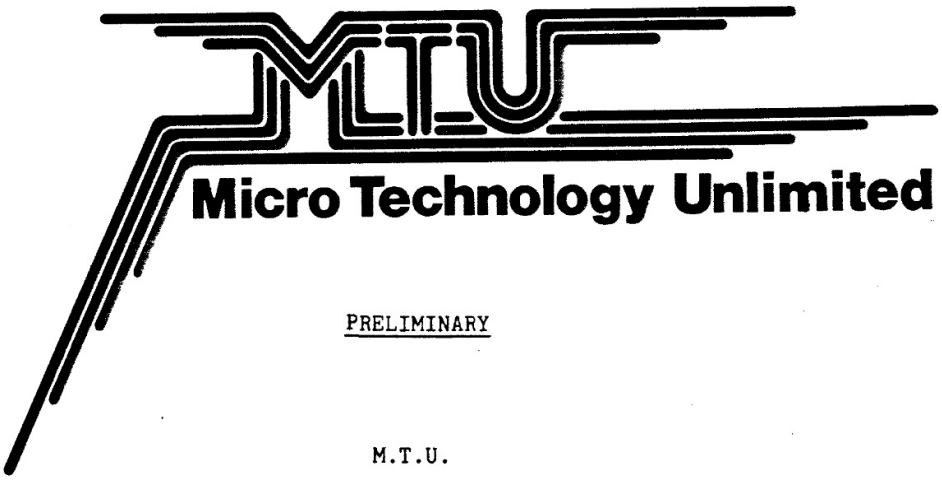


PRELIMINARY

M.T.U.

CODOS OPERATING SYSTEM

USER'S MANUAL

Release 1.0

August, 1980

Micro Technology Unlimited
2806 Hillsborough Street
P.O. Box 12106
Raleigh, NC 27605

(919) 833-1458

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 K-1013 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. During the many months of development of the CODOS system, we did not experience a single (hard) read error during hundreds of hours of use, following the rules below:

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.

6. Keep the diskette away from strong magnetic fields.

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 security. However, satisfactory results can usually be obtained with single-density diskettes of good quality, due to the exceptionally high-quality data separator used in the MTU K-1013 double-density controller, and the automatic error recovery software built in to CODOS.

INTRODUCTION TO CODOS

CODOS (Channel-Oriented Disk Operating System), formerly called APEX-65, is an extremely powerful and versatile disk operating system for 6502-based microcomputers using the M.T.U. K-1013 Floppy Disk Controller. 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 M.T.U. Disk Controller and takes full advantages of its many engineering achievements. An on-board Bootstrap PROM brings up CODOS in the onboard 8K bytes of "system" RAM, and enables the write-protect to help prevent errant user programs from inadvertently "crashing" the system.

CODOS provides true device-independent I-O over logical "channels", as found in many large 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 applications programs, which do not need to provide buffers, "File Control Blocks", or other artifaces 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.

The CODOS Monitor provides 28 built-in commands with free-format input, which can be easily extended to include many User-defined commands. English-language error messages help pinpoint user errors. Commands may also be read from disk files or other devices, and a special disk file called STARTUP.J is read by the system automatically during booting up, facilitating system customizing and turnkey applications.

Interfacing for user-written Assembly-language programs is provided in the form of "Supervisor Call" Pseudo-instructions (SVCs), which simplify program development and portability between different CODOS-based systems.

Several Utility programs are provided for copying files, making duplicate copies of the operating system, testing disks, etc.

HARDWARE REQUIREMENTS:

This is the initial release of CODOS. Additional configurations and more different disk types will be supported in forthcoming versions. The initial release supports:

(1) AIM-65 computers with 1 or 2 8-inch double density disk drives and 16K of additional RAM memory addressed at $0000.

(2) KIM-1 computer with 1 or 2 8-inch double density disks, a terminal device, and 8K or more of additional memory at $2000.

Either single or double sided disks may be used; however, double sided-disks will only be supported in the single-sided mode of operation in this release.

A software driver is included with CODOS for either system to allow an M.T.U. K-1008 Visible Memory High-resolution graphics Board plus a TV Monitor and ASCII keyboard to function as a terminal device.

This is a preliminary manual. We suggest you read the entire manual before attempting to use the CODOS system, then follow the First-time power-up instructions for your machine.

CODOS FIRST-TIME STARTUP PROCEDURE: AIM-65

THIS SECTION PERTAINS ONLY TO THE AIM-65 COMPUTER. IF YOU DO NOT HAVE AN AIM-65, SKIP TO THE SECTION PERTAINING TO YOUR SYSTEM.

Follow the following steps carefully:

1. Before attempting to use CODOS, we strongly suggest you carefully read the System Concepts section, chapter 3, and skim chapters 4, 5, and 6, which describe the Commands and Utility programs available on your system. These Commands and Utilities will be used during the first-time powerup procedure.

2. The purpose of the first-time power-up procedure is to copy the distribution disk to another disk and "customize" the new operating system on this disk to yourparticular machine needs. Once this is accomplished, you can use the new disk to bring up your "custom" system automatically using the bootstrap ROM. The objective of the first and most essential level of "System Generation" is to establish the number of disks in your system and the type of I-O desired for your console device. Later, more customization can be performed to "fine tune" the system and improve performance, as well as add support for other devices.

3. Insure that you have an operational disk system as described in the K-1013 hardware manual, with the on-board memory jumpered to the following addresses:

"SYSTEM RAM" at $8000.
"USER RAM" at $4000.

Additional RAM is recommended at $0000 through $3FFF, but not essential. An MTU K-1016 16 K memory board can fill this need easily. If the MTU "Visible Memory" is to be used, it should be addressed at $6000. A complete driver program for using the visible memory as a conole output with the AIM-65 is provided on the distribution disk.

4. Power up the AIM. Using the AIM Monitor, verify the existence of memory at $4000 and $8000. If you get a "MEMORY FAIL" when depositing into the $8000 block of memory, it means that the Hardware write-protect is set. This can be disabled by writing a 00 into $9FE8 (which will also give a "MEMORY FAIL" message, that can be ignored). It is not necessary to disable the write-protect to bring up the system; you are only doing it to make the initial check for correct board addressing.

5. Insert the CODOS distribution disk in drive 0 (label towards the moveable part of the door). Close the door.

6. Turn on the AIM printer.

7. Using the AIM Monitor, begin execution at $9F00. This is the on-board bootstrap ROM.

8. The disk should show immediate activity and after about two seconds the "Sign-on" message should be printed:

```
M.T.U. CODOS V1.0
ENTER DATE (EXAMPLE:
04-JUL-76)?=
```

9. Enter today's 9 character date in the indicated format and depress RETURN. Every CODOS keyboard entry needs to be terminated by a RETURN.

10. The display should show the CODOS prompt character, " ". This character indicates that CODOS is ready to accept a Command. You are now ready to make a backup copy of the Distribution disk. If you have a two-drive system, put a new disk into drive 1. If you have a one-drive system, enter:

```
UNPROTECT
SET 880F=1
```

If you do not have a 16K memory at $0000, redefine the Large Buffer address as described in chapter 9.

11. Type:

```
FORMAT
```

and then carefully follow the directions for the FORMAT Utility in chapter 6. When the new disk is formatted and the System copied, use the FILES command described in chapter 4 to ascertain which files were not copied by FORMAT.

12. Use the COPYF (if you have 2 or more drives) or COPYF1DRIVE (if you have only one drive) Utility programs described in chapter 6 to copy all remaining files from the distribution disk.

13. Remove the Distribution diskette from drive 0 and put it away in its envelope in a safe place. You should not need it again. Use only copies of the system for your normal activities.

14. Power off the AIM completely. Power back up the AIM, and insert the newly-formatted disk in drive 0.

15. Execute the bootstrap loader as before using the AIM Monitor to begin execution at $9F00. The system should come up exactly as with the Distribution disk.

16. If you have a Visible Memory board at $6000, you will probably want to type:

```
VMT
```

This loads and executes a special driver program which changes the Console device for CODOS from the AIM keyboard and Display/Printer to the AIM keyboard and Visible Memory CRT. The printer will still function normally and can be turned off using CNTRL-PRINT in the ususal fashion. The display will display only garbage, however. To restore the normal AIM display, type:

```
VMTOFF
```

which disables the Visible Memory Terminal driver program.

17. You may now enter any legal Commands, make additional backup copies, or "customize" your system using the guidelines in chapter 9. If you have a one-drive system, be sure to follow the directions in chapter 9 to make the permanent disk copy of the system "know" you have only 1 drive.

CODOS FIRST-TIME-POWER-UP PROCEDURE: KIM-1

THIS SECTION PERTAINS ONLY TO THE COMMODORE KIM-1 COMPUTER. IF YOU DO NOT HAVE A KIM-1, SKIP TO THE SECTION PERTAINING TO YOUR COMPUTER.

