29 ($1D).

PURPOSE: To scan a device or file name/drive in preparation for ASSIGNment to a channel, or to ascertain a file's status.

ARGUMENTS:

Y = Index to start of file or device name in buffer

U5 = Pointer to input buffer

ARGUMENTS RETURNED:

Y = Index to first character after file name/drive or device name in buffer.

A = status information as described below.

flags: Cy set if parsed name was a device name; Cy clear if parsed name was a file name. The sign and overflow flag are set according to bits 7 and 6 respectively of the accumulator as described below.

U3 = Points to first character of file or device name in the buffer.

The status information returned depends on whether the device specified was a file name (with or without drive specified) or a device name:

If the name was a device name (Cy set) then:

Bit 7 is set to 1 if the specified device does not exist.
Bits 6-0 contain the ASCII device name (1 character).

If the name was a file name (Cy clear) then:

Bit 7 is set to 1 if an illegal name or drive number was specified.
Bit 6 is set to 1 if the specified (or default) drive is not open.
Bit 5 is set to 1 if the file exists (i.e., an old file).
Bit 4 is set to 1 if the drive is write-protected.
Bits 3 and 2 are not used.
Bits 1 and 0 contain the drive number selected.

DESCRIPTION:

SVC #29 is a multi-purpose SVC which performs the following activities:

1. It scans and validates an input string of characters specifying a file or device name.

2. It returns status bits so that the application program can gracefully recover from any common file/device specification errors without aborting the program, and can determine what type of file/device was specified.

3. It helps setup the necessary registers for using SVC #21 (Assign).

SVC #29 is normally used to parse a character string which specifies the name of a file or device which is to be assigned to a channel for I-O. SVC #29 scans the string, determines if the name is a device name or a file name, and checks it for legality.

If the name is a legal device name, it checks to see if the specified device exists on the system. For example, a printer may or may not be present. The device name is returned in the A register so that the user may immediately use SVC #21 to assign the device if all is in order.

If the name is a file name, it is checked for legality. An optional drive number may be specified as part of the name if the name is terminated by a colon followed by a digit. Blanks may be present between the colon and the digit. If no drive is given as part of the string, the default drive (usually drive 0) will be assumed. If the file name (and optionally the drive number) are legal, then SVC #29 checks to see if the drive is open. If it is, then the disk is checked for write protection. The directory is then searched for the file name. The status bits of the A register return the results of these operations to the application program. Upon completion of the SVC, the application program should verify that the status bits returned in A reflect the desired conditions. If they do, then all bits of A except bits 0 and 1 should be masked off and SVC #21 invoked to assign the file. Note that SVC #29 automatically sets U3 to point to the file name.

EXAMPLE:

Suppose that a program is to serve as a user-defined command, with the first argument to be the name of any device or file (optionally with drive number) which is to be read for input to the program.

```
USRCOM    .BYTE   0,12      ;SVC #12 = QUERY I-O BUFFERS AND COMD ARG PTR IN Y
          .BYTE   0,29      ;PARSE THE ARGUMENT
          BMI     NOGOOD    ;BRANCH IF ILLEGAL OR NON-EXISITANT DEVICE/FILE/DRV.
          BCS     ASGNIT    ;BRANCH IF DEVICE SPECIFIED
          BVS     NOTOPN    ;BRANCH IF DRIVE ISN'T OPEN
          CMP     #$20
          BCC     NOSUCH    ;BRANCH IF NO SUCH FILE PRESENT
          AND     #$03      ;ELSE DISCARD ALL BUT DRIVE NUMBER
ASGNIT    LDX     #5        ;CHANNEL 5 TO BE USED
          .BYTE   0,21      ;ASSIGN CHANNEL TO DEVICE OR FILE
          ...
```

This program segment first uses SVC #12 to get the address of the system input buffer and a pointer to the argument which is presumably a device or file name. It then uses SVC #29 to parse the first argument encountered. The return arguments from SVC #29 are checked to make sure the file or device exists and is ready to be read. If everything is OK, then control passes to ASGNIT which assigns the file or device name just scanned to channel 5 in preparation for reading. If there is a problem with the syntax of the argument or the file/device name, branches are taken to appropriate error handling routines within the program (not shown).

NOTES:

1. SVC #29 does not indicate whether a file is locked. The ASSIGN SVC (#21) however does indicate the locked/unlocked status of a file when assignment of a channel to a file is completed.

2. SVC #29 is only available on MTU-130 versions of CODOS.

SVC #30 ($1E).

PURPOSE: To obtain the current date.

ARGUMENTS:

Y = Index to desired start of nine character date field in buffer.

U6 = Pointer to desired output buffer.

ARGUMENTS RETURNED:

Y = Index to next available character after the date.

DESCRIPTION:

SVC #30 is used to obtain the current date, as entered during the normal CODOS signon-procedure or by execution of the DATE command. The date field will be nine characters long.

EXAMPLE:

```
SHODT     .BYTE     0,12              ;SVC #12 GETS BUFFER POINTER INTO U6
          .BYTE     0,2,6,'TODAY IS ',0 ;SVC #2 = OUTPUT MSG (ON CHANNEL 6)
          LDY       #0                ;SET INITIAL INDEX TO BEGIN OF BUFFER
          .BYTE     0,30              ;GET 9 CHARACTER DATE
          LDX       #6                ;CHANNEL 6
          .BYTE     0,6               ;OUTPUT LINE
          ...
```

This program segment outputs line with a message such as:

TODAY IS 24-JAN-82

assuming that the date previously entered was "24-JAN-82".

NOTES:

1. The default date field is "*UNDATED*"

2. SVC #30 is available only on MTU-130 versions of CODOS.

CHAPTER 7.

16-BIT ARITHMETIC PSEUDO-PROCESSOR

This chapter describes the operation of the CODOS 16-bit Pseudo-Processor ("PP"), which is invoked by using SVC #27 (see Chapters 5 and 6 for general information on SVCs).

GENERAL DESCRIPTION

The Pseudo-processor is a software simulation of a 16-bit Central Processing Unit. The programmer can view it as an extension to the 6502 which provides 16 bit arithmetic and utility functions. The PP uses the Pseudo-registers U0 through U7 in page 0 as its "registers", and uses the rest of memory as ordinary memory. Executing an SVC #27 causes the PP to begin processing its instruction set beginning with the next byte. It continues executing until a PP "UEXT" instruction is encountered, at which time normal 6502-processing resumes at the next byte. The primary value of the Pseudo-Processor to the programmer is that it provides a very compact and easy-to-use method of performing 16-bit arithmetic, including multiplication and division, and also provides a simple way of computing 24-bit file positions for random access files.

PP REGISTERS

The PP has eight "Pseudo-registers", U0 through U7, as shown in figure 7-1. These are the same P-registers used for SVC processing. The Pseudo-processor uses U0 as its 16-bit accumulator. It operates in a manner similar to the 6502's "A" register, but operates on 16-bits instead of 8 bits of data. The P-registers U1 through U6 are 16-bit general purpose registers which can be used to hold operands for arithmetic operations or for holding addresses. In terms of the actual location of the P-registers in memory, all P-registers have the least-significant byte first and the most-significant byte at the next higher address, in the same fashion as conventional 6502 addresses. P-register U7 is 24 bytes long. The low-order 16 bits of U7 can be used just like any of the other general-purpose pseudo-registers U1 through U6. The high byte of U7 is only affected by one special instruction, specifically designed for the computation of a 24-bit file position, which is explained later.

INSTURCTION SET

The PP has a very simple instructions set, to facilitate hand-assembly, with 16 instructions in the set. Three of the instructions are three bytes long; all the rest are one byte instructions. The first byte is the operation code for the instruction, and always contains two subfields: a 4 bit operation field and a 4-bit register number. Thus every opcode is easily defined as two hex digits where the first hex digit tells which of the 16 possible operations is to be performed, and the second hex digit tells which register is to be used. The complete instruction set is defined in table 7-1. For example, a $13 instruction tells the PP to add the contents of Pseudo-register U3 to the Pseudo-accumulator, U0, and store the result in the Pseudo-accumulator. Notice that unlike the 6502 instruction set, arithmetic operations operate on U0 and another register, rather than the accumulator and memory.

PSEUDO-PROCESSOR REGISTERS

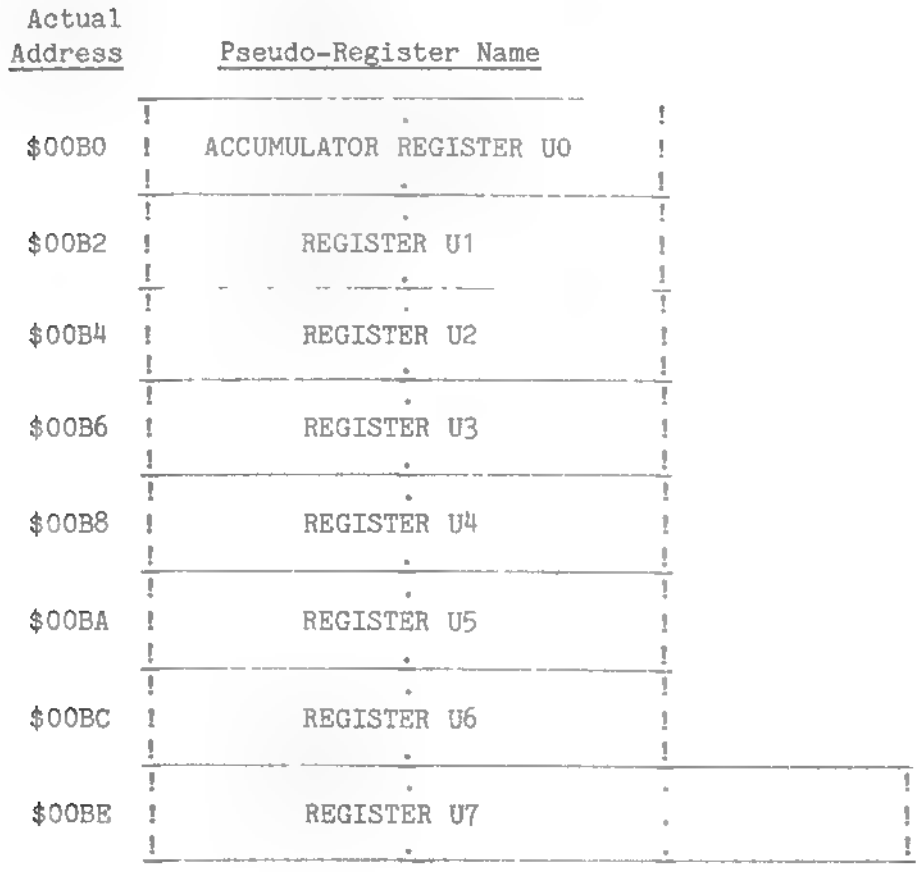


Figure 7-1

NOTES FOR FIGURE 7-1:

1. Values should be deposited with the least significant byte first.

2. Register U7 is three bytes long, with the least significant byte first and the most significant byte at the highest address.

TABLE 7-1: CODOS PSEUDO-PROCESSOR (PP) INSTRUCTION SET

Operation Code $HI	$LO	#	Mnemonic	Description
0	n	1	UEXT n	Exit PP mode. Set the Z and N flags to reflect the value in register Un, and return to normal 6502 execution mode. The Carry flag reflects the last UADD or USUB result.
1	n	1	UADD n	U0 = U0 + Un. 16-bit add. Carry flag set if carry occurs out of most significant bit of result; otherwise, carry is cleared.
2	n	1	USUB n	U0 = U0 - Un. 16-bit subtract. Carry flag is cleared if borrow occurred out of the most significant bit of result; otherwise, carry is set.
3	n	1	UMUL n	U0 = U0 * Un. 16-bit multiply. Carry not affected. Low 16 bits of product is in U0, and the high-order 16 bits is in Un.
4	n	1	UDIV n	U0 = U0 / Un. 16-bit divide. Carry not affected. quotient is in U0, and remainder is in Un. Aborts on divide by 0.
5	n	1	UNU0 n	U0 = Un. 16-bit move, Un to U0. Un remains unchanged.
6	n	1	U0UN n	Un = U0. 16-bit move, U0 to Un. U0 remains unchanged.
7	n	1	USWP n	U0 exchanged with Un. 16-bit exchange.
8	n	3	ULDI n,val	Un = val. 16-bit load immediate. First byte of val is low-order byte, second is high-order byte.
9	n	3	ULDA n,addr	Un = (addr). 16-bit load. The data at address addr is placed in the low-order byte of Un and the data at addr+1 is placed in the high-order byte.
A	n	1	U0LD n	U0 = (Un). 16-bit load U0 register-indirect. The data pointed to by register Un is loaded into the low byte of U0, and the next byte is loaded into the high byte of U0.
B	n	3	USTA n,addr	(addr) = Un. 16-bit store. The low byte of Un is stored in memory at address addr, and the high byte is stored at addr+1.
C	n	1	U0ST n	(Un) = U0. 16-bit store U0 register-indirect. The low byte of U0 is stored at the address in Un, and the high byte is stored at the next higher address.
D	n	1	UNU7 n	U7 = U0, Un. 24-bit move. Register U0 is moved to the low-order 16 bits of U7, and the low-order 8 bits of register Un is moved to the most significant (3rd byte) of U7. See note 3.
E	n	1	UJSR n	Call subroutine at (Un). Execute 6502-subroutine whose address is in register Un. See note 5.
F	n	1	UNOP n	No operation. Reserved for future extensions, treated as no-operation.

NOTATION USED: "n" = pseudo register number, 0 to 7, for U0 through U7 respectively. "Val" is 16-bit value or address. addr is 16-bit address. "#" means number of bytes in the instruction. "$HI" is high nybble (4 bits) of opcode, and "$LO" is low-order nybble.

NOTES FOR TABLE 7-1:

1. The $On (UEXT) instruction exits the Pseudo-processor and SVC #27. When normal 6502 operation is resumed, the 6502 registers will be preserved in the state they were in on executing SVC #27, except for the Carry (C), Negative (N) and Zero (Z) flags in the processor status word. The Carry is returned in the state which resulted from the last $1n (UADD) or $2n (USUB) operation. No other PP operations affect the Carry. If no UADD or USUB was executed, the Carry will be clear. The Z flag will be set if the register specified on the $On (UEXT) instruction was 0 (all 16 bits are 0); otherwise, the Z flag will be cleared. The N flag will be set if bit 15 of the Pseudo-register specified on the $On (UEXT) instruction was 1; otherwise it will be cleared. This is the sign bit for 16-bit two's complement arithmetic. The remaining flags in the processor status word are not affected.

2. All operations are performed in binary, regardless of the setting of the decimal mode flag when SVC #27 is executed.

3. The $En (UNU7) instruction is normally used immediately after $4n (UMUL) to obtain the first 24 bits of a 32-bit product in U7. This can be used to easily compute the desired file position in a file of fixed-length records. See Example 3.

4. The byte following the $On (UEXT) instruction should contain the first byte of normal 6502 code.

5. The $En (UJSR) instruction executes a user-defined 6502 machine-language subroutine whose address is in pseudo register n. On entry to the subroutine, register A will contain the low-order byte of U0, register X will contain the index needed to address register Un relative to register U0 (e.g., 0 if n = 0, 2 if n = 1, 8 if n = 4, etc.), and Y will be 0. The carry flag will reflect the status of the last UADD or USUB operation; the N and Z flags will reflect the value of the low order byte of U0. The decimal mode flag will be clear. The subroutine can destroy any registers but must return via an RTS with the stack intact. The user subroutine may <u>not</u> use any SVCs.

6. Unlike the normal 6502 ADC and SBC instructions, the setting of the carry has no effect on UADD or USUB.

7. The Pseudo-processor resides in one of CODOS's system overlays. Normally, therefore, the first time SVC #27 is executed in a program, CODOS will load the PP from disk and execute it automatically. Although this loading is quite rapid (typically a fraction of a second), it will be much longer than would be required if the PP was already loaded. If the PP is to be used in a time-critical portion of the program, you may want to "preload" the PP by executing a dummy SVC #27 prior to the time-critical portion of the code. CODOS will not reload the PP overlay if it is already in memory.

8. Multiplication and division operations are unsigned.

EXAMPLES AND APPLICATIONS:

Assume the following lines of initialization and definitions preceed all example solutions below:

```
SVCENB  =       $EE     ;ADDRESS OF SVC ENABLE FLAG
;
U0      =       $B0     ;16-BIT ACCUMULATOR FOR PP
U1      =       $B2     ;16-BIT PSEUDO-REGISTERS...
U2      =       $B4
U3      =       $B6
U4      =       $B8
U5      =       $BA
U6      =       $BC
U7      =       $BE     ;24 BIT REGISTER
        ...
        LDA     #$80
        STA     SVCENB  ;ENABLE SVCS
```

Example 1: In preparation for using SVC #5 and SVC #6 for line-oriented I-O, setup Pseudo-registers U5 and U6 to point to two 80-character buffers starting at MYBUFS.

```
DEFBUF  BRK
        .BYTE  27       ;SVC 27 ($1B) = ENTER 16-BIT PSEUDO PROCESSOR
        .BYTE  $85      ;ULDI 5, (LOAD U5 IMMEDIATE...)
        .WORD  MYBUFS   ;MYBUFS (..WITH DESIRED INPUT BUFFER ADDRESS)
        .BYTE  $86      ;ULDI 6, (LOAD U6 IMMEDIATE...)
        .WORD  MYBUFS+80 ;MYBUFS+80 (...WITH ADDRESS OF OUTPUT BUFFER)
        .BYTE  $00      ;UEXT 0 (EXIT PP MODE)
```

Note that above method uses 9 bytes of code, compared with 16 using the equivalent conventional 6502 code shown in the example for SVC #5 and SVC #6.

Example 2: Compute 10 times the 16-bit value at $2000 through $2001 and store the result at $2002 through $2003. If the result is 0, replace it with 1.

```
TENX    BRK
        .BYTE  27       ;SVC 27 ($1B) = ENTER 16-BIT PSEUDO-PROCESSOR
        .BYTE  $81      ;ULDI 1, (LOAD U1 IMMEIDATE...)
        .WORD  10       ;10 (...WITH 10)
        .BYTE  $90      ;ULDA 0, (LOAD U0 ABSOLUTE...)
        .WORD $2000     ;$2000  (...WITH 16-BIT NUMBER AT ADDRESS $2000)
        .BYTE  $31      ;UMUL 1 (MULTIPLY U0 = U0 * U1)
        .BYTE  $B0      ;USTA 0, (STORE U0...)
        .WORD  $2002    ;$2002 (...INTO ADDRESS $2002, $2003)
        .BYTE  $00      ;UEXT 0 (TEST U0 AND EXIT PP MODE)
        BNE    CONTIN   ;BRANCH IF U0 IS NOT 0
        LDA    #1       ;ELSE REPLACE RESULT WITH 1
        STA    $2002
CONTIN  ...
```

This example illustrates how flags set by the PP may be used after the SVC #27.

Example 3: Channel 5 is assigned to a file containing fixed-length records of 325 bytes each. Given that the desired record number (0 through 999) is in U1, read the selected record into a buffer at $2400.

```
RDREC   BRK
        .BYTE  27       ;SVC 27 ($1B) = ENTER 16-BIT PSEUDO-PROCESSOR.
        .BYTE  $80      ;ULDI 0, (LOAD U0 IMMEDIATE...)
        .WORD  325      ;325  (...WITH RECORD SIZE)
        .BYTE  $62      ;U0UN 2 (COPY RECORD SIZE TO U2 FOR SVC 15 LATER)
        .BYTE  31       ;UMUL 1 (MULTIPLY RECORD SIZE * RECORD NUMBER)
        .BYTE  $D1      ;UNU7 1 (SET 24 BITS OF U7 TO RESULT)
        .BYTE  $81      ;ULID 1 (LOAD U1 IMMEDIATE...)
        .WORD  $2400    ;$2400 (...WITH DESIRED BUFFER ADDRESS)
        .BYTE  00       ;UEXT 0 (EXIT PP MODE)
        LDX    #5       ;CHANNEL 5
        BRK
        .BYTE  19       ;SVC 19 ($13) = POSITION FILE.
        BRK
        .BYTE  15       ;SVC 15 ($0F) = READ RECORD
```

This example illustrates how the PP can be used to setup for file positioning in files, even for files with greater than 65,535 bytes (the example file has 325,000 bytes of data).

CHAPTER 8.

KEYBOARD AND TEXT DISPLAY I/O DRIVER (IODRIVER.Z)

The Keyboard and Text Display I/O Driver is a program that interfaces the keyboard and display hardware of the MTU-130 computer with CODOS, language interpreters, assembly language programs, and in fact any program that does not contain its own keyboard and display driver routines. In the case of user-written assembly language programs, most normal interaction with the keyboard and display may be performed through the SVC facility of CODOS. Please refer to sections 5 and 6 for details on the functions available through SVCs. More sophisticated interaction with these devices may be performed through direct calls to the Keyboard and Text Display Driver as described in this section.

The keyboard and text display I/O drivers are contained in the file IODRIVER.Z which is normally loaded into memory by the START UP.J file. This driver package has a number of entry points, each of which performs a specific function. A jump table is provided for these entry points in the System Communications area in low memory. These entry point addresses are fixed and are not expected to change over the life of the MTU-130 product. The location and function of each entry point is described below.

THE KEYBOARD DRIVER

The MTU-130 computer uses a software- scanned keyboard for flexibility. The keyboard input driver has five entry points that make it easy for a program to use the keyboard and which essentially hide its software-scanned nature. Three of these entry points are used for inputting or testing keys individually. The other two entry points are used for inputting or editing entire lines of text. These routines also update the display to reflect the information being entered. Timer T1 of the SYS1 6522 I/O chip is used by the keyboard driver to time the repeat rate. The 4 cursor direction keys, space bar, BACKSPACE, DELETE, and RUBOUT will automatically repeat if held down. All other keys may be forced to repeat by pressing the REPEAT key simultaneously with the character key. See appendix H for the character code generated by each key. Rollover and debouncing are handled fully by the keyboard driver.

Several parameters located in fixed locations in the System Communication area influence the operation of the keyboard driver. All parameters are set to default values when CODOS is "booted up" but the programmer may wish to change them.

PARAMETER	LOCATION	DESCRIPTION
QLN	$00F0	Pointer to line-buffer used for INLINE and EDLINE
KBECHO	$020F	If bit 7=1 then "echo" each key to the display.
NOCLIK	$0213	If bit 7=1 then no click when a key is pressed.
DBCDLA	$0220	Wait time in milliseconds allowed for contact bounce.
RPTRAT	$0221	Intercharacter repeat delay in 256uS units.
CURDLA	$0222	Determines cursor blink speed, 0=no blink.
CLKPER	$0224	Click waveform period in units of 200 microseconds.
CLKVOL	$0225	Click volume, $00 = minimum, $7F = maximum.
CLKCY	$0226	Click duration in units of complete waveform cycles.
YLNLIM	$0238	Line size limit for INLINE and EDLINE entry points.
UKINLN	$023A	If bit 7=1 then unrecognized keys are accepted for entry points INLINE and EDLINE.

The keyboard driver entry points are described below. To use an entry point, simply execute a JSR to the indicated address.

ENTRY POINT: GETKEY $0306

PURPOSE: To wait until a keyboard key is struck and return with character in A.

ARGUMENTS: None (see Chapter 10 for operational parameters)

ARGUMENTS RETURNED: A = Character code of key struck; X and Y preserved.

DESCRIPTION:

This entry point will wait indefinitely for a key to be pressed. While waiting, a flashing text cursor will normally be displayed unless suppressed by parameter setting.

ENTRY POINT: TSTKEY $030C

PURPOSE: To test if a key is pressed; has multiple recognition lockout.

