IBM Instruments Inc.

PRELIMINARY

Computer System
Operating System Reference Manual
Part 1. Operating System

Release 1.0

GC22-9199

Preliminary Edition Only (February 1983)

The contents of this edition are preliminary and subject to change. Changes will be included in subsequent Technical Newsletters or editions of this publication.

Requests for copies of IBM Instruments, Inc., publications should be made to your IBM Instruments, Inc., representative or by calling, toll-free, 800-243-3122 (in Connecticut, call collect 265-5791).

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Instruments, Inc., Department 79K, P.O. Box 332, Danbury, CT 06810. IBM Instruments, Inc. may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever.

00208830

01234567890

PREFACE

This manual describes the operating system -- CS-OS -- of the IBM Instruments Computer System. It consists of five chapters and an appendix.

- Chapter 1 -- "Introduction to the Operating System" -- describes the operating system and the system commands used with it.
- Chapter 2 -- "Computer System Text Editor" -- describes the main features of the text editor, including the various commands that can be used to create and edit program and data files.
- Chapter 3 -- "Computer System Macro Assembler" -- describes the instructions, instruction formats, addressing modes, and related aspects of the two-pass macro assembler that runs under the operating system.
- Chapter 4 -- "Linker, Locater, and Library Manager" -- describes the three programs available to the user for the development of modular code.
- Chapter 5 -- "CS-Debug" -- describes the special debugging utility designed to run in the multitasking environment of CS-OS.
- Appendix -- "Error Messages" -- lists and defines the various error messages that may be generated in the operation of CS-OS.

Related Publications:

Publications that discuss related aspects of the Computer System are:

Computer System Product Description, GC22-9183

Computer System BASIC Reference Manual, GC22-9184

Computer System Operating System Reference Manual
Part 2: Logical I/O and System Services, GC22-9200

Computer System Problem Isolation Manual, GC22-9192

CONTENTS

1.0 INTRODUCTION TO THE OPERATING SYSTEM

The IBM Instruments Computer System has a disk-based multitasking operating system -- CS-OS -- that supports standard peripherals such as a line printer, CRT display, floppy disks, hard disks, and auxiliary consoles.

This chapter is designed to get you started using CS-OS. It is neither a tutorial on operating systems nor an exhaustive treatment of how to use or modify the software, but it should tell you what you need to know to begin working with the software.

1.1 GENERAL INFORMATION

1.1.1 COMMAND STRUCTURE

Commands in CS-OS consist of a command name and optional parameters. Some commands are resident in memory and will execute immediately; others are transient (stored on disk) and must be loaded from disk before they are executed. User-defined commands are invoked by entering their full names. These command files must be binary type with transfer addresses (access type 01).

Where CS-OS requires numeric values, either decimal or hexadecimal notation may be used. Hex values must be preceded by a dollar sign ($). The operator prompt is:

```
0> _
```

The digit before the ">" symbol is the drive number of the default disk.

1.1.2 NAMING FILES

A fully specified filename consists of four fields: A volume label, a catalog name, a filename and an extension. When filenames are specified

in system commands, specific delimeters must be used to separate the fields.

1.1.2.1 Filename Format

The standard filename format for use in system commands is either

<volume:><catalog.>filename.ext

or

<drive:><catalog.>filename.ext

where

volume — is a field of one to six alphanumeric characters and is always terminated by a colon. This field can be omitted, in which case the default volume is used. A volume name cannot be a single numeric digit.

drive — is a single digit number corresponding to the last digit of the disk device name. For drive #FD00 the drive number would be 0, for #FD01 it would be 1, and so on.

catalog — is a field of one to eight alphanumeric characters (with the leading character alphabetic) and is always terminated with a period. This field can be omitted, in which case the default catalog is used.

Note: For the current release of the operating system there is no supprt for the catalog field; however, it is defined for use in future releases. If specified as part of a command it will be flagged as an error.

filename — is a field of one to eight alphanumeric characters with the leading character alphabetic. It is always followed by a period and the filename extension.

ext — is a field of one to three alphanumeric characters with the leading character alphabetic.

EXAMPLES OF VALID FILENAMES

INPUT.TXT DOREEN1.REL H1.H LINDAS.FIL

0:INPUT.TXT 1:INPUT.HEX 0:INPUT2.TXT

123456:COLORS:BLUE.SRC CLYDE:TESTCASE.BIN

VOL7:Z1.HERRYPGM.REL POLLY.TEMPFILE.SRC

RESTRICTIONS ON THE USE OF FILENAMES FOR SYSTEM COMMANDS

1. The catalog name is not supported in this release of the operating system. It should not be used in commands.

2. The volume field is presently used only in the COPY program. You should use the drive number as a means of selecting diskettes in all other system operations.

1.1.2.2 Entering Filenames

To specify a file, give the disk drive number, filename, and extension. The drive number is given as a decimal digit followed by a colon. The following are examples of unique files:

```
0:INPUT.TXT  1:INPUT.TXT  1:INPUT.HEX  0:INPUT2.TXT
```

The system maintains a default drive. If a file is on the default drive, the drive number and colon may be omitted from the file specification. Normally, the default drive is set to zero. Hence, the following two file descriptors are identical:

```
0:MELS.BIN  MELS.BIN
```

Using the SET command, the user may modify the default drive. Any drive in the system may become the default drive.

Note that only alphanumeric characters may appear in filenames or extensions. The following are invalid filenames:

```
1:TERRYSFILE.HEX      (name more than 8 characters)
2:TEMP.FILE           (extension more than 3 characters)
0 TEST.TMP            (colon missing after drive number)
BASIC+.BIN            (+ is a nonalphanumeric character)
EDITOR                (file extension missing)
```

1.1.3 WILDCARD FEATURE

CS-OS permits manipulation of classes of files. The mechanism for forming such classes is called wildcarding. Two wildcard characters perform unique identification tasks. The asterisk (*) matches an entire string of characters of arbitrary length. Since a complete filename consists of two strings (a name and an extension) the wildcard filename *.* expresses all possible filenames. The wildcard filename *.BLD expresses all filenames with the extension BLD.

The second wildcard character is the question mark (?). This character substitutes for any single character (including any blanks the system may have incorporated to "fill out" the filename to it maximum legal length). Hence, the filename TEST?.HEX is equivalent to TEST.HEX or TESTP.HEX or TEST2.HEX. It is not equivalent to TESTING.HEX. The filename *.* is equivalent to ????????.???.

The asterisk character (*) can be used to match any remainder of a string. When it is used in positions other than the first in a string, it is equivalent to the number of "?" characters sufficient to fill out the string (up to 8 for filenames, up to 3 for extensions). For example, the following wildcard file specifications match all files on the default drive whose names begin with the letters "WILD" and whose extension begins with the letter "T."

WILD*.T* WILD????.T* WILD????.T??

1.1.4 CTRL-ALT-DEL FUNCTION

There may be times when the user wishes to restart the system without resetting it: a device may not be working, an erroneous command may have been typed, and so forth. CS-OS uses a three-key sequence as an abort mechanism: Ctrl-Alt-Del. This set of keystrokes causes a restart that closes any open files, flushes any pending I/O, prints the start-up banner, and readies the system for new commands. Any pending SUBMIT file is terminated.

WARNING: This abort function should be used sparingly.

1.1.5 TASKS AND MULTITASKING

A "task" (sometimes termed "process") is a program that is run under the control of CS-OS. In fact, parts of CS-OS are themselves "tasks." Tasks run concurrently: that is, they appear to share the resources of the computer. (Such resources include the processor itself, console, memory areas, disk files, etc.) Each task is associated with a data structure called a Process Control Block (PCB), which contains fields that store information about the task and provide the mechanisms for the support of concurrency. Tasks call on the features of CS-OS to gain access to system resources in a controlled manner. CS-OS schedules tasks so as to give each a share of the computer's time and resources. Synchronization mechanisms and intertask communication channels are provided through system calls described elsewhere in this manual.

Each task in the system has an identifying name of up to 8 alphanumeric characters. This name is the means for calling upon system tasks (by means of system commands that will be described later in this chapter. No two tasks in the system may have duplicate names. No wildcarding is permitted in task names.

1.1.5.1 Predefined Tasks (System and Idle)

The task name SYSTEM is predefined. This task performs all CS-OS commands and actually constitutes the "system" with which the user interacts. The SYSTEM task begins running when CS-OS is started. The user may issue commands to SUSPEND or change the priority of the SYSTEM task, but should do so with care.

There is a second predefined task in CS-OS: the "idle" task. This task is an exception to the rule that tasks have names. The "idle" task has no name and does not appear in the TASKS display, but it is always in the system. The "idle" task has the lowest priority possible -- it runs only when all other tasks (including SYSTEM) cannot run for some reason. The user cannot SUSPEND or KILL the "idle" task. There is no need to in any case, since "idle" will not run if there are any other tasks that are ready to run.

1.1.5.2 Task Priority, Task Status, and Task Interrupts

Each task has a priority value associated with it. The priority is a number between 1 and 127, with the higher numbers representing tasks of greater priority. Within CS-OS, tasks are ordered by priority, the higher numbers run ahead of lower ones. Tasks of equal priority are scheduled on a round-robin basis. A task's priority is set when the task is created, but may be changed if desired. The priority of SYSTEM is 64, while the priority of "idle" is 0.

Each task has an associated status byte. This byte indicates the current status of the task and may take on one of the following values:

0 - No task (PCB is unallocated or task has been killed)
1 - Task is ready to run (on "ready queue")
2 - Task is delayed (SUSPENDed or DELAYed)
3 - Task is waiting for a resource (in a monitor)

The time at which a task is created (as determined by the time-of-day clock) is part of the task PCB. SYSTEM is always shown at the time that CS-OS was started or restarted.

Tasks are switched on each real-time clock interrupt (every twentieth of a second). A task switch could also occur when the running task must wait for a system resource or I/O device. The highest-priority task on the ready queue (which could be the task that had been running) is dispatched and the new task begins its execution. This task switching is transparent to the user except for the time delays that become involved when more than one task shares the computer. If a task has a priority greater than SYSTEM (>64), then SYSTEM will run only when the higher-priority task is waiting for some system resource or I/O device. This could make it appear that SYSTEM is not running at all. Similarly, if a user's task has priority lower than SYSTEM (<64), it may appear that the user task never runs. Actually, the user task runs whenever SYSTEM must wait for a system resource or I/O device. Task priorities must be chosen with care.

It is possible to delay a task (make it stay off the ready queue) for a specified number of real-time clock "ticks." The SUSPEND command and the DELAY system-call provide this facility. A SUSPENDed task will be placed back on the ready queue after a specified number of "ticks." The RESUME command and WAKEUP system call provide a means to immediately place a SUSPENDed (or DELAYed) task back on the ready queue. These commands and

system calls give the user more flexibility in the control of task execution.

Tasks are started up by using the RUN command with an initial priority. The task priority can be changed at any time by using the PRIority command.