Follow the following steps carefully:

1. Before attempting to use CODOS, we strongly suggest you carefully read the System Concepts section, chapter 3, and skim chapters 4, 5, and 6, which describe the Commands and Utility programs available on your system. These Commands and Utilities will be used during the first-time powerup procedure.

2. The purpose of the first-time power-up procedure is to copy the Distribution disk to another disk and "customize" the new operating system on this disk to you particular machine needs. Once this is accomplished, you can use the new disk to bring up your "custom" system automatically using the bootstrap ROM. The objective of the first and most essential level of "System Generation" is to establish the number of disks in your system and the type of I-O desired for your Console device. Later, more customization can be performed to "fine tune" the system and improve performance, as well as add support for other devices.

3. Insure that you have an operational disk system as described in the K-1013 hardware manual, with the on-board memory jumpered to the following addresses:

"SYSTEM RAM" at $C000.
"USER RAM" at $6000.

At least 8K additional RAM, and preferably 16K, should be available starting at $2000. An MTU K-1016 16K RAM board can supply this need conveniently.

Unfortunately, there is very little standardization of memory layouts or I-O devices on KIM based systems. MTU will make available several other configurations of CODOS in the near future to support the most common configurations.

4. CODOS requires a Console device for input-output (a terminal). The KIM keypad cannot be used for this purpose. Any type of terminal can be used, provided it is capable of supplying ASCII characters and displaying them. The user must provide the software driver routines for the terminal device. If you do not have a terminal, a very low-cost versatile terminal can be made by using the MTU K-1008 "Visible Memory" with a Monitor to display output, and an ASCII keyboard with parallel input. A complete driver program called the "Visible Memory Terminal" (VMT) supporting both text and high-resolution graphics output is provdied on the CODOS Distribution disk. If you wish to use it, the Visible Memory board should be addressed at $A000.

5. If you do not wish to use the VMT Console, you must load the necessary software drivers into memory using the KIM Monitor. These drivers can reside in any part of memory not needed by CODOS (see memory map), and can be in PROM or RAM. You will need a subroutine to input a character from the keyboard, and a subroutine to output a character to your display device. The specific requirements for the driver routines are given in Appendix E, together with a sample driver.

6. Once you have your input-output Console drivers loaded, you will need to communicate their location to CODOS, along with some other information. Using the KIM Monitor, carefully deposit the necessary information in memory as shown in Table 2-1.

TABLE 2-1: DEFINING KIM-1 SYSTEM TO CODOS

At Address...	Deposit this information...
$0100	Number of disk drives in system, 0 or 1.
0101	Flag. Set to non-0 if you want to use VMT Visible Memory Terminal (requires Visible Memory at $A000)
0102-0103	Address of your Keyboard Character-in driver subroutine.
0104-0105	Address of your Display Character-Out driver subroutine, if you do not want to use the VMT driver provided.
0106-0107	Starting address of the Large Buffer Area used by file copying Utility programs, etc. See description below.
0108-0109	End address of the Large Buffer Area.

NOTES:

1. All 2-byte quantities should be depositied in the conventional low-byte, high-byte order.

2. The Large Buffer Area should be defined as an area of available RAM not in the reserved memory area for CODOS nor overlapping your I-O drivers, which CODOS can use for temporary storage during copying operations by the utility programs. It must not overlap the area into which the Utiltiy programs load ($2000 through 28FF) The Buffer should be as large as possible to increase efficiency, especially on a one-drive system, since the size of the buffer determines the number of disk-swaps needed to copy a file. This buffer is only used for scratch storage by the FORMAT and file-copying Utility programs.

3. If you choose to use the VMT, it will be loaded into memory automatically during booting-up. You will still need to provide the Keyboard Driver.

EXAMPLE:

A KIM system has 1 drive, 16K of RAM at $2000, and uses a CRT terminal for the Console. The CRT Input-character routine is located at $1780 and the output entry point is $1783. The VMT will not be used. Deposit memory as follows:

$0100 01
0101 00
0102 80
0103 17
0104 83
0105 17
0106 00
0107 29
0108 FF
0109 3F

7. When you you are certain your I-O drivers are operational and the necessary information is properly installed in zero-page, you may insert the distribution disk into drive 0 (label side toward moveable part of door) and close the door.

8. Using the KIM Monitor, begin execution at address $FF00, which is the Bootstrap PROM. The disk should show immediate signs of activity. After about two seconds, your Console should display:

```
M.T.U. CODOS V1.0
ENTER DATE (EXAMPLE:04-JUL-76)?=
```

If the disk shows activity but the message does not appear, check your device drivers for the Console. If the disk does not show activity, check your board addressing for the disk controller.

9. Enter today's 9 character date in the indicated format and depress RETURN. Every CODOS keyboard entry needs to be terminated by a RETURN.

10. The display should show the CODOS prompt character, " ". This character indicates that CODOS is ready to accept a Command. If the characters do not appear on the console as you type them, check your keyboard driver. If you get double characters, don't worry about it. You can fix this later as described in chapter 9. Assuming all is well, you are now ready to make a backup copy of the Distribution disk. If you have a two-drive system, put a new disk into drive 1.

11. Type:

```
FORMAT
```

and then carefully follow the directions for the FORMAT Utility in chapter 6. If the system "crashes" or behaves strangely after starting the copying of the operating system, it is because your Large Buffer Addresses were wrong. When the new disk is formatted and the System copied, use the FILES command described in chapter 4 to ascertain which files were <u>not</u> copied by FORMAT.

12. Use the COPYF (if you have 2 drives) or COPYF1DRIVE (if you have only one drive) Utility program described in chapter 6 to copy all remaining files from the distribution disk.

13. Use the CLOSE command to close drives 0 and 1 if you have a two-drive system, or drive 0 if you have a one-drive system. Remove the Distribution disk and put it away in its envelope in a safe place. Work only with the copied disk.

14. Put the new disk into drive 0 and OPEN it. Use the FILES command to verify the existence of the system files.

15. Right now the address of your device drivers and the other information is only temporarily patched into the CODOS system memory image. You will now want to make it a permanent part of the disk copy of the memory image, so that you can boot-up without any preamble. If your device-drivers are in RAM, you will also want to have these loaded into memory for you from disk automatically when you boot-up. To do this, follow the instructions in Chapter 9.

16. Once you have performed the necessary "System Generation", it will only be necessary to insert the "customized" disk in drive 0 and begin execution at $FF00 to bring up the system.

CODOS SYSTEM CONCEPTS

The CODOS Operating System is a powerful computer program for managing the resources of a 6502-based microcomputer. In particular, it provides a convenient method for storing and retrieving other programs and data on floppy disk storage. The user will normally interact with CODOS principally through two built-in facilities:

1. The SYSTEM MONITOR;
2. The SVC PROCESSOR.

The SYSTEM MONITOR provides a simple method for the user to interact directly with CODOS by typing commands from the 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 user 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 utilize the functions of the SYSTEM MONITOR to some degree. In addition, however, programmers will also wish to utilize the facility which permits programs to interact with the operating system. For example, programmers will wish to be able to display messages on the display 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 programs called the SUPERVISOR CALL (SVC). Although not found on microcomputers, SVC's are found extensively on the finest mainframe computers. Instead of a JSR instruction to a system routine, the SVC consists of a BRK ($00) instruction followed by a data byte which identifies the function desired. There are several advantages to this method, which are described later; the most important advantage of the SVC is that SVCs are address-independent. This means that a program using SVCs will run without modification regardless of the location of the operating system. Thus, for example, a program written on an AIM with CODOS at $8000 can be run without modification on a KIM with CODOS at $E000. SVCs are discussed in detail in a later section.

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 re-run such that the output is directed instead to a printer, without any modification to the program. Input or output can also be re-directed to a file on disk. This provides an exceptionally powerful capability. The devices to be used can be selected by a simple MONITOR command, or by an 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 printer is normally an output-only device, but the system console (terminal) can both send and receive data.

Certain channels have pre-defined meanings, and other channels have been given suggested standard meanings in the interest of uniformity among applications. These channel definitions are given in Table 3-1.

The way in which channels are used will become clearer in following sections which introduces the CODOS Monitor Commands. The section on interfacing to user programs, chapter 7, describes the use of channels from a programmer's point of view.

TABLE 3-1: STANDARD CHANNELS

Channel 0: Reserved for internal CODOS operation.
Channel 1: Input commands to SYSTEM MONITOR.
Channel 2: Output from SYSTEM 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:

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

2. The notation "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.

TABLE 3-2: 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, other than console (e.g., a Teletype).
V	Visible memory high-resolution graphcis driver.