ARGUMENTS: None

ARGUMENTS RETURNED: Carry is set if a key was down, clear if not. A = character code of key seen down, if any. X and Y are preserved.

DESCRIPTION:

This entry point will scan the keyboard once looking for a key that is down. If one is found down that has not been previously recognized as down, its code is loaded into A and the Carry flag is set. If no keys are found down, the Carry flag is cleared and A is loaded with an undefined value. The difference between this entry point and the IFKEY entry point is that a key is recognized as down only once until the operator releases it. This also applies to a key still down after recognition by the GETKEY entry point.

ENTRY POINT: IFKEY $0360

PURPOSE: To test if a key is pressed without multiple recognition lockout.

ARGUMENTS: None

ARGUMENTS RETURNED: Carry is set if a key was down, clear if not. A = character code of key seen down, if any. X and Y are preserved.

DESCRIPTION: This entry point will scan the keyboard once looking for a key that is down. If one is found down, its code is loaded into A and the Carry flag is set. If no keys are found down, the Carry flag is cleared and A is loaded with an undefined value. The difference between this entry point and the TSTKEY entry point is that a key may be repeatedly recognized as being down as long as the operator holds it down.

ENTRY POINT: INLINE $031E

PURPOSE: To input an entire line from the keyboard, with editing permitted.

ARGUMENTS: None (see requirements for QLN below).

ARGUMENTS RETURNED: A = number of characters in the line, Y = 0, X preserved;
QLN points to the complted line.

DESCRIPTION:

This entry point accepts one line of text from the keyboard, and allows all the line-editing functions permitted by CODOS. Editing functions include CTRL-B for recalling prior lines, and automatic replacement of function keys with pre-defined macro character strings. Please see Table 2-4 for a complete list of editing keys available. In order to use this entry point, the calling routine MUST preset location $00F0 to the desired input-line buffer location before calling the entry point. If you wish to use the normal CODOS line buffer, you should then set $00F0 to $00 and $00F1 to $05. Once called, the routine will not return until a complete line has been entered, terminated by a carriage return. Function keys are automatically expanded to the equivalent character string using the global function key string table at address $0400, as described for the CODOS ONKEY command.

Normally this routine will simply ignore any unrecognized control keys (such as CTRL-D, for example) or unrecognized extended characters (such as the PF1 key, for example). Setting the UKINLN flag at $023A will instead allow such characters to be returned in the line buffer. The "strange" characters will not be echoed to the screen, however.

The maximum number of characters which can be returned in the line is determined by the value of YLNLIM (address $0238). The standard input buffer at $0500 is large enough for 191 (decimal) characters, and this is the default value for YLNLIM.

ENTRY POINT: EDLINE $0321

PURPOSE: To edit an entire line using the keyboard.

ARGUMENTS: Y=indexes the implied CR at the end of the line to be edited; QLN (address $00F) points to start of line to be edited.

ARGUMENTS RETURNED: A = number of characters in the line, Y = 0, X preserved;
QLN points to the completed line.

DESCRIPTION:

This entry point is similar to INLINE, above, except that the input line buffer is assumed to already hold a line to be edited when the entry point is called. The keyboard can be used to edit or accept the line. The full range of editing keys accepted by CODOS are available. To use this entry point, copy the desired line into the buffer, install the address of the buffer in QLN (address $00F0), set Y to the character count of the line, and call EDLINE.

THE TEXT DISPLAY DRIVER

The MTU-130 computer uses a bit-mapped dot matrix display format for extreme flexibility in text and graphics display. Thus software is responsible for drawing the individual dots that make up character patterns when conventional text display is required. The text display driver has numerous entry points that make it easy for a program to display "normal text" and essentially hide the display's software intensive nature.

The video display is organized as 480 horizontal dots by 256 vertical dots. Dot locations are defined by cartesian coordinates (X,Y) in the usual fashion where (0,0) is the lower left hand corner of the screen and (479,255) is the upper right. Calls to both the text display driver and the graphics display driver (described in section 9) may be intermixed to provide the desired display.

Text is displayed in a text window which is always 80 characters wide. The text window may be defined from 1 to 25 lines long, and defaults to 24. The top of the text window is defined by the constant YTDOWN, measured in vertical dots down from the top. YTDOWN may be defined from 0 to 245 and defaults to 0. The current position of the text cursor within the text window is kept in variables COL (1 to 80) and LINE (1 to 25-10*YTDOWN). Please refer to section 9 if you wish to place characters in arbitrary X and Y locations on the screen which do not fall into the normal 24x80 "grid".

Characters are displayable horizontally only using a 5 wide by 7 high dot matrix in a 6 by 10 cell of dots. Where required, lower case characters are displayed with descenders which drop 2 Y positions below the baseline. Characters may be underlined automatically under the control of a flag or may be underlined individually by overstriking the underline character. For special applications, a separate user supplied font using the same cell size may be selected by setting a flag. The entire cell may be displayed in reverse video if desired, controlled by a flag.

The text cursor normally consists of a blinking reverse video block displayed at the current cursor position. It is automatically blinked by the keyboard-input routine while waiting for input. Therefore the cursor is normally only displayed when the computer is ready to accept keyboard input. The cursor can be drawn at will by explicit calls to a routine, and can be totally disabled by setting a flag. The keyboard input routine normally blinks the cursor at a selectable rate which defaults to about 2Hz.

The text display driver also controls display of the function key legends. When displayed, they occupy the bottommost 16 vertical dots for the full screen width. The legends consist of two groups of 4 rectangular boxes each of which has enough room to display an 8 character label. The text I/O driver draws the boxes and displays the labels. Association of function key depression with specific functions can be done by the user program, or can use automatic substitution of character strings as is available in normal CODOS command entry.

Several parameters located in fixed locations in the System Communication area influence the operation of the text display driver. All parameters are set to default values when IODRIVER.Z is executed. Some parameters, such as the current cursor position, change during normal operation. You may also wish to alter or examine certain of these parameters.

PARAMETER	LOCATION	DESCRIPTION
COL	$0200	Current column location of text cursor, 1 - 80.
LINE	$0201	Current line number of text cursor, 1 - NLINET.
NLINET	$021E	Number of text lines in the text window.
YTDOWN	$021F	255-(Y coordinate of top of the text window).
CURDLA	$0222	Determines cursor blink speed, 0=no blink.
NOLFCR	$0210	If bit 7=1 then no automatic line feed after CR.
NOSCRL	$0211	If bit 7=1 then instead of scrolling, the text window is cleared and the cursor is homed when text goes beyond the bottom line.
UNDRLN	$0212	If bit 7=1 then all characters underlined when drawn.
NOBELL	$0214	If bit 7=1 then BEL character is ignored.
RVIDEO	$0215	If bit 7=1 then characters are drawn in reverse video.
SHODEL	$0216	If bit 7=1 then display DEL (RUBOUT) as a character shape.
SHOUL	$0217	If bit 7=1 then character cell is erased before the underline character is drawn.
EXCCP	$0218	If bit 7=1 then call user control character processor.
EXTHI	$0219	If bit 7=1 then call user routine to process all characters with bit 7 set.
EXFONT	$021A	If bit 7=1 then use external font table (see User Defined Fonts at the end of this chapter).
BELPER	$0227	Bell sound waveform period in units of 200 microseconds.
BELVOL	$0228	Bell sound volume, $00 = minimum, $7F = maximum.
BELCY	$0229	Bell sound duration in units of complete waveform cycles.
QEXCC	$022F	Address of external control character processor if used.
QEXFNT	$0231	Address of external font table if used.
QEXHI7	$0233	Address of external processor for characters with bit 7=1.
EXFTBK	$0237	Memory bank number containing external font table.
TABTBL	$06E0-06FF	Tab stop table. A tab stop is located at the column number specified by each non-zero byte. See OUTCH entry point for details.
LEGTBL	$05C0-05FF	Function key legend table. Contains 8 groups of 8 characters which are the displayed legends for the 8 function keys. The label for the f1 key is first. See DRWLEG entry point for details.
KEYSTR	$0400-04FF	Function key substitution string table. Contains 8 groups of 32 characters which represent the the character strings to be substituted for the associated function keys when using the input-line or edit-line functions. See INLINE entry point for details.

The following describes the entry points into the text display driver.

ENTRY POINT: OUTCH $0309

PURPOSE: To display a printable character or interpret a control character.

ARGUMENTS: Character to be displayed or interpreted in A.

ARGUMENTS RETURNED: None, A, X, and Y registers preserved.

DESCRIPTION:

This entry point is used for general text display. If the character is printable (i.e., not a control or extended character), it is displayed at the present cursor position as defined by COL and LINE. The cursor is then moved one position to the right (COL incremented). If the cursor tries to go beyond the right end of the present row (COL 80), it is returned to the left edge of the screen (COL=1) and down one line (LINE incremented). If the cursor tries to go beyond the text window bottom (LINE NLINET), then the entire text window scrolls upward instead. If NOSCRL is set, the text window would be cleared and the cursor placed in the upper left corner (COL=1, LINE=1) instead. If RVIDEO is set, the character will be drawn in black against a white background. If UNDRLN is set, the character will be underlined as it is drawn. All printable characters except the underline first erase the 6 by 10 character cell before being drawn.

If the character is an ASCII control character, action is according to the following table:

CODE	NAME	ACTION
$0D	CR	Carriage return, moves cursor to left screen edge (COL=1) and also performs LF function unless NOCRLF flag is set.
$0A	LF	Line feed, advances to next lower line without affecting column. If already on bottom line (LINE=NLINET) then scrolls display up one line unless NOSCRL flag is set. If NOSCRL is set then text window is cleared and LINE is set to 1.
$08	BS	Backspace, move cursor left 1 character without erasing the character.
$09	HT	Tab, move COL to next tab stop in tab table. If beyond last valid tab stop, no action.
$18	CAN ^X	Cancel, clear current display line, set COL to 1. Does not change LINE.
$07	BEL ^G	Bell, sound audio tone unless NOTONE is set.
$0C	FF ^L	Form feed, clear text window, home the cursor to the upper left corner (sets COL=1 and LINE=1).
$7F	RUBOUT (DEL)	Rubout, performs equivalent to Backspace, Space, Backspace unless SHODEL flag is set in which case it displays a checkerboard character shape.

If the character is an extended character (bit 7=1, presumably from one of the special keys on the keyboard), action is according to the following table:

CODE	NAME	ACTION
$A0		Cursor up, move cursor up one line unless already at top of text window (LINE=1).
$A1		Cursor left, translate to BS and process as a BS character ($08).
$A2		Cursor right, move cursor right one without erasing unless already at right edge of screen (COL=80).
$A3		Cursor down, translate to LF and process as a LF character ($0A).
$A4	HOME	Cursor home, move cursor to left screen edge at top of text window (COL=1, LINE=1).
$8A	X	Multiply, print * symbol on screen.
$8B	/	Divide, print / symbol on screen.
$8C	-	Subtract, print - symbol on screen.
$8D	+	Add, print + symbol on screen.
$8E	ENTER	Translate to CR and process as a CR character ($0D).
$B4	SHIFT/HOME	Shifted HOME, translate to FF and process as a FF character($0C).

If the EXTHI flag is set, a user supplied external subroutine is called to handle extended characters instead.

ENTRY POINT: CLRDSP $0312

PURPOSE: To clear the entire 480 by 256 screen.

ARGUMENTS: None.

ARGUMENTS RETURNED: None, A, X, and Y registers preserved.

DESCRIPTION:

This entry point is used to clear the entire screen including the legend boxes. If you want to clear only the text window, use the CLRHTW entry point, or use the OUTCHR entry point with an argument of $0C (form feed).

ENTRY POINT: DRWLEG $0315

PURPOSE: To draw the function key legend boxes and their labels.

ARGUMENTS: The 64 bytes of legend labels from $05C0 - $05FF.

ARGUMENTS RETURNED: None, registers not preserved.

DESCRIPTION:

This entry point erases the existing legend area (bottom 16 scan lines of the display area) and draws 8 legend boxes each containing an 8-character function key legend. The boxes and legends are displayed in two groups of 4 like the function keys on the MTU-130 keyboard. The legends to be used are ASCII strings of exactly 8 characters each, which must be predefined in the legend table ($5C0-$5FF). Non-displayable characters (control or characters with bit 7 set) are treated as if they were blanks. The characters are drawn in normal mode (no underlining or reverse video). If an external font table is specified (EXFONT flag set), then it is used to draw the legend characters. The position of the text cursor (COL, LINE) is not affected.

ENTRY POINT: INITIO $030F

PURPOSE: To clear the screen and set default values of display parameters.

ARGUMENTS: None.

ARGUMENTS RETURNED: None, registers not preserved.

DESCRIPTION:

This routine completely re-initializes the text display driver and is useful if its previous state is unknown. It performs the following functions:

1. Clears entire 480 by 256 screen area.
2. Clears all function key legends to blanks.
3. Clears function key substitution strings to the null string ($80).
4. Draws the function key legend boxes.
5. Sets text window to 24 lines starting at top of the screen.
6. Text cursor is placed in the home position (COL=1, LINE=1)
7. All text driver flags (NOLFCR, NOSCRL, UNDRLN, etc.) are cleared.
8. The keyboard driver is initialized.
9. The audio DAC is initialized.

ENTRY POINT: INITTW $0363

PURPOSE: To initialize the text window to 24 lines and clear the text window only.

ARGUMENTS: None

ARGUMENTS RETURNED: Registers are not preserved.

DESCRIPTION:

This routine re-initializes the text display driver. It performs the following functions:

1. Sets text window to 24 lines starting at top of the screen.
2. Text cursor is placed in the home position (COL=1, LINE=1)
3. All text driver flags (NOLFCR, NOSCRL, UNDRLN, etc.) are cleared.
4. The audio DAC is initialized.

ENTRY POINT: DEFTW $0366

PURPOSE: To set the position and size of the text window.

ARGUMENTS: A = desired number of text lines, 1 - 24.
Y = position of top of window (number of scan lines down from top).

ARGUMENTS RETURNED: None, only X register is preserved.

DESCRIPTION:

This routine defines the size and position of the text window. The requested number of text lines must fit between the specified top position and the bottom of the screen (255) at the rate of 10 scan lines per character line.

ENTRY POINT: CLRHTW $0369 873

PURPOSE: To clear the text window and home the cursor.

ARGUMENTS: None

ARGUMENTS RETURNED: None, all registers preserved.

ENTRY POINT: HOMETW $036C

PURPOSE: To place the cursor in the home position (COL=1, LINE=1)

ARGUMENTS: None

ARGUMENTS RETURNED: None, all registers preserved.

ENTRY POINT: CRLF $036F

PURPOSE: To move cursor to the left screen edge and down one line.

ARGUMENTS: None

ARGUMENTS RETURNED: None, all registers preserved.

DESCRIPTION:

This routine performs the same as the OUTCH entry point with an argument of $0D (CR).

ENTRY POINT: CLRTW $0372 882

PURPOSE: To clear the text window without moving the cursor.

ARGUMENTS: None

ARGUMENTS RETURNED: None, all registers preserved.

ENTRY POINT: CLRLEG $0375 (885)

PURPOSE: To clear the legend display area (bottommost 16 scan lines).

ARGUMENTS: None

ARGUMENTS RETURNED: None, only X and Y are preserved.

ENTRY POINT: CLRTLN $0378

PURPOSE: To clear a specified text line.

ARGUMENTS: A = line number to be cleared, 1 <= A <= NLINET

ARGUMENTS RETURNED: None, only Y is preserved.

ENTRY POINT: LINEFD $037B

PURPOSE: To move the cursor down one text line.

ARGUMENTS: None.

ARGUMENTS RETURNED: None, only X and Y are preserved.

DESCRIPTION:

This routine performs the same as the OUTCH entry point with an argument of $0A (LF).

ENTRY POINT: OFFTCR $037E

PURPOSE: To turn the text cursor off if it is on.

ARGUMENTS: None.

ARGUMENTS RETURNED: None, all registers are preserved.

ENTRY POINT: ONTCR $0381

PURPOSE: To turn the text cursor on.

ARGUMENTS: None.

ARGUMENTS RETURNED: None, all registers are preserved.

ENTRY POINT: FLPTCR $0384

PURPOSE: To flip the video sense of the cursor at the cursor position.

ARGUMENTS: COL = column number of character to be flipped.
LINE = line number of character to be flipped.

ARGUMENTS RETURNED: None, all registers are preserved.

ENTRY POINT: BEEP $038D (409)

PURPOSE: To sound an audible beep.

ARGUMENTS: A = volume in range of $00 (silence) to $7F (maximum), $40 is normal.
X = duration in complete waveform cycles, 1-255, 0=256.
Y = Waveform period in units of 200 microseconds.

ARGUMENTS RETURNED: None, all registers are preserved.

USER DEFINED FONTS

The Text Display I/O Driver has provisions for use of an external user-defined character font. To use an external font table, first store the address of the external table in location $0231 (low) and $0232 (high). Install the bank number containing the font table into location $0237 (default is 0). Then to enable the external table, set bit 7 of the EXFONT flag at location $021A to a one. Following these actions, all characters subsequently drawn on the screen will refer to the external font table until the EXFONT flag is cleared. You may switch back and forth between the normal and alternate font by merely toggling bit 7 of EXFONT.

The text display driver uses a basic cell structure of 7 rows of 5 dots for a character. The font table therefore consists of 7 bytes for each character where the leftmost 5 bits of each byte represent the 5 dots for that row. For example, the font table entry for a letter "A" would be as follows:

Byte 0	0 0 1 0 0 0 # *	$20	
.	0 1 0 1 0 0 0 0	$50	* This bit is the descender
.	1 0 0 0 1 0 0 0	$88	flag, see text.
.	1 1 1 1 1 0 0 0	$F8	
.	1 0 0 0 1 0 0 0	$88	# This bit is the "J dot"
.	1 0 0 0 1 0 0 0	$88	flag, see text.
Byte 6	1 0 0 0 1 0 0 0	$88	

The first 7 byte table entry corresponds to the ASCII code $20, the next 7 bytes to $21, etc. up to $7F.

Normally the character is drawn so that the bottommost dot row rests on the baseline. If the descender flag bit is on (see * note above), the entire 5 by 7 matrix is shifted two dot rows downward as it is drawn. This is normally used when drawing lower case characters such as j,g,y,p and q. For descended characters requiring a dot outside the 5X7 matrix area (such as the lower case j) the "J dot" flag bit should be a one. This will cause a centered dot to be placed two rows above the top row of the descended character.

EXTERNAL HANDLING OF ASCII CONTROL CHARACTERS

The Text Display I/O Driver has provision for external handling of the ASCII control characters, i.e., those between $00 and $1F inclusive. To use an external control character processor, first place its address in location $022F (low) and $0230 (high). Then to enable the external processor, set bit 7 of the EXCCP flag at location $0218 to a one. Following these actions, all control characters received by the OUTCH entry point will be passed to the external routine until the EXCCP flag is cleared. You may switch back and forth between noraml and special handling of control characters by merely toggling bit 7 of EXCCP.

The external routine is entered with the ASCII control character code in the A register, and X and Y undefined. The external routine must save and restore any register it uses, and may also make balanced use of the stack. Any of the entry points described earlier except OUTCH may be called by the external routine. The external routine must exit by using an RTS instruction.

EXTERNAL HANDLING OF EXTENDED CHARACTERS

The Text Display I/O Driver has provision for external handling of the extended characters, i.e., those that have bit 7 set. To use an external extended character processor, first place its address in location $0233 (low) and $0234 (high). Then to enable the external processor, set bit 7 of the EXTHI flag at location $0219 to a one. Following these actions, all extended characters received by the OUTCH entry point will be passed to the external routine until the EXTHI flag is cleared.

The external routine is entered with the extended character code in the A register, and X and Y undefined. The external routine must save and restore any register it uses, and may also make balanced use of the stack. Any of the entry points described earlier except OUTCH may be called by the external routine. The external routine must exit by using an RTS instruction.

CHAPTER 9.

GRAPHICS DISPLAY I/O DRIVER (GRAPHDRIVER.Z)

The Graphics Display I/O Driver is a program that interfaces the display generator, light pen, and keyboard of the MTU-130 computer with assembly language programs and language interpreters that need to utilize the graphics capabilities of these devices. The graphics functions provided are as follows:

1. Plotting points and vectors given cartesian X and Y coordinates.

2. Drawing text characters at arbitrary X and Y locations.

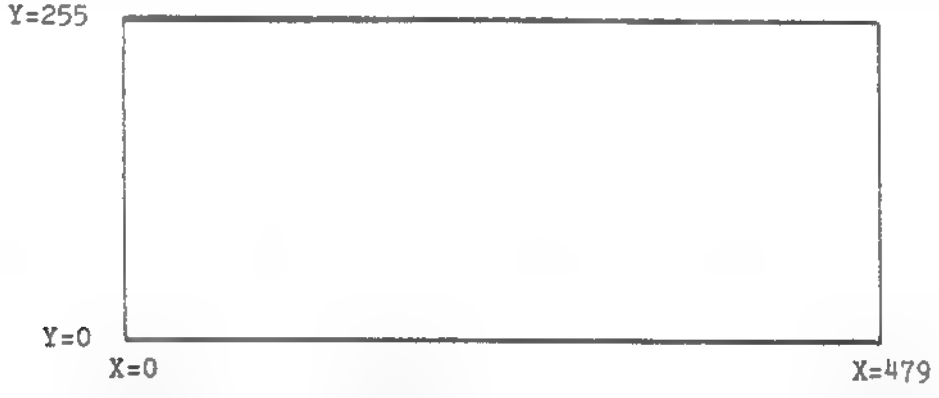
3. User input of position information using a graphic cursor and the keyboard.

4. User input of position information using the light pen.

The Graphics Display I/O Driver (file GRAPHDRIVER.Z) has a number of entry points which may be called to perform various functions. A jump table for these entry points is provided in the system communications area of memory, and is not expected to change. The Graphics driver software also requires the presence of the standard text I-O driver software (file IODRIVER.Z). Both these files are noramlly loaded by the STARTUP.J file. In order for the graphics drivers to work, the GRAPHDRIVER.Z file must be loaded AFTER the IODRIVER.Z file has been loaded and initialized (by executing it).

GRAPHICS PARAMETERS

The MTU-130 display screen is actually just a very large matrix of dots, each of which may be on (a tiny spot of light) or off. Each of the 122,880 dots is independent of the others which means that any kind of image can be constructed with the proper software. The dot array consists of 256 rows of dots with 480 dots in each row. The location of any dot can be specified by giving its X and Y coordinates as illustrated below:



X defines the column number and can range between 0 and 479. Y defines the row number and can range between 0 and 255. Both coordinates are considered to be two byte quantities even though only X really requires two bytes. The least significant byte of coordinates is always first, like 6502 memory addresses.

Arguments to the graphics routines are generally stored in dedicated locations in memory. These locations are described below:

PARAMETER	LOCATION	DESCRIPTION
XC	$0202	X coordinate of the graphic cursor position, 2 bytes.
YC	$0204	Y coordinate of the graphic cursor position, 2 bytes.
XX	$0206	X graphic coordinate "register", 2 bytes.
YY	$0208	Y graphic coordinate "register", 2 bytes.
GMODE	$020A	Graphic drawing mode, $00=move, $40=erase, $80=draw, $C0=flip. Add $20 for dashed lines.
DSHPAT	$020B	Recirculating dashed line pattern, each 1 bit=dot on, 2 bytes.

NOTE: On all entry points, automatic bounds checking is performed as follows. If XC or XX is negative, it is set to 0 or if greater than 479 it is set to 479. If YC or YY is negative, it is set to 0 or if greater than 255 it is set to 255. For relative coordinates, the sum of the cursor and displacement coordinates is corrected as above.