1.2 SYSTEM COMMANDS

1.2.1 USER TRANSIENTS

Any access-type 01 binary file can be executed directly as part of the SYSTEM task. For example, if drive #1 has a program file called PGM.BIN, then the program can be run with the command:

```
1:PGM
```

The system loads the transient file into memory and jumps to its transfer address. (Invoking the program is thus equivalent to a LOAD command.) If there is no transfer address, control returns to the system. Parameters required by the user transient may be input on the same command line that invokes it. A space, comma, or other nonalphanumeric delimiter must separate the parameters from the file specification. For example,

```
1:COPY.BIN INPUT.TXT,MYVOL:OUTPUT.TXT
```

invokes the COPY program on drive 1 and specifies the input and output files to be used. The COPY transient must parse the command line starting with the delimiter.

1.2.2 ATCHDEV (RESIDENT)

The ATCHDEV command attaches a device to an existing driver that is part of the system. The format of the command is:

```
ATCHDEV device
```

where 'device' is the device name to be attached. The first character must be a '#' to indicate a device name. The next three characters must match the device driver name to which it will be attached. An error will occur if the driver does not support additional devices. Upon successful completion of the command, the list of attached devices is displayed via the LISDEV command.

ERROR MESSAGES:

```
SYNTAX ERROR
INVALID COMMAND FOR DRIVER
DEVICE ATTACH ERROR
```

1.2.3 ATCHDRV (RESIDENT)

The ATCHDRV command attaches a device driver to the system by loading a binary image file from disk and executing it. The file is assumed to contain the device driver code. If no extension is specified in the filename, the command assumes '.DVR'. The command examines register D7.B on return for an indication of the driver's success at initialization. A nonzero value indicates failure. The format of the command is:

```
ATCHDEV filename.ext
```

where 'filename.ext' is the name of the file containing the driver code.

ERROR MESSAGES:

```
DRIVER ATTACH ERROR
NO DRIVER INITIALIZATION ADDRESS
```

1.2.4 CLRDSPL (RESIDENT)

This command clears the display page of graphics memory. It should be used with discretion. The format of the command is:

```
CLRDSPL
```

1.2.5 COPY (TRANSIENT)

The COPY program transfers data from one device or file to another. The format of the command is:

```
COPY source,destination[;options]
```

where source and destination can be a device specification (Part 2, Section 1.2.5):

```
#device
```

or a file specification (Part 2, Section 1.2.4):

```
[volume:]filename.ext
```

In the case of a file specification, the volume identifier field is optional. If omitted, the default volume identifier is used.

As an additional alternative, the drive number may be entered in place of the volume identifier. In this case, the system looks for the volume identifier of the disk mounted in the given drive. Currently 0 through 3 specify diskette drives, and 4 through 7 specify hard disk drives. Directly accessing the disk drives through their device names (#FD0X or #HD0X) is not accepted by COPY command.

The wildcard character asterisk (*), may be used within or in place of the filename, the extension fields, or both. The wildcard feature may be used only in a file-to-file copy. If both source and destination are files, the following options may be implemented:

V -- verify the file contents of the destination diskette.

C -- compare files. A byte-by-byte comparison is made between the two files specified.

The format of the output is:

RELATIVE SECTOR	OFFSET	FILE 1 BYTE	FILE 2 BYTE
XXXXXXXX	XXXX	XX	XX

All values are printed in hexadecimal. If ten or more mismatches are encountered, a message is printed and the file comparison is aborted.

When the destination of data is a device, the following keyboard control characters are accepted:

Ctrl/Break -- quit
Ctrl/Numlock -- halt until any key pressed.

If the fields following the COPY command are omitted, copy will prompt the user for the desired input, as shown in the following example:

```
COPY
CS/9000   COPY       1.0    1/06/83
COPYRIGHT 1982 IBM INSTRUMENTS, INC.
ENTER SOURCE DEVICE OR FILE SPECIFICATION:  *.SMP
ENTER DESTINATION DEVICE OR FILE SPECIFICATION:  TEST:*.SRC
ENTER OPTIONS: V
COPY (Y/N/Q)  RAMVOL: GRTEST02.SMP TO TEST  : GRTEST02.SRC ? Y
FILE EXISTS: OVERWRITE (Y/N) TEST  : GRTEST02.SRC ? Y
COPY (Y/N/Q)  RAMVOL: PRTEST00.SMP TO TEST  : PRTEST00.SRC ? Q

ANOTHER COPY ?  N
```

This prompts all files with the extension .SMP on the default volume (here it was RAMVOL) that is to be copied to the volume identifier TEST, with the extension .SRC. If the file already exists on the destination diskette, the user is asked whether it should be overwritten. Each file is verified after it has been copied. Y = Yes, N = No, Q= Quit.

```
COPY 1:SAMPLE.SRC,#PR
```

This copies the text file SAMPLE.SRC (file type 3) from the volume identifier mounted in drive 1, to the printer.

```
COPY #CON,#SER00
```

This transfers characters typed in from the keyboard, to the device connected to the auxiliary port. The terminating character is CTRL-D (hex 04).

```
COPY #CON,TEST:MYTEXT.SRC
```

The text file MYTEXT.SRC is created on the volume identifier TEST. It contains all the data typed in from the keyboard, until an end-of-file character (CTRL-D) is typed.

ERROR MESSAGES:

SYNTAX ERROR
ILLEGAL DEVICE NAME

```
ILLEGAL VOLUME NAME
ILLEGAL FILENAME
CANNOT HAVE AMBIGUOUS FILENAME
FILENAMES MUST BE SAME FORMAT
ILLEGAL DEVICE FOR COPY PROGRAM
INPUT MUST BE TEXT FILE
*******: FILE ALREADY EXISTS
*******: VOLUME NOT FOUND
*******: FILE NOT FOUND
*******: OPEN INPUT DIRECTORY ERROR
*******: READ DIRECTORY ERROR
*******: OPEN INPUT ERROR
*******: OPEN OUTPUT ERROR
*******: OUTPUT DIRECTORY FULL
*******: READ ERROR
*******: WRITE ERROR
*******: CLOSE OUTPUT ERROR
FILE SIZES ARE NOT THE SAME
STATUS= XXXX (value in hexadecimal)

(******* is a device name or a volume name)
```

1.2.6 DELETE (RESIDENT)

This command removes a file from disk. Wildcard characters in filenames can be used to remove categories of files. The format of the command is:

```
DELETE [drive:] filename.ext
```

where the drive number will be given the default value (0) if no drive is specified. This command will delete only those files with an access code of 00. (See the listing of access codes under the SECURE command>) The filename and extension fields may contain wildcard characters. When a named file is found, the system issues a prompt that gives the user a chance to save the file.

```
DELETE MYFILE.TMP

DELETE- 0:MYFILE.TMP ? YES
```

The YES response assures the operating system that this is the file to be deleted. The YES can be abbreviated to Y; any other input is interpreted as a NO.

```
DELETE *.TMP

DELETE-0:MYFILE.TMP ? NO
```

would be the correct response if MYFILE.TMP was not the one that was to be deleted. This strategy saves files from being wiped out accidentally by typographical errors. If there are several matches -- due to the use of a wildcard character in the filename -- each will be prompted in turn, and any of the matches may be removed. Suppose, for example, that drive 1 has three TXT files named TEST1, TEST2, and TEST3. Then the following command sequence removes files TEST2.TXT and TEST3.TXT but not TEST1.TXT:

```
DELETE 1:TEST?.TXT

DELETE-1:TEST1.TXT ? NO

DELETE-1:TEST2.TXT ? YES

DELETE-1:TEST3.TXT ? YES
```

1.2.7 DIR (RESIDENT)

The DIR command provides a list of the files on a specified disk. The listing prints on the console device unless it is directed to the line printer. The

```
DIR        (goes to console)

DIR /L     (goes to line printer)
```

The directory listing has the following format:

```
NAME TYPE-CODE ACCESS-CODE FIRST-SECTOR LAST-SECTOR SECTOR-COUNT TIME
```

The type code, access code, first sector, and so on are output in hexadecimal. The number of sectors is output in decimal.

The type codes defined in CS-OS are:

00	binary file
01	binary file with transfer address
02	unused
03	text file (hex file)
04	unused
05	Contiguous file (used only by DIR.DIR)

For access-code definitions, see the section on the SECURE command.

Filenames are listed as 8-character strings with 3-character extensions. Following the directory list, the total number of disk sectors used by the listed files is given in decimal.

The DIR command allows several levels of file qualification for listing categories of files.

```
DIR [/L] [drive] [filename.ext]
```

If the drive number is not specified, the default value is assumed.

```
DIR 1
```

will list the entire directory. The filename and extension may use wildcards. For example:

```
DIR 1:*.HEX
```

will list on the console all files from disk 1 that have the extension HEX. Another example:

```
DIR /L TEST?.*
```

will list on the line printer all files from the default disk that have names beginning with TEST followed by a character (or blank).

You can stop the displayed directory from scrolling by pressing Ctrl-numlock. Press any other key to resume.

1.2.8 DISKCOPY (TRANSIENT)

The DISKCOPY command copies the contents of the source diskette to the destination diskette. The sectors are copied, starting with logical sector 1 and copying through the total number of sectors per diskette. The volume label sector is not copied. When the diskette copy is completed, the destination diskette volume identifier may be changed. If both source and destination diskettes have identical volume identifiers, a warning is printed. Both source and destination diskettes must have the same format, (e.g., 2-sided, double density). The format of the command is:

```
DISKCOPY source,destination[;options]
```

where source and destination are drive numbers (0, 1, 2 or 3), or they may be omitted. In this case the user is prompted for source, destination, and desired options. Legal options are:

V -- verify destination diskette

DISKCOPY prompts the user, providing a chance to save the contents of the destination diskette. For example:

```
DISKCOPY
WARNING:  DESTINATION DISKETTE CONTENTS
          WILL BE DESTROYED
COPY FROM DRIVE 0 TO DRIVE 1? Y
CURRENT DESTINATION VOLUME IDENTIFIER IS:  FOL1
DO YOU WANT TO CHANGE THE VOLUME IDENTIFIER (Y/N)?  Y
ENTER NEW VOLUME IDENTIFIER:  TEST
DISKCOPY COMPLETED
```

For a system with only one diskette drive, DISKCOPY prompts the user to change from input diskette to output diskette in the drive specified. The program utilizes the maximum amount of memory space available in the system, in order to read and write a multiple number of sectors at a time. This enables the user to change diskettes the least number of times.

```
DISKCOPY 0
```

ERROR MESSAGES:

SYNTAX ERROR
DISKETTE FORMATS DO NOT MATCH
END OF MEMORY ADDRESS ERROR
CHANGE VOLUME IDENTIFIER ERROR

```
OPEN INPUT ERROR
OPEN OUTPUT ERROR
READ ERROR
WRITE ERROR
STATUS: ****
VERIFY ERROR:  LOGICAL SECTOR *********
```

1.2.9 DISKUTIL (TRANSIENT)

The DISKUTIL program allows the user to examine the contents of diskette sectors. The format of the command is:

```
DISKUTIL
```

The sector contents are displayed in both hexadecimal and ASCII notation. On entry, DISKUTIL prompts for the drive number, followed by the sector number that you wish to display. The sector number may be issued in decimal or hexadecimal (e.g., 21 decimal = $ 15 hexadecimal). After the first file sector is entered, the user may type 'F' or 'B' to step forward or backward through the file as indicated. Typing 'Q' exits the program.