NOTES:

1. Other devices may be named as desired during System Generation, using a single letter for each. See Chapter 9.

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 at System Generation (SYSGEN), 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 terminal controlling the system (normally a CRT plus keyboard), 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 voluminous output, but you do not want any output, you can merely assign the null device and run the program.

Additional devices 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 3-2 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.

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 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 facilitate searches for files, as will be discussed later. 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. Although CODOS does not enforce any particular convention, Table 3-3 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.

Remember that file names must have at least two characters; this is how CODOS tells the difference between a file name and a device name (which can have only one letter).

NOTE: The Underline character is not available on the AIM-65 keyboard. The "$" is used instead on AIM systems.

TABLE 3-3: FILE EXTENSIONS

Extension	Meaning
A	Assembly language source program.
B	BASIC Program source.
C	Command (User-defined command programs and System Utility programs).
D	Data.
G	Graphic data.
H	Hex file (i.e., paper tape-type format).
J	Job file (i.e., a text file of CODOS commands).
L	Listing.
T	Text.
X	Executable code other than a command (e.g., subroutine package).
Z	CODOS reserved system file.
5	AIM BASIC ROM source program (Pertains to AIM only).

Notes:
1. If the extension is not given, ".C" will be assumed.
2. Other extensions may be devised by the user as needed.

CODOS SYSTEM MONITOR

The CODOS Monitor is an interactive program which allows the user to enter commands to the system. The Monitor 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, and a file called STARTUP.J is 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, ">" 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 3-4.

There are two main types of commands in CODOS: User-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.

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 "=" 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. In order to have a uniform method of 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 tne ususal operators, "+", "-", "*", "/" and "\". "\" is the remainder operator. All expressions are evaluated left-to-right without any hierarchy. 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).
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).
```

TABLE 3-4: COMMAND EDITING CHARACTERS

Character	Meaning
DEL or CNTRL-H	Backspace 1 character.
CNTRL-X	Delete entire line (start line over).
RETURN	End-of-command.
;	Comment. Any characters after ";" are ignored.
!	Command abbreviation character. See text.
blank	Separator between arguments.
CNTRL-S	Temporarily suspend output display.
CNTRL-C	Command abort (during display)

TABLE 3-5: COMMANDS

Command	Purpose
ASSIGN	Display or alter channel assignments for I-O.
BEGINOF	Position channel to beginning-of-data.
COPY	Copy memory block.
CLOSE	Close-out operations on disk specified.
DATE	Set date.
DISK	Display attributes of disks.
DRIVE	Designate default drive .
DUMP	Display contents of memory.
ENDOF	Position channel to end-of-file.
DELETE	Delete file from disk directory.
FILES	List names of files on disk.
FILL	Fill block of memory with a constant.
FREE	Release channel if assigned.
GET	Load program into memory from disk.
GETLOC	Display load addresses of loadable file.
GO	Begin execution of program in memory.
LOCK	Enable write-protect on disk file.
NEXT	Resume execution of program in memory.
OPEN	Open-up operations on a disk.
PROTECT	Enable hardware write-protect on system memory.
REG	Dispaly contents of registers.
RENAME	Change the name of a file.
SAVE	Save program on disk.
SET	Set memory to value(s).
SVC	Enable or disable Supervisor Call Processor.
TYPE	Display contents of file.
UNLOCK	Disable write-protect on file.
UNPROTECT	Disable hardware write-protect on system memory.
⟨name⟩	Execute User-defined command or system Utility.

COMMAND NAME: OPEN.

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

SYNTAX: OPEN ⟨drive⟩ ...

ARGUMENTS:

⟨drive⟩ = disk drive number, 0 to 3 to be opened. Defaults to drive 0.

EXAMPLES:

OPEN

opens the disk in drive 0 for operations.

OPEN 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 at the time. 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.

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 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 permissable.

COMMAND NAME: CLOSE.

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.

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.

If you forget to CLOSE a disk before removing it, you will get a system error message of "PREVIOUS DISK NOT CLOSED (OR RESET HIT)" when you attempt another disk operation. At this point you can do one of two things:

1. Put the old diskette back in and CLOSE it (be sure to get the right disk or you'll kill it!!!), or;

2. You can ignore the error and OPEN the new disk. This will usually have no ill effects on the disk which you failed to close. At worst, files which had all of the following characteristics on the old disk may be truncated:

a. The file was assigned and not freed;
b. The file was written to by a user (not system) program;
c. The last operation was a write (not a read).

Thus it is unlikely that there will be any adverse affect on the un-closed disk. HOWEVER, IF YOU FAIL TO DO EITHER (1) or (2) ABOVE IN RESPONSE TO THE ERROR MESSAGE, YOU CAN EXPECT UNPREDICTABLE AND INVARIABLY UNPLEASANT RESULTS ON THE NEW DISK!

COMMAND NAME: DISK.

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

SYNTAX: DISK

ARGUMENTS: none.

EXAMPLE:

```
DISK
```

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

```
11 FILES:0 458K FREE
79 FILES:1 102K FREE
```

which indicates that drives 0 and 1 are open, with 11 files on drive 0 and 79 files on drive 1. There is 458 K bytes (1 K = 1024 bytes; therefore about 468,992 bytes remain available on drive 0).

NOTES:

1. Disk space is allocated and displayed in blocks of 2K bytes.

2. The file count is in decimal.

COMMAND NAME: FILES.

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, one name per line.

FILES 1

displays the names of all the files on drive 1.

NOTES:

1. The DIR utility program can be used when it is desirable to display information about selected files. See chapter 6.

2. As with any command, CNTRL-S can be used to temporarily suspend the display. CNTRL-C can be used to abort the command.

COMMAND NAME: DRIVE.

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

2. The DRIVE command only effects the drive for file name references. It does not effect 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: ASSIGN.

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.

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 (terminal). 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" of "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 mispelled 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 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 conosle 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. Chpater 9 discusses STARTUP.J Jobs in some detail.

COMMAND NAME: FREE.

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 4.

NOTES:

1. It is permissable to free an unassigned channel.

COMMAND NAME: SAVE.

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 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.

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 normally 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.

COMMAND NAME: GET.

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=100 =1B00
```

will load the file OLD_PROG.X from drive 1 into memory. The first block (which is whatever size was SAVEd on the file) will be loaded starting at address $0100, 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. 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.

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

3. 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.

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

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

6. 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.

7. 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.

8. 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.

9. 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.

COMMAND NAME: GETLOC.

PURPOSE: To display the Entry point, Starting and Final load addresses 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 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=500B 5000 588E

which indicates that VMT loads into addresses $5000 through 588E inclusive, and execution starts at $500B.

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
            0300 03DD
            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, $0300 through $03DD, 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.

COMMAND NAME: DELETE.

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 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!

****

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.

COMMAND NAME: RENAME.

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 exisitng file. Defaults to the 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: BEGINOF

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 permissable 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: 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: LOCK.

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 default disk, 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 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: UNLOCK.

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 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: TYPE.

PURPOSE: To display or print 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 Default drive, ususally 0.
<channel> = desired pre-assigned source channel number, 0 to 9.
<dest.device> = desired output device name. Defaults to Concole ("C").
<dest.file> = desired file to recieve 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 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 a device, for example the console, CNTRL-Z is used to enter and 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 the beginning of the file, so be careful!

COMMAND NAME: DATE.

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.

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: DUMP.

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

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

ARGUMENTS:

<from> = desired starting address.
<to> = desired ending address (see note 1 below). Default is <from> +7.
<channel> = desired channel on which to display the output. Defaults to automatically assigned available channel to Console.

EXAMPLES:

```
DUMP 1000
```

displays 8 bytes of memory starting at $1000.

```
DUMP 200 213
```

displays memory starting at $0200 and will include memory through $0213. The resulting display might look similar to:

```
0200 00 21 00 AA 00 AA 00 76  .!.....v
0208 34 87 41 42 43 AA 00 AA  4.ABC...
0210 10 FF 55 FF 55 FF 55 FF  ..U.U.U.
```

Of course, the actual values displayed will depend on the contents of memory. The eight 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  7
```

dumps 500 (decimal) bytes starting at $1000. The display will be output to the device or file currently assigned to channel 7.

NOTES:

1. A complete line is always displayed even if the from address is not an even multiple of 8 bytes. Sufficient complete lines will be displayed to insure that <from> is included in the display.

2. As with any command, CNTRL-S can be used to temporarily suspend the display. CNTRL-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 ASCII printable ASCII characters except blank, provided bit 7 is 0.

COMMAND NAME: 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 or expression to be deposited.
<character> = an ASCII character to be depositied.

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 200 80-.10 "   " 80-.20 '"'
```

sets $0200 to $76, $0201 through $0203 to $20 (ASCII blank), $0204 to $6C, and $0205 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 ~~APEX-65~~ COPOS. If reading the byte back from memory results in a bad compare to the value deposited, an error message is issued.