ENTRY POINTS FOR GRAPHIC DISPLAY DRAWING

The following entry points are used for drawing points, lines, and characters on the MTU-130 display. Calls to these entry points may be intermixed with calls to the Text Display Driver described in section 8.

ENTRY POINT: SDRAW $0324

PURPOSE: To draw a solid vector from the cursor to (XX,YY).

ARGUMENTS: XC,YC = coordinates of initial endpoint.
XX,YY = coordinates of final endpoint.

ARGUMENTS RETURNED: XC,YC are set to the coordinates of the final endpoint.
GMODE is set to $80.
A, X, and Y are preserved.

DESCRIPTION:

SDRAW unconditionally draws a solid white line between specified endpoints. The setting of GMODE and DSHPAT have no effect on the line drawn. When drawing is complete, XC and YC will be set to the final endpoint coordinate in preparation for another vector connected to the vector just drawn.

ENTRY POINT: SMOVE $0327

PURPOSE: To move the graphic cursor to (XX,YY) without drawing.

ARGUMENTS: XX,YY = coordinates of point to move to.

ARGUMENTS RETURNED: XC,YC contain a copy of XX,YY.
A, X, and Y are preserved.

ENTRY POINT: SVEC $0330

PURPOSE: To draw a vector from the cursor to (XX,YY) according to GMODE and DSHPAT

ARGUMENTS: XC,YC = coordinates of initial endpoint.
XX,YY = coordinates of final endpoint.
GMODE = Type of line to be drawn.
DSHPAT = Dashing pattern if GMODE specifies a dashed line.

ARGUMENTS RETURNED: XC,YC are set to the coordinates of the final endpoint.
A, X, and Y are preserved.

DESCRIPTION:

SVEC draws a line starting at XC,YC and ending at XX,YY. When drawing is complete, XC and YC will be set to the final endpoint coordinate in preparation for another vector connected to the vector just drawn. The appearance of the line drawn depends on the setting of GMODE according to the table below:

GMODE	LINE TYPE
$00	Move
$40	Erase (draw black line)
$80	Draw (draw white line)
$C0	Flip (flip pixel state along line)
$60	Erase dashed
$A0	Draw dashed
$E0	Flip dashed

The DSHPAT parameter can be considered to be a 16 bit loop-around shift register. The register is rotated left one position for every pixel plotted when GMODE calls for a dashed line. If the bit looped around DSHPAT is a 1, then the pixel is "drawn" according to GMODE. If the bit looped around is a 0, then the pixel is skipped. DSHPAT is not reset by any entry point; it just recirculates whenever a dashed line is drawn. The default contents of DSHPAT are $F0F0 which gives dashes 4 dots long separated by 4 dots of blank space.

ENTRY POINT: SDOT $0336

PURPOSE: To draw a single dot (pixel) at (XX,YY) according to GMODE.

ARGUMENTS: XX,YY = coordinates of dot to draw.
GMODE = Type of dot to be drawn.

ARGUMENTS RETURNED: XC,YC are set to the coordinates of the dot.
A, X, and Y are preserved.

DESCRIPTION:

SDOT plots a single dot at (XX,YY) according to GMODE. After the dot is plotted, XX,YY will be copied into XC,YC. The appearance of the dot depends on the setting of GMODE according to the table below:

GMODE	DOT TYPE
$00	Move (no display change)
$40	Erase (draw black dot)
$80	Draw (draw white dot)
$C0	Flip (flip dot from white to black or black to white)

ENTRY POINT: SDRAWR $032A

PURPOSE: To draw a solid white vector relative to the cursor.

ARGUMENTS: XC,YC = coordinates of initial endpoint of the vector.
X = Signed X displacement of final endpoint from XC (-128 to +127).
Y = Signed Y displacement of final endpoint from YC (-128 to +127).

ARGUMENTS RETURNED: XC,YC and XX,YY are set to the absolute coordinates of the final endpoint.
A, X, and Y are preserved.

DESCRIPTION:

SDRAWR is similar to SDRAW except that the final endpoint coordinates are determined differently. The line drawn starts at XC,YC and ends at XC+X,YC+Y where X and Y are the contents of the X and Y machine registers. After drawing, both XC,YC and XX,YY are updated to the coordinates of the final endpoint in preparation for another line.

ENTRY POINT: SMOVER $032D

PURPOSE: To move the graphic cursor relative to its present position.

ARGUMENTS: XC,YC = coordinates of initial position graphic cursor.
X = Signed X displacement of final position from initial positon.
Y = Signed Y displacement of final position from initial position

ARGUMENTS RETURNED: XC,YC and XX,YY are set to the absolute coordinates of the final position.
A, X, and Y are preserved.

DESCRIPTION:

SMOVER is similar to SMOVE except that the final position coordinates are determined differently. The new cursor position is XC+X,YC+Y where XC,YC is the old position and X and Y are the contents of the X and Y machine registers. After moving, both XC,YC and XX,YY are updated to the new position.

ENTRY POINT: SVECR $0333

PURPOSE: To draw a vector relative to the cursor according to GMODE and DSHPAT.

ARGUMENTS: XC,YC = coordinates of initial endpoint of the vector.
X = Signed X displacement of final endpoint from XC.
Y = Signed Y displacement of final endpoint from YC.
GMODE = Type of line to be drawn.
DSHPAT = Dashing pattern if GMODE specifies a dashed line.

ARGUMENTS RETURNED: XC,YC and XX,YY are set to the absolute coordinates of the final endpoint.
A, X, and Y are preserved.

DESCRIPTION:

SVECR is similar to SVEC except that the endpoint coordinates are computed as in SDRAWR. See SVEC and SDRAWR descriptions for details.

ENTRY POINT: SDOTR $0339

PURPOSE: To draw a single dot (pixel) at a position relative to the cursor according to GMODE.

ARGUMENTS: XC,YC = Present cursor position.
X = Offset of point from cursor position in X direction.
Y = Offset of point from cursor position in Y direction.
GMODE = Type of point to be drawn.

ARGUMENTS RETURNED: XC,YC and XX,YY are set to the coordinates of the dot.
A, X, and Y are preserved.

DESCRIPTION:

SDOTR is similar to SDOT except that the coordinates of the dot are determined differently. The dot coordinates are XC+X,YC+Y where XC,YC is the cursor position and X and Y are the contents of the X and Y machine registers. After plotting the point, both XC,YC and XX,YY are updated to its position. See SDOT discription for more information.

ENTRY POINT: SDRWCH $0345

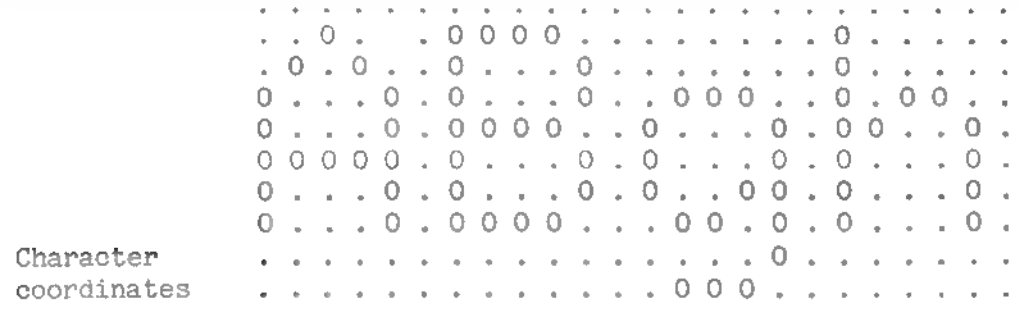
PURPOSE: To draw a single character at (XX,YY).

ARGUMENTS: XX,YY = Coordinates of lower left corner of 6 by 10 character matrix.
A = ASCII character code in range of $20 - $7F.

ARGUMENTS RETURNED: XC,YC position of character just drawn (copy of XX,YY).
XX,YY position of next character to draw (XX=XX+6, YY=YY).
A, X, and Y are preserved.

DESCRIPTION:

SDRWCH may be used to draw characters at any arbitrary location on the screen. The character cell used is 6 dots wide by 10 dots high into which a character 5 dots wide by 7 dots high is written as illustrated below:



The character's coordinates refer to the lower left corner of the 6 by 10 cell. The character's baseline is normally 2 dot rows above the bottom of the cell but lower case characters with descenders (g,j,p,q,y) will extend down to the bottom of the cell. The entire 6 by 10 character cell is cleared before the character is drawn so it may be desirable to draw characters first and then any graphics that might overlap portions of the cell. After the character is drawn, XX is incremented by 6 in preparation for another character. Thus labels for charts and graphs may be drawn easily by repeated calls to SDRWCH. The XX value must be between 0 and 474 inclusive and the YY value must be between 0 and 247 inclusive. If either is out of range, the character will not be drawn at all. Only printable ASCII character codes ($20-$7F) may be drawn, all other codes will not be drawn.

ENTRY POINT: SISDOT $0348

PURPOSE: To determine whether pixel at (XX,YY) is on or off.

ARGUMENTS: XX,YY = Position of pixel to test.

ARGUMENTS RETURNED: A=0 if pixel is off, nonzero if on.
XC,YC set equal to XX,YY
X and Y are preserved.

ENTRY POINT: SONGC $034E

PURPOSE: To turn on the graphic crosshair cursor.

ARGUMENTS: XC,YC = Position of graphic cursor.

ARGUMENTS RETURNED: None, A, X, and Y are preserved.

DESCRIPTION:

The graphic crosshair cursor consists of a full screen height vertical line drawn at the horizontal position specified by XC and a full screen width horizontal line drawn at the vertical position specified by YC. The crosshair cursor is drawn in flip mode which means that parts of an image it covers will be restored when it is later turned off with the SOFFCC entry point. Flip mode also means that the cursor will show up regardless of the background color of the screen.

ENTRY POINT: SOFFGC $034B

PURPOSE: To turn off the graphic crosshair cursor.

ARGUMENTS: XC,YC = Position of graphic cursor.

ARGUMENTS RETURNED: None, A, X, and Y are preserved.

DESCRIPTION:

This entry point turns the graphic cursor off that had been previously turned on by the SONGC entry point. For proper operation, the value of XC and YC must be the same as they were when the cursor was turned on.

ENTRY POINTS FOR USER COORDINATE INPUT

The following entry points are used for convenient operator input of position data using either the light pen or the keyboard. The light pen is best for pointing out objects that already exist on the screen or for very rapid coordinate input, i.e., drawing directly on the screen. The keyboard is best for very precise (to the pixel) location of coordinates where speed of input is less important.

ENTRY POINT: SGRIN $033C

PURPOSE: To allow user coordinate input by maneuvering a cursor with the keyboard cursor control keys.

ARGUMENTS: XC,YC = Initial position of graphic cursor.

ARGUMENTS RETURNED: XX,YY = user selected position of cursor.
A = ASCII code of key pressed to terminate the input.
XC, YC, X and Y are preserved.

DESCRIPTION:

This routine activates a rapidly blinking full-screen crosshair cursor which can be maneuvered using the cursor keys on the keyboard. It remains active until a non-cursor key is struck. It then returns the coordinates and ASCII code. The shifted cursor control keys move the cursor 5 times as fast as the unshifted cursor keys. HOME is not considered a cursor key by SGRIN.

ENTRY POINT: SLTPEN $033F

PURPOSE: Activate light pen for one frame and return coordinates of hit, if any.

ARGUMENTS: None.

ARGUMENTS RETURNED: Cy set if pen saw light, cleared if not.
XX,YY = Coordinates of hit, if any.
XX, YY, X and Y are preserved only if no hit.

DESCRIPTION:

This entry point first waits for the end of the current screen scan and then begins checking for a light pen "hit" (response to light from the screen). If light is seen during the next screen scan, the X and Y coordinates of the beam position when the hit occurred are placed in XX,YY and the carry flag is turned on. If no light is seen during the scan, the carry flag is turned off. The maximum amount of time spent in this routine is 33 milliseconds when no light is detected. The time varies from less than 1 to a maximum of 32 milliseconds when light is detected. The X and Y coordinates returned have resolution to the pixel level but a random variation up to + or - 2 coordinate units can be expected.

Light pens of course can only respond to areas of the screen that emit light. For this the light pen is well suited for quickly selecting one object from a group on the screen merely by pointing at it. Two methods may be used for "drawing" lines and curves on a blank screen. One is to display a "tracking" pattern such as a solid square 5 to 7 pixels high and wide. If the point coordinates returned by SLTPEN are not the center of the tracking pattern, it is erased and redrawn centered around the new coordinates. By doing this in a loop, the pattern will appear to "follow" the pen's movement and the program can save successive positions of the pattern. The other method is to simply fill a portion or all of the screen with white and then store the series of coordinates generated by repeated calls of SLTPEN. The points could even be plotted in black as they are generated with little effect of the pen's operation.

IMPORTANT NOTE:

The light pen will generally not respond to features less than 2 pixels wide horizontally. Thus single dots and vertical lines that are only one pixel wide will be invisible to the pen unless the screen brightness is very high.

ENTRY POINT: SINTLP $0351

PURPOSE: Wait for end of frame and then activate the light pen.

ARGUMENTS: None.

ARGUMENTS RETURNED: None, X and Y are preserved.

ENTRY POINT: STSTLP $0354

PURPOSE: Test for light pen hit and return coordinates if a hit.

ARGUMENTS: None.

ARGUMENTS RETURNED: Cy set if pen saw light previously, cleared if not.
XX,YY = Coordinates of hit, if any.
XX, YY, X and Y are preserved only if no hit.

DESCRIPTION:

STSLP makes an immediate test of the light pen "hit" status and then quickly returns. Any amount of time may elapse between a light pen hit and when STSTLP is called to compute the X and Y coordinates of the hit.

CHAPTER 10.

SYSTEM CUSTOMIZATION

System customization (often called "system generation") is the procedure for "customizing" CODOS to a particular machine configuration and set of operator preferences. Since the MTU-130 may be equipped with from 1 to 4 disk drives and any of a variety of printers (or no printer at all) and other I/O devices, provision has been made for accommodating alterations with a minimum of difficulty. The Setup and Installation manual tells how to get your system going the first time. This chapter tells how to avoid unecessary preamble when starting-up your system. Once "Customized", the modifications become a permanent part of the system on disk. When the system is booted-up, the operating system will be immediately ready to respond to your needs. The MTU-130 is equipped with a number of utility programs and built-in capabilities for self-modification, which are described briefly below and in more detail later in this section.

First-time Power-up Procedures

The first-time-power-up procedures are discussed in the Setup and Installation manual which is the first manual in the MTU-130 system notebook. In particular, the startup procedures establish, on a temporary basis, the number of disk drives in the system. This attribute is only patched into memory, however, and will be lost as soon as power is removed from the system. Therefore two interactive Utility programs are provided to permanently update the operating system on disk.

System Generation Utility Programs

The SYSGENDISK Utility program is provided to adjust CODOS for the number of disks in your system, and to "fine tune" the system to get the most out of your particular disk drives. You should run SYSGENDISK after you have copied the Distribution diskette to your first working diskette. Normally, you will only run this Utility program once, unless you change the number or type of disk drives in your system. SYSGENDISK is described in detail later in this section.

The SYSGENDEVICE Utility program is provided to add new Input-Output devices to CODOS. Once you have defined a new I-O device, you can assign channels to it and perform any input or output desired. You will want to run this Utility program whenever you add a new peripheral I-O device such as a printer to your system. SYSGENDEVICE is described in detail later in this section.

The SYSGENPRINTR Utility program is provided to automatically generate a machine language printer driver routine for almost any kind of printer which you might want to use on the MTU-130. It is a conversational program which asks you questions about your printer and then writes a machine language program which you may use immediately. SYSGENPRINTR is described in detail later in this section.

STARTUP.J File

Another feature of CODOS which greatly facilitates customization is the STARTUP.J file. When CODOS is booted-up, it first loads the operating system into memory. It then will read a list of CODOS Monitor commands from a file called STARTUP.J, and execute them just as though they were typed by you at the Console. Therefore if you have any special needs for your system, they can be attended to without operator intervention at this time. For example, if you need to load your various device-drivers into memory or wish to modify the keyboard repeat speed, you can let the STARTUP.J file do this for you. More information on modifying the STARTUP.J file is provided later in this section.

CUSTOMIZING DISK ATTRIBUTES WITH THE SYSGENDISK UTILITY PROGRAM

The version of CODOS shipped with the MTU-130 computer is set up to be usable on the broadest possible range of MTU-130 hardware configurations. This means however that as received, CODOS is unlikely to be optimized for your particular hardware configuration and desires. The SYSGENDISK utility is provided to allow you to easily optimize CODOS. Note that in all cases except one, running SYSGEN-DISK is optional; if the default parameters are acceptable, it need not be run. The one exception is systems that have only one disk drive. If you don't run SYSGENDISK to customize CODOS for single drive operation, you will have to make the first-time- power-up patch described in the Setup and Installation manual every time you turn the system on.

SYSGENDISK allows you to modify the following disk system parameters (values in parentheses are the default values on the Distribution disk):

1. The number of disk drives in the system (2)

2. The number of disk buffers in the system (6)

3. The disk drive track-to-track step time (8 milliseconds)

4. The disk drive head load time (50 milliseconds)

The meaning and selection of each of these parameter values is described in the following sections, followed by a description of SYSGENDISK operation.

Number of Disk Drives

CODOS needs to know how many disk drives are available in the system. In particular, it needs to know whether there is just one drive or more than one so that the proper copy routine is used by the FORMAT utility. The OPEN command also needs to know the number of drives so that it can properly flag as an error any attempt to open a non-existant disk drive. For specialized applications, you can specify fewer drives than are actually present. Note however that there is a very substantial advantage in having at least two drives available because copy and backup operations are much more automatic. CODOS on the Distribution disk is set up for two disk drives. Up to four drives are permitted.

Number of Disk Buffers

In order to do disk operations, CODOS requires a number of disk buffers. Each disk buffer is a 256 byte region of memory. These are always embedded in system memory above $C000 and so do not directly tie up any user memory below $C000. Every disk drive above two requires a buffer to hold the block allocation map for that drive. Every file that is assigned to a channel also needs a buffer. There is enough room in system memory for a maximum of 8 disk buffers. The default is 6 buffers. This leaves 2 buffers (512 bytes) unused during normal CODOS operations. These two buffers are used, however, by certain utility programs such as DISKETTE and BACKUP. Therefore it is recommended that the default number of buffers be retained unless you have a specific need for more simultaneously-active files. Reducing the number of buffers used will increase the amount of memory which is available for other uses. If fewer than 6 disk buffers are specified, then fewer files may be simultaneously active. The extra memory freed up by specifying fewer than 6 buffers may be used for custom I/O device drivers if desired. See Tables 10-1 and 10-2 below for specifics regarding the effect of changing the number of disk buffers available.

TABLE 10-1: NUMBER OF SIMULTANEOUSLY ACTIVE FILES

# DRIVES IN SYSTEM	NUMBER OF BUFFERS 2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
2	2	3	4	5	6	7	8
3	*	2	3	4	5	6	7
4	*	*	2	3	4	5	6

NOTES FOR TABLE 10-1:

1. The values shown are the maximum number of simultaneously active files for the listed combination of buffer count and disk drive count.

2. An entry of "*" indicates an illegal combination and that more buffers must be specified.

TABLE 10-2: FREE SYSTEM RAM ADDRESSES

# OF DISK BUFFERS	ADDITIONAL FREE RAM AVAILABLE
7	$D300-D3FF
6	$D300-D4FF (Normal configuration)
5	$D300-D5FF
4	$D300-D6FF
3	$D300-D7FF
2	$D300-D7FF, $E000-E0FF

NOTES FOR TABLE 10-2:

1. The DISKETTE and BACKUP utilities use $D300-$D4FF for special buffers. You should therefore use these utilities with caution if you increase the number of buffers to greater than 6. In this case you should ensure that you FREE all channels assigned to files before running the utilities to avoid conflicts.

Track-to-Track Step Time

The disk controller hardware needs to know how fast the disk drives are able to respond to track seek step commands. If the disk controller tries to step too fast, the disk drive will "loose its place" and not position the head over the correct track. The default value of 8 milliseconds is slow enough to accomodate all disk drives supplied by MTU. If you have double-sided drives supplied by MTU or have supplied your own drives with a faster seek time than 8 milliseconds, you may wish to change the step time. Decreasing the step time will substantially improve the performance of the system. The optimum settings for drives supplied by MTU are shown below:

MTU System Number	Disk Drive Type	Track-to-Track Step Time
MTU-130-1S	Single-sided	8
MTU-130-1D	Double-sided	3
MTU-130-2S	Single-sided	8
MTU-130-2D	Double-sided	3

If you have disk drives from another source, you will have to find the rated track-to-track step time in the drive manual or continue to operate with the default. Remember that the SYSGENDISK utility requires the step time in hexadecimal so a step time of 10 milliseconds would be entered as an A.

Head Load Time

The disk controller hardware needs to know how fast the disk drives are able to press their read/write head against the diskette. If the disk controller tries to read or write too soon after commanding the head to load, read or write errors are likely. The default value of 50 milliseconds is slow enough to accomodate all MTU disk drives but may be slower than is necessary. Longer than necessary head load time usually has a small impact on system performance except when copying small records from one drive to another. The table below may be referred to if you have disk drives supplied by MTU:

MTU System Number	Disk Drive Type	Head Load Time
MTU-130-1S	Single-sided	36 ($24)
MTU-130-1D	Double-sided	50 ($32)
MTU-130-2S	Single-sided	36 ($24)
MTU-130-2D	Double-sided	50 ($32)

If you have disk drives from another source, you will have to find the rated head load time in the drive manual or continue to operate with the default. DO NOT experiment to find the fastest allowable head load time since read errors are hidden by CODOS and write errors may not be noticed until long after they are made. Remember that the SYSGENDISK utility requires the head load time in hexadecimal so a value of 30 milliseconds would be entered as 1E.

Any changes you make using SYSGENDISK are made to the copy of the operating system on the disk in drive 0 (not to the memory-resident image of CODOS currently being run), and therefore will not be activated until you reboot the system. The changes made are permanent (until you rerun SYSGENDISK). Any copies of the system made using FORMAT after running SYSGENDISK will also have the new attributes. Do not run SYSGENDISK on the Distribution disk.

The disk in drive 0 must not be write-protected, and the file CODOS.Z must be unlocked before executing SYSGENDISK. To begin, type the CODOS command:

```
SYSGENDISK
```

which initiates the program. The first prompt will be:

```
THIS UTILITY MODIFIES DISK ATTRIBUTES
FOR CODOS ON DRIVE 0 DISK
DEFAULTS SHOWN IN ( ).
WANT TO PROCEED (Y)?=
```

If your reply is just a carriage return or starts with "Y", the program will proceed; otherwise, it terminates. The "Y" displayed in parentheses indicates the default reply if a carriage return is entered. The disk in drive 0 will be accessed momentarily, and the Utility will display:

```
# OF DRIVES (2)?=
```

The number in parentheses is the present number of drives known to CODOS. Enter the number of disk drives in your system, 1 to 4. Double-sided drive counts as one drive, not two. The next prompting message displayed is:

```
# DISK BUFFERS (6)?=
```

which requests the number of disk buffers for the system, 2 to 8. Normally you will simply reply with a carriage return to this prompt.