ERROR MESSAGES:

```
INVALID SECTOR NUMBER
OPEN ERROR
READ ERROR
STATUS = *****  (***** = error code in hexadecimal)
```

1.2.10 DTCHDEV (RESIDENT)

The DTCHDEV command detaches a device from an existing driver. The format of the command is:

```
DTCHDEV device
```

where 'device' is the device name that is to be detached. Upon successful completion of the command, the list of remaining attached devices is displayed via the LISTDEV command.

ERROR MESSAGES:

```
SYNTAX ERROR
INVALID COMMAND FOR DRIVER
DEVICE DETACH ERROR
```

1.2.11 DTCHDRV (RESIDENT)

The DTCHDRV command detaches a device driver from the operating system. The format of the command is:

```
DTCHDRV device
```

where 'device' is the master device name of the driver. All control blocks associated with the device driver are disabled. These include Physical Device Blocks (PDB), Time Out Blocks (TOB), Interrupt Descriptor Blocks (IDB), etc. Upon successful completion of the command, the list of remaining attached devices is displayed via the LISTDEV command.

ERROR MESSAGES:

```
SYNTAX ERROR
INVALID COMMAND FOR DRIVER
DRIVER DETACH ERROR
```

1.2.12 FORMAT (TRANSIENT)

The FORMAT command creates physical sectors on a diskette in the density, number of sides, and sector size specified. The configuration information and the given volume identifier, are placed in the volume label of an 8" diskette according to IBM Diskette Standards. The information is placed in logical sector 0 for a 5 1/4-inch diskette (PASS 1). Then the program initializes the sectors to zero and creates a chain of logically linked free sectors that are available to the user. A level number is placed in logical sector 0, indicating the file access structure (1=linked sectors). The file entry 'DIR.DIR' is created in the directory, to be used for file access. The time portion of this file indicates when the diskette was last formatted (PASS 2). The format of the command is:

FORMAT

PROGRAM FUNCTIONS:
(1) Format diskette -- creates physical sectors on a new diskette in the specified format, then initializes and links the logical sectors.
(2) Initialize diskette -- clears and relinks existing sectors. It does not change physical attributes of the diskette, including volume identifier.
(3) Change Volume Identifier--does not destroy data on diskette.
(Q) End program.

DRIVE NUMBER: Diskette drive numbers are: 0, 1, 2, or 3

FORMATS SUPPORTED:
8-INCH DISKETTE
(1) Double sided, double density, sector size 256
(2) Single sided, single density, sector size 256
(3) Double sided, single density, sector size 256

5¼-INCH DISKETTE
(1) Double sided, double density, sector size 256
(2) Single sided, double density, sector size 256

Note: the default for both diskette types is choice 1 (2D diskette).

VOLUME IDENTIFIER: One to six digits or letters without blanks; e.g., TEST5

Note: Prohibited volume names: '0' through '9'

SECTOR SEQUENCE NUMBER: This is used as a fixed value when the physical sector numbers are incremented on the diskette. Accepted values are:

8-inch double density -- 1 to 13 (default =13)
8-inch single density -- 1 to 8 (default =8)
5 1/4-inch double density -- 1 to 8 (default=3)

ERROR MESSAGES:

OPEN ERROR
CONFIGURATION ERROR
WRITE TRACK ERROR
WRITE SECTOR ERROR
STATUS = **** (**** = error code in hexadecimal)

1.2.13 HELP (TRANSIENT)

The HELP command displays a list of CS-OS commands on the console screen. The HELP command is invoked by typing:

```
HELP
```

1.2.14 JUMP (RESIDENT)

This command allows the user to leave CS-OS and go to any arbitrary absolute address. If the program at that address does a subroutine return (RTS instruction), CS-OS will continue at the command level:

```
JUMP  E112
```

will go to the address E112 (hexadecimal).

```
JUMP 256
```

will go to the address 256 (decimal).

1.2.15 KILL (RESIDENT)

The KILL command removes tasks from the system. KILL works only on tasks that are running or ready to run (on ready queue). The syntax of the KILL command is:

```
KILL taskname
```

where 'taskname' is the name of any task currently in the system (except SYSTEM). If the SYSTEM task were to be KILLed, none of the CS-OS commands described in this manual would be available. If the KILL command is tried on the SYSTEM task, the error message will be displayed.

KILL should be used with great care since resources acquired by a task can be permanently lost if that task is KILLed. All the CS-OS devices are protected from this; if a task has acquired one or more of the SYSTEM resources, it cannot be KILLed until it releases the resources. The error message

```
TASK HOLDS RESOURCES
```

indicates that the task must release its resources before it can be KILLed. Similarly, the error message

```
NOT ON READY QUEUE
```

indicates that the task is not in the proper state, which probably means that it has acquired some user resource.

1.2.16 LISTDEV (RESIDENT)

The LISTDEV command lists all device drivers currently known to the system.

```
LISTDEV
```

A device driver is known if it has done an ATACHPDB to the IOMGR. The six-character identifier in the Physical Device Block (PDB) is displayed along with the driver logical attributes, one hexadecimal byte. If an alternate device identifier is supported, its six-character name is also displayed. The following example shows the format of the LISTDEV output.

```
LISTDEV
      #SCRN0          21        #SCRN1          21        #CNSL0          21
      #CON            30        #KPD            30        #PR             B0
      #GR             01        #SER00          30        #SER01          30
      #SER02          30        #PPU            B0        #BUSA?          12
      #FD00  TEST     1A        #FD01  DMS403   1A        #FD02           1A
      #FD03           1A
```

1.2.17 LOAD (RESIDENT)

This command puts programs into the transient area. They are not executed -- control returns to CS-OS command level. LOAD requires that files be binary type (00 or 01 type). The format of the command is:

```
LOAD [drive:] filename.ext
```

where the drive will default if omitted. No wildcard characters are permitted in the filename or extension.

```
LOAD 1:PROG1.BIN
```

loads PROG1.BIN into the transient area.

1.2.18 PAGDSPL (RESIDENT)

This command switches the displayed pages of graphics memory. That is, page 0 becomes page 1 and page 1 becomes page 0. The format of the command is:

```
PAGDSPL
```

1.2.19 PRI (RESIDENT)

This command changes the priority of a running task. The format of the command is:

```
PRI taskname,priority
```

where 'taskname' is the name of a running task, and priority is the new priority. Priorities can be in the range 1 to 127 with the SYSTEM task having priority 64.

If the priority is not in this range then the error message

```
INVALID PRIORITY
```

will be displayed.

1.2.20 RENAME (RESIDENT)

This command changes the name of a file without modifying its contents. The command format is:

```
RENAME [drive:] oldname.oldext,newname.newext
```

where the drive will default if omitted. The file access code must be 00 or 01 to allow renaming. The newname must not exist already with that extension on that disk. The following command, for example, will rename the file DAVES.OLD to DAVES.NEW:

```
RENAME DAVES.OLD,DAVES.NEW
```

No wildcard characters are permitted in either the new or old names or extensions.

1.2.21 RESUME (RESIDENT)

The RESUME command immediately places a SUSPENDed task in the ready queue. RESUME has no effect on a task already running or on the ready queue. The format of the command is:

```
RESUME taskname
```

where 'taskname' is the name of some task currently in the system. If "taskname" is not a current task, then the error message

```
NO SUCH TASK
```

will be issued by the operating system.

1.2.22 RUN (RESIDENT)

A process (task) is started from the console by using this command. The command format is:

```
RUN filename.ext,priority
```

where filename.ext is the name of a binary program file of type=01 (i.e., has a transfer address) and priority is the desired priority. After the task is loaded and has been placed on the READY queue, control returns to the system.

NOTE: A maximum of 6 tasks (in addition to the SYSTEM task) is allowed.

If the priority is not in the range 1 to 127 then the error message

INVALID PRIORITY

will be displayed. If there are no available PCB's for starting the task then the message

XXXXXXXXXX

will be displayed.

1.2.23 SAVE (RESIDENT)

This command saves an area of memory as a binary file. The command format is:

```
SAVE [drive:] filename.ext,startad,endad [,transfer ad]
```

where the drive defaults if omitted. The filetype of the save-file will be 00 if no transfer address is present, and 01 if a transfer address is supplied. For example, the following command will save the first 16k of memory as a system file to be entered at the address 10AC hexadecimal:

```
SAVE 1:USER.BIN, 9000, 9800, 9000
```

Addresses can also be entered in decimal notation. To save the first 256 bytes of memory:

```
SAVE BASEPAGE.SAV,0,256
```

No wildcard characters are permitted in the filename or the extension.

1.2.24 SECURE (RESIDENT)

The security of a file is determined by its access code (see below). The code permits protection of certain files from deletion or renaming. The format of the command is:

```
SECURE [drive:] filename.ext,access-code
```

where the drive will default if omitted. For example,

```
SECURE DOS.SYS,0
```

removes any protection from the file DOS.SYS on drive zero.

```
SECURE INIT.CMD,2
```

protects the file INIT.CMD from deletion.

```
SECURE INIT.CMD,1
```

allows INIT.CMD to be renamed but not deleted.

No wildcard characters are permitted in either the filename or extension.

The access codes defined in CS-OS are:

0	not protected
1	cannot be deleted
2	cannot be renamed
3	cannot be deleted or renamed
4	is read only (cannot be written)
5	is read only and cannot be deleted
6	is read only and cannot be renamed
7	is read only, cannot be deleted, and cannot be renamed.

1.2.25 SET (RESIDENT)

This command allows the user to control characteristics of the console and line-printer devices, and to change the default drive and volume identifier used in disk and file operation. The format of the command is:

SET parameter = value

where 'parameter' is one of the following two-letter mnemonics:

TS -- tab settings -- defines the tab stops to be used for the console. The default value is 8 columns.

LD -- depth -- set the number of lines per page for the line printer. The default value is 60 lines.

LW -- width -- sets the number of characters per line for the line printer. The default value is 132 characters.

DD -- default drive -- sets the disk drive to be used if the drive specification is omitted. This value is initially set for drive 0. The default volume is updated to the diskette volume identifier in the given drive.

DV -- default volume -- sets the volume identifier to be used when it is omitted in a file specification. The volume identifier must be mounted in a diskette drive in order to execute the 'SET DV' command. The default is initially set to the volume identifier of the diskette in drive 0. The default drive is updated to the drive number that contains the diskette with the given volume identifier.