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: 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 200 2FF 0
```

fills every byte between $0200 and $02FF inclusive with $00.

```
FILL 2301 2301+.10=" "
```

fills $2301 through $230B with $20 (an ASCII blank).

```
FILL 0 B '"'
```

fills $0000 though $000B 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: COPY.

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).

COMMAND NAME: REG.

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:

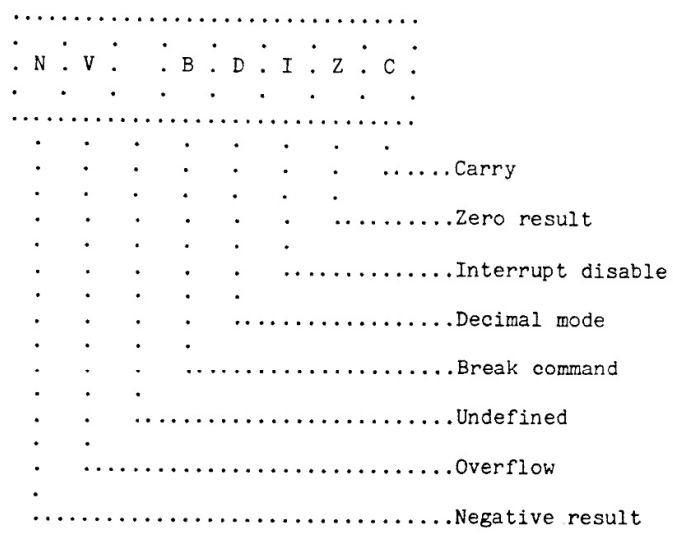
1. The REG command without arguments displays the user's registers in the format illustrated below:

```
.......Current Program Counter (P)
.
.       .......Contents of memory at P through P+2
.       .
.       .       ......Contents of Accumulator (A)
.       .       .
.       .       .
P=1B1F (201A17) A=2A X=05 Y=00 F=32 S=FD
                        .   .   .   .
                        .   .   .   .
Contents of X reg.......    .   .   .
                            .   .   .
    Contents of Y reg.......    .   .
                                .   .
         Contents of Flags(F).......    .
                                        .
             Current Stack pointer(S)....
```

All values are given in hexadecimal. The key letters given in the display are the same as the <reg. desig.> needed to set the register values. Naturally the actual values displayed will depend on the register contents at the time.

(continued...)

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.

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 200
```

begins execution of a machine language program at $0200.

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.

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.

COMMAND NAME: NEXT.

PURPOSE: To resume execution or 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 difference between GO and NEXT is that GO enters the program with a JSR after discarding any stack (i.e., sets S=FF), wheras 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 BRK has been encountered. A rudimentary form of breakpoints can be used by setting a $00 byte into a program and executing GO. When the BRK ($00) is encountered, control will return to the monitor. The registers can then be altered or displayed as desired. To resume execution, use the SET command to replace the BRK with the original opcode, use the REG command to set P to the address of the restored opcode, and type NEXT. The program will continue execution in the normal manner.

COMMAND NAME: PROTECT.

PURPOSE: To enable the memory-protect hardware on the System 8k block of memory on the disk controller board, 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 or the 8K block of system memory 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: UNPROTECT.

PURPOSE: To disable the hardware write-protect on the 8k of system RAM on the disk controller board, 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 overwrite 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. GET can load into System RAM after an UNPROTECT command, but not into page 0 or 1. The only way to load a program into page 0 or 1 is to use GET with a <dest.> argument specified elsewhere, and then COPY the memory image into the desired locations.

COMMAND NAME: SVC.

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 the Assembly Language Interface section.

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

EXTENSIONS FOR AIM-65 COMPUTERS

In addition to the standard features of CODOS, several extensions are included for AIM-65 systems. These extensions include I-O drivers for the AIM keyboard and display/printer, special keys on the keyboard, and interfacing for the AIM Monitor and ROMs to the disk. These functions are all included in the AIMEXT.Z file on the distribution disk, which must be loaded into memory during booting-up by the STARTUP.J file unless the features are not to be used (in which case alternate I-O console drivers must be defined).

Special keys:

ESC — Exits CODOS to the AIM monitor. This is the only proper way to exit CODOS to the AIM Monitor.

F3 — When in the AIM monitor, depressing F3 will exit the AIM monitor and re-enter CODOS (assuming CODOS has been booted up before).

CNTRL — Depressing CNTRL will temporarily suspend console output. Hit any key to resume.

NOTES:

1. Since BASIC uses almost all of Page 0, hitting F3 automatically copies the part of page 0 which conflicts with CODOS to another area. Depressing ESC automatically restores the saved page-0 image. In this way, CODOS will not interfere with BASIC operation.

2. If you use RESET to exit from CODOS, the conflicting BASIC page 0 will not be restored.

Interfacing AIM Monitor and ROMs to disk:

After booting up CODOS, you may return to the AIM Monitor using the ESC key. You may then load and dump ("L" and "D" commands), LOAD and SAVE BASIC programs, Edit and Assemble using CODOS disk files.

For example, to save a BASIC program, enter BASIC in the ususal fashion using the "5" command from the AIM Monitor. BE SURE TO SPECIFY THE MEMORY SIZE AS 22527 (OR 20479IF VMT IS IN USE). If you instead reply with RETURN, BASIC will wipe out the AIM I-O drivers and crash the system. When you are ready to save your program, type SAVE as usual. AIM BASIC will prompt:

```
OUT=
```

Reply with "U" (for user-device). This will invoke the CODOS interface routine which will prompt:

```
FILE=
```

Type in the desired file name, for example:

```
STARTREK.5:1
```

CODOS will save the file and return control to BASIC. In the event of an error you can use ESC and "6" to get back into BASIC to do it over.

In the same manner, you may load programs for BASIC. The speed at which programs load is limited by the display speed, but is still much faster than tape.

The same procedure can be used for loading and dumping memory. Keep in mind that the AIM saves memory images in a paper-tape format. These can be loaded using the AIM "L" command but not the CODOS GET command. You will probably want to use the CODOS SAVE command instead of "D" on the AIM.

If you have the AIM assembler ROM, you may use disk files too. You may assemble directly from disk files, and output either the listing or the object code to the disk file, but not both. Of course if you really want both on disk all you have to do is assemble twice. You may not use .FILE directives to link multiple disk files together; however, this is not much of a hardship since you can easily concatentate files together in CODOS before assembly, using the instructions given for the TYPE command. Again, remember that the output from the AIM assembler is a paper-tape format file, which can only be loaded into memory using the AIM "L" command. Once loaded, it can be saved as a User-command file by entering CODOS and using the SAVE command.

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 if desired.

2. The Utility programs do not execute in the System 8K of RAM, but elsewhere in memory. Most Utility programs use a fairly large amount of memory because the programs need a large buffer space to perform their function efficiently.

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

The standard Utilities supplied are listed in table 6-1, and are described in the following section.

TABLE 6-1. UTILITY PROGRAMS

Name	Function
DIR	Display file attributes for selected file(s), using "wildcard" name matching.
COPYF	Copy file on multiple-drive system.
COPYF1DRIVE	Copy file(s) on single-drive system.
FORMAT	Prepare a new disk or erase an existing disk; test and bypass defective sectors; copy operating system.

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 sting 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.

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

EXAMPLES:

DIR

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

```
APEX.Z              :0 L  11-AUG-80  $0018C0
AIMEXT.Z            :0 L  11-AUG-80  $0001BF
SVCPROC.Z           :0 L  11-AUG-80  $00018F
COPYF.C             :0 L  11-AUG-80  $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 IN?ENTORY.? 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. The underline character is not available on the AIM-65 keyboard. On the AIM, the "$" character can be used in lieu of "_" (underline). The "-" character

therefore matches any string terminated by a "$" on the AIM.

2. 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. To display all files on drive 1, the correct command is "DIR *.?:1".

3. "Wildcards" are not available in other commands or Utilities.

UTILITY NAME: COPYF.

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

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

ARGUMENTS:

<file> = desired file name to copy.
<drive> = disk drive where file is to be found. Defaults to 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 TURKEY

copies the file TURKEY.C from drive 0 onto drive 1. The new file on drive 1 will also be named TURKEY.C.

COPYF DATA.D:1

copies file DATA.D from drive 1 to drive 0, with the same name.

COPYF CLONE NEWCLONE:0

duplicates file CLONE on drive 0. 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.

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. Ususally 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 another 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 memory.

NOTES:

1. COPYF1DRIVE should never be executed from a job file, but only from the console.

UTILITY NAME: FORMAT.

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

SYNTAX: FORMAT

ARGUMENTS: none.

EXAMPLE:

    FORMAT

initiates the interactive FORMAT Utiltiy. The Utiltiy 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. 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 takes about 3 minutes to complete. It is normally not essential to test diskettes unless you have doubts about the integrity of the disk. 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.

    The next prompt issued by FORMAT is:

    DISK VOLUME SERIAL NO. (VSN)?=

Enter any hex 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 every disk should be given a unique VSN. It is suggested that the VSN also be copied onto the external disk label using a SOFT magic marker when the formatting is done. Although the VSN is not used by the system for any purpose in this version, future releases will use the VSN for certain verification operations. In any event, you may assign any VSN you wish. The system will next prompt:

    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 20 K 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 will want to use the COPYF Utiltiy to copy additional files. In particular, you will probably want to copy the COPYF Utiltiy and the FORMAT Utility, and any device drivers (such as VMT) 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. 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.

INTERFACING USER-WRITTEN ASSEMBLY-LANGUAGE PROGRAMS TO CODOS

Introduction

This section discusses 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 7-1. 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.

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 a KIM system with CODOS at $E000 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.

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.

TABLE 7-1: CODOS SVC NUMBERS

SVC#	Description	Pass Regs.	Returns Regs.
0	Return to CODOS Monitor	-	-
1	Not currently defined		
2	Output inline message (see text)	-	-
3	Input byte from channel	X	A, F
4	Output byte to channel	X	-
5	Input line from channel	X,U5	A,Y,F
6	Output line to channel	X,Y,U6	-
7	Output string on channel	X,Y,U6	-
8	Decode ASCII hex to value	X,Y,U5	A,Y,F,U0
9	Decode ASCII dec. to value	X,Y,U5	A,Y,F,U0
10	Encode value to ASCII hex	X,Y,U0,U6	Y
11	Encode value to ASCII dec.	X,Y,U0,U6	Y
12	Querry default buffer addr.	-	U5,U6,Y
13	Not currently defined		
14	Querry 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	Querry file position	X	X,U7
21	Assign channel to file/device	X,A,U3	A,F
22	Free Channel	X	-

Note: This is a preliminary list. Other functions will be added later.

TABLE 7-2: CODOS PSEUDO-REGISTERS

Reg.	Address	Special Useage or Function
U0	$B0-B1	Often used for passing numeric values
U1	B2-B3	Often used for passing an address
U2	B4-B5	Often used for passing address or size information
U3	B6-B7	---
U4	B8-B9	---
U5	BA-BB	Points to start of input-line-buffer
U6	BC-BD	Points to start of output-line-buffer
U7	BE-C0	24 bit register used for passing File Position Ordinal

NOTES:

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.

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 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
;
          .=      $200      ;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 7-1 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
;
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 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" or "SELECTED CHANNEL 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 Supervisor 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 the individual SVC descriptions. 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 ti 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.

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 7-2 . 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 the address of an input buffer and U6 to hold the address of 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 the location of 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 buffer 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 80 characters. The following problem illustrates line-processing.

PROBLEM: Write a program to copy line 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.

(Continued)

SOLUTION:

```
SVCENB    =       $EE
U5        =       $BA       ;P-REG U5
U6        =       $BC       ;P-REG U-6
;
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
          TAY               ;ELSE SAVE CHARACTER COUNT IN Y...
          TAX               ;...AND X TOO
LOOP      LDA     (U5),Y    ;COPY CONTENT OF INPUT BUFFER...
          STA     (U6),Y    ;...TO OUTPUT BUFFER
          DEY
          BPL     LOOP      ;...UNTIL WHOLE LINE COPIED
          TXA               ;THEN RECALL LINE LENGTH...
          TAY               ;...TO Y
          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 Backspace (DEL or CNTRL-H) and CNTRL-X (start line over), 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 is entered, terminated by a carriage return. 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. The carriage return is replaced in the line with a $00 byte, and the character count in A does not include it.

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-of-File is tested as before. Note that the SVC 5 function returns the character count in A and Y set to 0 (therefore ready to index the first character of the line). The character count of the line is transferred to the X register as a temporary save, and the line is copied (backwards) from the input buffer to the output buffer. The output buffer is then output over channel 6. Note that the character count must be passed in the Y register. In the example, this character count was recalled from X to Y through A.