After specifying the number of disk buffers in the system, the next prompt is:

```
TRACK STEP RATE ($08 MS)?=
```

The present setting of the step rate will be shown in parentheses. Entering a carriage return will keep the present step rate. The value you type must be an integer between 1 and $F (15 decimal).

```
HEAD LOAD TIME ($32 MS)?=
```

The present value is shown in parentheses and is retained if you type a carriage return. Legal values may range from 2 to $FE milliseconds in 2 millisecond increments. After you enter your value for head load time, the operating system file will be updated on disk and the Utility will terminate with the message:

```
SYSTEM MODIFIED.
CHANGES WILL BE ACTIVATED ON NEXT POWER-UP.
SUGGEST YOU LOCK CODOS.Z
```

This completes SYSGENDISK. To test the modified system, re-boot CODOS.

ADDING A PRINTER TO THE MTU-130

The first peripheral device most people add to their MTU-130 computer is a printer. The MTU-130 can accommodate almost any printer for small computers on the market today, with the appropriate interface cable. Either parallel (Centronics interface) or serial (RS-232 interface) printers can be used.

Adding a printer involves three steps:

(1) Make or buy an appropriate interface cable for your printer.
(2) Generate a software printer-driver program to run your printer.
(3) Tell CODOS that you have a printer and where the driver routine is.

To help you with the first step, please refer to tables 10-4 and 10-5 which specify the cable connection requirements for several popular printers. If your printer is not among these, you will need to consult the printer manual. If you do not wish to make your own cable, a local computer store can make one for you.

For many people, the most difficult part of adding a printer is writing the software "driver" subroutine in machine language. Luckily, the SYSGENPRINTR utility program will normally do all the work for this step for you! This program will ask you questions about your printer and write an output-only printer driver configured according to your answers. This should be sufficient to handle most of the general purpose printers available for microcomputers. SYSGENPRINTR is explained in detail later in this section. In rare instances printers will be too unusual to handle with SYSGENPRINTR. If your printer requires special control codes or error handling, the printer driver generated by the SYSGENPRINTR program may not be sufficient. Guidlines for creating your own printer driver from scratch are included later in this section.

Finally, you will need to permanenetly modify CODOS so that it "knows" that you have a printer available. To do this, run SYSGENDEVICE, which is another conversational utility program. SYSGENDEVICE is described in detail later in this section. Once you have performed these steps, your printer will be immediately available for use as soon as you power-up your MTU-130.

TABLE 10-3: INTERFACE ATTRIBUTES FOR SELECTED PRINTERS

Parallel Interface Printers

PRINTER	STROBE PULSE	BUSY SIGNAL	NOTES
MX70	N	H	
MX80	N	H	Select PRINT AND LINE FEED on buffer full.
IDS440	N	H	

Serial Interface Printers

PRINTER	DATA BITS	STOP BITS	PARITY	BAUD RATE	NOTES
ANACOM 150	8	2	N	9600	Set 9600 baud rate Set BUSY = - volts

TABLE 10-4: CONNECTIONS ON MTU-130 END OF PRINTER CABLE

PARALLEL		SERIAL	
Connectors - T&B/ANSLEY 609-36M		Conntectors - Type: DB-25P	
PIN #	SIGNAL NAME	PIN #	SIGNAL NAME
19	GROUND	7	GROUND
27	DATA 0 (PB0)	2	DATA OUT
9	DATA 1 (PB1)	5	CTS (Clear To Send)
28	DATA 2 (PB2)		
10	DATA 3 (PB3)		
29	DATA 4 (PB4)		
11	DATA 5 (PB5)		
30	DATA 6 (PB6)		
12	DATA 7 (PB7)		
31	BUSY (CB1)		
13	STROBE (CB2)		

NOTES FOR TABLE 10-X:

1. The parallel connections use CB2 for the strobe signal. It may be necessary to turn off the printer if you want to use the CB2 signal to generate sound, such as with BASIC's TONE command.

TABLE 10-5: CONNECTIONS FOR PRINTER END OF CABLE FOR SELECTED PRINTERS

	PARALLEL		SERIAL	
	EPSON MX70 MX80	IDS 440		ANACOM 150
SIGNAL NAME	PIN #	PIN #	SIGNAL NAME	PIN #
GROUND	19	7	GROUND	7
DATA 0	2	14	DATA	3
DATA 1	3	13	CTS	11
DATA 2	4	12		
DATA 3	5	11		
DATA 4	6	10		
DATA 5	7	9		
DATA 6	8	15		
DATA 7	9	-		
BUSY	11	19		
STROBE	1	3		

RUNNING SYSGENPRINTR

Make sure that you have a non-write-protected disk in drive 0. To run the SYSGENPRINTR program, enter the CODOS command:

SYSGENPRINTR

The program will inform you of its function and ask if you want to continue. To continue, enter "Y" and then carriage return, or just carriage return. If not, enter "N" and then carriage return.

The SYSGENPRINTR program will now ask you a series of questions about how to configure your printer driver, based on the characteristics of your printer. All responses consist of entering the appropriate key followed by a carriage return. In cases where a default selection is supported, you may respond with just a carriage return. The default selection is indicated by an underline. If none of the selections are underlined, then a default selection is not allowed. If a response is not valid, the question is repeated until a valid response is obtained. In the discussions below, the necessity of a carriage return may not be mentioned, but is always implied.

Question 1 - Do you want your printer driver to perform TAB expansion (Y/N)?

This question should be answered by entering "Y" or "N" to signify yes or no, respectively. Answering yes means that the hex code $09 (CNTL-I) will be interpreted as a tab character. The printer driver will count characters starting from the last carriage return or form-feed sent. When a TAB character is received, spaces will be output until the next tab position is reached. The tab positions are determined by the contents of the Global Tab Table found at $06E0. If beyond the last tab stop, no spaces are printed.

Answering no means that the characters will be sent to the printer with no special handling for tabs.

Question 2 - Do you want your printer driver to insert line feeds after carriage returns (Y/N)?

Answering yes means that after every carriage return ($0D) sent to the printer, the printer driver will automatically send a line feed ($0A). This assumes that your printer does not automatically line feed when a carriage return is received. Answering no will configure the printer driver to not send a line feed after a carriage return.

Question 2 - Does your printer have a parallel or serial interface (P/S)?

This question should be answered by entering "P" or "S" to signify parallel or serial, respectively. If parallel is selected, a sequence of questions dealing with the parallel interface will be asked. Skip to Parallel Printer Driver section for the remaining questions. If serial is selected, a different sequence of questions will be asked. Skip to Serial Printer Driver section for the remaining questions.

Parallel Printer Driver

The SYSGENPRINTR program will configure a parallel interface which sends 8 bits of data. A character will be transferred to the printer with a strobe pulse, to which the printer should respond by activating a Busy signal. This Busy signal should remain active until the printer is ready to receive another character.

You may refer to Table 10-3 to see if your printer is listed. If so, you may use the answers provided. If not listed, you should refer to your printer manual to determine the proper answers.

Question 3P - Does your printer require a negative or positive going STROBE pulse (N/P)?

This question should be answered by entering "P" or "N" to signify positive or negative, respectively. Answering positive will select a positive STROBE pulse. This means that the STROBE signal will normally be low, with a positive pulse occuring when a character is to be transferred to the printer. Answering negative will select a negative STROBE pulse. In this case the STROBE signal will normally be high, with a negative pulse occuring when a character is to be transferred. Answering this question with a carriage return will select the default, which is a negative STROBE pulse.

Question 4P - Does your printer generate an active high or low BUSY signal (H/L)?

This question should be answered by entering "H" or "L" to signify high or low, respectively. Answering high will cause the printer driver to look for a high-to-low transition of the BUSY signal to indicate when the printer is ready to receive another character. Answering low will cause the printer driver to look for a low-to-high transition of the BUSY signal to indicate when the printer is ready to receive another character. Answering with a carriage return will select the default, which is a high BUSY signal.

This concludes the parallel configuration questions. The configured printer driver will be written to the file PRINTDRIVER.Z. If the file already exists, you are given the choice of overwriting the file or aborting the writing of the file. The SYSGENPRINTR program then returns to CODOS. The file written will have two parts. The printer driver will occupy the region from $D280 to $D2FF. A second part, which is an initialization routine, will start at $B400. This initialization part can be freely overwritten by other programs (such as the utility programs), because it is executed only once when the driver is initialized.

Serial Printer Driver

The SYSGENPRINTR program will configure a serial interface which transmits the appropriate number of bits with the desired parity and baud rate. The printer driver will make use of the 6551 serial interface IC in the MTU-130 to perform the transmission. The 6551 IC expects the CTS (Clear to send) signal to go high when the printer is not ready to receive characters. (IMPORTANT NOTE: If the CTS line is raised while a character is being transmitted, the remaining untransmitted bits of the character are forced to ones. This will cause an incorrect character to be transmitted. The serial printer driver will wait 2 milliseconds after each character is transmitted to allow time for the printer to raise the CTS signal. If your printer does not update the CTS signal in this time frame, you will need to make other arrangements).

You may refer to Table 10-3 to see if your printer is listed. If so, you may use the answers provided. If not listed, you should refer to your printer manual to determine the proper answers.

Question 3S - Does your printer require 7 or 8 data bits (7/8)?

This question should be answered by entering "7" or "8" to signify 7 or 8 data bits, respectively. If 8 is selected, all 8 bits in each byte are transmitted. If 7 is selected the least significant 7 bits of each byte are transmitted. Answering with a carriage return will select the default, which is 8 data bits.

Question 4S - Does your printer require 1 or 2 stop bits (1/2)?

This question should be answered by entering "1" or "2" to signify 1 or 2 stop bits, respectively. Answering with a carriage return will select the default, which is 1 stop bit.

Question 5S - Does your printer require ODD, EVEN, MARK, SPACE or No parity (O,E,M,S,N)?

This question should be answered by entering "O", "E", "M", "S", or "N". The "O" and "E" keys selects odd or even parity, respectively. The "M" and "S" keys select transmission of a mark (1) or space (0) bit, respectively. The "N" key selects no parity. If you've previously selected 8 data bits and 2 stop bits, you will be informed that a response other than "N" will reduce the number of stop bits to 1. Answering with a carriage return will select the default of no parity.

Question 6S. What is the desired baud rate (50,75,110,134,150,300,600,1200,1800, 2400,3600,4800,7200,9600,19200)?

This question should be answered with a the desired baud rate. This baud rate should be one of those listed in parentheses.

This concludes the serial configuration questions. The configured printer driver will be written to the file PRINTDRIVER.Z. If the file already exists, you are given the choice of overwriting the file or aborting the writing of the file. The SYSGENPRINTR program then returns to CODOS. The file written will have two parts. The printer driver will occupy the region from $D280 to $D2FF. A second part, which is an initialization routine, will start at $B400. This initialization part can be freely overwritten by other programs (such as CODOS utility programs), because it is needed only during initialization.

FINAL STEPS FOR ADDING A PRINTER

To make CODOS aware of the printer device you must run the SYSGENDEVICE utility as described later in this section. Respond with "1" to the first question to add a device. Next, you must specify a single character device name. "P" is the standard name for a printer, but you may respond with any single letter except "C" or "N". The printer driver doesn't perform any input, so respond with a carriage return to the prompt for an input driver address. Respond with "D280" as the output driver address. This completes the configuration of CODOS for a printer driver. Before using the printer driver, you must first execute the command:

```
PRINTDRIVER.Z
```

which loads the printer driver code, and executes the initialization routine. You will normally want to add this command to the STARTUP.J file. You must execute this file (not just GET it) to perform the necessary initialization. You should also note that after pressing RESET on the keyboard, it will be necessary to re-execute PRINTDRIVER.Z before using the printer, because RESET clears the I-O ports on the MTU-130. Failure to initialize the printer driver will cause the system to "hang" or crash when an attempt is made to print.

WRITING AND ADDING YOUR OWN I/O DRIVERS

This section describes how you can write your own device drivers for additional printer types not already covered in this section or other output or input devices, and how to define them to CODOS. Up to six additional device drivers, each potentially capable of input and output, may be added. Except for a little bit of programming, adding a custom device driver is almost as easy as adding one of the standard ones described previously.

The following list outlines the steps necessary for interfacing a new device to CODOS:

1. Decide how the device will interface electrically with the system, i.e., parallel port, serial port, custom logic board, etc.

2. Wire the interface cable or build the interface board and test the electrical interface.

3. Write the driver program according to the guidelines to be described.

4. Decide where the driver program is to reside and then assemble it.

5. Run the SYSGENDEVICE utility to define the device and driver to CODOS.

6. Test the device and its driver interface to CODOS.

7. Modify the STARTUP.J file so that the new driver is automatically loaded when the system is booted up.

Each of these steps will now be described in detail.

ELECTRICAL INTERFACE

Generally the device itself will dictate the interface method to be used. A printer with a parallel interface for example would normally be interfaced through the User parallel port connector on the MTU-130 rear panel. With the parallel port, it is best to use the "A side" ports first and save the "B side" for later use. In particular you try to avoid using the CB1 and CB2 signals since some BASIC programs may use them for generating sounds. It is helpful to scan the Programming chapter of the Monomeg hardware manual to determine what the exact capabilities of the built-in parallel and serial I/O ports are.

INTERFACE CABLE

Typically a device interface will use only a portion of the parallel or serial I/O port signals. Therefore it is wise to construct the interface cable plug that mates with the MTU-130 so that another cable that connectes to another device and uses the remaining signals can be attached to the same plug at a later date. It is sometimes possible to obtain a parallel port mating connector that has both a plug and a socket together as well as an exit hole for the cable somewhat like a Christmas tree light set. This would allow two or more interface cable sets to be "stacked up" and the devices used simultaneously as long as there are no electrical conflicts among the devices.

Unlike many operating systems, CODOS makes very few demands on the device driver routines it interfaces to. A full bi-directional (both input and output) driver routine may have as many as 3 entry points whereas a simple output-only routine might have only one entry point.

The initialization entry point is optional. If the device or the driver routine must be initialized before it can be used for input or output, this entry point is expected to perform that initialization. The initialization entry point will be called by CODOS only once at the time the driver routine is loaded into memory. There are no arguments passed and the state of the registers upon return is immaterial. Stack usage should be balanced however and an RTS instruction should be used to return to CODOS when initialization is complete. The initialization entry point may use any CODOS facilities it desires such as SVCs.

The input byte entry point is present only if the device is capable of an input function. This entry point is called every time CODOS wishes to read a byte from the device. The driver is expected to wait until the device has a byte available, read it, and return it to CODOS in the A register. All 8 bits of the byte are significant so if the most significant bit needs to be masked off, the driver should do it. The carry flag should be returned clear unless you wish to signal end-of-file. Note that inputting a CNTRL-Z character ($1A) does not indicate end-of file. This allows all 256 possible codes to be input. If your input device is a character-oriented device, CODOS expects an ASCII CR (Carriage Return) to be used for end-of-line and for a feed to the next line. If the device uses a different convention (such as separate carriage return and line feed functions), the input driver should filter out a line feed immediately following a carriage return. The X and Y registers need not be saved or restored. Stack usage must be balanced and an RTS must be used to return to CODOS. The input byte entry point may not use SVCs. It is recommended that the stack or locations within the driver routine itself be used for temporary storage.

The output byte entry point is present only if the device is capable of an output function. This entry point is called every time CODOS wishes to write a byte to the device. The driver is expected to wait until the device is ready to accept a byte and then send it the byte CODOS has passed in the A register to the device. All 8 bits of the byte may be significant so if the device requires the most significant bit to be zero, the output driver should mask it off. CODOS uses an ASCII CR character to end a line and automatically feed to the next line. If the device requires separate LF characters to feed to the next line, the output driver should insert them after CR characters itself. The A, X, and Y registers need not be saved or restored. Stack usage must be balanced and an RTS must be used to return to CODOS. The output byte entry point may not use SVCs. It is recommended that the stack or locations within the driver routine itself be used for temporary storage.

The listing on the next page is an example of a simple printer driver. Of course every printer is different but this should serve to illustrate how a driver routine is written.

LISTING 10-1: SAMPLE DEVICE DRIVER

```
ANACOM PRINTER DRIVER FOR MTU-130 CODOS          MTU 6502 ASM 1.0   *UNDATED*

0001 0000                .PAGE   'ANACOM PRINTER DRIVER FOR MTU-130 CODOS'
0002 0000          ;
0003 0000          ;       THIS PRINTER DRIVER WORKS FOR THE ANACOM 150 PRINTER, AND MAY WORK
0004 0000          ;       FOR OTHER PRINTERS WITH SERIAL INTERFACE AS WELL.  TO MODIFY THE
0005 0000          ;       BAUD RATE AND OTHER TRANSMISSION PARAMETERS, SEE THE 6551 DATA SHEET
0006 0000          ;       AND PRINTER OWNERS MANUAL AND MODIFY THE BYTES SET BY "INITPR"
0007 0000          ;       APPROPRIATELY.  THIS VERSION USES 9600 BAUD WITH NO PARITY.
0008 0000          ;       TO INITIALIZE, EXECUTE ENTRY POINT INITPR.  TO OUTPUT CHARACTER,
0009 0000          ;       USE ENTRY POINT OUTPR WITH DESIRED CHARACTER IN A.  OUTPR RESTORES
0010 0000          ;       A, X, AND Y REGISTERS; INITPR RESTORES X AND Y.  NOTE THAT ANACOM
0011 0000          ;       REQUIRES A LF INSTEAD OF A CR FOR A NEW LINE; THIS WILL NOT BE TRUE
0012 0000          ;       FOR MOST OTHER PRINTERS.  THE DRIVER MUST BE RE-INITIALIZE AFTER ANY
0013 0000          ;       RESET OF THE SYSTEM.
0014 0000          ;
0015 D280  =       CODORG  =       $D280   ;STARTING LOCATION FOR ROUTINE
0016 0000          ;
0017 0000          ;       6551 CHIP EQUATES...
0018 0000          ;
0019 BFC8  =       DR      =       $BFC8   ;6551 TRANSMIT/RECEIVE DATA REGISTER
0020 BFC9  =       SR      =       DR+1    ;RESET/STATUS REGISTER
0021 BFCA  =       COMR    =       SR+1    ;COMMAND REGISTER
0022 BFCB  =       CTRR    =       COMR+1  ;CONTROL REGISTER
0023 0000          ;
0024 0000                  *=      CODORG  ;ENTRY POINT FOR NORMAL CHARACTER OUTPUT...
0025 D280          ;
0026 D280 48       OUTPR   PHA             ;SAVE CHARACTER TO PRINT
0027 D281 ADC9BF   OUTPR1  LDA     SR      ;EXAMINE STATUS REGISTER
0028 D284 2910             AND     #$10    ;ISOLATE "TRANSMITTER READY" BIT
0029 D286 F0F9             BEQ     OUTPR1  ;WAIT TILL ITS READY
0030 D288 68               PLA             ;THEN RECALL CHARACTER
0031 D289 C90D             CMP     #$0D    ;IS IT A CARRIAGE RETURN?
0032 D28B F004             BEQ     OUTPR4  ;IF SO BRANCH
0033 D28D 8DC8BF           STA     DR      ;ELSE OUTPUT TO TRANSMITTER
0034 D290 60               RTS             ;RETURN TO CALLER
0035 D291          ;
0036 D291 A90A     OUTPR4  LDA     #$0A    ;REPLACE CR WITH ASCII LF
0037 D293 8DC8BF           STA     DR      ;OUTPUT TO TRANSMITTER
0038 D296 A90D             LDA     #$0D    ;RESTORE REG
0039 D298 60               RTS             ;RETURN TO CALLER
0040 D299          ;
0041 D299          ;       ***COME HERE TO INITIALIZE WHEN DRIVER IS LOADED...***
0042 D299          ;
0043 D299                  .ENTRY
0044 D299 8DC9BF   INITPR  STA     SR      ;STORE ANYTHING TO RESET CHIP
0045 D29C A90B             LDA     #$0B    ;NO PARITY, NO ECHO, RTS ON, NO INTERRUPT,
0046 D29E 8DCABF           STA     COMR    ;RTS ON.
0047 D2A1 A91E             LDA     #$1E    ;8 DATA BITS, 1 STOP BIT, 9600 BAUD.
0048 D2A3 8DCBBF           STA     CTRR
0049 D2A6 60               RTS             ;RETURN TO CALLER
0050 D2A7          ;
0051 D2A7                  .END
   0 ERRORS IN PASS 2
```

WHERE TO PUT THE DRIVER

Generally I/O drivers are expected to remain in memory ready for use regardless of what programs may have been run since CODOS was last booted up. To meet this goal it is necessary to store the driver in an area of memory away from the user area that extends from $0700 through BFFF. CODOS has a 128 byte area from $D280 through $D2FF reserved for this purpose. This 128 bytes is generally enough to hold a printer driver and is often adequate for two or three simple drivers.

Another area that may be used, particularly on systems with only two disk drives, is one or more of the disk buffers. On dual drive systems you can use the 512 bytes from $D300-D4FF and still have enough buffers for 6 simultaneously active files. However, certain utilities such as DISKETTE and BACKUP also use this area, so you will need to reload your driver routines after running such utilities. Device drivers are expected to reside in memory bank 0.

SYSGENDEVICE UTILITY

This utility program is used to interface your device driver program to CODOS. The utility will first ask you what single letter name you wish to associate with the device. "P" for printer is obvious but any letter that is not already used is acceptable. Please note that "C" and "N" have already been assigned to the console and null device respectively. It will next ask you what the entry point address for input is. If your driver can do input, enter the hex address of the input entry point. If it cannot, reply with a carriage return to prevent CODOS from ever trying to input from this device. Finally it will ask the entry point address for output. If your driver can do output, enter the address of the output entry point; otherwise just enter a carriage return. Remember that SYSGENDEVICE only modifies CODOS on the disk in drive 0, it does not affect the copy currently in memory. You will have to re-boot to get the modified version of CODOS in memory before you can test the driver with CODOS. SYSGENDISK is described in more detail later in this secion.

The initialization entry point should be the entry point address specified when the driver program object code is saved on disk. Then if initialization is necessary, the driver is loaded and initialized simply by giving its name, either from the keyboard or a job file. If there is no initialization entry point, then the GET command would be used to load the driver.

TESTING

After CODOS is modified with SYSGENDEVICE and the driver itself is loaded into memory (and initialized if necessary), it is ready to be tested. For an output device like a printer, you can do this simply by entering the command: TYPE C P (assuming the device name was "P"). This "connects" the keyboard to the device and every line you type will be sent to the device when you hit carriage return. The connection is broken by typing a cntl/Z. For an input device, you could enter: TYPE T C (assuming the device name was "T") and input from the device will appear on the console display line-by-line. Receipt of an ASCII SUB (cntl/Z or $1A) or pressing the INT key will restore normal console operation. Of course these suggestions only apply to text-oriented devices that use the ASCII character code. Other device types will have to be tested with a program.

LOADING WITH THE STARTUP.J FILE

After the new device and driver is throrughly tested, you will probably want the driver to be automatically loaded whenever the system is turned on so that the device will be ready for use. This is accomplished by adding a line to the STARTUP.J file that loads the driver into memory and runs it if initialization is necessary. Please refer to the STARTUP.J section of this chapter for further information on this procedure.

INTERRUPT DRIVEN I/O

You may use interrupt-driven devices if desired. In this case, the device's interrupt service routine should input or output bytes into a local buffer. The Device-In or Device-Out driver should transfer one byte between the accumulator and this buffer during each call from CODOS.