With the exception of DV, all parameters take a value that may be either decimal or hexadecimal. To view the current value of the parameter, type 'SET' followed by the parameter name and a carriage return. The following are valid commands:

SET DD = 2 (set default to drive 2, update default volume)

SET DV (displays default volume)

SET LW = 80 (set printer line width to 80 characters)

ERROR MESSAGES:

SYNTAX ERROR

BAD PARAMETER

VALUE TOO LARGE -- a number larger than 255 was used in the SET command

VOLUME IDENTIFIER NOT MOUNTED -- on SET DV = name, the volume identifier is not mounted in any drive

SET DEFAULT VOLUME ERROR -- an I/O error occurred during an attempt to set the default volume

DRIVE NOT FOUND -- on SET DD = n, the drive number was not valid

SET DEFAULT DISKETTE ERROR -- an I/O error occurred during an attempt to set the default disk.

1.2.26 SHOW (RESIDENT)

The SHOW command lists all logical unit numbers and the associated devices currently opened for the task specified in the command line. The format of the command is:

```
SHOW taskname
```

The system finds the task identifier, searches the list of Device Control Blocks (DCBs) for that task, and prints out the logical unit number and device name for each entry in the list. The following example shows the opened devices for the system task.

```
SHOW SYSTEM
   00246 #CON      00249 #SCRN0      00250 #CNSL0      00251 #SCRN1
   00252 #FD00     00253 #FD01       00254 #FD02       00255 #FD03
```

ERROR MESSAGES:

NO SUCH TASK

The printer spooler is a multitasking facility that allows the user to retain use of the system while ASCII files are copied to the printer. Requests to the spooler are queued on a first come first serve basis. Upon receipt of the first request to spool a file, a task is involved that removes files from the spool queue and copies them to the printer. Copying proceeds until the queue is exhausted or an error occurs. If an error occurs it is logged, and copying is terminated for the file being processed. The contents of the queue remain intact following an error and the spooling task can be restarted by issuing a spool command. The contents of the spool queue may be cleared by using the SPOOLC command (see separate listing of this command). Information on the spooler task, spooler queue and error log may be obtained by the SPOOLQ command (see separate listing of this command).

1.2.27 SPOOL (RESIDENT)

The spool command invokes the spooler task. If a filename accompanies the command, it will be verified, located, and added to the spooler file queue. If errors are encountered in the filename or if the file is not active (on the system), an error will result. The format of the command is:

SPOOL [<VOLUME>:<FILENAME>.<EXT>]<CR>

where

<VOLUME> is a six-character volume label
<FILENAME> is an eight-character filename
<EXT> is a three-character file extension
<CR> is a carriage return

Example:

SPOOL SYSVOL:TEXTFILE.TXT<CR>

1.2.28 SPOOLC (RESIDENT)

The SPOOLC command clears the spooler queue of all files but the one currently being copied. The format of the command is

SPOOLC<CR>

where

<CR> is a carriage return

Example:

SPOOLQ<CR>

For more information on spooler commands, see Spool.

1.2.29 SPOOLQ (RESIDENT)

The SPOOLQ command displays status information about the spooling task on the console screen. This information includes

SPOOLER TASK STATUS (ACTIVE/INACTIVE)
SPOOLER ERROR LOG
SPOOLER FILE QUEUE CONTENTS

The format of the command is:

SPOOLQ<CR>

where

<CR> is a carriage return

Example:

SPOOLQ<CR>

For more information on spooler commands, see Spool.

Notes on Using the Spooler The spooler task runs in operating system space and thus does not interfere with user task memory allocation. However, the spooler does use operating system resources, and care should be taken to avoid conflict with other user activities. Resources used by the spooling task include

#PR Printer (nonshareable device)
(once the spooler has acquired the printer other tasks may not use it until it has been released)

Disk files at time of submission and copy
(do not remove disk from drive until all spooled files on the disk have been copied to the printer)

The spooler task currently runs at priority level 10 (A).

1.2.30 SUBMIT (RESIDENT)

This command allows the use of a file containing CS-OS command lines as a source of console commands. The text lines in the file are executed as though they were typed at the console. The memory resident SUBMIT can invoke any other command under CS-OS. The file must be a text file (type 03). The format of the command is:

```
SUBMIT [drive:] filename.ext [,param,...,param]
```

where the drive defaults if omitted. All commands from the file will be echoed as they are read.

SUBMIT files use a special macro indicator -- the ampersand symbol (&). This macro indicator permits a SUBMIT file to use parameters from the SUBMIT command line as it executes. There can be up to 10 parameters in a given SUBMIT command line. All parameters must be set off with commas. Within the SUBMIT file, a parameter is called out by the macro indicator and a single decimal digit. Hence, &0 is the first parameter and &9 is the tenth parameter. The parameter from that position in the command line will be substituted in the SUBMIT file text in place of the macro indicator and parameter number. If the parameter does not exist, or the parameter number is bad, the parameter value will default to a carriage return.

No wildcard characters are permitted in the filename or extension of the SUBMIT file. We recommend that the filename extension "SUB" be used for SUBMIT files.

1.2.31 SUSPEND (RESIDENT)

The SUSPEND command causes a task to lose its place in the ready queue and to wait for a specified number of real-time clock "ticks." The task will not run during this time, even if its priority is higher than other tasks in the system. Once the specified time has passed, the task will resume its place in the ready queue. The format of the command is:

```
SUSPEND taskname [,ticks]
```

where 'taskname' is the name of some task currently in the system. The 'tick' parameter is optional; its default value is FFFFFFFF. (This is a delay measured in years, effectively "forever.") The tick count may be any numeric value. Some examples of SUSPEND are:

```
SUSPEND SYSTEM,200     (ten seconds)
SUSPEND TASK1          ("forever")
SUSPEND MYTASK, 8000   (about 27 minutes)
```

1.2.32 TASKS (RESIDENT)

The TASKS command displays the current state of the tasks in the system. The format of the command is:

```
TASKS
```

The tasks are listed in order of their PCB (Process Control Block) number, which will probably also correspond to the order in which they were generated. There will be a single line of console output for each task in the system (except "idle"). The task name, priority, status, and time of generation will be printed. The priority is output in hex notation. Recall that the status of a task in the system can take the following values:

```
01 - on the ready queue
02 - delayed or SUSPENDed
03 - waiting for a resource
```

An example of the TASKS command is:

```
TASKS

SYSTEM  40 01 13 NOV 82  07:30:05
TASK1   07 01 13 NOV 82  07:31:01
MYTASK  11 03 13 NOV 82  08:01:20
SPOOLER 50 02 13 NOV 82  08:15:01
```

1.2.33 TIME (RESIDENT)

This command displays the date and current time of day in the following format:

dwk dm mon yr hh:mm:ss

The day of the week (dwk) and month (mon) are represented by three-letter abbreviations. The other time elements are represented by numbers.

The TIME value is pre-set upon delivery. The system keeps proper elapsed time by use of a battery-operated clock. To display the current TIME, simply use the command name:

```
TIME
```

At noon on Christmas day 1982, this command would produce:

SAT 25 DEC 82 12:00:00

When files are created under CS-OS, the current time value will be included in their directory entry.

To change the TIME setting, use the command "TIME=" followed by the desired new TIME value in the format same used for displaying the time, as shown above. Any legal delimiter may be used (see Part 2, Section 2.4.9), with the addition of spaces used to delimit the date fields only. If an incorrect syntax or an invalid TIME parameter is encountered, the current TIME will be printed, along with the appropriate error message.

2.0 COMPUTER SYSTEM TEXT EDITOR

The Computer System text -- CS-Edit -- editor is a line-oriented editor for creating and updating text and program and data files. CS-Edit commands are divided into two groups: those that move data to and from temporary processor storage, and those that modify the data in temporary storage. The temporary storage used by the editor is called the "text buffer". Input commands bring a file or part of a file into the text buffer for editing. Edit commands modify or change the contents of the text buffer. Output commands write the corrected file from the text buffer to the output file after the contents have been updated. Input and output commands are described in Section 2.4. Edit commands are described in Section 2.5.

2.1 USING THE TEXT EDITOR

CS-Edit is invoked from the operating system (CS-OS) by typing EDIT on the keyboard. The operating system responds by loading and executing the editor, which then prompts for a filename:

FILE TO BE EDITED

When you respond with a filename (and extension) followed by a carriage return, CS-Edit does one of two things depending on whether the file named exists on disk or not. (A carriage return alone will return control to the operating system.)

1. If the file already exists, CS-Edit saves the original (input) file as a backup by changing its extension from "TXT" to "BAK". In addition, it creates a temporary (output) file for storing the results of the editing session. This file has the same name and contents as the original file, but its filename extension is TMP.

   NOTE: If a file with the same name as the temporary file already exists on disk, the disk file is automatically deleted by the editor.

2. When a new file is being created, (that is, when no file with the specified name and extension can be found on disk), CS-Edit responds with the following prompt:

CREATE NEW FILE?

This query serves to prevent the accidental creation of a file (as the result, for example, of mistyping a filename).

If you respond with a Y, the editor will open a new file having the name and extension specified and will use that file to store the results of the editing session.

If you respond with an N, the initial prompt (FILE TO BE EDITED) will be repeated.

When you finish editing (that is, when you type the ZIP command and return to the command mode of CS-Edit), the file you worked with (under the extension TMP) will be given the name and extension you specified.

The following table summarizes the manipulation of "old" and "new" files.

you specify	CS-Edit creates	after editing, you have
OLDFILE.TXT	OLDFILE.BAK and OLDFILE.TMP	OLDFILE.BAK and OLDFILE.TXT
NEWFILE.TXT	NEWFILE.TMP	NEWFILE.TXT
		NOTE: The above "TXT" files include the changes made during the editing session.

Additional Notes on Running the Editor:

It is important to remember that the automatic creation of the backup file (BAK) by the Editor will result in the deletion of any file having the same name.

When a new file is created by the Editor, it does not create any backup (BAK) file. At the end of the edit session there will only be one file that has the filename and extension that was specified in response to FILE TO BE EDITED.

2.2 EDITING MODES

2.2.1 COMMAND MODE

The initial prompt (FILE TO BE EDITED) in an editing session indicates that CS-Edit has entered the command mode. It is in this mode that most CS-Edit commands are executed. The prompt character for the command mode is ">". As each command is completed, CS-Edit responds with a ">", indicating that it is ready to accept another command. During execution of a command string (see below), the ">" is displayed only when the entire sequence of commands has been completed.

In command mode, any line that is not a legal command generates the following message on the display:

COMMAND ERROR

2.2.2 INPUT MODE

When you invoke an APPEND, INSERT, or REPLACE command, CS-Edit enters the input mode. The editor will now accept input lines of text and store them in the text buffer. When you type a line consisting only of the Esc character, the editor returns to command mode.

2.3 COMMAND STRUCTURE AND COMMAND STRINGS

A command directs CS-Edit to perform an operation. With two exceptions, a command consists of a single-letter code. (The exceptions have a two-letter code.) The command code can have an optional integer argument (n) following it, separated from the code by at least one blank space, or it can be followed by an optional string argument. The range of allowed values for the integer argument is 1 to 65,535 decimal.