An alternative to copying the input buffer's contents to the output buffer would simply be to copy the contents of 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 7-1, 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
        BRK
        .BYTE   12      ;SVC 12 = GET BUFFER ADDRESSES
        LDX     #5      ;CHANNEL 5 FOR INPUT
        BRK
        .BYTE   5       ;SVC 5 = INPUT LINE
        BRK
        .BYTE   8       ;SVC 8 = DECODE HEX VALUE TO U0
        RTS
```

EXPLANATION:

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 a six line routine. 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). Note that the end-of-line delimiter is returned as $00.

Note: 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 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 copy the address of the buffers to U5 and U6, instead of using SVC 12.

SVC DESCRIPTIONS

SVC #0 ($00)

PURPOSE: Return to CODOS Monitor.

ARGUMENTS: None.

ARGUMENTS RETURNED: None.

DESCRIPTION:

SVC #0 returns control to the CODOS MONITOR. It has two possible advantages over simply using an RTS to return to the Monitor:

1. It can be executed anywhere, even in a subroutine, provided that SVCs are enabled;

2. 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.

EXAMPLE:

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

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 effect 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. 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.

2. 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 = 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 insure 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.

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 console, this will ring the bell (if the console is so equipped), 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.

SVC #5 ($05)

PURPOSE: Input line of text from channel.

ARGUMENTES:

X = Channel number to read from.

U5 = Address of desired input buffer for line.

ARGUMENTS RETURNED:

A = Count of characters in line.

Y = 0.

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

DESCRIPTION:

SVC 5 inputs a line of text from the file or device assigned to channel 5. The text will be deposited in a buffer whose address is specified in U5. The line of text will be terminated by a $00 byte. 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 $00 terminator. The Y register is always returned as 0 to facilitate user processing of the line using Indirect, Y addressing. If the channel selected is assigned to a device, then End-of-Line is defined as the first carriage return ($0D) encountered. This carriage return is converted to the $00 terminator in the buffer, and is not included in the character count in A.

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 80 ($50) characters. If an SVC 5 attempts to input a line with more than 80 characters, then the system will automatically add an end-of-line character after 80 characters are read. This is to prevent 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 80 characters. The system buffers are only 80 characters long, however, so the user will have to provide a buffer elsewhere and not use SVC 12 to define the buffer address.

2. The following editing characters are recognized by SVC 5. These editing characters are not returned in the line, but instead perform the function indicated:

BACKSPACE, DEL, or RUBOUT ($08 or $7F): Backspace one character. Will not backspace beyond beginning-of-buffer.

CNTRL-X or CAN ($18): Delete entire line (start over).

RETURN ($0D): End-of-line.

CNTRL-Z ($1A): End-of-File (applies only if entered from device, not in file; must follow carriage return).

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 the address of 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 (which is added by the system).

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 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 80 characters. Failure to limit the amount put into the system input 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 the address of 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.

SYSTEM GENERATION

System Generation is the procedure for "customizing" CODOS to a particular machine configuration. 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 the needs of the user. In particular, the system needs to know the number of disk drives and location of the input-output device driver subroutines, especially the Console. Chapter two tells how to get your system going the first time. This chapter tells how to make the changes a permanent part of the system.

One of the features of CODOS which greatly facilitates customization is the booting-up procedure. When CODOS is booted-up, it first loads the operating system memory image into memory. It then will read a list of CODOS commands from a file called STARTUP.J. 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, you can let the STARTUP.J file do this for you.

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 operating system image in memory after it is loaded. Since CODOS comes up in protected mode, you will need to UNPROTECT first, though.

You can write your own STARTUP.J file by simply using the TYPE command. To be on the safe side, we suggest you first TYPE the command file under another name, and then, when you are sure its correct, DELETE the old STARTUP.J and RENAME your new file STARTUP.J. This will be illustrated shortly by an example.

Generally, you can design your STARTUP.J file as you please, but you should keep in mind the following considerations:

1. The system initially assigns the Null output device to channel 2, and assigns channel 1 to STARTUP.J. It reads commands from channel 1 and outputs to channel 2. Thus no output will appear at the terminal unless channel 2 is reassigned. Normally, this should be done by executing DATE as the last command on the STARTUP.J file. DATE will assign channel 2 to the Console, give the signon message, and prompt for the date.

2. The STARTUP.J file must GET SVCPROC.Z if you plan to use SVCs.

3. You can't do any input or output to a device until its driver subroutines are loaded.

4. Any error detected by the system causes the system to stop reading the STARTUP.J file and try to issue an error message. Thereafter it will try to read from the Console.

5. In an AIM system, you must GET AIMEXT.Z if you want to use the keyboard and display on the AIM. AIMEXT contains the device drivers for the AIM plus the code needed to implement the disk support for the Editor, BASIC and Assembler ROMs, etc.

Table 9-1 is a list of the addresses of certain important parameters which you may wish to patch in the operating system. These addresses are subject to change as new releases are made. You must patch the addresses of the input and output drivers as a minimum in a KIM system. You must UNPROTECT before using SET to make the patches. It is strongly suggested you PROTECT before the end of STARTUP.J.

TABLE 9-1: CODOS V1.0 SYSTEM PARAMETER ADDRESSES

AIM-65 Addr.	KIM-1 Addr.	Description of contents
$8607-8608	C607-C608	CONSOLE KEYBOARD DRIVER SUBROUTINE ADDRESS (SEE NOTE 1).
860A-860B	C60A-C60B	CONSOLE DISPLAY DRIVER SUBROUTINE ADDRESS (SEE NOTE 1).
860D-860E	C60D-C60E	ADDRESS OF DRIVER ROUTINE TO DETERMINE IF A KEY IS DOWN ON CONSOLE (SEE NOTE 1). FOR KIM ONLY, SET C60C=4C.
8615-863C	C615-C63C	DEVICE NAME TABLE AND I-O DISPATCHERS. (SEE NOTE 2.)
87E9	C7E9	CODE FOR DISK TRACK-TO-TRACK SEEK TIME AND HEAD UNLOAD TIME (SEE NOTE 3).
87EA	CFEA	CODE FOR DISK HEAD LOAD TIME. (SEE NOTE 3.)
880F	C80F	NUMBER OF DISK DRIVES IN SYSTEM (1 OR 2).
883A	C83A	FLAG. IF BIT 7 = 1 THEN SYSTEM WILL IGNORE IRRECOVERABLE DISK READ ERRORS (NOT RECOMMENDED).
883B	C83B	FLAG. IF BIT 7 = 1 THEN PERMITS SAVE COMMAND TO OVERWRITE AN EXISITNG FILE WITH THE SAME NAME.
8845	C845	FLAG. IF BIT 7 = 1 THEN NO ECHO OF KEYBOARD INPUT WILL BE SENT TO THE CONSOLE OUTPUT (ELIMINATES DOUBLE-CHARS.)
8846	C846	FLAG. IF BIT 7 = 1 THEN A LINE FEED WILL BE SENT TO THE CONSOLE OUTPUT EACH TIME CR IS SENT.
8848	C848	ASCII CHARACTER TO BE USED IN LIEU OF CNTRL-C.
8849	C849	ASCII CHARACTER TO BE USED IN LIEU OF CNTRL-S.
884A	C84A	ASCII CHARACTER TO BE USED IN LIEU OF CNTRL-X.
884B	C84B	ASCII CHARACTER TO BE USED IN LIEU OF CNTRL-Z.
884C	C84C	ASCII CHARACTER TO BE USED IN LIEU OF UNDERLINE.
8856	C856	CODE FOR NUMBER OF SIMULTANEOUSLY-ACTIVE FILES.(SEE NOTE 5.)
8857	C857	MAXIMUM RECORD LENGTH FOR INPUT LINE.
885A	C85A	NUMBER OF BYTES TO DUMP PER DISPLAY LINE.
8868-8869	C868-C869	ADDRESS OF SYSTEM INPUT LINE BUFFER.
886A-886B	C86A-C86B	ADDRESS OF SYSTEM OUTPUT LINE BUFFER.
886C-886D	C86C-C86D	ADDRESS OF LARGE TRANSIENT BUFFER FOR COPYF, ETC.
886E-886F	C86E-C86F	SIZE (NOT FINAL ADDR.) OF LARGE TRANSIENT BUFFER.
8870-8871	C870-C871	ADDRESS OF USER-DEFINED INTERRUPT SERVICE ROUTINE.
8872-8873	C872-C873	ADDRESS OF USER-DEFINED ERROR RECOVERY ROUTINE.

NOTES:

1. See listing 9-1 for a description of the requirements for Console input and output routines.

2. See listing 9-2 for a description of the memory allocated for the Device Name table and Dipatcher.

3. The system is shipped with the seek time and head load time sufficiently long to satisfy any known disk drive. A very substantial performance increase can be realized by setting the correct seek time and head load time for your system. The times required for your disk are given in the specifications for the disk drive provided by the manufacturer. The code to install into memory is described in the K-1013 Disk Controller Hardware Manual page 37 and 29, for the "Specify" FDC command.

4. The above addresses are subject to change in future revisions.

5. The CODOS system is set up to allow simultaneous buffered access to a number of files. The system can be configured to allow 3, 4, 5, or 6 simultaeously-active files. Each simultaneously-active file requires a 256-byte buffer on an exact page boundary in the on-board System or User RAM on the disk controller. For a one or two drive system, there is sufficient room in the System 8 k for three such buffers. Up to three additional buffers can be allocated in User RAM. These buffers are managed automatically by the system. The standard system comes setup to allow up to the full 6 simultaneous files. Since the need for 6 simultaneously-active files rarely if ever occurs, users may wish to delete some or all of the buffer space which is allocated at the top of the User 8K of ram. Reducing the number of simultaneously-active files to 3 will free up all of the User-RAM buffers. To redefine the number of buffers available to the system, set the indicated address to 13 (decimal) times the number of buffers desired,plus one. For example, to delete all the Buffers in the user area on an AIM, use the command:

```
SET 8856 =.13*3+1   ;SELECT 3 SIMULTANEOUSLY-ACTIVE FILES.
```

This value should be set during SYSGEN if desired and not altered. Do not specify a value for less than 3 or greater than 6 simultaneously-active files. This value has no effect on the number of files which can be stored on the disk, nor on the number of I-O devices or channels.

SAMPLE STARTUP.J FILES

EXAMPLE 1:

An AIM system has one Siemans disk drive with a seek time of 6 ms. The VMT is to be used for the Console. The user has RAM available from $0000 to $3FFF in addition to the Visible memory at $6000 and the User RAM at $4000.

To create a new STARTUP.J, the user types:

```
TYPE C STARTUPNEW.J
```

This allows him to type his new command-list from the keyboard and save it on the STARTUPNEW.J file. The commands he enters are:

```
GET AIMEXT.Z     ;LOAD AIM I-O AND EXTENTIONS
UNPROTECT        ;UNPROTECT SYSTEM RAM
SET 880F = 1     ;JUST ONE DRIVE
SET 87E9 = AF 20 ;SET SEEK, UNLOAD, LOAD TIMES
VMT              ;LOAD, INITIALIZE VISIBLE MEMORY DRIVERS
GET SVCPROC.Z    ;LOAD SVC PROCESSOR
PROTECT          ;RESTORE SYSTEM WRITE PROTECT
DATE             ;ACTIVATE CONSOLE, ISSUE SIGNON MESSAGE, GET DATE
```

The user terminates the file with a CNTRL-Z. He then checks the file for accuracy by entering:

```
TYPE STARTUPNEW.J
```

When the user is convinced he has created the file correctly, he makes it the new STARTUP.J file by entering:

```
UNLOCK STARTUP.J
DELETE STARTUP.J
RENAME STARTUPNEW.J STARTUP.J
```

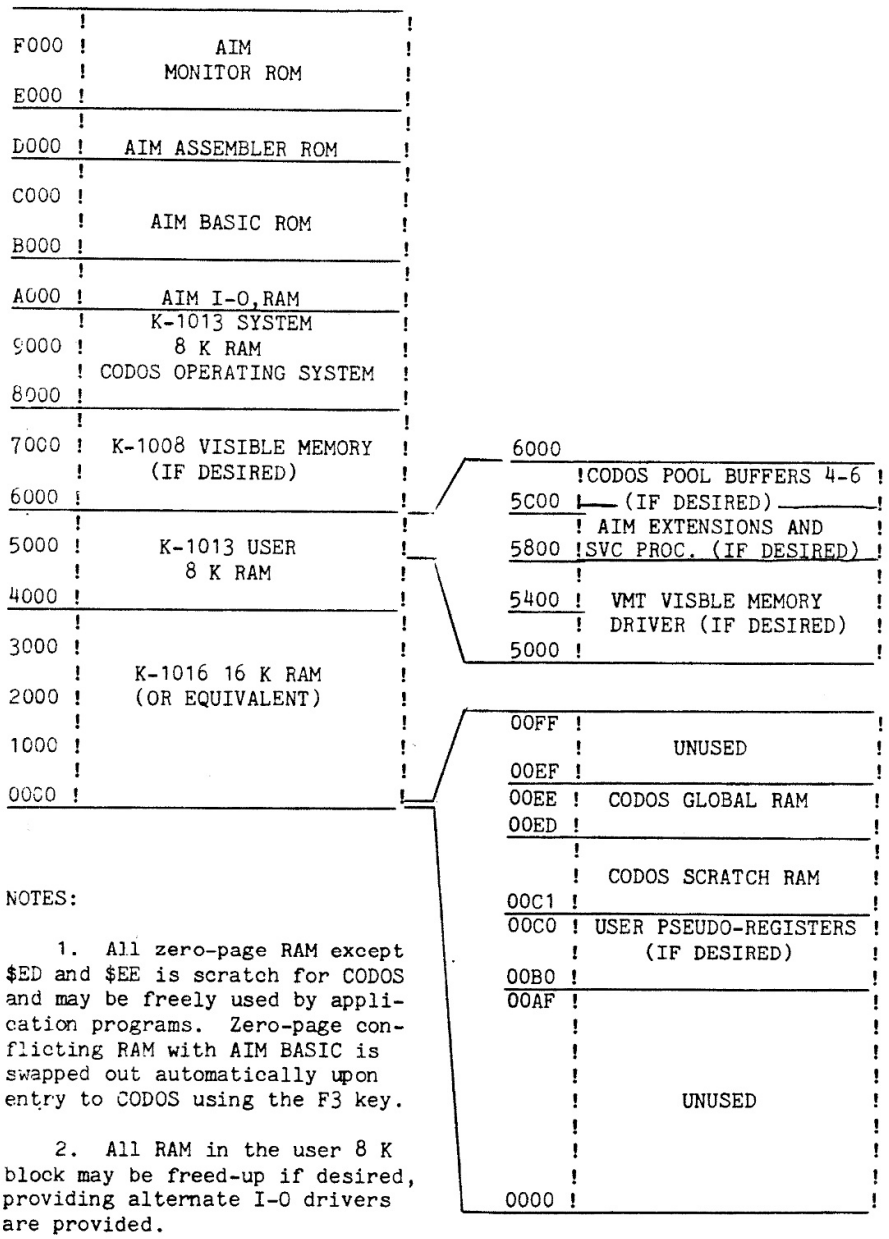
From this point on, if the Bootstrap loader is executed, the system will execute all the commands the user typed before accepting input from the Console.

```
ODOS  RELEASE 1.0
UMP TABLE


                                .PAGE 'JUMP TABLE'
* 178 00EF                      .=    X'8600    ;*** TABLE ORIGIN ***
  179                   ;
  180                   ;       JUMP-TABLE TO USER AND SYSTEM DEPENDENCIES
  181                   ;
  182                   ;       DEFINITIONS...
  183                   ;
* 184                   ;       JCHIN:  JUMP TO USER'S CONSOLE-CHARACTER-IN SUBROUTINE. MUST
  185                   ;       RETURN ASCII CHARACTER IN A WITH BIT 7 = 0 (UNLESS KEYBOARD
  186                   ;       MASK, KBMASK, HAS BEEN ALTERED FROM ITS DEFAULT 0 VALUE FOR
  187                   ;       A SPECIAL KEYBOARD).  USER SUBROUTINE CAN DESTROY ANY REGISTERS
  188                   ;       BUT MUST RETURN WITH STACK INTACT.  THE DRIVER DOES NOT HAVE
  189                   ;       TO ECHO THE CHARACTER TO THE DISPLAY.  THIS IS DONE BY CODOSSTORE 1005
  190                   ;       (UNLESS THE HALF-DUPLEX FLAG, HFDPLX, HAS BEEN CHANGED FROM
  191                   ;       ITS DEFAULT OF 0 TO $80).
  192                   ;
  193                   ;       JCHOUT:  JUMP TO USER'S CONSOLE-CHARACTER-OUTPUT SUBROUTINE.
  194                   ;       CHARACTER TO BE OUTPUT IS PASSED IN A REGISTER, WITH BIT 7=0.
  195                   ;       THIS ROUTINE CAN CLOBBER ANY REGISTERS BUT MUST RETURN WITH
  196                   ;       THE STACK INTACT.
  197                   ;
  198                   ;       JCHIF:  JUMP TO USER ROUTINE TO DETERMINE IF A KEY IS DEPRESSED
  199                   ;       ON THE CONSOLE KEYBOARD.  IF NOT, THE ROUTINE SHOULD SET BIT 7
  200                   ;       OF THE A REGISTER TO 1 (ASSUMING DEFAULT VALUE OF KBMASK) AND
  201                   ;       RETURN.  THE REMAINING BITS OF A ARE "DON'T CARE".  IF A KEY
  202                   ;       IS DEPRESSED, THEN THE ASCII KEY SHOULD BE RETURNED IN A WITH
  203                   ;       BIT 7 SET TO 0 (AGAIN, ASSUMING DEFAULT SETTING OF KBMASK).
  204                   ;       THIS ROUTINE CAN CLOBBER ANY REGISTERS BUT MUST RETURN WITH
  205                   ;       STACK INTACT.  IF YOUR CONSOLE DEVICE CANNOT SUPPORT THIS
  206                   ;       FEATURE, THEN JUST SET BIT 7 OF A TO 1 AND RETURN.
  207                   ;
  208                   ;       JCINIT:  JUMPS TO USER'S CONSOLE-INITIALIZATION ROUTINE.
  209                   ;       THIS ROUTINE WILL BE CALLED BY THE SYSTEM ON STARTUP PRIOR
  210                   ;       TO EXECUTING THE COMMANDS ON THE "STARTUP.J" FILE.  THEREFORE,
  211                   ;       IF THE DRIVERS FOR THE CONSOLE ARE LOADED BY A COMMAND IN THE
  212                   ;       STARTUP.J FILE, IT WILL BOMB THE SYSTEM IF YOU JUMP TO THE
  213                   ;       INITIALIZATION ROUTINE VIA JCINIT.  FOR THIS SITUATION, INCLUDE
  214                   ;       EXECUTION OF THE INITIALIZATION ROUTINE IN THE "STARTUP.J"
  215                   ;       COMMAND INSTEAD.  JCINIT DEFAULTS TO A RTS.  IT IS NORMALLY
  216                   ;       USED WHEN ROM-BASED DRIVERS ARE USED FOR THE CONSOLE WHICH
  217                   ;       CAN BE INITIALIZED BEFORE "STARTUP.J" IS READ BY THE SYSTEM.
  218                   ;
  219                   ;       JSINIT:  JUMP TO SYSTEM-DEPENDENT CODE FOR PARTICULAR SYSTEM.
  220                   ;       THIS ROUTINE IS PROVIDED BY MTU FOR EACH SYSTEM (AIM, KIM, ETC)
* 221                   ;       AND IS NOT NORMALLY ALTERED BY THE USER.
  222                   ;
  223 8600 4CA188   JCOLD:  JMP   COLD      ;JUMP TO COLD START (CANT BE RE-ENTERED!)
  224 8603 4C628A   JWARM:  JMP   COMD      ;JUMP TO WARM START ENTRY POINT
  225                       .IF    AIM       ;*****AIM SYSTEM ONLY...
  226 8606 4CCB58   JCHIN:  JMP   AIMKB     ;TO CONSOLE CHARACTER-INPUT SUBROUTINE
* 227 8609 4CFCEE   JCHOUT: JMP   AIMPR     ;TO CONSOLE CHARACTER-OUTPUT SUBROUTINE
  228 860C 4CF958   JCHIF:  JMP   AIMKD     ;TO CONSOLE KEY-DEPRESSED TEST SUBROUTINE
  229 860F 60       JCINIT: RTS             ;TO CONSOLE INITIALIZATION. NOT NORMALLY USED.
  230 8610 EA               NOP
  231 8611 EA               NOP
```