RUNNING SYSGENDEVICE

To ammend CODOS for new devices, first make sure you have a non-write protected disk in drive 0 with CODOS.Z unlocked. Then execute the CODOS command:

```
SYSGENDEVICE
```

which starts the interactive program for changing the names and characteristics of I-O devices on your system. Once these modifications are made, you will be able to assign a CODOS channel to the device and perform input-output. The modifications which you make become permanent (until you run SYSGENDEVICE again), and any copies of the modified system made using the FORMAT utility will also possess the modified I-O device attributes. Table 10-7 lists requirements for device-driver subroutines under CODOS. Once you have written or obtained the necessary device-driver subroutine, you can add your new device to CODOS' device table by executing SYSGENDEVICE. The Utility prompts:

```
THIS UTILITY PERMANENTLY MODIFIES THE
DEVICE DRIVER TABLE IN CODOS ON DRIVE 0.
DO YOU WANT TO:
(0) QUIT,
(1) ADD A DEVICE, or
(2) DELETE A DEVICE, OR,
?=
```

Enter the appropriate number, 0 to 2, and a carriage return. For example, assuming that you wished to add a line printer device, you would enter "P" for the line printer name. The Utility will respond with:

```
INPUT DRIVER ADDR.  (CR=NONE)?=
```

If your device does not have an input capability (for example, a line printer), respond with a carriage return. Otherwise, enter the address of the machine-language driver subroutine for inputting a character from your new device. See note 3 below. The next prompt is:

```
OUTPUT DRIVER ADDR.  (CR=NONE)?=
```

In a like manner, enter the address of the character-output driver routine. The output driver address for the standard printer driver generated by SYSGENPRINTR is $D280. You will enter:

```
D280
```

The program will terminate with the message:

```
MODIFICATION COMPLETE.
SUGGEST YOU LOCK CODOS.Z
```

This completes the procedure. The new device will be available to your system as soon as you re-boot CODOS.

Deleting an existing device is accompished in a similar manner by responding with a "2" to the initial SYSGENDEVICE prompt and then indicating the device to delete.

NOTES:

1. The file CODOS.Z must be unlocked prior to executing SYSGENDEVICE or no changes will be made and an error message will be issued.

2. The modifications made to the system are made on the disk copy of the system in drive 0; therefore, the changes will not become effective until the system is booted up.

3. The requirements for device drivers are summarized in Table 10-7.

4. To modify an existing device, first delete it and then rerun SYSGENDEVICE to add the same device.

5. You may not delete the Console or Null devices.

6. Naturally, the SYSGENDEVICE Utiltiy does not automatically save your device driver(s) on disk; it is your reponsibility to see that they are loaded into memory before being executed. This can be done automatically during booting-up, if desired, as explained elsewhere in this section.

7. You may add up to 6 custom devices besides the null device and Console.

8. When answering questions which have a "no change" option for a reply, remember that the "present" status of the system is the status of the system on disk 0, not the present memory-resident CODOS image.

TABLE 10-7: CONSOLE AND DEVICE DRIVER REQUIREMENTS

Driver Subroutine	Function and Requirements
Device-In	Input byte from device. This routine must return the byte of data from the device in the A register. It does not have to restore X or Y before returning. The Carry should be cleared. See note 1 below.
Device-Out	Output byte to device. This routine should output the contents of the accumulator to the device. It does not have to restore the X or Y registers before returning. See note 4 below.

NOTES FOR TABLE 10-7.

1. CODOS can input or output all 256 possible byte codes to devices. If an input device returns the carry set, however, the system will interpret it as an end-of-file indication.

2. I-O drivers may not use SVC's and should return in non-decimal mode.

3. You may use interrupt-driven devices if desired. In this case, the device's interrupt-service routine should input or output bytes into a local buffer. The Device-In or Device-Out driver should transfer one byte between the accumulator and this buffer during each call from CODOS.

4. The line terminator is a CR character. If your device needs a LF, your driver should add it.

USING THE STARTUP.J FILE

Using the STARTUP.J file is perhaps the most flexible and powerful method of system modification. Since the STARTUP.J file can contain any list of commands (built-in or user-defined), you can include SET commands to automatically "patch" the parameter area or operating system image in memory after it is loaded. The I/O driver parameters you would most likely want to change are described later in this section. CODOS parameters are described in Appendix E.

Since the STARTUP.J file is nothing more than a file of ASCII text, you can write your own STARTUP.J file by simply using the TYPE command the text editors. To be on the safe side, we suggest you create your new command file under another name, and then, when you are sure its correct, DELETE the old STARTUP.J and RENAME your new file as STARTUP.J. Generally speaking, you can design your own STARTUP.J file as you please, but you should keep in mind the notes listed below when doing so. Never modify the STARTUP.J file on the Distribution disk.

NOTES:

1. Keep in mind that only CODOS itself is automatically loaded by the bootstrap loader PROM. All other programs and subroutines needed for system operation (such as the keyboard and text display I/O drivers) must be loaded by commands in the STARTUP.J file.

2. You can't do any input or output to a device until its driver subroutines are loaded (or executed if initialization is needed).

3. Any error detected by the system causes CODOS to stop reading the STARTUP.J file and to try to issue an error message. Thereafter it will try to read from the Console. It is normally a good idea to laod and initialize your Console device drivers (by executing IODRIVER.Z) as soon as possible in the STARTUP.J sequence, so that if you have a mistake in the STARTUP.J file or other problem, you will be able to see the error message.

4. The STARTUP.J file must GET SVCPROC.Z if you plan to use SVCs. Almost all Utility programs including the Editor require the SVC processor.

5. The STARTUP.J file must GET GRAPHDRIVER.Z if you plan to use graphics. All of the Graphics Libraries for BASIC except KGL assume that this file has been loaded into memory. The Editor also requires GRAPHDRIVER.Z.

6. You cannot LOAD or SET into reserved memory unless you UNPROTECT first.

7. It is a good practice to re-PROTECT the system after you are done with any modifications.

8. Do not change the STARTUP.J file on the Distribution disk provided by MTU.

SAMPLE STARTUP.J #1:

Below is a listing of the standard STARTUP.J file supplied on the Distribuiton disk:

```
;This is the default STARTUP.J file for MTU-130 CODOS 2.0...
IODRIVER.Z ;Load & initialize Console I-O drivers.
GET SVCPROC.Z ;Load SVC Processor.
GET GRAPHDRIVER.Z ;Get Graphics Drivers (Needed by EDIT).
DATE ;Prompt for entry of date.
```

The IODRIVER.Z file is loaded and run in the second line to insure that the console I/O devices are initialized prior to being used. The SVC processor and graphics subroutines don't require initialization so they are just loaded by the next two lines. The last line executes the CODOS DATE command. Since it is the last line in the job file, CODOS will read commands from the console keyboard after the date is entered.

SAMPLE STARTUP.J #2:

This is similar to the standard startup file except that the user prefers somewhat different parameter values for the keyboard. He has also connected a printer to the system and wishes to automatically start executing an assembly language application program after the date is entered.

```
IODRIVER.Z  ;LOAD AND INITIALIZE CONSOLE I-O DRIVERS
GET SVCPROC.Z  ;LOAD SVC PROCESSOR
GET GRAPHDRIVER.Z  ;LOAD GRAPHICS DISPLAY ROUTINES
PRINTDRIVER.Z  ;LOAD AND INITIALIZE MY PRINTER DRIVER
SET 221 .150  ;WANT ABOUT 25CPS KEYBOARD REPEAT RATE
SET 213 80  ;I LIKE SILENT KEYBOARDS
DATE
BASIC
RUN STOCKANAL ;RUN MY PORTFOLIO ANALYSIS PROGRAM
```

The first three lines are the same as example 1. The fourth line loads and initializes a driver program for a printer that has been added to the system. The next 2 lines redefine some of the keyboard and sound parameters of the system. The parameter addresses and their effects may be found in the next section or in Chapter 8. The last lines will cause the BASIC program called STOCKANAL to be loaded and executed automatically.

SAMPLE STARTUP.J #3:

This might be the STARTUP.J file that goes with an integrated laboratory data aquisition and analysis software package. The startup file defines the function keys such that pressing one will run a corresponding program from the package.

```
IODRIVER.Z  ;LOAD AND INITIALIZE CONSOLE I-O DRIVERS
GET SVCPROC.Z  ;LOAD SVC PROCESSOR
GET GRAPHDRIVER.Z  ;LOAD GRAPHICS DISPLAY ROUTINES
DATE  ;ASK FOR DATE
ONKEY 1 'NMR READ' 'NMR READ' ;Program to operate our NMR instrument.
ONKEY 2 'GC READ'  'GC READPROG' ;Program to operate our gas chromatagraph.
ONKEY 3 'INTEGRAT' 'PEAKINTGRATE'  ;Program to estimate fraction quantity.
ONKEY 4 'VIB ANAL' 'DO VIBRATEANAL.J'  ;Link to vibration analysis.
ONKEY 5 ' BASIC' 'MTUBASIC'  ;Allow general purpose computer use too.
```

The first 4 lines are the same as the standard startup file in example 1. The next 3 lines set up function keys 1, 2, and 3 to load and execute an assembly language program when they are pressed. The definition of function key 4 illustrates some of the power of function keys and job files. Pressing key 4 will cause a job file called VIBRATEANAL.J to be executed. This job file could in turn redefine the function keys for various component programs of a vibration analysis package. The last line sets up key 5 to simply put the user in the BASIC interpreter for general purpose computing.

I/O DRIVER PARAMETERS

The system parameter area in low memory ($0200-027F) contains a number of parameters that affect the "feel" of the console to a great extent. Default values of these parameters have been selected that hopefully will satisfy most users. If you wish to change any of these parameters, refer to the guidelines below for help in determining their values. It is most convenient to include SET commands to set the parameters in the STARTUP.J file but you may also enter SET commands through the console any time CODOS is in control. Programs can also change the parameters while they run. Only the most commonly altered parameters are explained here, a complete list may be found in Chapter 8.

PARAMETER: RPTRAT - Keyboard repeat rate.

ADDRESS: $221

DEFAULT VALUE: $C3 (195)

DESCRIPTION: This parameter determines how fast the keyboard repeats. The default value of $C3 gives a rate of approximately 20 characters per second. To make the rate slower, increase the value up to a maximum of $FF. To make it faster, decrease the value. The parameter is actually the repeat period (time between repeats or 1/rate) in units of .000256 second. Remember that the repeat rate will slow down if character processing takes longer than the repeat period.

PARAMETER: CURDLA - Cursor flash rate.

ADDRESS: $222

DEFAULT VALUE: $06

DESCRIPTION: This parameter determines how fast the cursor flashes while waiting for keyboard input. To make the rate slower, increase the value. To make it faster, decrease the value. To eliminate the cursor altogether, set it to zero. Note that the cursor stays on when it is moving so slower flash rates will not materially affect the cursor's maneuverability.

PARAMETER: NOCLIK - Presence of audible key click.

ADDRESS: $213

DEFAULT VALUE: $00

DESCRIPTION: This parameter determines whether the keyboard will click when keys are pressed. It is normally zero which allows clicks. If it is set to $80, then clicks will be suppressed and permit silent keyboard operation.

PARAMETER: CLKVOL - Volume of audible key click.

ADDRESS: $225

DEFAULT VALUE: $20

DESCRIPTION: This parameter determines how loudly the keyboard will click (provided clicking is enabled). To make it louder, increase the value up to a maximum of $7F. To make it softer, decrease the value. The waveform period (CLKPER) and duration (CLKCY) will also affect the apparent loudness to some extent.

PARAMETER: CLKPER - Pitch of audible key click.

ADDRESS: $224

DEFAULT VALUE: $05

DESCRIPTION: This parameter determines the pitch of the keyboard click. To increase the pitch, reduce the value. To decrease the pitch, increase the value. Note that a value of 0 is interpreted as 256. The actual tone frequency in Hertz is 500C/N where N is the parameter value. When using the higher pitches, you may wish to increase the duration (CLKCY) or volume (CLKVOL) to retain good audibility.

PARAMETER: CLKCY - Duration of audible key click.

ADDRESS: $226

DEFAULT VALUE: $02

DESCRIPTION: This parameter determines the duration of the keyboard click. To increase the duration, increase the value up to a maximum of $7F. To reduce the duration, reduce its value. The number of waveform cycles produced is one plus the duration parameter value. Note that the time required to produce the click tone is added to the character processing time so if the duration is excessive, keyboard response will seem sluggish.

PARAMETER: BELVOL - Volume of audible bell tone.

ADDRESS: $228

DEFAULT VALUE: $40

DESCRIPTION: See the CLKVOL parameter description for details.

PARAMETER: BELPER - Pitch of audible bell tone.

ADDRESS: $227

DEFAULT VALUE: $05

DESCRIPTION: See the CLKPER parameter description for details.

PARAMETER: BELCY - Duration of audible key click.

ADDRESS: $229

DEFAULT VALUE: $0C

DESCRIPTION: See the CLKCY parameter description for details.

PARAMETER: TABTBL - Tab stop table.

ADDRESS: $6E0-6FF

DEFAULT VALUE: $09, $11, $19, $21, $29, $31, $39, $41, $49, $00, . . .

DESCRIPTION: This parameter is actually a list of up to 32 tab stops. The values stored actually represent the column number that a tab stop is located on. The first zero value marks the end of the table. The values must be stored in ascending sequence. Remember that the leftmost character position is column one, not zero.

APPENDIX A
CODOS ERROR MESSAGES

CODOS Error #	Error Message
1	Command not found.
2	File not found.
3	Drive needed is not open.
4	Syntax error in command argument.
5	Missing or illegal disk drive number.
6	Drive needed is not ready.
7	Locked file violation.
8	Missing or illegal channel number.
9	Channel needed is not assigned.
A	Diskette is write-protected.
B	Missing or illegal device or file name.
C	Missing or illegal file name.
D	Not a loadable ("SAVEd") file.
E	<from> address missing or illegal.
F	<to> address missing or illegal.
10	<from> address greater than to address.
11	Reserved or protected memory violation.
12	<value> out of range (greater than $FF or less than 0).
13	Arithmetic overflow.
14	<entry> address missing or illegal.
15	New file on write-protected diskette.
16	Illegal or unimplemented SVC number.
17	Memory verify failure during SET or FILL.
18	<value> missing or illegal.
19	New file name is already on selected diskette.
1A	Missing or illegal character string delimiter (' , ").
1B	<destination> address missing or illegal.
1C	Missing or illegal register name.
1D	All buffers in use (free a chan. assigned to a file).
1E	Unformatted disk or irrecoverable read/write error.
1F	Breakpoint table full (3 BP's already set).
20	Write-protected disk or formatting error.
21	Input from output-only device, or visa-versa.
22	Not enough channels are free for specified function.
23	No CODOS on drive 0, or system overlay load error.
24	Illegal entry into CODOS system.
25	Required software package not loaded in memory.
26	Diskette is full; all blocks already allocated.
27	Diskette is full; no room left in directory.
28	Unformatted diskette or drive went not-ready.
29	Unformatted diskette or irrecoverable seek error.
2A	Unformatted diskette or hardware drive fault.
2B	System crash: illegal system overlay number.
2C	System crash: illegal sector on disk.
2D	System crash: directory/file table check error.
2E	System crash: file ordinal check error.
2F	System crash: illegal track on disk.
30	System crash: NEC 765 chip command phase error.
31	System crash: NEC 765 chip result phase error.
32	System crash: Directory redundancy check failed.
33	Missing or illegal memory bank number.
34	Missing or illegal function key number.

When an error is detected by CODOS, the program being executed is aborted and an error number is displayed on channel 2 (the Console). If the error occurred in a built-in command, CODOS will display the erroneous command and an "up arrow" character pointing to the next character of the command which CODOS was going to examine. Note that this is not necessarily the location of the error! The error could be anywhere before the up-arrow. If the error occurred during a user-program the registers will be displayed in the state they were in when the offending SVC was issued. If the error occurred during a CODOS Utiltiy, the registers show the location where the error was detected in the Utility.

CODOS will issue an English error message detailing the problem if it can. These error messages reside on the text file SYSERRMSG.Z. Therefore if the system can't read this file from drive 0 it won't issue the message. Keeping the error messages on disk greatly reduces the amount of memory required for the operating system.

Provision has been made for user-defined error recovery in lieu of the default error recovery by CODOS. This capability is provdied by SVC number 25 and is described in Chapter 6.

APPENDIX B

CODOS FILE FORMATS

From the programmer's viewpoint, a CODOS file is simply an array of bytes with a pointer to the current file position. Reads and writes always take place starting at the current file position and advance the file position pointer by the number of bytes read or written. The size of the file can be freely increased or decreased at any time. Any write-operation which crosses the current end-of-file will automatically increase the size of the file. A file can be truncated so as to make the present file position end-of-file. The file can be repositioned at will (but not beyond the present end-of-file) by using SVC #19. This structure is called a byte-addressable file and is the most versatile file organization available on a computer.

There are no reserved "codes" for end-of-file, end-of-line, etc. You may freely write and read all 256 possible bytes at any position in the file. This lets you decide the file organization that makes sense for your application, rather than letting the operating system dictate restrictions that make life simple for it. CODOS keeps track of the present End-of-File by an internally-maintained pointer. You can always determine the present End-of-File position from within a program by positioning the file to End-of-File and executing SVC #20 to read the file position.

Normal CODOS text files consist of variable-length lines terminated by an ASCII CR (carriage return = $0D).

Loadable files (such as are generated by the SAVE command) have the following format:

Size	Description
1 byte	$58 = ASCII "X" = CODOS loadable file header byte.
1 byte	Overlay number, normally $00 (can be defined by assembler .OVL pseudo-op).
1 byte	Memory bank number, either 0, 1, 2, or 3.
1 byte	$00 = Reserved for future; always $00.
2 bytes	entry address. Entry point into module. If not applicable, set to same as from address, below.
2 bytes	from address. Starting load address for block in memory.
2 bytes	Size (not final address!) of memory block to be loaded.
n bytes	Actual memory image to be loaded.

If multiple blocks are stored on the file, the above format is merely repeated as many times as necessary. The GET command continues loading until End-of-File is encountered or until a non-0 overlay number is encountered in the header. A special overlay loader would presumably process blocks with non-zero overlay numbers.

EXAMPLE:

The standard CODOS Utility COPYF loads into $B400 though $B698 of bank 0 with an entry point at $B400. The first few bytes of the file are (in hex):

58 00 00 00 00 B4 00 B4 99 02 A9 01 8D ...

where the last three bytes (A9 01 8D) are the first three bytes of the actual program.

APPENDIX C

BOOTSTRAP LOADER OPERATION

The CODOS system is loaded into RAM by the 256-byte bipolar PROM on the Floppy Disk Controller Board. This PROM occupies addresses $FF00-$FFFF in bank 0. This means that the Reset, Maskable interrupt (IRQ) and non-maskable (NMI) vectors are also located in this PROM. The Reset vector points to the beginning of the PROM ($FF00), the IRQ vector points to location $02FD, and NMI points to location $02FA. The PROM operates as follows:

1. Clear decimal mode, define stack pointer = $FF.
2. Test the keyboard "MOD" key and jump to $0300 if not down, otherwise continue.
3. Copy disk controller command strings from PROM into memory from $00C3 through $00D6.
4. Read track 0, sector 0 of drive 0 into locations $FE00 through $FEFF.
5. Determine loading information for the actual program to be loaded by examining the following addresses:
   FE3C = FINALS = Final sector number for the load.
   FE3D = DMAPG = DMA Address code for loading of the first sector.
   FE3E, FE3F = ENTRY = Address-1 of entry point into program.
6. Load sectors 1 through FINALS from track 0 into memory starting at the address corresponding to the DMA code DMAPG. See note 3 below.
7. Jump to address ENTRY+1.

Users with the necessary technical expertise may wish to use this information to boot programs other than CODOS.

EXAMPLE:

The Standard MTU-130 CODOS program is supplied on the distribution disk ready for execution by the bootstrap loader. The memory image to be loaded is stored on sectors 1 though 25 (sectors are numbered starting with 0) on track 0. Track 0 sector 0 does not contain any useful information except for the following bytes:

$3C = $19 = Final sector to be loaded into memory from track 0.
$3D = $98 = DMA address code for $E600.
$3E = $FF = Low address byte of entry point-1 ($E600).
$3F = $E5 = High address byte of entry point-1.

NOTES:

1. The bootstrap loader uses page 0 as follows:
   $00C3-$00D6 = NEC-765 Command Strings.
   $00D7-$00DF = Scratch RAM, Result phase readouts from NEC-765.

2. Except for the four bytes described in step 7 above, the Bootstrap loader does not use the information in track 0 sector 0 in any way.

3. The DMA code is the byte which is sent to the DMA address register, as discussed in the Disk Ccontroller Hardware manual. It identifies which 64-byte boundary in the disk controller RAM is to be used as the starting address for the transfer.

4. The Bootstrap PROM assumes the disk is formatted for double density operation on drive 0 with 256-byte sectors, 26 sectors per track.

5. If the Bootstrap Loader detects a disk error, it moves 5 bytes (an $AA, followed by the 4 disk controller status registers) to the beginning of display memory and then retries the disk operation. It will do this indefinitely until the disk operation is successful or Reset is hit.

6. The keyboard MOD key is used to distinguish between a cold Reset (key down) and a warm reset (key up). The warm Reset is intended to re-enter CODOS through a vector at $0300 without re-booting CODOS. A time-delay circuit effectively "presses" the MOD key on power-up to trigger a cold Reset.

APPENDIX D

SAMPLE APPLICATION PROGRAM

HIGH-SPEED, INTERRUPT-DRIVEN, DIRECT-TO-DISK DATA AQUISITION USING CODOS

Two features unique to CODOS are its high speed operation and its interruptability. Listing D-1 is a complete application program which illustrates the use of SVCs for high-speed, interrupt-driven data aquisition using a parallel port. The program is assembled for the MTU-130 computer using the User 6522 VIA device, but can easily be modified. The program itself is on the CODOS Distribution disk with the filename DAQDEMO.C. In an actual application, the interrupts would probably be generated by the input device itself (such as an A-D converter), but for purposes of illustration, the 6522's internal timer is used to generate interrupts at precisely timed intervals. This interval can be easily modified by changing the value of the constant DELAY at the end of the program. When the program starts, it inputs a value every 250 microseconds (using the value of DELAY given) and stores it on the disk, until 50,000 bytes have been read (12.5 seconds elapsed time). The values are read from the 6522 A port, which is assumed to be connected to the device of interest.

The most important point illustrated by this program is that no data is lost while the operating system is writing to disk, because CODOS can be interrupted at any time without harm. It also illustrates that a large volume of data can be written to disk in a short time.

Before using the program, you will need to prepare a new disk formatted with a "data-aquisition skew", by typing:

```
FORMAT S
```

and proceeding with the FORMAT Utiltiy in the usual manner. The "S" argument tells the operating system to arrange the sectors slightly differently from normal. This has no effect on normal operation of the disk. The reason for this operation is explained in note 1, below. Once you have FORMATted the disk and copied any desired programs onto it, ASSIGN channel 6 to the file you wish save the data on. Then execute the program. It will take about 12.5 seconds to complete, using the DELAY and NSAMP (number of samples) values given.

The program itself is composed of two separate parts: a main program, and an interrupt service routine. The interrupt service routine collects the data samples into two buffers by filling first one and then the other. As each buffer becomes full, the service routine sets a "Buffer Full" flag. The main program performs some initialization, and then waits for a "Buffer Full" condition. As soon as a buffer becomes full, the main program writes the entire buffer to channel 6 (the disk file) as one CODOS record. While the "full" buffer is being written to disk, the interrupt service routine is busy filling up the other buffer, one byte at a time. When the main program is done writing the first buffer to disk, it clears the "Buffer Full" flag and waits for the service routine to set the "Buffer Full" flag for the other buffer. This operation is called double-buffering. The flags used for handshaking between the service routine and main program are called semaphores, beause they tell when the program can "proceed". If the service routine discovers that one buffer has become full before the other buffer has been emptied by the main program, it aborts the program with a "BUFFER OVERRUN" message. This condition occurs when you decrease DELAY to the point where the service

routine is stealing such a high percentage of the machine cycles that the main program and CODOS can no longer complete all the operations needed to perform the disk write in the time it takes to fill a buffer. Another flag called "DONE" is set when the desired number of samples have been placed in the buffer. This flag tells the main program to "flush" the final, partially-filled buffer to disk, disable the timer interrupts, and free channel 6. Without this flag, the last partial buffer-full of data would never be transferred from the buffer to disk.