Two or more commands can be entered on a single line for sequential execution. The resulting series of commands is called a command string. Each command must be delimited from the previous command by the Esc

character. The last command in the string must be followed by a carriage return.

If an error is encountered during the execution of a command string, the Editor returns to command mode and does not execute the error-causing command or any command that follows it.

2.4 THE ESCAPE AND BACKSPACE KEYS

The Esc key has two functions:

1. Pressing the Esc key while the editor is in input mode (and the input line consists of the Esc character only) returns it to command mode.

2. Esc is used to delimit commands in a command string.

You may depress the backspace key (<--) at any time to "rub out" a character. This action deletes the last typed character each time the backspace key is depressed.

2.5 EDITOR INPUT AND OUTPUT COMMANDS

The input and output commands described below can be used to read files or parts of files into the text buffer or to write files from the text buffer to an output file.

EOF

The (EOF) end-of-file command writes an EOF indicator to the output file.

Command format: E

Response: Control returns to the editor

Error messages: none

GET

The GET command reads n lines of text from the input file into the text buffer and inserts them after the last text line in the buffer. If an end-of-file (EOF) is reached, the message "EOF ON READ" is shown on the display to indicate that the entire file has been read.

Command format: G n

Response: >

Error messages: TEXT BUFFER IS FULL

HTAB

The HTAB command sets the tab stops to every n columns. If n is set to 0, no tabs are set. The tab character (09 hex) is stored in the text buffer but does not take up any "text space" when data is entered.

Command format: H n

Response: >

Error messages: none

WRITE

The WRITE command writes the first n lines of text from the text buffer to the specified output file. After the lines are written, they are deleted from the text buffer.

Command format: W n

Response: >

Error messages: DSK ERROR:nn

XMIT

The XMIT command writes the entire contents of the text buffer to the output file.

Command format: X

Response: >

Error messages: none

YANK

The YANK command reads lines from the input file into the text buffer until it is full. The lines read in are inserted after the last line in the text buffer.

Command format: Y

Response: >

Error messages: none

ZIP

The ZIP command is the standard way of completing the editing of a file. First the contents of the text buffer are written to the output file. The remaining lines of the input file are then read into the text buffer and written to the output file. Finally, an EOF is written and control passes back to the editor.

Command format: Z

Response: Control is returned to the editor.

Error messages: None

2.6 EDITING COMMANDS

The editing commands described below can be used to modify the contents of the text buffer.

APPEND

The APPEND command puts the editor in input mode and allows you to add text to the end of the text buffer from the console. The editor will accept input lines not exceeding 79 characters. When the Esc key is pressed, the editor returns to command mode.

Command format:	A
Response:	A carriage return; the editor is waiting for input
Error messages:	TEXT BUFFER IS FULL
	LINE LENGTH EXCEEDED

BOTTOM

This command moves the internal pointer to the last line of text in the buffer.

Command format:	B
Response:	>
Error messages:	None.
Status message:	BOTTOM OF BUFFER

CHANGE

This command can be used to change one character string to another. If the character "G" (for "global") is present, every occurrence of "oldstring" is replaced with "newstring". If the "G" is not present, only the first occurrence of "oldstring" is replaced with "newstring".

The slash character (/) is the delimiter or "quote" character. (If "/" appears in the text you wish to change, the quote character may be any ASCII character not appearing in "oldstring" or "newstring"). The two character strings may be of different lengths.

Command format: C /oldstring/newstring/[G]

Response: changed line(s)

Error messages: SYNTAX ERROR

STRING NOT FOUND

For the following examples, the text is assumed to be in the text buffer:

```
A computer provides a means of
managing a collection of data
```

If the command

```
C /computer/Computer System/
```

were entered, the resulting text would be

```
A Computer System provides a means of
managing a collection of data
```

If the command

```
C / a / the /G
```

were entered, the resulting text would be

```
A Computer System provides the means of
managing the collection of data
```

Note that both occurrences of "a" were altered (because G was specified) and that the upper case A was not altered. When changing single characters or small character strings, be sure that you include spaces in the command; otherwise strings within words may be altered. For example, C/a/the/G would yield:

```
A Computer System provides the methens of
mthentheging the collection of dthetthe
```

DELETE

This command deletes n lines of text from the text buffer starting with the current line. If n is left unspecified, only the current line is deleted.

Command format: D n

Response: >

Error messages: TEXT BUFFER IS EMPTY

FIND

This command finds and moves the first line containing the specified character string to the cursor line. The slash character (/) is the standard delimiter. (If "/" is part of the string you wish to find, the delimiter may be any ASCII character not appearing in the specified character string).

Command format: F /string/

Response: displays first line containing the specified character string

Error messages: SYNTAX ERROR

STRING NOT FOUND

As with the CHANGE command, be sure to include blank spaces where necessary. "F /thing/", for example, would locate "something", "anything", "things", or "thing", whichever appears first.

INSERT

This command puts the editor in insert mode and allows you to insert a line or lines into the text buffer preceeding the line at which the internal pointer is currently positioned. The buffer size is 16,384 bytes. The maximum line length is 79 characters of data and a carriage return. If the insertion exceeds this maximum, the line will be truncated to 79 characters. The editor returns to command mode when the Esc key is pressed.

Command format: I n (default = 1)

Response: carriage return, linefeed

Error messages: TEXT BUFFER IS FULL

LINE LENGTH EXCEEDED

LIST

This command lists the entire contents of the text buffer on the system console. (To display individual lines of text, refer to the PRINT command).

Command format: L

Response: displays lines from the text buffer

Error messages: none

MOVE-FROM/MOVE-TO/MOVE-LINES

The MOVE-FROM and MOVE-TO commands are used in conjunction with the MOVE-LINES command to reposition material in a test file.

MOVE-FROM saves the location of the correct line. When the ML command is executed this line will be the first line of the text material moved.

Command format: MF

Response: >

Error messages: none

The MOVE-TO comand saves the location of the current line. When the ML command is executed, this line will be used as the insert location (i.e., data will be positioned before this line).

Command format: MT

Response: >

Error messages: none

The MOVE LINES command moves n lines of text from the line pointed to by the MF command and inserts them before the line pointed to by the MT command.

Command format: ML n

Response: >

Error messages: COMMAND ERROR

MOVE FEWER LINES

MT OR MF POINTER NOT SET

EXAMPLE:

For the following example, the text below represents the total contents of the text buffer and has been listed (see LIST command).

```
a aaaa aaaaaa aa aaaaa aaaa aa          (line 1(MF)
bbbbbb bbb bbbb bbbbb bb bbb b          (line 2)
ccccc cc ccccc ccccc ccccccccc          (line 3)
dddd ddddddddd dd ddddd dddd dd         (line 4(MT)
```

Any line of listed text can be positioned at the current-line locaton by means of the NEXT or UP command. If line 1 is brought to the current line location and the MF command entered, line 1 will be the "move from" location. If line 4 is moved to the current-line location and the MT command entered, it becomes the "move to" location. (These locations are marked only internally. The "MF" and "MT" markers shown in the example do not appear on the display itself).

The command

```
ML 2
```

would move two lines (starting at line 1) to the location preceeding line 4, producing the following text.

```
ccccc cc ccccc ccccc ccccccccc          (line 3)
a aaaa aaaaaa aa aaaaa aaaa aa          (line 1)
bbbbbb bbb bbbb bbbbb bb bbb b          (line 2)
dddd ddddddddd dd ddddd dddd dd         (line 4)
```

NEXT

This command moves the location of the current line forward, in effect moving the text upward on the display. An integer value (n) is used to specify the number of lines to be skipped over. If n is unspecified, it is assumed to be one (1) and the line moved to the current-line location will be the next line. (See the UP command.)

Command format:	N n
Response:	displays nth line after the current line
Error messages:	TEXT BUFFER IS EMPTY
	BOTTOM OF BUFFER

If, for example, you were positioned at the top of the text buffer and wished to see the text on line 10, you would enter N9.

PRINT

This command displays n lines from the text buffer on the CRT, starting with the current-line. The current-line is not changed by this command. If n is unspecified, it is assumed to be one (1). (To display the entire contents of the text buffer, refer to the LIST command.)

Command format:	P n
Response:	displays line(s)
Error messages:	TEXT BUFFER IS EMPTY

If, for example, you wished to see lines 10 through 29 of a file in the text buffer and you were positioned at the top of the text buffer, you would enter N9 to move down to line 10 and P10 to display ten lines of text beginning with line 10.

QUIT

QUIT terminates editing and returns the user to the operating system command mode.

Command format:	Q

Response:	Control is returned to the operating system
Error messages:	none

REPLACE

This command puts the editor in input mode and causes the current line to be replaced by a line or lines of text that you enter. You are, in effect, "typing over" the line. The editor will return to the command mode when the Esc key is pressed.

Command format:	R
Response:	A carriage return; the editor is waiting for input
Error messages:	TEXT BUFFER IS EMPTY LINE BUFFER EXCEEDED

SCRATCH

This command erases the contents of the text buffer and removes all internal pointers, preparing the buffer for new work.

Command format:	S
Response:	>
Error messages:	none

TOP

This command positions the first line in the text buffer at the current-line location.

Command format:	TOP
Response:	displays the first line of the text buffer
Error messages:	TOP OF TEXT BUFFER

UP

This command moves the current line backward from its present position in the text buffer. An integer value (n) is used to specify the number of lines to be skipped over. If n is unspecified, it is assumed to be one and the preceding line is moved to the current-line location. (See the NEXT command.)

Command format:	U n
Response:	displays nth line after current line
Error messages:	TEXT BUFFER IS EMPTY

2.7 SUMMARY OF COMMANDS

A	Append text to the end of the text buffer.
B	Move cursor to the end of the text buffer.
C /oldstring/newstring/[G]	Change the occurrence(s) of "oldstring" to "newstring"
D n	Delete n line(s).
E	Write out an EOF.
F /string/	Find the first occurrence of "string" in the text buffer.
G n	Get n lines of text from the input file.
H n	Set the tab stops to every n columns.
I n	Insert line(s) of text. (Input Mode)
L	List the contents of the text buffer.
MF	Save the move-from location.

MT	Save the move-to location.
ML n	Move n lines of text in the text buffer to a specified location.
N n	Skip ahead n line(s).
P n	Display n line(s) of the text buffer.
Q	Return to the system.
R	Replace the current line with line(s) of text.
S	Clear the text buffer.
T	Make the first line in the text buffer the current line.
U n	Skip back n line(s).
W n	Write n line(s) of text from the text buffer to the output file.
X	Write the entire contents of the text buffer to the output file.
Y	Read lines of text from the input file into the text buffer until it is full.
Z	Write the contents of the text buffer to the output file; write the remaining lines of text from the input device to the output device; write an EOF mark to the output device.

3.0 COMPUTER SYSTEM MACRO ASSEMBLER

3.1 INTRODUCTION

The Computer System macro assembler described in this manual is a two-pass macro relocating assembler that runs on the Computer System processor with the CS-OS.