```
)DOS  RELEASE 1.0
EVICE DRIVER TABLES

                                .PAGE 'DEVICE DRIVER TABLES'
235                     ;
236                     ;       DNT: DEVICE NAME TABLE.
237                     ;
238 8615 4E             DNT:    .BYTE 'N'       ;"N" = NULL DEVICE DRIVER
239 8616 43                     .BYTE 'C'       ;"C" = CONSOLE DEVICE
240 8617 00                     .BYTE 0
241 8618 00                     .BYTE 0         ;RESERVED FOR CUSTOM DEVICES...
242 8619 00                     .BYTE 0
243 861A 00                     .BYTE 0
244 861B 00                     .BYTE 0
245 861C 00                     .BYTE 0
246                     ;
247                     ;       DDTI: DEVICE DRIVER DISPATCH TABLE FOR INPUT.
248                     ;
249                     ;       NOTE 1...FDERTT IS ADDRESS OF ERROR PROCESSING WHEN A PROGRAM
250                     ;       ATTEMPTS TO INPUT OR OUTPUT ON A DEVICE WHICH DOES NOT HAVE THE
251                     ;       CORRESPONDING DRIVER DEFINED (E.G., INPUT FROM PRINTER).
252                     ;
253 861D 0095           DDTI:   .WORD NULDVR    ;NULL DRIVER DEVICE (DTI=X'80)
254 861F 4A9D                   .WORD CIN       ;CONSOLE INPUT ROUTINE (DTI=$82)
255 8621 1E89                   .WORD FDERTT
256 8623 1E89                   .WORD FDERTT    ;CUSTOM DRIVER ADDRESSES...
257 8625 1E89                   .WORD FDERTT
258 8627 1E89                   .WORD FDERTT
259 8629 1E89                   .WORD FDERTT
260 862B 1E89                   .WORD FDERTT
261                     ;
262                     ;       DDTO: DEVICE DRIVER DISPATCH TABLE FOR OUTPUT.
263                     ;
264 862D 0095           DDTO:   .WORD NULDVR    ;NULL DRIVER (DTI=X'80)
265 862F 5F9D                   .WORD COUT      ;CONSOLE OUTPUT ROUTINE (DTI = $82)
266 8631 1E89                   .WORD FDERTT
267 8633 1E89                   .WORD FDERTT    ;CUSTOM DEVICE DRIVERS...
268 8635 1E89                   .WORD FDERTT
269 8637 1E89                   .WORD FDERTT
270 8639 1E89                   .WORD FDERTT
271 863B 1E89                   .WORD FDERTT
272
```

TO ADD A NEW DEVICE, INSERT THE SINGLE CHARACTER ASCII NAME INTO AN AVAILABLE BYTE IN TABLE DNT, AND THE ADDRESS OF THE INPUT DRIVER IN THE CORRESPONDING ENTRY IN DDTI. IF THE DEVICE HAS AN OUTPUT CAPABILITY, INSERT THE ADDRESS OF THE OUTPUT DRIVER ROUTINE IN THE CORRESPONDING ENTRY IN DDTO.

CODOS MEMORY MAP - AIM-65



NOTES:

1. All zero-page RAM except $ED and $EE is scratch for CODOS and may be freely used by application programs. Zero-page conflicting RAM with AIM BASIC is swapped out automatically upon entry to CODOS using the F3 key.

2. All RAM in the user 8 K block may be freed-up if desired, providing alternate I-O drivers are provided.

3. In standard form, CODOS uses $1000 to $2FFF for the large scratch buffer for the file copy Utility program; may be altered if desired.

4. Bootstrap loader uses $0000 through $00EE for scratch RAM during the booting-up process.

MEMORY MAP FOR CODOS (AIM)

Address	Usage
$9FE8-9FFF	Disk controller I-O. Write protect port at 9FE8.
9F00-9FE7	Bootstrap PROM.
8600-9EFF	CODOS Operating system code and tables in write-protected RAM.
8000-85FF	Pool disk buffers, Tables, and Directory buffers for CODOS in write-protected RAM.
5D00-5FFF	Default location for pool disk buffers 4, 5, 6. If pool buffers 4-6 are disabled, then unused.
5CAF-5CFF	Default location for system Output Line Buffer. Otherwise unused.
5C5E-5CAE	Default location for system Input Line Buffer. Otherwise unused.
5A70-5C5D	SVC processor, if desired. Otherwise unused.
5A12-5A6F	Zero page swap area for AIM Extension package, if desired. Otherwise unused.
5890-5A11	AIM I-O drivers and Extended AIM support package. If not desired, then unused.
5000-58EF	Optional VMT (Visible Memory Terminal) driver package, if desired. Includes text and high-resolution vector drawing capability, and AIM BASIC USR Function interface. If not enabled, then unused.
1000-2FFF	Default location of Large Buffer used by file-copying and FORMAT Utility programs. Otherwise, unused.
00EF	Used for Print-flag by optional VMT package. Otherwise, unused.
00EE	SVC Enable flag for CODOS.
00ED	CODOS global zero-page RAM. Do not use.
00C1-00EC	Scratch zero-page used by CODOS. Can be freely used by applications programs for scratch RAM. Conflicting AIM BASIC page zero RAM is automatically swapped and preserved by CODOS.
00B0-00C0	User Pseudo-Registers, if desired. Otherwise, unused.
0094-00AF	Scratch zero-page used by optional VMT package. Automatically swapped in/out, preserves AIM BASIC conflicting RAM. If not enabled, then unused.

NOTES:

1. By appropriate System Generation, it is possible to free all User RAM ($4000-5FFF), if desired.

2. Effectively, CODOS uses only location ED and EE in zero page for storage of permanent information. All other page 0 is essentially available.

LISTING 9-3: SAMPLE CONSOLE KEYBOARD DRIVER ROUTINE

```
    1                           .TITLE INCH, ' KIM KEYBOARD DRVR'
    2                   ;
    3                   ;       KIM ASCII KEYBOARD DRIVER FOR DATA TO PORT A BITS 0 TO 7,
    4                   ;       NEGATIVE GOING STROBE TO BIT 7.
    5                   ;
    6 1700              PAD     =       X'1700    ;KIM PORT A DATA
    7 1701              PADD    =       X'1701    ;KIM PORT A DATA DIRECTION
    8                   ;
    9 0000                      .=      X'1780    ;*****ORG***
   10                   ;
   11 1780 A900         INCH:   LDA     #X'00
   12 1782 8D0117               STA     PADD      ;INPUTS PLEASE
   13 1785 AD0017       INCH1:  LDA     PAD
   14 1788 30FB                 BMI     INCH1     ;WAIT FOR STROBE
   15 178A C961                 CMP     #97       ;LOWER CASE A
   16 178C 9007                 BCC     INCH4     ;BRANCH IF NOT L.C.
   17 178E C97B                 CMP     #123      ;L.C.Z+1
   18 1790 B003                 BCS     INCH4
   19 1792 38                   SEC
   20 1793 E920                 SBC     #X'20     ;FOLD LOWER TO UPPER CASE ALPHA
   21 1795 2C0017       INCH4:  BIT     PAD
   22 1798 10FB                 BPL     INCH4     ;WAIT FOR END-OF-STROBE
   23 179A 60                   RTS
   24                   ;
   25 0000                      .END
NO ERROR LINES
```

APPENDIX B: ERROR MESSAGES

CODOS Error NO.	Meaning
1	System crash: Floppy disk controller chip command phase error.
2	System crash: Floppy disk controller chip result phase error.
3	Missing or illegal disk drive number specified.
4	Unformatted disk or hardware drive fault.
5	Selected drive (or drive needed for System overlay) is not ready.
6	Disk drive seek error.
7	Hardware disk read/write error.
8	Irrecoverable disk read/write error.
9	Disk is write-protected.
A	Missing or illegal channel number specified.
B	Specified channel is not assigned.
C	Disk is hardware write-protected, or disk formatting error.
D	Can't FORMAT an open drive.
E	Illegal track specified on disk.
F	CODOS-reserved memory violation.
10	System crash: Bad file table file ordinal.
11	System crash: bad entry in block assignment table.
12	Disk is full; all blocks are allocated.
13	No room for new file; disk directory is full.
14	System crash: directory entry /file table mismatch.
15	Specified file was not found on selected drive.
16	System crash: defective directory entry.
17	File table is full (free some channels assigned to files).
18	Specified file does not exist.
19	Illegal or missing file name.
1A	Numeric overflow (greater than $FFFF).
1B	From argument is greater than To argument.
1C	Illegal sector specified on disk.
1D	Previous disk was not closed (or Reset hit).
1E	Guarded file violation.
1F	File is not a loadable-format file (not made by SAVE).
20	Illegal or missing character in command.
21	Value evaluates to greater than 255 ($FF) or less than 0.
22	Missing or illegal device or file name.
23	Selected disk drive (or drive needed for system overlay) is not open.
24	From address argument missing or illegal.
25	To address argument missing or illegal.
26	Entry point address argument missing or illegal.
27	Can't assign a new file to write-protected disk.
28	Command not found.
29	Value argument missing or illegal.
2A	Missing or illegal string delimiter.
2B	Memory verification failure during SET or FILL.
2C	Missing or illegal register name.
2D	Interrupt through undefined User-Interrupt Vector.
2E	Illegal or unimplemented SVC.
2F	Insufficient free channels.
30	Illegal or missing disk Volume Serial Number (VSN)
31	Illegal transfer of control into CODOS.
32	System crash: illegal system overlay number.
33	System crash: overlay did not load, or no system on disk in drive 0.
34	File name specified for SAVE already exists.
35	Missing or illegal Destination address argument.
36	Tried to input on output-only device, or visa-versa.