When studying the program, you will notice that the buffers used were quite large (8K bytes each). This is highly desirable when high-speed disk operations are desired. CODOS can usually write one record of 8K bytes considerably faster than it can write, say, 8 records of 1K bytes each. This is because each time you use an SVC to write a record, CODOS has to perform a considerable amount of "overhead", such as processing your SVC number, checking to see if the channel specified is legal and assigned, etc. This overhead may take enough time that the desired sector has already passed under the write-head on the disk, thus requiring another full one-sixth of a second for a complete disk revolution.

NOTES:

1. The CODOS system is carefully optimized to give fast loading of programs. The FORMAT Utility program numbers the sectors on the disk such that sectors that are numbered sequentially are physically located on alternating sectors on the disk, as shown below:

#0 #13 #1 #14 #2 #15 #3 #16 ... #12 #25

The numbering of sectors in this fashion is called an "interleave" of 2. When CODOS is transferring large blocks of information to or from disk starting at sector 0, it sets up for the next DMA transfer of sector 1 while sector 13 is passing under the head. If sector 1 was physically adjacent to sector 0, sector 1 would already be under the head before the system was ready to actually perform the transfer. This would mean that the disk could only access one sector per revolution instead of 13 sectors per revolution (for 26 sectors per track). Most of the time spent by CODOS is used to move data from the user's record to the system buffer in the onboard DMA memory. Moving 256 bytes to or from the user's record to the system buffer actually takes up virtually all the time available between the end of the transfer of sector 0 and the beginning of the transfer for sector 1, even with an interleave of 2. If interrupts occur during this time, the interrupt service routine may easily steal enough time so that CODOS can't complete the transfer in the time available. In this case, what is needed is an increased interleave, so that two sectors intervene between sectors 0 and 1 on the disk instead of 1. Then the timing requirements are relaxed and a large percent of the time available can be spent in the interrupt service routine. Specifying the "S" argument on the FORMAT command generates a disk with an interleave of 3 instead of 2. This will not impair operation of the system on that disk in any way; the software and hardware do not care about the physical location of the sectors on the track. The disk controller simply keeps searching till it finds the desired sector number. The only consequence to normal operations is that program loading will be somewhat slower. This difference will normally be imperceptible except for very large programs. The "S" option will permit a large number of interrupts to be made during disk accesses without a substantial performance degredation in thoughput. The interleave can only be changed at the time the disk is formatted.

2. Note that the sample program uses the 6522 timer in the free-running, interrupt mode. If the program does not complete properly for any reason, the timer may continue to interrupt. You may need to RESET to clear this condition.

3. Sample rates up to 7KHz have been obtained using this program. However, for rates above 4KHz, you should use a freshly formatted disk or one from which no files have been deleted since it was last formatted.

4. Refer to section 5.6 in the Monomeg Single Board Computer Hardware Manual for additional information on programming the 6522 I/O interface chip and alternate methods of controlling the sample rate and connecting to the data source.

DOCUMENTATION MTU 6502 ASSEMBLER 1.0

```
0002 0000                .PAGE  'DOCUMENTATION'
0003 0000        ;       CODOS DEMONSTRATION PROGRAM. 12/31/80   B. CARBREY
0004 0000        ;                                    8/29/81    REVISED H. CHAMBERLIN
0005 0000        ;
0006 0000        ;       HIGH SPEED, INTERRUPT-DRIVEN, DIRECT-TO-DISK DATA AQUISITION
0007 0000        ;
0008 0000        ;       THIS PROGRAM USES A DOUBLE-BUFFERED, INTERRUPT SERVICE ROUTINE
0009 0000        ;       TO COLLECT DATA FROM PARALLEL PORT A OF A 6522 VIA AND STORE
0010 0000        ;       IT ON A CODOS DISK FILE.  THE 6522 TIMER IS USED TO SAMPLE THE
0011 0000        ;       PORT AT USER-DEFINED INTERVALS AND STORE THE DATA READ ON DISK.
0012 0000        ;       THE SAMPLING RATE FOR THE PORT CAN BE MODIFIED BY CHANGING THE
0013 0000        ;       CONSTANT "DELAY" (APPROX. TIME BETWEEN SAMPLES IN MICROSECONDS)
0014 0000        ;       AT THE END OF THE PROGRAM.  THE TOTAL NUMBER OF SAMPLES TO BE
0015 0000        ;       TAKEN CAN BE VARIED FROM 1 TO 65,535 BY ADJUSTING THE CONSTANT
0016 0000        ;       "NSAMP" AT THE END OF THE PROGRAM.  DEPENDING ON THE SEEK TIME
0017 0000        ;       AND HEAD LOAD TIME OF YOUR DISK DRIVES, THIS PROGRAM CAN BE
0018 0000        ;       USED TO SAMPLE IN EXCESS OF 5000 BYTES PER SECOND WITHOUT LOSS
0019 0000        ;       OF DATA.  IF THE AQUISITION RATE IS INCREASED BEYOND THE
0020 0000        ;       MAXIMUM RATE WHICH THE PROGRAM CAN HANDLE, THE PROGRAM WILL
0021 0000        ;       ABORT WITH THE MESSAGE "BUFFER OVERRUN ERROR", INDICATING THAT
0022 0000        ;       ONE BUFFER WAS FILLED BEFORE THE OTHER COULD BE EMPTIED TO DISK.
0023 0000        ;
0024 0000        ;       *****IMPORTANT*****.  WHEN USING THIS PROGRAM BE SURE TO USE A
0025 0000        ;       DISK WHICH HAS BEEN FORMATTED USING "FORMAT S" TO INCREASE THE
0026 0000        ;       SKEW BETWEEN LOGICALLY ADJACENT SECTORS.  IF YOU USE A DISK
0027 0000        ;       FORMATTED WITH THE STANDARD SKEW THE ALLOWABLE AQUISITION RATE
0028 0000        ;       WILL BE VERY LOW BECAUSE THE RELATIVELY LONG SERVICE ROUTINE
0029 0000        ;       WILL "STEAL" ENOUGH CYCLES FROM THE NORMAL CODOS WRITE-RECORD
0030 0000        ;       SVC THAT IT WILL NOT BE ABLE TO TRANSFER ALL THE BYTES FROM THE
0031 0000        ;       DOUBLE BUFFER TO THE DMA BUFFER BEFORE THE READ-WRITE HEAD HAS
0032 0000        ;       PASSED THE DESIRED SECTOR, THUS REQUIRING ANOTHER FULL 1/6TH OF
0033 0000        ;       A SECOND FOR EACH SECTOR WRITTEN.
0034 0000        ;
0035 0000        ;       IN ACTUAL PRACTICE, THE SERVICE ROUTINE SHOULD BE SHORTENED AS
0036 0000        ;       MUCH AS POSSIBLE AND THE BUFFERS MADE AS LARGE AS POSSIBLE, TO
0037 0000        ;       MAXIMIZE THE CONTINUOUS THROUGHPUT TO THE DISK.  SHORTENING THE
0038 0000        ;       SERVICE ROUTINE MAY BE EASY, BECAUSE MOST "REAL" DEVICES WILL
0039 0000        ;       PROVIDE THEIR OWN INTERRUPT (E.G., A/D), SO THAT THE TIMER WONT
0040 0000        ;       BE NEEDED, AND THE DECREMENTING OF "COUNT" MAY NOT BE NEEDED.
0041 0000        ;       ALSO, PROPER HOOKUP TO THE 6522 WILL ALLOW THE ACT OF READING
0042 0000        ;       THE DATA REGISTER ALSO CLEAR THE INTERRUPT.
0043 0000        ;
0044 0000        ;       THE MOST IMPORTANT FEATURE OF THIS PROGRAM IS THAT IT PROVES
0045 0000        ;       THAT YOU CAN INTERRUPT CODOS FREELY, EVEN DURING DISK
0046 0000        ;       OPERATIONS, WITHOUT ANY PROBLEMS.
0047 0000        ;
0048 0000        ;       *DIRECTIONS: BEFORE EXECUTING, WIRE YOUR INPUT SOURCE TO PORT
0049 0000        ;       A OF THE 6522.  SET THE DESIRED SAMPLE FREQUENCY AND TOTAL
0050 0000        ;       NUMBER OF SAMPLES AT THE END OF THE PROGRAM.  ASSIGN CHANNEL
0051 0000        ;       6 TO THE DISK FILE DESIRED.  BEGIN EXECUTION.
0052 0000        ;
0053 0000        ;       THIS PROGRAM IS SET UP FOR PORT A OF THE USER PARALLEL PORT ON
0054 0000        ;       THE MTU-130 COMPUTER, BUT CAN EASILY BE RE-ASSEMBLED FOR OTHER
0055 0000        ;       DEVICES OR OTHER ADDRESSES.
```

```
0056 0000                .PAGE  'EQUATES AND PAGE 0'
0057 0000          ;
0058 0000          ;       CODOS EQUATES...
0059 0000          ;
0060 00B0 =        U0      =       $B0             ;PSEUDO REGISTER 0
0061 00B2 =        U1      =       U0+2            ;PSEUDO REGISTER 1
0062 00B4 =        U2      =       U1+2            ;PSEUDO REGISTER 2
0063 00EE =        SVCENB  =       $EE             ;SVC ENABLE FLAG
0064 0000          ;
0065 0000          ;       6522 VIA EQUATES...
0066 0000          ;
0067 BFD0 =        IO      =       $BFD0           ;6522 ADDRESS FOR USER PORT
0068 BFD0 =        UDRB    =       IO+0            ;PORT B DATA
0069 BFD1 =        UDRAH   =       IO+1            ;PORT A DATA
0070 BFD2 =        UDDRB   =       IO+2            ;PORT B DIRECTION
0071 BFD3 =        UDDRA   =       IO+3            ;PORT A DIRECTION
0072 BFD4 =        UT1L    =       IO+4            ;TIMER 1...
0073 BFD5 =        UT1CH   =       IO+5
0074 BFD6 =        UT1LL   =       IO+6
0075 BFD7 =        UT1LH   =       IO+7
0076 BFD8 =        UT2L    =       IO+8            ;TIMER 2...
0077 BFD9 =        UT2H    =       IO+9
0078 BFDA =        USR     =       IO+$A           ;SHIFT REG
0079 BFDB =        UACR    =       IO+$B           ;AUX CONTROL
0080 BFDC =        UPCR    =       IO+$C           ;PERIPHERAL CONTROL
0081 BFDD =        UIFR    =       IO+$D           ;INTERRUPT FLAGS
0082 BFDE =        UIER    =       IO+$E           ;INTERRUPT ENABLES
0083 BFDF =        UDRA    =       IO+$F           ;PORT A DATA, NO HANDSHAKE
0084 0000          ;
0085 0000          ;       PROGRAM EQUATES...
0086 0000          ;
0087 1000 =        BUF0    =       $1000           ;BUFFER 0 STARTING ADDRESS
0088 0020 =        NPAGE   =       32              ;NUMBER OF PAGES IN BUFFER 0 (8K BYTES)
0089 0030 =        B0PGLM  =       BUF0/256+NPAGE ;PAGE LIMIT FOR BUFFER 0
0090 3000 =        BUF1    =       256*NPAGE+BUF0;BUFFER 1 STARTING ADDRESS
0091 0050 =        B1PGLM  =       BUF1/256+NPAGE ;PAGE LIMIT FOR BUFFER 1
0092 0000          ;
0093 0000          ;       *0-PAGE RAM...*
0094 0000          ;
0095 0000                  *=      $10             ;***0-PAGE ORG
0096 0010          BUFPAG  *=      *+2             ;POINTER TO CURRENT BUFFER PAGE
0097 0012          YBUF    *=      *+1             ;INDEX WITHIN CURRENT PAGE
0098 0013          COUNT   *=      *+2             ;COUNTER OF SAMPLES LEFT TO TAKE
0099 0015          B0FULL  *=      *+1             ;FLAG, BUFFER 0 IS FULL
0100 0016          B1FULL  *=      *+1             ;FLAG, BUFFER 1 IS FULL
0101 0017          DONE    *=      *+1             ;FLAG, ALL DATA SAMPLES TAKEN
0102 0018          INTYSV  *=      *+1             ;USED TO SAVE Y DURING INTERRUPT
0103 00EC =        INTASV  =       $00EC           ;A SAVE LOCATION USED BY CODOS INT/BRK
0104 0019                                          ;PROCESSOR
```

```
0105 0019                .PAGE  'MAIN PROGRAM - INITIALIZATION'
0106 0019        ;
0107 0019                *=     $0700          ;*PROGRAM ORG*
0108 0700        ;
0109 0700        ;       BEGIN EXECUTION HERE AFTER ASSIGNING CHANNEL 6 TO A FILE.
0110 0700        ;
0111 0700 D8     DATAIN  CLD
0112 0701 A900           LDA    #BUF0&$FF
0113 0703 8510           STA    BUFPAG         ;DEFINE PAGE POINTER FOR SERVICE ROUTINE
0114 0705 A910           LDA    #BUF0/256
0115 0707 8511           STA    BUFPAG+1
0116 0709 A000           LDY    #0
0117 070B 8412           STY    YBUF           ;DEFINE INDEX WITHIN PAGE OF BUFFER
0118 070D 8415           STY    B0FULL         ;CLEAR ALL FLAGS...
0119 070F 8416           STY    B1FULL
0120 0711 8417           STY    DONE
0121 0713 A900           LDA    #0             ;DEFINE STARTING COUNT OF SAMPLES TO TAKE
0122 0715 38             SEC                   ;=2'S COMPLEMENT OF REQUESTED NUMBER
0123 0716 ED4A08         SBC    NSAMP
0124 0719 8513           STA    COUNT
0125 071B A900           LDA    #0
0126 071D ED4B08         SBC    NSAMP+1
0127 0720 8514           STA    COUNT+1
0128 0722 38             SEC                   ;ENABLE CODOS SVCS
0129 0723 66EE           ROR    SVCENB
0130 0725 A9D3           LDA    #SERVC&$FF
0131 0727 85B0           STA    U0             ;SET U0 = ADDRESS OF INTERRUPT SERVICE
0132 0729 A907           LDA    #SERVC/256
0133 072B 85B1           STA    U0+1
0134 072D 00             BRK
0135 072E 18             .BYTE  24             ;SVC #24 = DEFINE IRQ VECTOR
0136 072F 78             SEI
0137 0730        ;
0138 0730        ;       SETUP 6522 TIMER IN FREE-RUNNING INTERRUPT MODE...
0139 0730        ;
0140 0730 A900           LDA    #0
0141 0732 8DD3BF         STA    UDDRA          ;SET DATA DIRECTION = IN ON PORT
0142 0735 A9C0           LDA    #$C0
0143 0737 8DDBBF         STA    UACR           ;FREE RUN TIMER 1
0144 073A 8DDEBF         STA    UIER           ;ENABLE TIMER INTERRUPTS
0145 073D AD4808         LDA    DELAY
0146 0740 8DD4BF         STA    UT1L           ;LOAD DESIRED DELAY INTO TIMER
0147 0743 AD4908         LDA    DELAY+1
0148 0746 8DD5BF         STA    UT1CH          ;ACTIVATE TIMER
0149 0749 58             CLI                   ;LET 'ER RIP!
```

```
0150 074A                .PAGE  'MAIN PROGRAM - DISK WRITE LOOPS'
0151 074A          ;     COME HERE WHEN WAITING FOR BUFFER 0 TO BE FILLED BY INTERRUPT
0152 074A          ;     SERVICE ROUTINE.  SETUP FOR WRITE OF BUFFER 0 AS CODOS RECORD.
0153 074A          ;
0154 074A A900     SETUP0 LDA    #0
0155 074C 85B4           STA    U2              ;DEFINE SIZE OF RECORD = ENTIRE BUFFER 0...
0156 074E A920           LDA    #NPAGE
0157 0750 85B5           STA    U2+1
0158 0752 A900           LDA    #BUF0&$FF
0159 0754 85B2           STA    U1              ;DEFINE START ADDR. OF RECORD = BUFFER 0
0160 0756 A910           LDA    #BUF0/256
0161 0758 85B3           STA    U1+1
0162 075A A206           LDX    #6              ;DISK IS ON CHANNEL 6
0163 075C          ;
0164 075C          ;     COME HERE TO TEST SEMAPHORES FROM IRQ SERVICE ROUTINE...
0165 075C 2415     WAIT0  BIT    BOFULL          ;TEST "BUFFER 0 FULL" FLAG
0166 075E 3007           BMI    WRITE0          ;BRANCH IF BUFFER IS FULL
0167 0760 2417           BIT    DONE            ;ELSE TEST "AQUISITION DONE" FLAG
0168 0762 302F           BMI    FINI0           ;EXIT IF DONE
0169 0764 4C5C07         JMP    WAIT0           ;ELSE REPEAT
0170 0767          ;
0171 0767          ;     COME HERE WHEN BUFFER 0 IS FULL. WRITE IT TO DISK (CHAN 6)...
0172 0767 00       WRITE0 BRK
0173 0768 10             .BYTE  16              ;SVC #16 = WRITE RECORD
0174 0769 A900           LDA    #0
0175 076B 8515           STA    BOFULL          ;CLEAR "BUFFER 0 FULL" SEMAPHORE
0176 076D          ;
0177 076D          ;     SETUP FOR NEXT WRITE FROM BUFFER 1...
0178 076D          ;
0179 076D A900           LDA    #0
0180 076F 85B4           STA    U2              ;DEFINE RECORD SIZE = WHOLE BUFFER 1
0181 0771 A920           LDA    #NPAGE
0182 0773 85B5           STA    U2+1
0183 0775 A900           LDA    #BUF1&$FF
0184 0777 85B2           STA    U1              ;DEFINE RECORD START = BUFFER 1 ADDRESS
0185 0779 A930           LDA    #BUF1/256
0186 077B 85B3           STA    U1+1
0187 077D A206           LDX    #6              ;CHANNEL 6
0188 077F          ;
0189 077F          ;     COME HERE TO TEST SEMAPHORES FOR BUFFER 1 FROM IRQ ROUTINE...
0190 077F 2416     WAIT1  BIT    B1FULL          ;TEST "BUFFER 1 FULL" FLAG
0191 0781 3007           BMI    WRITE1          ;BRANCH IF FULL
0192 0783 2417           BIT    DONE            ;ELSE TEST "AQUISITION DONE" FLAG
0193 0785 301E           BMI    FINI1           ;EXIT IF THROUGH
0194 0787 4C7F07         JMP    WAIT1           ;ELSE REPEAT
0195 078A          ;
0196 078A          ;     COME HERE WHEN BUFFER 1 IS FULL.  WRITE IT TO DISK.
0197 078A 00       WRITE1 BRK
0198 078B 10             .BYTE  16              ;SVC 16 = WRITE RECORD TO CHANNEL
0199 078C A900           LDA    #0
0200 078E 8516           STA    B1FULL          ;CLEAR "BUFFER 1 FULL" FLAG
0201 0790 4C4A07         JMP    SETUP0          ;GO REFILL BUFFER 0
0202 0793          ;
0203 0793          ;     COME HERE WHEN FINISHED WHILE FILLING BUFFER 0...
0204 0793          ;
```

```
0205 0793 A512   FINIO   LDA    YBUF          ;RECALL INDEX TO NEXT BYTE IN BUF
0206 0795 38             SEC
0207 0796 E900           SBC    #BUF0&$FF     ;COMPUTE FRACTIONAL PAGE FILLED
0208 0798 85B4           STA    U2            ;DEFINE RECORD SIZE...
0209 079A A511           LDA    BUFPAG+1
0210 079C E910           SBC    #BUF0/256
0211 079E 85B5           STA    U2+1
0212 07A0 00             BRK
0213 07A1 10             .BYTE  16            ;SVC 16 FOR FINAL WRITE OF PARTIAL RECORD
0214 07A2 4CB407         JMP    EXITOK
0215 07A5        ;
0216 07A5        ;       COME HERE WHEN FINISHED WHILE FILLING BUFFER 1...
0217 07A5 A512   FINI1   LDA    YBUF          ;RECALL INDEX TO NEXT BYTE IN BUF
0218 07A7 38             SEC
0219 07A8 E900           SBC    #BUF1&$FF     ;COMPUTE FRACTIONAL PAGE FILLED
0220 07AA 85B4           STA    U2            ;DEFINE RECORD SIZE...
0221 07AC A511           LDA    BUFPAG+1
0222 07AE E930           SBC    #BUF1/256
0223 07B0 85B5           STA    U2+1
0224 07B2 00             BRK
0225 07B3 10             .BYTE  16            ;SVC 16 FOR FINAL WRITE OF PARTIAL RECORD
0226 07B4        ;
0227 07B4        ;       COME HERE FOR NORMAL EXIT
0228 07B4        ;
0229 07B4 A206   EXITOK  LDX    #6            ;CHANNEL 6
0230 07B6 00             BRK
0231 07B7 16             .BYTE  22            ;SVC 22 = FREE CHANNEL 6
0232 07B8        ;
0233 07B8 00             BRK
0234 07B9 02             .BYTE  2             ;SVC 2 = INLINE MESSAGE
0235 07BA 02             .BYTE  2             ;...ON CHANNEL 2
0236 07BB 0D             .BYTE  13            ;CARRIAGE RETURN
0237 07BC 4143..         .BYTE  'ACQUISITION COMPLETE.'
0238 07D1 00             .BYTE 0
0239 07D2 60             RTS
```

```
0240 07D3                .PAGE  'INTERRUPT SERVICE ROUTINE'
0241 07D3          ;
0242 07D3          ;       INTERRUPT SERVICE ROUTINE FOR IRQ...
0243 07D3          ;
0244 07D3 8418   SERVC   STY    INTYSV          ;SAVE Y, A SAVED IN INTASV BY CODOS
0245 07D5 ADD4BF         LDA    UT1L            ;CLEAR 6522'S INTERRUPT FLAG
0246 07D8 ADD1BF         LDA    UDRAH           ;INPUT DATA BYTE FROM PORT
0247 07DB A412           LDY    YBUF            ;RECALL INDEX TO BUFFER
0248 07DD 9110           STA    (BUFPAG),Y      ;STORE BYTE INTO BUFFER
0249 07DF C8             INY                    ;BUMP INDEX
0250 07E0 8412           STY    YBUF            ;SAVE INDEX
0251 07E2 D022           BNE    SERV6           ;BRANCH IF NOT CROSSING PAGE BOUNDARY
0252 07E4 A411           LDY    BUFPAG+1        ;ELSE RECALL HI BYTE OF BUFFER POINTER
0253 07E6 C8             INY                    ;ADVANCE
0254 07E7 C030           CPY    #BOPGLM         ;CHECK FOR END OF BUFFER 0
0255 07E9 D00C           BNE    SERV3           ;BRANCH IF NOT AT END OF BUFFER 0
0256 07EB 38             SEC
0257 07EC 6615           ROR    BOFULL          ;ELSE SET "BUFFER 0 FULL" SEMAPHORE
0258 07EE 2416           BIT    B1FULL          ;TEST "BUFFER 1 FULL" SEMAPHORE
0259 07F0 3031           BMI    OVERUN          ;BRANCH IF OTHER BUFFER NOT EMPTIED YET
0260 07F2 A030           LDY    #BUF1/256       ;ELSE SETUP POINTER TO OTHER BUFFER
0261 07F4 4C0408         JMP    SERV5
0262 07F7          ;
0263 07F7 C050   SERV3   CPY    #B1PGLM         ;CHECK FOR END OF BUFFER 1
0264 07F9 D009           BNE    SERV5           ;BRANCH IF NOT AT END OF BUFFER 1
0265 07FB 38             SEC
0266 07FC 6616           ROR    B1FULL          ;ELSE SET "BUFFER 1 FULL" SEMAPHORE
0267 07FE 2415           BIT    BOFULL          ;TEST "BUFFER 0 FULL"
0268 0800 3021           BMI    OVERUN          ;BRANCH IF OTHER BUFFER NOT EMPTIED YET
0269 0802 A010           LDY    #BUF0/256       ;ELSE SETUP FOR OTHER BUFFER
0270 0804          ;
0271 0804 8411   SERV5   STY    BUFPAG+1        ;REDEFINE POINTER TO START OF OTHER BUFFER
0272 0806          ;
0273 0806 E613   SERV6   INC    COUNT           ;INCREMENT 2'S COMP. OF SAMPLE COUNT
0274 0808 D004           BNE    SEREND          ;GO RETURN IN NOT DONE
0275 080A E614           INC    COUNT+1
0276 080C F005           BEQ    FINI            ;GO SIGNAL CONPLETION IF INCREMENT TO ZERO
0277 080E          ;
0278 080E A418   SEREND  LDY    INTYSV          ;RESTORE Y
0279 0810 A5EC           LDA    INTASV          ;RESTORE A
0280 0812 40             RTI                    ;EXIT SERVICE ROUTINE
```

```
0281 0813                .PAGE  'ALTERNATE EXITS FROM SERVICE ROUTINE'
0282 0813          ;
0283 0813          ;     COME HERE WHEN DESIRED NUMBER OF SAMPLES HAVE BEEN INPUT
0284 0813          ;
0285 0813 38       FINI  SEC
0286 0814 6617           ROR   DONE          ;SET "DONE" SEMAPHORE
0287 0816 A900           LDA   #0
0288 0818 8DDBBF         STA   UACR          ;KILL TIMER INTERRUPTS
0289 081B A940           LDA   #$40
0290 081D 8DDEBF         STA   UIER
0291 0820 4C0E08         JMP   SEREND        ;THEN NORMAL EXIT
0292 0823          ;
0293 0823          ;     COME HERE IN CASE OF BUFFER OVERRUN.  DATA WAS AQUIRED FASTER
0294 0823          ;     THAN IT COULD BE WRITTEN TO DISK.
0295 0823          ;
0296 0823 A900     OVERUN LDA   #0
0297 0825 8DDBBF         STA   UACR          ;KILL TIMER INTERRUPTS
0298 0828 8DDEBF         STA   UIER
0299 082B 78             SEI
0300 082C 00             BRK
0301 082D 02             .BYTE  2           ;SVC 2 = MESSAGE
0302 082E 02             .BYTE  2           ;...ON CHANNEL 2
0303 082F 0D             .BYTE  13          ;CR
0304 0830 4255..         .BYTE  'BUFFER OVERRUN ERROR.'
0305 0845 00             .BYTE 0
0306 0846 00             BRK
0307 0847 00             .BYTE  0           ;RETURN TO CODOS, SHOW  REGS
0308 0848          ;
0309 0848          ;     ***INSTALL DESIRED DELAY BETWEEN SAMPLES (MICROSECS), AND***
0310 0848          ;     ***TOTAL NUMBER OF SAMPLES TO BE TAKEN HERE...          ***
0311 0848          ;
0312 0848 FA00     DELAY .WORD  250          ;=4000 SAMPLES PER SECOND
0313 084A 50C3     NSAMP .WORD  50000        ;QUIT AFTER 50000 SAMPLES (12.5 SEC)
0314 084C          ;
0315 084C                .END
   0 ERRORS IN PASS 2
```