Pass 1 of the assembler defines every user symbol. The symbol table built in pass 1 generates code in pass 2. Errors are trapped in both passes, and listings and object code are output from pass 2.

The format of the object code output from the assembler is that defined for the Computer System LINK and LOCATE programs. The operating instructions for LINK and LOCATE are included in Chapter 4 of this manual.

3.2 OPERATING INSTRUCTIONS

The assembler is executed as a command under CS-OS. To invoke the Computer System macro assembler, enter the following command on the keyboard:

```
ASMB FILENAME.EXT,OPTIONS,DEVICE
```

where

filename.ext	specifies the name and extension of the file to be assembled (e.g., 1:GRTEST01.SMP).
options	are any combination of the letters L, M, O, S, and X (e.g., LSO).
device	is the name of the listing/error-message device and can be any of the following:

```
CON
LPT
DSK 0
DSK 1
DSK 2
DSK 3
```

For example:

ASMB_1:GRTEST01.SMP,LSO,DSK

The available 'options' are:

"L" -- Listing produced on "device." If the device is DSK, then a file is produced named filename.LST.

"M" -- Macro expansions listed in the listing if the "L" option was also selected.

"O" -- Object code produced on disk named filename.REL.

"S" -- Sorted symbol table listing produced on the same device as the listing output.

"X" -- Cross-reference file produced on disk named filename.CRF. The XREF utility produces the cross-reference listing (see below).

Note: When specifying DSK as the device, the drive number used is the drive number of filename.ext. If a different drive is desired, specify DSK n. Specifying LPT sends output to #PR.

EXAMPLES:

1. ASMB 1:TEST.ASM,LSO,LPT
2. ASMB 1:TEST.ASM,LSO,DSK
3. ASMB 1:TEST.ASM,,CON

Example 1: The listing and symbol table go to the LPT device, and the object code to 1:TEST.REL.

Example 2: The listing and symbol table go to 1:TEST.LST, and the object code to 1:TEST.REL.

Exampl #3: No options are set, and any error messages are sent to the console device.

3.2.1 CROSS-REFERENCING PROGRAM LABELS WITH XREF

Note: The XREF function cannot be used if the Computer System does not have a line printer attached.

It is often useful to have a sorted list of the labels in a program, along with all the references made to those labels.

To generate a cross-reference listing, you must first run the assembler with the "X" option, which produces an output file containing the unsorted symbols. This file will be the same as the input file except that its filename suffix is "CRF".

The next step is to execute the XREF utility program as follows:

```
XREF filename.CRF
```

For example, to produce a cross-reference list for the SAMPLE program, you would type:

```
XREF SAMPLE.CRF
```

3.3 INSTRUCTION FORMATS

A source language statement consists of a label, an operation code, an operand, and comments. The label is used when needed as a reference by other statements. The operation code may be a mnemonic machine operation, an assembly directing pseudo-op, or a macro call label. An operand may be an expression consisting of an alphanumeric symbol, a number, a special character, a string of one to four characters, or any of these combined by arithmetic operations; In certain instances there may be no operand at all. The comments may also be left out. The fields in a source statement are separated by at least one space character.

3.3.1 STATEMENT CHARACTERISTICS

The fields of every source-program line appear in the following order:

```
Label     Opcode     Operand(s)  Comments
```

3.3.2 FIELD DELIMITERS

One or more spaces separate the fields in a statement. An end-of-statement mark (carriage return) terminates the entire statement. A single space following an end-of-statement mark from the previous statement is the null field indicator of the label field.

3.3.3 CHARACTER SET

The characters recognized by the assembler are:

```
A through Z
0 through 9
* (asterisk)    comment, multiply, or loc. counter
+ (plus)        addition
- (minus)       subtraction
/ (slash)       division
$ (dollar)      hexadecimal indicator
()(parentheses)
' (apostrophe) quoted-string delimiter
, (comma)       operand delimiter
# (pounds)      immediate-mode indicator
& (ampersand)  macro-parameter indicator
. (period)      alphabetic character
  (space)
```

Any other ASCII characters may appear in the comments field. The letters A through Z, the numbers 0 through 9, and the period (.) may be used in an alphanumeric symbol. In the first position of the label field, an asterisk indicates a comment line; in the first position of an operand, it stands for the location-counter value; otherwise, it is the multiplication operator. The plus, minus, slash, and asterisk are used as operators in arithmetic expressions.

The pounds sign indicates the immediate addressing mode; the dollar sign ($) indicates hexadecimal numbers; the apostrophe indicates ASCII strings; the ampersand indicates substitutable parameters in macro definitions; the comma separates operands. Parentheses are used in the syntax of some of the addressing modes.

Spaces separate fields of a statement and may also be used to format the output listing.

3.3.4 STATEMENT LENGTH

A statement may be up to 80 characters in length.

3.3.5 LABEL FIELD

The label field serves to identify the statement and may be used as a reference by other statements in the program.

The field starts immediately following an end-of-statement mark and is terminated by a space. A space in the first position indicates that the statement is unlabeled.

3.3.6 LABEL SYMBOL

A label is composed of from one to eight characters. The first one must be an alpha character. The remaining characters must be alphanumeric characters. If the label is composed of more than eight characters the assembler truncates the symbol to eight characters.

An asterisk in position one indicates that the entire statement is a comment. An asterisk in any position of the label field other than the first position is illegal.

3.3.7 OPCODE FIELD

The operation code defines an operation to be performed by the computer or by the assembler (opcodes, pseudo-ops, or macro calls). The opcode field follows the label field and is separated from it by at least one space. If there is no label, the opcode/pseudo-op/macro call may begin anywhere after position one. The opcode field is terminated by a space immediately following the mnemonic.

3.3.8 OPERAND FIELD

The meaning and format of the operand field is dependent on the type of operation code used in the source statement.

This field follows the opcode field and is separated from it by at least one space. A set of reserved labels identify the processor registers. The set of 8-bit data registers is:

D0.B, D1.B, ... D7.B

The set of 16-bit data registers is:

D0.W, D1.W, ... D7.W

The set of 32-bit data registers is:

D0.L, D1.L, ... D7.L

which may be abbreviated to:

D0, D1, ... D7

The set of 32-bit address registers is:

A0, A1, ... A7

The set of special registers is:

PC	(program counter)
SP	(stack pointer = A7)
USP	(user-mode stack pointer)
SR	(16-bit status register)
CCR	(8-bit condition-code register)

The operand field is terminated by a blank space or a carriage return. Any data between a blank space and a carriage return will be interpreted as a comment.

3.3.9 SYMBOLIC TERMS

A symbolic term follows the same rules as those that govern the formation of labels. A symbol used in the operand field must be a symbol that is defined elsewhere in the program. An asterisk can refer to the value of the location counter at the time the source statement is encountered.

3.3.10 NUMERIC TERMS

A numeric term may be either decimal or hexadecimal. A decimal number is represented by one to ten decimal digits and may range from 0 to 2147483648. A hexadecimal number is indicated by one to eight hexadecimal digits within the range 0 to FFFFFFFF and is preceded by a dollar sign ($).

3.3.11 EXPRESSION OPERATORS

The asterisk (location counter), symbols, and numbers may be joined by the four arithmetic operators (+ - * /) to form arithmetic expressions. The assembler evaluates expressions from left to right without regard to precedence or operator hierarchy. A fractional result, if obtained during the evaluation of an expression, is truncated to an integer value. The + and - symbols may be used as unary operators. All expressions are evaluated using 32-bit unsigned arithmetic. No blanks are permitted in an expression.

Quoted character strings containing one to four characters may also be used in expressions. Their value is formed from the 7-bit ASCII code, which represents the characters. The characters are right-justified in the 32-bit value and any unused high-order bits are cleared. The quote character (apostrophe) may not appear in the string.

Some examples of valid expressions are:

```
ALPHA+1        $FF        'AB'-3        */256        $123456

4096*16/2      'ZZZZ'-'AAAA'           BETA-GAMMA/DELTA

-99999         -10+'stuff'             *+22         *-LABEL
```

3.3.12 MACRO CALL ARGUMENT LISTS

Arguments are passed to macros by placing them in the operand field; individual arguments are separated by commas. The actual arguments are substituted as character strings into the positions of the corresponding dummy arguments in the macro definition. If comments are to be included in the statement, a comma must follow the last argument. A macro may have up to 10 arguments, numbered 0 to 9.

3.3.13 EVALUATION OF SYMBOLS AND EXPRESSIONS

Because of the 2-pass nature of the assembler, only one level of forward referencing is legal when using symbols and expressions in the operand field of source statements. All expressions are evaluated using 32-bit unsigned arithmetic, ignoring any overflows or underflows. Instructions using byte operands use the low-order byte of the expression value. Count fields used in the COMMON and DS pseudo-ops are 16-bit only. The SET and EQU pseudo-ops generate 32-bit values that are matched with their symbolic names. The location counter is an unsigned 32-bit value.

3.3.14 COMMENT FIELD

A comment may be included in a source statement as long as it is separated by at least one space from the operand field. Any printable character may appear in a comment.

3.4 ADDRESSING MODES

There are eleven basic types of addressing permitted by the Computer System processor. Some of these have multiple forms, and some instructions take special addressing formats not permitted in other cases. The CS-OS assembler is quite particular in that it requires the operands to match the types allowed by the operation mnemonic. CS-OS will not reform the mnemonic in response to the operands found -- it will simply flag such operands as syntax errors. Hence, the assembled operation will always be the one indicated by the mnemonic.

For more detailed information on the addressing modes allowed by particular instructions you may want to obtain a copy of the 68000 16-Bit Microprocessor User's Manual.

3.4.1 DATA REGISTER DIRECT ADDRESSING

This addressing mode uses one of the data registers as the source (or destination) operand. Some examples are:

```
MOVE.W D0,D1
ADDI.L #$123,D4
```

3.4.2 ADDRESS REGISTER DIRECT ADDRESSING

This addressing mode uses one of the address registers as the source (or destination) operand. Some examples are:

```
MOVEA.L TABLE,A2
MOVE.L D0,A3
```

3.4.3 ADDRESS REGISTER INDIRECT ADDRESSING

This addressing mode uses one of the address registers as the address of the source (or destination) operand. A 16-bit signed offset in the instruction is added to the contents of the address register to form the effective address used. The offset is formed from an expression included in the operand field of the instruction. This offset cannot be Relocatable or Common, nor can it exceed the range allowed by a signed 16-bit value. Some examples are:

```
MOVE.B 10(A3),D0
LEA (A4),A2 Note:  expression omitted = 0
```

If the expression is omitted, the value "zero" will be used.