APPENDIX E

CODOS SYSTEM ADDRESSES

Table E-1 gives the addresses of certain important memory locations within the CODOS operating system nucleus which the advanced user may wish to examine or modify. Casual modification of values listed may crash the system and cause unpredictable side effects. See sections 8 and 9 for further information.

TABLE E-1: IMPORTANT CODOS SYSTEM ADDRESSES (HEXADECIMAL)

Address	Size	Description of contents
00EC	1	Accumulator save during SVC or IRQ processing.
00ED	1	Error number for user-defined error recovery.
00EE	1	SVC enable flag.
E603	3	Jump to warm re-entry point for CODOS Monitor.
E615	3	Jump to "REQUIRED SOFTWARE NOT LOADED" error message.
E621	3	Jump to console-character-out routine with CTRL-S/Q (XON?XOFF).
E6F3	9	Todays date (as entered by operator from DATE command).
E74F	1	Number of disk drives in system, 1 to 4 (See Note 2).
E75C	2	Address+1 where last error was detected by CODOS.
E763	1	Cumulative count of soft disk read errors.
E764	1	Cumulative count of soft disk write errors.
E765	1	Cumulative count of recalibrate commands issued to disk controller during read/write error recoveries.
E766	1	Sector number for last disk error causing a recalibrate.
E767	1	Track number for last disk error causing a recalibrate.
E779	1	Flag. If bit 7 = 1 then system will ignore (continue after) irrecoverable disk read errors (use a last resort only).
E77A	1	Flag. If bit 7 = 1 then permits save command to overwrite an existing file with the same name.
E780	1	Flag. If bit 7 = 1 then program executing was invoked by SVC #13.

E788	1	Keyboard echo flag for CODOS. Set to $80 to enable echo.
E793	1	Current ASCII default file extension character ("C").
E796	1	Current default drive number (Set by DRIVE command).
0238	1	Maximum record length for input line.
E798	1	Number of file names per line for FILES command (5 or less).
E799	1	Number of bytes to dump per display line.
E79F	1	ASCII character to be used in lieu of Backslash.
E7BE	2	Pointer to start of system input line buffer.
E7C0	2	Pointer to start of system output line buffer.
E7C2	2	Pointer to large transient buffer for COPYF, ETC.
E7C4	2	Size (NOT final address) of large transient buffer.
E7C6	2	Pointer to user-defined interrupt service routine.
E7C8	2	Pointer to user-defined error recovery routine.
0303	3	Jump executed when CNTRL-C is entered from console.

NOTES FOR TABLE E-1:

1. The above addresses are valid for CODOS 2.0 only and are subject to change in future revisions.

2. The SYSGENDISK utility must be used if the system is to be changed to support more than 2 disk drives.

3. Additional parameter and subroutine addresses can be found in Chapters 8, 9, 10, and Appendix F.

NOTES FOR TABLE E-1:

1. The above addresses are valid for CODOS 2.0 only and are subject to change in future revisions.

2. See Chapter 10 for a description of the requirements for Console and Device input and output routines.

3. The SYSGENDISK utility must be used if the system is to be changed to support more than 2 disk drives.

4. The flag for ignoring "strange" control keys is normally set so that control codes such as CNTRL-L (for clearing the display) can be embedded harmlessly in a CODOS command. See SVC #5 description for details.

5. Additional parameter and subroutine addresses can be found in Chapters 8, 9, 10, and Appendix F.

APPENDIX F

MTU-130 STANDARD SYSTEM MEMORY MAP

Bank 0 Address	Contents
FF00-FFFF	Bootstrap ROM, Vectors, Disk controller registers.
FE00-FEFF	CODOS System overlay RAM area.
E600-FDFF	CODOS nucleus.
E300-E5FF	CODOS Block-assignment tables for drives 1 and 0 and directory buf.
E000-E2FF	Pool disk buffers 0, 1, and 2.
DD20-DFFF	SVC Processor (see chap. 5)
D800-DD1F	Command Processor (can be overlaid; change $D800 to non-$D8 if so).
D500-D7FF	Pool disk buffers 5, 4, and 3. (can be overlaid if not used)
D300-D4FF	Optional pool buffers for 3- and 4-drive systems or UNUSED
D280-D2FF	Printer driver or UNUSED (see chap. 10)
C5B0-D27F	Console I-O dirver (can be overlaid if not using console)
C000-C5AF	Screen Graphics Drivers or UNUSED (see chap. 9)
BE00-BFFF	System I-O (or UNUSED ordinary RAM). Normally I-O is enabled.
B400-BDFF	CODOS Utility area (when needed only)
A000-B3FF	Default Large Transient Buffer (when needed only)
0700-BDFF	Normally Available RAM (or used by BASIC and BASIC program).
06E0-06FF	Tab stop table (up to 32 tab stops)
0600-06DF	System output line buffer (224 decimal bytes)
05C0-05FF	Function key legends (8 each, 8 bytes in length, see chap. 8)
0500-05BF	System input line buffer (192 decimal bytes)
0400-04FF	Function key strings (8 each, 32 bytes in length)
0306-03FF	Jump table to I-O routines, graphics, etc.
0303-0305	Jump to CNTRL/C processor
0300-0302	Jump to operating system warm reset entry
02FD-02FF	Jump to IQR and BRK processor
02FA-02FC	Jump to NMI processor
02F9	I-O space enable semaphore ("SEEIO")
02E0-02F8	OPEN (reserved for more scratch)
02B0-02DF	Scratch ram used by Console I-O and graphics drivers
0280-02AF	Scratch RAM for CODOS.
0240-027F	OPEN (reserved for more global varaibles)
0200-023F	Global variables, constants & flags for Console I-O and graphics (see chap. 8, 9, and 10)
0113-01FF	Stack.
0100-0112	CODOS bank switch/restore routine.
00F0-00FF	Scratch RAM for console I-O.
00ED-00EF	Global RAM used by CODOS
00C1-00EC	Scratch RAM used by CODOS nucleus, SVC Processor and Command Proc.
00B0-00C0	Pseudo registers U0-U7 (or available RAM).
0000-00AF	UNUSED

Bank 1	
C000:1-FBFF	Bit-mapped CRT display RAM
FC50:1-FD4F	Backtrack buffer (for lines recalled by CNTRL-B).
FD50:1-FFEF	Standard 96-character ASCII font table for CRT.

APPENDIX G

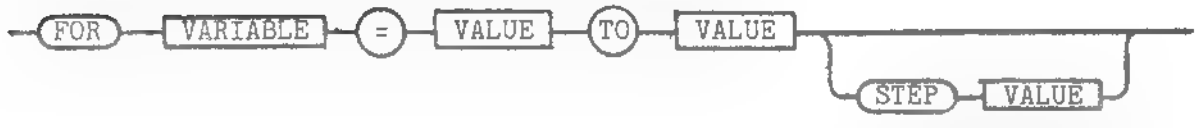
SYNTAX DIAGRAMS FOR CODOS BUILT-IN COMMANDS

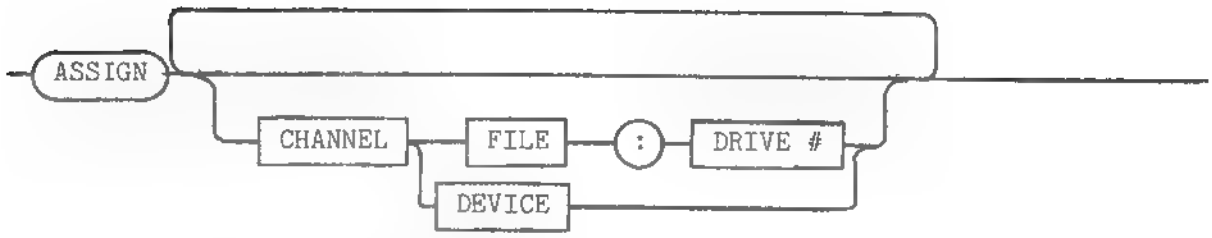
Figure G-2 provides syntax diagrams which unambiguously define how CODOS built-in commands may be legally constructed. These diagrams are called Wirth diagrams, in honor of Niklaus Wirth, who popularized the use of these diagrams in order to define the Pascal programming language.

To construct a legal CODOS command, you may follow any path indicated by the diagram. The rounded-enclosures and circles are CODOS keywords and delimiters which must be entered exactly as shown. Rectangles enclose names of entities which must be provided by the user. For example, figure G-1 shows a Wirth diagram for a BASIC language FOR statement. If you were to "read" this diagram "out loud", you might read it as follows:

"A BASIC 'FOR' statement is the keyword 'FOR', followed by a variable, followed by an "=", followed by a value, followed by a 'TO' keyword, followed by a value, optionally followed by the keyword 'STEP' and a value."

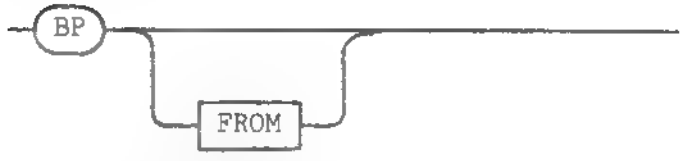
FIGURE G-1: WIRTH DIAGRAM FOR BASIC "FOR" STATEMENT

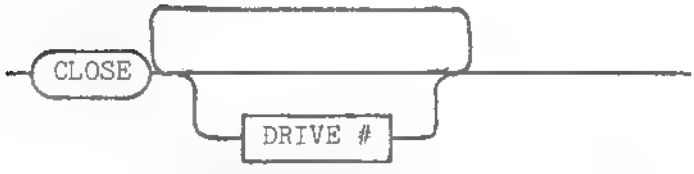


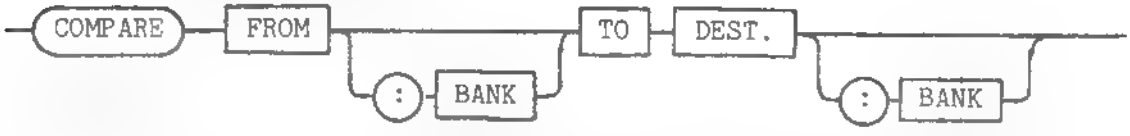

ASSIGN
CHANNEL
FILE
:
DRIVE #
DEVICE

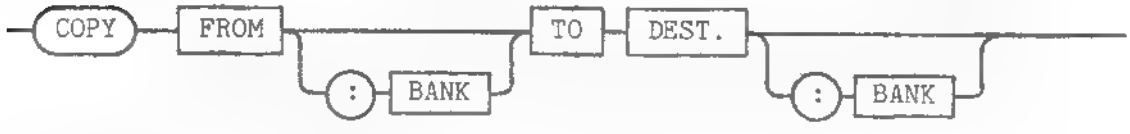

BEGINOF
CHANNEL

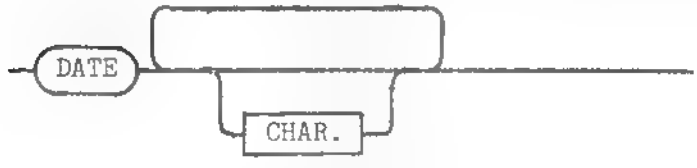

BOOT

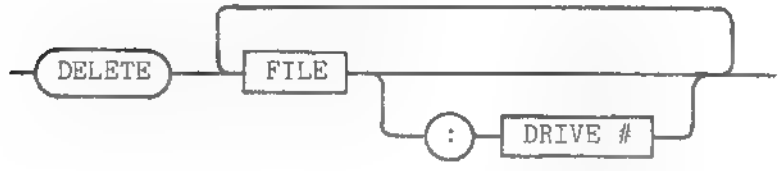

BP
FROM


CLOSE
DRIVE #

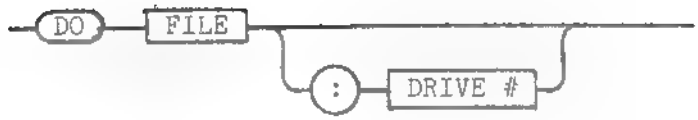

COMPARE
FROM
:
BANK
TO
DEST.
:
BANK


COPY
FROM
:
BANK
TO
DEST.
:
BANK

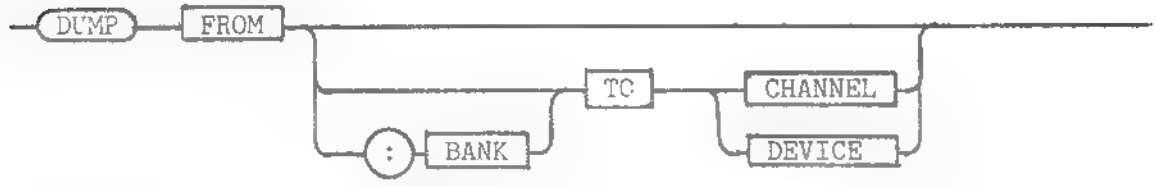

DATE
CHAR.


DELETE
FILE
:
DRIVE #

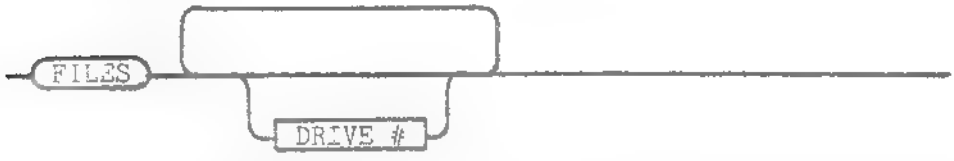

DISK

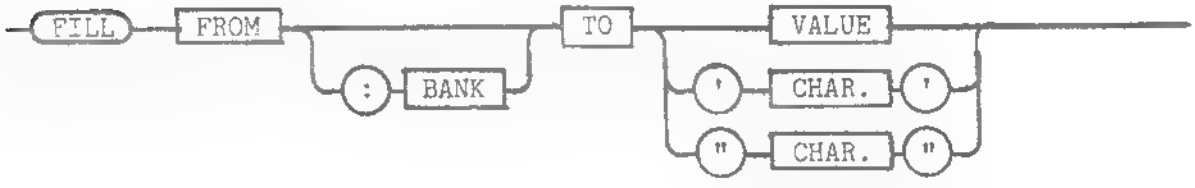

DO
FILE
:
DRIVE #


DRIVE
DRIVE #

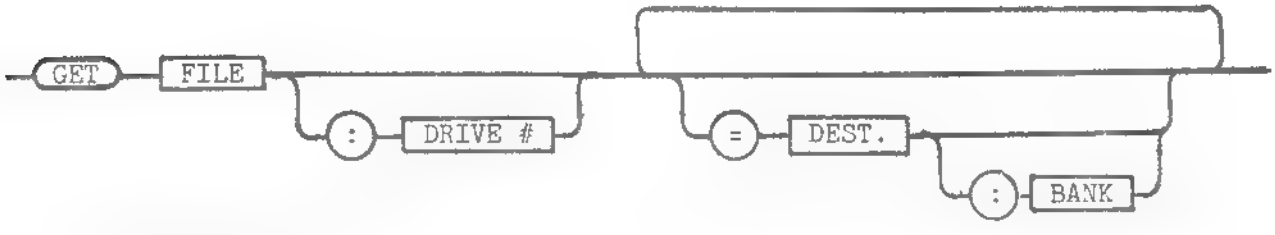

DUMP
FROM
:
BANK
TO
CHANNEL
DEVICE

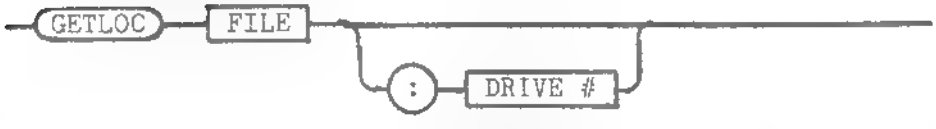

ENDOF
CHANNEL


FILES
DRIVE #

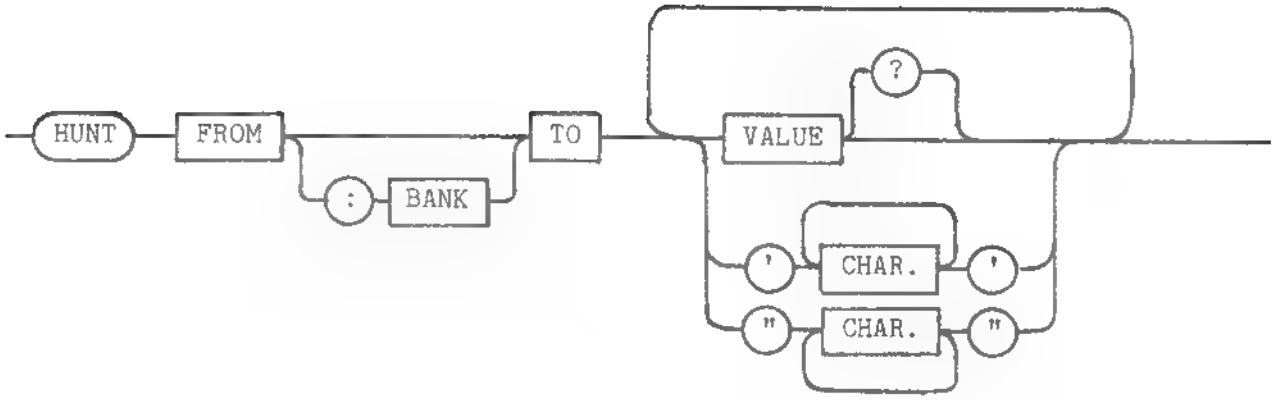

FILL
FROM
:
BANK
TO
VALUE
'
CHAR.
'
"
CHAR.
"

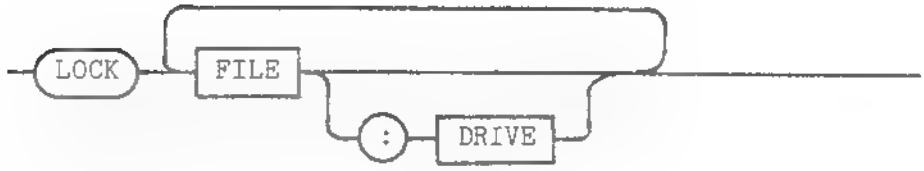

FREE
CHANNEL

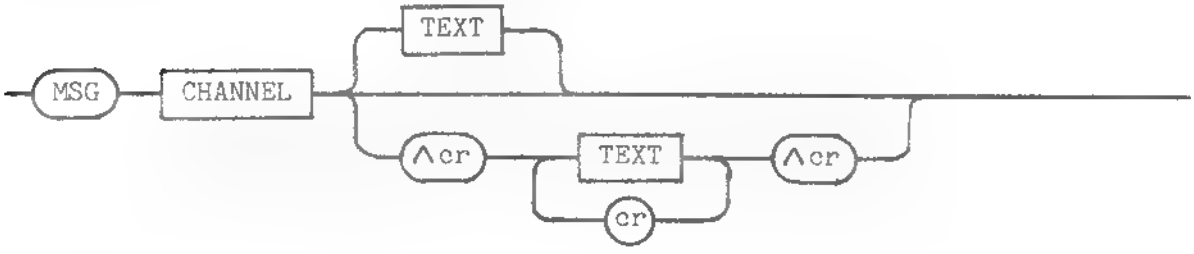

GET
FILE
:
DRIVE #
=
DEST.
:
BANK

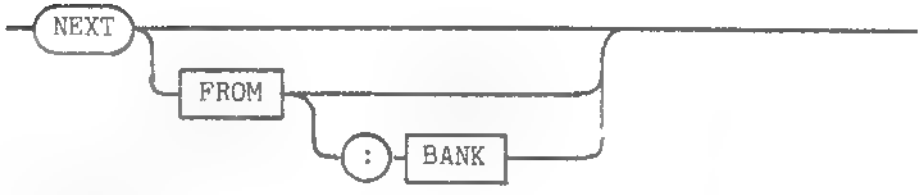

GETLOC
FILE
:
DRIVE #

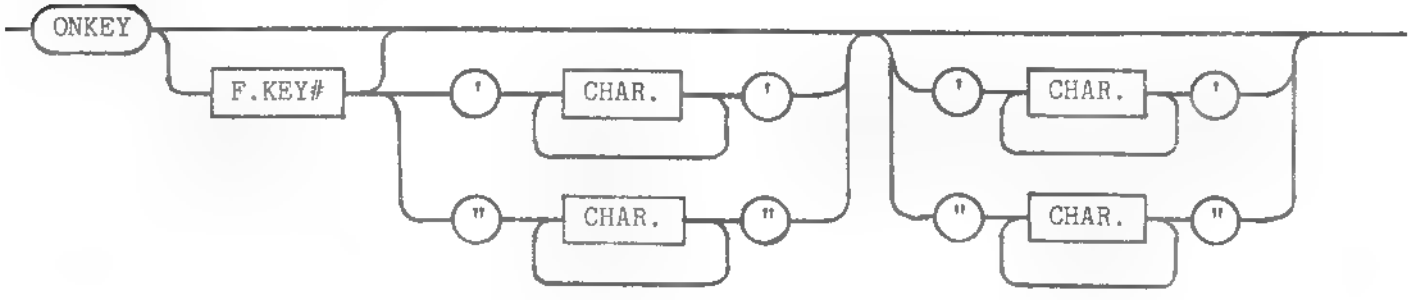

GO
FROM
:
BANK

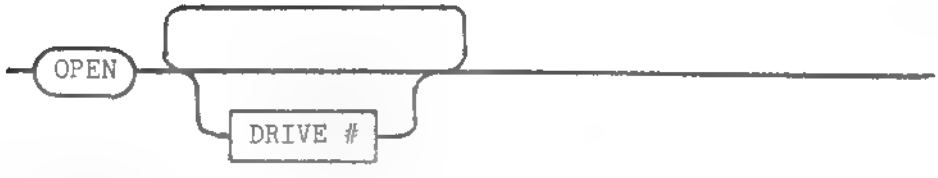

HUNT
FROM
:
BANK
TO
VALUE
?
'
CHAR.
'
"
CHAR.
"