3.4.4 ADDRESS REGISTER PREDECREMENT ADDRESSING

This addressing mode uses the contents of an address register as the address of a source (or destination) operand. It is like address register indirect addressing, except that no expression is allowed. The value in the address register is decremented before the effective address is computed. Some examples are:

```
CMP.W -(A4),D0
MOVE.L D0,-(A6)
```

3.4.5 ADDRESS REGISTER POSTINCREMENT ADDRESSING

This addressing mode uses the contents of an address register as the address of a source (or destination) operand, then increments the address

register value. Some examples are:

```
CMPA.L (A4)+,A2
MOVE.W (A3)+,(A5)+
```

3.4.6 ADDRESS REGISTER WITH INDEX ADDRESSING

This addressing mode uses the contents of two registers to generate the effective address. The first register specified must be an address register. The index register may be either a data or address register. If a data register is used as an index, it may be considered as either a signed-word or long-word value. The assembler assumes that the index is a long-word unless the index register is specifically declared as a word register. An 8-bit displacement expression must also be included in the address. This expression must not be Relocating or Common, and it must not exceed the range of a signed byte. Some examples are:

```
ADD.W C1(A1,D1),D2
MOVE.L $12(A2,D3.W),0(A3,D4)
```

where A1, A2, and A3 are address registers; D1, D3, and D4 are index registers; and C1, $12, and 0 are 8-bit displacement expressions.

3.4.7 ABSOLUTE SHORT ADDRESSING

This mode of addressing uses a 16-bit signed value in the instruction as the source (or destination) address. The value comes from an expression in the operand. The expression must not be Relocating or Common, and it must not exceed the range of a signed word. Some examples are:

```
JSR $0400
JMP *-LABEL (Note: LABEL is relocating)
```

3.4.8 ABSOLUTE LONG ADDRESSING

This mode of addressing uses a 32-bit value in the instruction as the source (or destination) address. The value comes from an expression in the operand. Relocation and Common expressions are permitted. Some examples are:

```
JSR LABEL
MOVE.W *+2,LABEL-4
```

3.4.9 PROGRAM-COUNTER RELATIVE ADDRESSING

This addressing mode uses the program counter register (PC) in a form similar to that of address register indirect addressing described above. The offset value is computed as the displacement between the expression value and the current PC value. This mode is useful for generating position-independent code. Some examples are:

```
LEA 2(PC),A4
JMP TAG(PC)
```

Note: The offset expression may contain Relocating symbols. The resulting effective address is not relocating.

Another form of program-counter relative addressing is used in the branch instructions. Here the assembler computes a displacement value that is incorporated in the instruction itself. Some examples are:

```
BNE.S LABEL
BSR   *+5
```

3.4.10 PROGRAM-COUNTER-WITH-INDEX ADDRESSING

This addressing mode parallels the address-register-with-index addressing except that the program-counter register (PC) is used as the address register, and the 8-bit displacement is computed as the difference between the expression value and the present location. Note that, as in program counter indirect addressing, the expression may contain Relocating symbols. Some examples are:

```
MOVE.L ARG(PC,D1),D3
JMP TABLE(PC,D2.W)
```

3.4.11 IMMEDIATE ADDRESSING

In this mode the instruction itself contains the necessary data. This mode is always indicated by the pounds sign (#) preceding the immediate expression. The restrictions on the magnitude of the immediate value depend on the instruction in use. Only those instructions requiring 32-bit immediate data may have Relocating or Common immediate values. Some examples are:

```
MOVEQ #'C',D2
ADDI.L #$123456,2(A3)
```

3.4.12 SPECIAL-REGISTER ADDRESSING

This addressing mode is used only with certain instructions. The source (or destination) operand is one of the special registers. The MOVE instruction can use the SR, CCR, and USP registers. Some examples are:

```
MOVE.W D2,SR
MOVE.B CCR,(A2)+
MOVE.L USP,A3
MOVE.L A4,USP
```

The ANDI, ORI, and EORI instructions may use the SR and CCR registers. Some examples are:

```
ANDI.B #1,CCR
ORI.W #$1F,SR
```

3.4.13 REGISTER-LIST ADDRESSING

This addressing mode is only permitted in the MOVEM instruction. A list of registers (data or address) is specified. The assembler builds a bit-map that indicates which registers are to be operated upon. Two types of syntax are allowed in a register list. First, specific registers may be specified, separated by slashes. Second, ranges of registers may be specified by giving the first register name, a minus sign, and the last register name. The ordering assumed is D0, D1, ... D7, A0, A1, ... A7. Hence, D0-A7 implies all registers. Both forms may be combined in a single register list as follows:

```
MOVEM.L (A7)+,D0/D1/D5-A0  regs. D0,D1,D5,D6,D7,A0
MOVEM.W A5-D2/D7,(A6)      regs. A5,A6,A7,D1,D2,D7
```

3.5 INSTRUCTION SET SUMMARY

This section summarizes the mnemonics provided in this assembler and the syntax allowed for each operation. Some abbreviations used in this section are:

```
An........an address register
Dn........a data register
Rn........a register (data or address)
<data>....an expression
<ea>......an address operand (effective address)
          (see Section 3.4, "Addressing Modes")
cc........one of the set of conditionals

          CC-carry clear
          CS-carry set
          EQ-equal
          F -always false
          GE-greater than or equal (signed)
          GT-greater than (signed)
          HI-greater than (unsigned)
          LE-less than or equal (signed)
          LS-less than or equal (unsigned)
          LT-less than (signed)
          MI-less than 0
          NE-not equal
          PL-greater than or equal to 0
          T -always true
          VC-no overflow
          VS-overflow

d(Ay).....Address Register Indirect addressing
.s .......Size specification (.B, .W, or .L)
```

MNEMONIC	OPERATION	ASSEMBLER SYNTAX
ABCD	Add decimal extend	ABCD Dy,Dx ABCD -(Ay),-(Ax)
ADD.B (.W, .L)	Add binary	ADD.s <ea>,Dn ADD.s Dn,<ea>
ADDA.W (.L)	Add to address reg.	ADD.s <ea>,An
ADDI.B (.W, .L)	Add immediate value	ADD.s #<data>,<ea>
ADDQ.B (.W, .L)	Add value quick	ADDQ.s #<data>,<ea>
ADDX.B (.W, .L)	Add with extend	ADDX.s Dy,Dx ADDX.s -(Ay),-(Ax)
AND.B (.W,.L)	Logical AND	AND.s <ea>,Dn AND.s Dn,<ea>
ANDI.B (.W, .L)	Logical AND Immed.	ANDI.s #<data>,<ea> ANDI.B #<data>,CCR ANDI.W #<data>,SR
ASL.B (.W, .L) ASR.B (.W, .L)	Arithmetic shift	ASd.s Dx,Dy ASd.s #<data>,Dy ASd.W <ea>
Bcc (.S)	Conditional Branch	Bcc <label>
BCHG	Test bit and change	BCHG Dn,<ea> BCHG #<data>,<ea>
BCLR	Test a bit and clear	BCLR Dn,<ea> BCLR #<data>,<ea>
BRA (.S)	Branch always	BRA <label>
BSET	Test a bit and set	BSET Dn,<ea> BSET #<data>,<ea>
BSR (.S)	Branch to subroutine	BSR <label>
BTST	Test a bit	BTST Dn,<ea> BTST #<data>,<ea>
CHK	Check bounds	CHK <ea>,Dn

CLR.B (.W, .L)	Clear operand	CLR.s <ea>
CMP.B (.W, .L)	Compare	CMP.s <ea>,Dn
CMPA.W (.L)	Compare Address	CMPA.s <ea>,An
CMPI.B (.W, .L)	Compare Immediate	CMPI.s #<data>,<ea>
CMPM.B (.W, .L)	Compare Memory	CMPM.s (ay)+,(Ax)+
DBcc (Note: DBRA accepted for "test false" = DBF)	Test, decr. and branch	DBcc Dn,<label>
DIVS	Signed division	DIVS <ea>,Dn
DIVU	Unsigned division	DIVU <ea>,Dn
EOR.B (.W, .L)	Exclusive OR	EOR.s Dn,<ea>
EORI.B (.W, .L)	Exclusive OR Immed.	EORI.s #<data>,<ea> EORI.B #<data>,CCR EORI.W #<data>,SR
EXG	Exchange Regs.	EXG Rx,Ry
EXT.W (.L)	Extend Sign	EXT.s Dn
JMP	Jump	JMP <ea>
JSR	Jump to Subroutine	JSR <ea>
LEA	Load Effective Addr.	LEA <ea>,An
LINK	Link and Allocate	LINK An,#<data>
LSL.B (.W, .L) LSR.B (.W, .L)	Logical Shift	LSd.s Dy,Dx LSd.s #<data>,Dy LSd.W <ea>
MOVE.B (.W, .L)	Move Data	MOVE.s <ea>,<ea> MOVE.B <ea>,CCR MOVE.W <ea>,SR MOVE.W SR,<ea> MOVE.L USP,An MOVE.L An,USP
MOVEA.W (.L)	Move to Address	MOVEA.s <ea>,An

MOVEM.W (.L)	Move Multiple Regs.	MOVEM.s <reg-list>,<ea> MOVEM.s <ea>,<reg-list>
MOVEP.W (.L)	Move Peripheral Data	MOVE.s d(Ay),Dx MOVE.s Dx,d(Ay)
MOVEQ	Move Data Quick	MOVEQ #<data>,Dn
MULS	Signed Multiplication	MULS <ea>,Dn
MULU	Unsigned Multiply	MULU <ea>,Dn
NBCD	Negate Decimal extend	NBCD <ea>
NEG.B (.W, .L)	Negate	NEG.s <ea>
NEGX.B (.W, .L)	Negate with extend	NEGX.s <ea>
NOP	No operation	NOP
NOT.B (.W, .L)	Logical Inverse	NOT.s <ea>
OR.B (.W, .L)	Logical OR	OR.s <ea>,Dn OR.s Dn,<ea>
ORI.B (.W, .L)	Logical OR Immed.	ORI.s #<data>,<ea> ORI.B #<data>,CCR ORI.W #<data>,SR
PEA	Push Effective Address	PEA <ea>
RESET	Reset Ext. Devices	RESET
ROL.B (.W, .L) ROR.B (.W, .L)	Rotate	ROd.s Dx,Dy ROd.s #<data>,Dy ROd.W <ea>
ROXL.B (.W, .L) ROXR.B (.W, .L)	Rotate with Extend	ROXd.s Dx,Dy ROXd.s #<data>,Dy ROXd.W <ea>
RTE	Return from Exception	RTE
RTR	Return and Restore CC	RTR
RTS	Return from Subroutine	RTS
SBCD	Subtract decimal	SBCD Dy,Dx

		SBCD -(Ay),-(Ax)
Scc	Set by Cond. Codes	Scc <ea>
STOP	Load SR and Stop	STOP #<data>
SUB.B (.W, .L)	Subtract Binary	SUB.s <ea>,Dn SUB.s Dn,<ea>
SUBA.W (.L)	Subtract Address	SUBA.s <ea>,An
SUBI.B (.W, .L)	Subtract Immediate	SUBI.s #<data>,<ea>
SUBQ.B (.W, .L)	Subtract Data Quick	SUBQ.s #<data>,<ea>
SUBX.B (.W, .L)	Subtract with Extend	SUBX.s Dy,Dx SUBX.s -(Ay),-(Ax)
SWAP	Swap Register Halves	SWAP Dn
TAS	Test and Set	TAS <ea>
TRAP	Trap	TRAP #<data>
TRAPV	Trap on Overflow	TRAPV
TST.B (.W, .L)	Test an Operand	TST.s <ea>
UNLK	Unlink	UNLK An

3.6 PSEUDO INSTRUCTIONS

COMMON

This pseudo-op is used to reserve or declare an area in Common for interprogram data communication. The syntax is:

```
COMMON   SYMBOL,operand
```

The operand may be a number, symbol, or expression. The operand is evaluated, and the value obtained is used to reserve or declare that amount (in bytes) in the Common area. The low-order 16 bits of the operand value are used. The COMMON pseudo-op must not be labeled.