LOCK
FILE
:
DRIVE

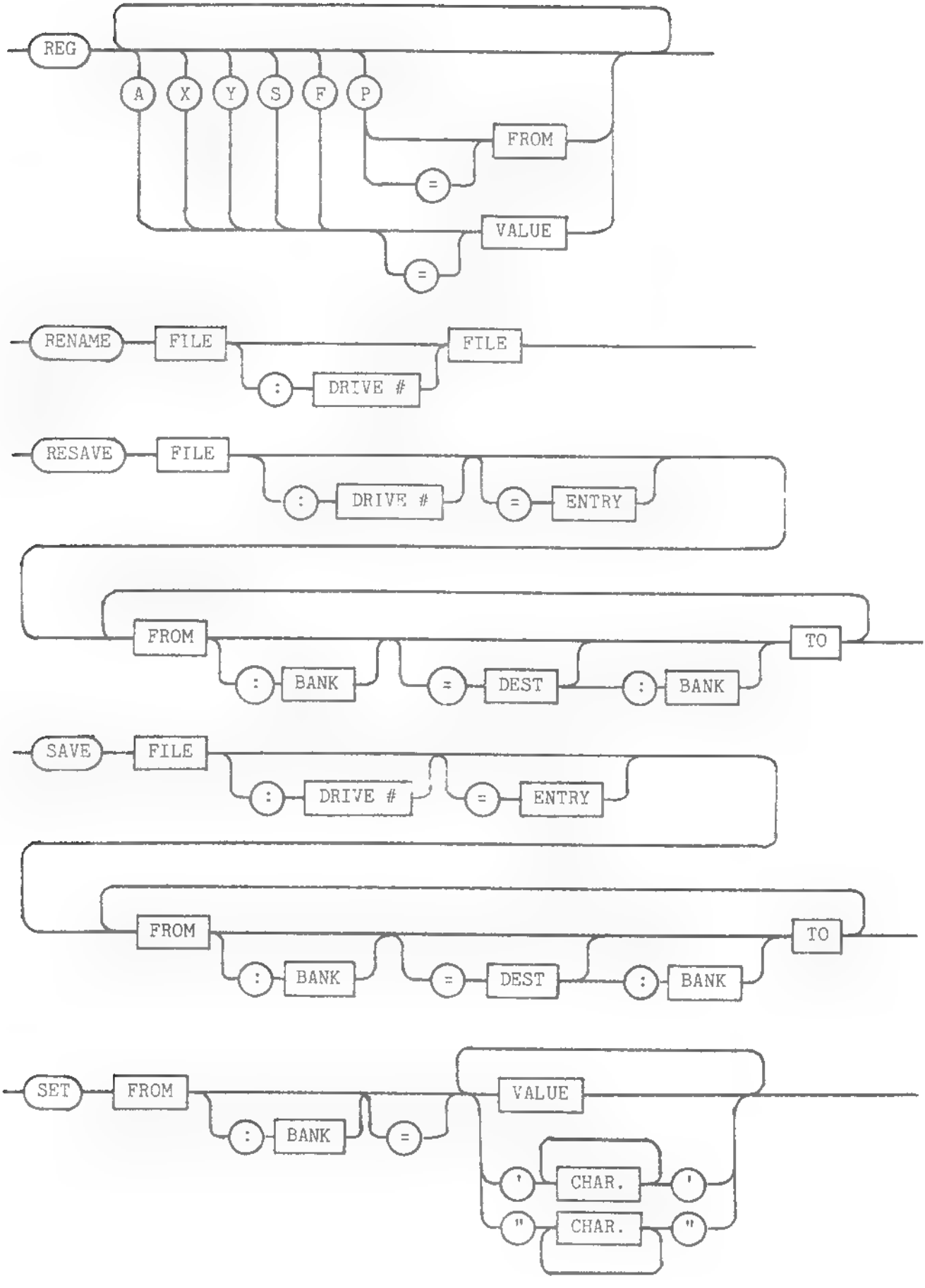

MSG
CHANNEL
TEXT
^cr
TEXT
^cr
cr

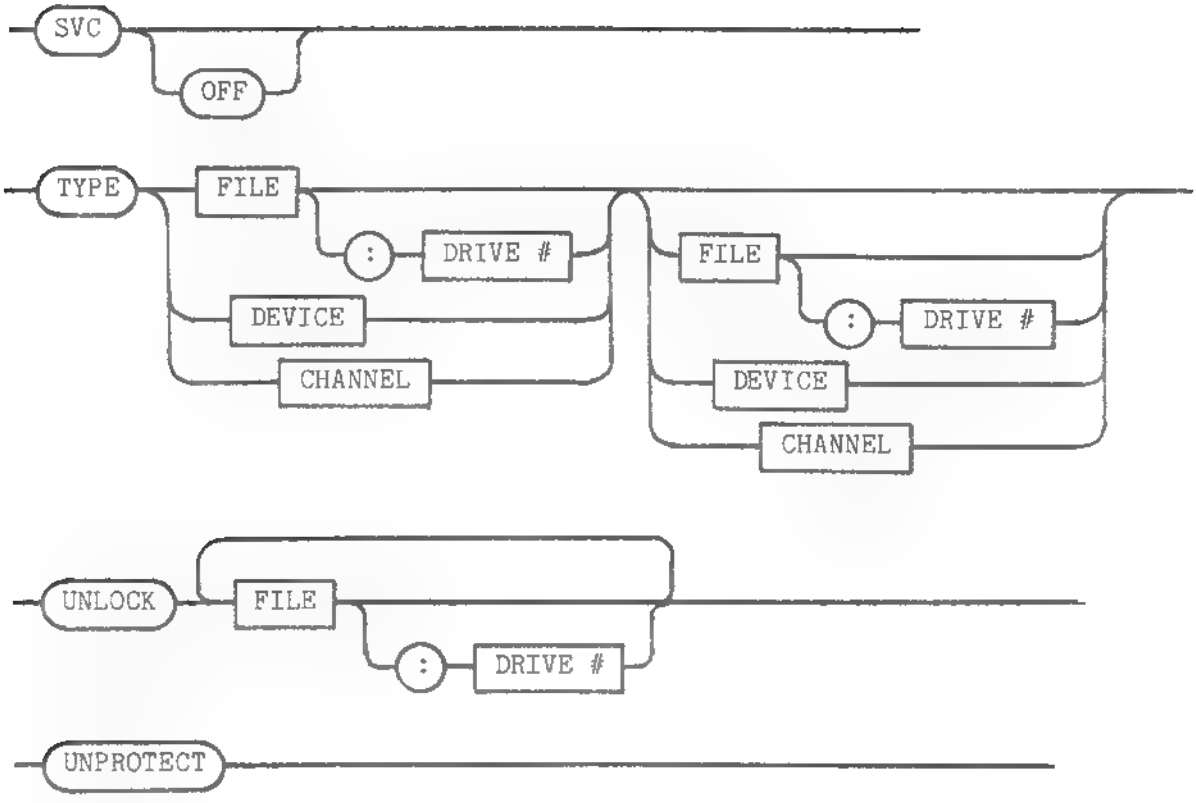

NEXT
FROM
:
BANK


ONKEY
F.KEY#
'
CHAR.
'
"
CHAR.
"
'
CHAR.
'
"
CHAR.
"


OPEN
DRIVE #


PROTECT


REG
A
X
Y
S
F
P
=
FROM
=
VALUE
RENAME
FILE
:
DRIVE #
FILE
RESAVE
FILE
:
DRIVE #
=
ENTRY
FROM
:
BANK
=
DEST
:
BANK
TO
SAVE
FILE
:
DRIVE #
=
ENTRY
FROM
:
BANK
=
DEST
:
BANK
TO
SET
FROM
:
BANK
=
VALUE
'
CHAR.
'
"
CHAR.
"



Descriptions of Identifiers Used:

CHANNEL	I-O Channel number, 0 to 9.
FILE	Filename, 2 to 12 characters plus optional 1-character extension.
DRIVE #	Disk drive number, 0 to 3.
DEVICE	I-O device name, 1 character.
FROM	Starting address.
BANK	Memory bank number, 0 to 3.
TO	End address.
DEST.	Destination address.
CHAR	An ASCII character.
VALUE	A numeric expression evaluating between 0 and $FF.
F.KEY #	Function key number, 1 to 8.
ENTRY	Entry point address.
cr	The ASCII carriage return character, $0D.

APPENDIX H

MTU-130 CHARACTER CODE CHART

The MTU-130 Computer and CODOS uses the standard ASCII character code for all internal operations. Special keys on the keyboard are given non-ASCII codes which can be recognized by the fact that bit 7 is a 1 whereas all ASCII codes have bit 7 set to a zero.

CHAR CODE	GRAPHIC OR NAME	HOW GENERATED
00	NUL	CTRL/ SPACE
01	SOH	CTRL/ A
02	STX	CTRL/ B
03	ETX	CTRL/ C
04	EOT	CTRL/ D
05	ENG	CTRL/ E
06	ACK	CTRL/ F
07	BEL	CTRL/ G
08	BS	BACKSPACE
09	HT	TAB
0A	LF	LINE FEED
0B	VT	CTRL/ K
0C	FF	CTRL/ L
0D	CR	RETURN
0E	SO	CTRL/ N
0F	SI	CTRL/ O
10	DLE	CTRL/ P
11	DC1	CTRL/ Q
12	DC2	CTRL/ R
13	DC3	CTRL/ S
14	DC4	CTRL/ T
15	NAK	CTRL/ U
16	SYN	CTRL/ V
17	ETB	CTRL/ W
18	CAN	CTRL/ X
19	EM	CTRL/ Y
1A	SUB	CTRL/ Z
1B	ESC	ESC
1C	FS	CTRL/SHIFT/ ,<
1D	GS	CTRL/ =+
1E	RS	CTRL/SHIFT/ .>
1F	VS	CTRL/SHIFT/ /?

CHAR CODE	GRAPHIC OR NAME	HOW GENERATED
20	SPACE	Space bar
21	!	SHIFT/ 1!
22	"	SHIFT/ '"
23	#	SHIFT/ 3#
24	$	SHIFT/ 4$
25	%	SHIFT/ 5%
26	&	SHIFT/ 7&
27	'	' "
28	(	SHIFT/ 9(
29	)	SHIFT/ 0)
2A	*	SHIFT/ 8*
2B	+	SHIFT/ =+
2C	,	,<
2D	-	-_
2E	.	.>
2F	/	/?
30	0	0)
31	1	1 !
32	2	2 @
33	3	3 #
34	4	4 $
35	5	5 %
36	6	6 carret
37	7	7 &
38	8	8 *
39	9	9 (
3A	:	SHIFT/ ;:
3B	;	; :
3C	¼	SHIFT/ ,<
3D	=	= +
3E	½	SHIFT/ .>
3F	?	SHIFT/ /?

CHAR CODE	GRAPHIC OR NAME	HOW GENERATED
40	@	SHIFT/ 2@
41	A	SHIFT/ A
42	B	SHIFT/ B
43	C	SHIFT/ C
44	D	SHIFT/ D
45	E	SHIFT/ E
46	F	SHIFT/ F
47	G	SHIFT/ G
48	H	SHIFT/ H
49	I	SHIFT/ I
4A	J	SHIFT/ J
4B	K	SHIFT/ K
4C	L	SHIFT/ L
4D	M	SHIFT/ M
4E	N	SHIFT/ N
4F	O	SHIFT/ O
50	P	SHIFT/ P
51	Q	SHIFT/ Q
52	R	SHIFT/ R
53	S	SHIFT/ S
54	T	SHIFT/ T
55	U	SHIFT/ U
56	V	SHIFT/ V
57	W	SHIFT/ W
58	X	SHIFT/ X
59	Y	SHIFT/ Y
5A	Z	SHIFT/ Z
5B	[	[]
5C	\	\ |
5D	]	SHIFT/ [ ]
5E	^	SHIFT/ 6^
5F	_	SHIFT/ -_

CHAR CODE	GRAPHIC OR NAME	HOW GENERATED
60	`	` @
61	a	A
62	b	B
63	c	C
64	d	D
65	e	E
66	f	F
67	g	G
68	h	H
69	i	I
6A	j	J
6B	k	K
6C	l	L
6D	m	M
6E	n	N
6F	o	O
70	p	P
71	q	Q
72	r	R
73	s	S
74	t	T
75	u	U
76	v	V
77	w	W
78	x	X
79	y	Y
7A	z	Z
7B	{	{}
7C	|	SHIFT/ \ |
7D	}	SHIFT/ {}
7E	~	SHIFT/ ` @
7F	DEL	RUBOUT

CHAR CODE	GRAPHIC OR NAME	HOW GENERATED
80		f1
81		f2
82		f3
83		f4
84		f5
85		f6
86		f7
87		f8
88		PF1
89		PF2
8A		×
8B		÷
8C		-
8D		+
8E		ENTER
8F		
90		SHIFT/ f1
91		SHIFT/ f2
92		SHIFT/ f3
93		SHIFT/ f4
94		SHIFT/ f5
95		SHIFT/ f6
96		SHIFT/ f7
97		SHIFT/ f8
98		SHIFT/ PF1
99		SHIFT/ PF2
9A		SHIFT/ ×
9B		SHIFT/ ÷
9C		SHIFT/ -
9D		SHIFT/ +
9E		SHIFT/ ENTER
9F		

CHAR CODE	GRAPHIC OR NAME	HOW GENERATED
A0		cursor up
A1		cursor left
A2		cursor right
A3		cursor down
A4		HOME
A5		DELETE
A6		INSERT
A7		
A8		
A9		
AA		
AB		
AC		
AD		
AE		
AF		
B0		SHIFT/ cursor up
B1		SHIFT/ cursor left
B2		SHIFT/ cursor right
B3		SHIFT/ cursor down
B4		SHIFT/ HOME
B5		SHIFT/ DELETE
B6		SHIFT/ INSERT
B7		
B8		
B9		
BA		
BB		
BC		
BD		
BE		
BF		

Code values from $C0 through $FF have no defined function and may not be generated by the MTU-130 keyboard.

APPENDIX I

USING EXTENDED MEMORY ADDRESSING

The MTU-130 computer and CODOS 2.0 both support extended memory addressing beyond the normal 64K limit of the 6502 microprocessor. This feature is implemented such that it is completely transparent to the programmer (even machine language programmer) if it is not used. It is important to realize that the extended addressing feature is not a simple bank switching scheme. Instead it is driven by the addressing mode used by instructions and actually allows a single program plus its data to exceed 64K without programmer hardship. Hardware level programming details of the extended memory addressing feature may be found in section 4.6 of the Monomeg Single Board Computer hardware manual.

The 256K addressing capacity of the MTU-130 is divided into four banks of 64K bytes each. Bank 0 is the normal or system bank which is automatically selected and assumed when a bank is not specified or the programmer wishes to ignore extended addressing. Bank 0 is also special because it contains the CODOS operating system, display and keyboard I/O drivers, all of the system I/O addresses, the system parameter area, and most importantly, the stack and the base (zero) page. All of the memory maps and system addresses given in this manual refer to locations in bank 0 unless otherwise noted. Even with this heavy usage, the amount of memory left to the user in bank 0 is as large or larger than on other competitive 6502 based systems.

The simplest use of extended addressing by an assembly language program is to specify a data bank that is different from 0. In this usage, the user program still resides in bank 0 but now any large data arrays are stored in another bank thus allowing the actual program to become much larger. The data bank is set by the program by altering the least significant two bits of location BFE0. The settings are 11 for bank 0, 10 for bank 1, 01 for 2, and 00 for 3. When changing the data bank select bits, be careful not to disturb any of the other bits. Now that the data bank is set differently from the program bank (which is still 0), any instruction using the (INDIRECT,X) addressing mode or the (INDIRECT),Y addressing mode will refer to the selected data bank for its data (the indirect address pointer is still in page 0 of bank 0 as always). The execution of all other "normally fetched" 6502 instructions is unaffected by the data bank selection. After a little thought and study of the 6502 instruction set it should be obvious that these two addressing modes must be used for addressing large (greater than 256) data arrays and are seldom if ever used for addressing small lists or individual datums.

The CODOS SVC facility may be freely used with the data bank set to anything (except SVC #15 and #16) since the SVC processor will save and restore the data bank setting. SVC #15 (read record) and #16 (write record) will transfer to/from memory in the data bank that was selected when the SVC was executed. This provides an ideal way of saving or retrieving those large data arrays on disk.

It is also possible to run an assembly language program in a bank other than bank 0. When CODOS starts a user program, it will automatically select the correct program bank and will also set the data bank equal to the program bank. The program bank may also be set by the user program by altering bits 2 and 3 of location BFE0 to 11 for bank 0, 10 for 1, 01 for 2, and 00 for bank 3. Program bank selection generally affects all memory references made during program execution except those described previously in connection with the data bank. There are other exceptions described in the next paragraph.

Program bank selection does not affect references to page 0 or the stack however. Thus page 0 and the stack (page 1) always reside in bank 0 regardless of what the program bank is set to. This is useful if one wishes to jump between banks since the code that modifies the program bank and then jumps into the new bank will not be affected by any intermediate setting of the program bank select bits. Actually, jumping between bank 0 and the user's program bank may be fairly frequent since CODOS imposes the following restrictions on program bank usage:

1. SVCs may only be issued by a program running in bank 0.

2. The standard display driver and keyboard subroutines reside in bank 0.

3. The I/O registers are in bank 0 (although you could reach them indirectly by temporarily setting the data bank to 0).

Because of these restrictions, it is recommended that bank 0 be used for program storage unless the program becomes so large that it cannot fit. If it does overflow, the main program should stay in bank 0 with large subroutines moved to another bank.

When an interrupt is recognized (either IRQ, NMI, or a BRK instruction), an "interrupt mode flip-flop" is set which temporarily forces the program bank to 0 but leaves the data bank selection in effect. Therefore, interrupt service routines must also reside in bank 0. The effect of the program bank is restored when an RTI instruction is executed. If the interrupt service routine uses instructions affected by the data bank selection, it will be necessary for the program to save and restore the data bank selection.

APPENDIX J

EFFECT OF CONSOLE INTERRUPT AND RESET

The MTU-130 console keyboard has three keys in the extreme upper right corner that control the system Reset function and the Non-maskable interrupt. The following describes what CODOS does and does not do in response to pressing these keys.

Interrupt Key

The Interrupt key (labelled INT) unconditionally triggers a non-maskable interrupt sequence in the 6502 microprocessor when it is pressed. This interrupt will unconditionally interrupt any running program, even CODOS itself, and jump to $029A which in turn normally contains a jump to the CODOS NMI processor unless a user-defined NMI service routine is used instead. CODOS's non-maskable interrupt service entry point does the following:

1. Assigns channel 2 to the console display and channel 1 to the keyboard.
2. Prints the letters "NMI" and then the registers which represent the machine state at the time of interrupt (see REG command in Chapter 3).
3. Waits for the operator to enter a CODOS command.

Program execution can be safely resumed at the point of interruption by entering the NEXT command if the following is true:

1. The program expects channels 1 and 2 to be assigned to the console.
2. There is no chance that an SVC was being executed at the time of interrupt.
3. That the point of interruption did not occur in the system or the I-O drivers.

Note that execution of certain illegal opcodes will freeze the 6502 CPU and make it unresponsive to the INT key. In this case the RESET key is the only way to restore control.

Reset Key

The Reset key (labelled RESET) unconditionally triggers a system reset when it is released after having been pressed for at least 1/4 second. Pressing Reset for less than 1/4 second (perhaps by accident), will have no effect. The following hardware related events happen in response to Reset:

1. All system I/O ports (including the user parallel port) revert to inputs.
2. The Reset signal on the User parallel I/O port connector goes low.
3. The display turns on and reverts to 480x256 black and white.
4. The Program and Data banks revert to bank 0.
5. The disk controller is reset.
6. The serial interface is reset.
7. The CPU enters the Bootstrap PROM on the Disk Controller board.

The bootstrap program in the PROM next looks at the state of the MOD key on the keyboard. If the MOD key is pressed, it loads CODOS from the disk in drive 0 just as if power had been turned on. Most of memory (except $C000-$FFEF and $00C3-$00DF) is left alone however (see Appendix C).

If the MOD key is not pressed, the bootstrap program jumps to location $0300 which jumps to the "warm reset" entry point of CODOS which performs the following:

1. Clear decimal mode.

2. Discard stack (S=$FF).

3. Force all drives to "closed" status (does not update open files on disk) which also forces any channels assigned to files to the "free" condition.

4. Forces console on channels 1 and 2.

5. Restores default drive to 0, banks to 0, default extension to "C", normal error processing, and restores normal NMI, IRQ, and cntl/C vector settings.

6. Re-initializes text display to normal (24 lines, 80 characters, normal video).

7. Opens drive 0.

8. Displays "RESET" on the console and then waits for a command.

NOTES:
1. Does not re-initialize any I/O drivers except the console. Any drivers that use the parallel ports and do not set the direction registers and mode on every character must be re-initialized. Any driver that uses the serial port will have to be re-initialized.

2. Does not relaod CODOS, I/O drivers, or other programs.

3. If Reset is pressed during disk writes, files on disk may be left in an undefined state.

APPENDIX K

NON-OVERLAY CODOS COMMANDS

Most CODOS built in commands are loaded into memory from disk only as needed, typically in less than one half second. These commands are called system overlays, and reduce the size of the operating system by more than 4K bytes. Since the overlays are not always in memory, it is normally necessary to maintain an open disk in drive 0 with the CODOS overlays present. On rare occasions you may wish to operate without an open disk in drive 0 for some reason. The following CODOS built in commands are not overlays and are always available whenever the CODOS system has been booted-up:

BOOT
BP
CLOSE
DELETE
DRIVE
FREE
GET
GO
NEXT
OPEN
RESAVE
SAVE