DC.B

This pseudo-op forms constant bytes of object code. The operands may be expressions or quoted character strings. If the operand is an expression, then the low-order byte of the expression's value is used. The expression must not be Relocating or Common. If the operand is a quoted string, then a byte will be formed for each character in the string. A quote character (') may be formed by using two quotes in a row. The bytes formed by DC.B need not lie on an even-word boundary. The DC.B pseudo-op may have multiple operands, separated by commas. The listing will show only the first expression value or the first four characters of the quoted string. Some examples of this pseudo-op are:

```
DC.B 1,2,3,4
DC.B 'This is a character string'
DC.B 22+LAB1, 'I don't believe this'
```

The DC.B pseudo-op may be labeled.

DC.L

This pseudo-op forms 32-bit long-words of constant data. The assembler forces these long-words to lie on even-byte boundaries by outputting null bytes as necessary. Like DC.B, the DC.L pseudo-op may take either expressions or quoted strings as operands. Quoted strings will be padded with nulls at the right to make the boundaries correct. The full 32-bit values of expressions will be used. These expressions may be Relocating or Common. The listing will show the first expression value or the first four characters of the first quoted string. The DC.L pseudo-op may be labeled. Some examples of DC.L are:

```
LABEL1  DC.L *-LABEL1,LABEL1-2,$12345
        DC.L 'This string has lots of character'
```

DC.W

This pseudo-op, like DC.B and DC.L, is used to form constant data. The DC.W pseudo-op forms words of data, using either expressions or quoted strings as operands. The assembler forces the words to lie on even-byte boundaries. Null bytes are used to fill out character strings. The low-order 16-bit values of expressions are used. Expressions must not be Relocating or Common. The listing will show the first expression value or the first four characters of the first character string. Some examples of DC.W are:

```
LABEL2 DC.W $123,$321,'ABCD'
LABEL3 DC.W LABEL3-LABEL2,'example of DC.W '
```

DS.B

This pseudo-op is used to reserve a block of memory whose length in bytes is the value of the operand. The syntax is:

```
[Label]    DS    operand
```

The operand may be a number, symbol, or expression. The DS.B pseudo-op may be labeled and the block of memory reserved is cleared to zeroes. Symbols used in the operand field must have been previously defined in the program. Only the low-order 16 bits of the operand value are used.

DS.L

This pseudo-op is like DS.B except that memory is allocated in 32-bit long-word units. The location counter is adjusted to force even-byte boundary alignment.

DS.W

This pseudo-op is like DS.B except that memory is allocated in word (16-bit) units. The location counter is adjusted to force even-byte boundary alignment.

END

This pseudo-op terminates a program. It marks the physical end of the source language program. The last statement of a program must be an END statement. The END statement must not be written with a label; it generates no object code and has no operand.

ENDIF

This pseudo-op is used as a terminator to an IF pseudo-op. It must be unlabeled and have no operand.

ENDM

This pseudo-op is used to indicate the end of a macro definition. It must not have a label or an operand.

ENTRY

This pseudo-op declares a symbol that may be referenced by separately assembled programs. The syntax is:

```
ENTRY   SYMBOL
```

This pseudo-op must not be labeled and the operand field must contain a symbol that is defined elsewhere in the program.

EQU

This pseudo-op assigns to a symbol a value other than the value normally assigned by the program-location counter. The syntax is:

```
Label    EQU    operand
```

The EQU statement must be labeled. The operand field may contain a number, symbol, or expression. Symbols appearing in the operand field must be previously defined in the source program. The full 32-bit operand value is used. This pseudo-op generates no object code.

Note: Once a symbol has been defined with an EQU statement, it cannot be redefined.

EXTERNAL

This pseudo-op declares a symbol that may be referenced by the program but is defined in some other program. The syntax is:

```
EXTERNAL   SYMBOL
```

This pseudo-op must not be labeled. SYMBOL in the operand field must be declared by an ENTRY pseudo-op in the program in which it is defined.

IF

This pseudo-op causes the assembler to process the following code normally if the value of the operand is not zero but to ignore all source statements until a matching ENDIF statement is encountered if the value of the operand is zero. The syntax is:

```
IF      operand
```

The operand may be a number, symbol, or expression. The IF pseudo-op must not be labeled and must have an operand.

INCLUDE

This pseudo-op is used to specify source files that are to be included in the input source stream to the assembler. The syntax is:

```
INCLUDE filename.ext
```

The filename can be any source file present on the same disk as the input source file.

MACRO

This pseudo-op is used in the definition of a macro. All statements following the MACRO pseudo-op up to the next ENDM pseudo-op are stored in the Macro Table as a macro definition. The syntax is:

```
Label        MACRO [C]
```

MACRO statements require labels. The label is the symbol (name) by which a macro is expanded or called. The operand field may contain a "C," which specifies whether or not comment lines in the macro definition are to be stored in the Macro Table. If the "C" is present then the comment lines are stored and thus expanded with the rest of the lines in the macro definition when it is called. By omitting the "C," the user can lower the main-memory requirements needed to store the macro definition.

Note: Macro definitions may not be nested but may contain calls to other macros.

NAME

This pseudo-op names the program. The syntax is:

```
NAME  SYMBOL
```

SYMBOL in the operand field is passed to the Linking Loader as an Entry point. It must not be used as a label in the program. A NAME pseudo-op must be included in each program as the first statement.

NOPRINT

This pseudo-op turns off the L option for selected portions of a listing and is used in conjunction with the PRINT option. The syntax is:

```
NOPRINT [n]
```

where n is a decimal number. The default value is 1.

PAGE

This pseudo-op causes the listing device to advance to the top of the next page. This statement does not appear on the listing, generates no object code, and must not be labeled. The syntax is:

```
PAGE [n]
```

where n is a decimal number. The default value is 1.

PRINT

This pseudo-op is used to turn on the L option previously turned off by a NOPRINT statement. The syntax is:

```
PRINT [n]
```

SET

This pseudo-op assigns to a symbol a value other than the value normally assigned by the program-location counter. The syntax is:

```
Label    SET    operand
```

The SET statement must be labeled. The operand field may contain a number, symbol, or expression. Symbols appearing in the operand field must have been previously defined. This pseudo-op is like the EQU pseudo-op except that symbols may be defined more than once. The entire 32-bit operand value is used.

3.7 MACROS

Macros are sections of code that are defined once at the beginning of a program and used and referenced by a mnemonic code, with or without parameters.

Usually the code in a macro contains statements that are repeated many times throughout a program. Macros provide a shorthand notation for repeating these sections of code.

The statements to be repeated are grouped in one place at the beginning of the program and are preceded by the MACRO pseudo-op and followed by the ENDM pseudo-op. The macro is named by placing the name in the MACRO statement's label field. A macro is called by placing its name in a statement's opcode field along with any parameters to be passed to the Macro in the operand field (separated by commas).

The effect of a macro operation is the same as an open subroutine in that it produces in-line code. The in-line code is inserted in the normal flow of the program so that the generated statements are assembled with the rest of the program.

3.7.1 MACRO PROTOTYPE

The macro definition is known as the prototype. The source statements included in the prototype may be any legal assembler or processor instruction except for another MACRO pseudo-op.

Macro prototypes are of the form:

```
Header          MNAME   MACRO   [C]
                ................
Body            ................
                ................
Termination             ENDM
```

where:

MNAME is the name of the Macro.

C is an optional operand to control the storing of comment lines from the prototype body.

Body is the sequence of source statements.

Termination is the line containing the pseudo-op ENDM.

ENDM is recognized by the assembler as the end of the macro definition.

3.7.2 EXAMPLE OF A MACRO PROTOTYPE

```
line 1          LOOP    MACRO C
line 2                  MOVEQ #&0,D1    LOAD D1 WITH ARGUMENT 0
line 3                  MOVEQ #&1,D2    LOAD D2 WITH ARGUMENT 1
line 4          *
line 5                  SUBQ.L #1,D2    COUNT DOWN D2
line 6                  BNE.S *-2
line 7          *
line 8                  SUBQ.L #1,D1    COUNT DOWN D1
line 9                  BNE.S *-8
line 10         *
line 11                 RTS             ALL DONE
line 12                 ENDM
```

Line 1 is the header. It names the macro as LOOP and specifies that comment lines in the body are to be stored in the macro definition in the Macro Table.

Lines 2 and 3 are source statements with substitutable arguments (parameters) in the variable field (&0, &1). A substitutable argument is recognized by the presence of the ampersand. The digit after the "&" is the argument number. Ten is the maximum number of arguments usable in a single macro (0-9).

Lines 4-11 make up the rest of the prototype body.

Line 12 is the termination line.

Lines 2 through 11 are stored in the Macro Table by the assembler for later use. If the "C" was not on the header statement, then lines 4, 7, and 10 will not have been saved.

The macro name "LOOP" is stored in the symbol table with a pointer to the location of the macro definition in the Macro Table.

A typical call to the macro LOOP might be:

```
        LABEL     LOOP 100,12
```

This would expand into the following:

```
                MOVEQ #100,D1          LOAD D1 WITH ARGUMENT 0
                MOVEQ #12,D2           LOAD D2 WITH ARGUMENT 1
*
                SUBQ.L #1,D2           COUNT DOWN D2
                BNE.S *-2
*
                SUBQ.L #1,D1           COUNT DOWN D1
                BNE.S *-8
*
                RTS                    ALL DONE
```

The parameter argument "100" is substituted for 0 and the parameter argument "12" is substituted for &1. The parameters can be any character string at all. They may be used anywhere in the macro body to stand for the character string they will represent when the macro is expanded.

3.8 INTERPROGRAM LINKAGE

Linking pseudo instructions are used to provide a means of communications between a main program and its subroutines or among several subprograms that are to be linked together to run as a single program.

3.8.1 COMMON

COMMON reserves a block of storage locations that may be used in common by several programs. Each SYMBOL identifies a segment of the block for the subprogram in which the COMMON statement appears. The operand is the length of the related segment. The format of the statement is:

```
COMMON     SYMBOL,operand
```

Any number of COMMON statements may appear in a subprogram. Storage locations in common are assigned contiguously. The length of the clock is equal to the sum of the lengths of all segments named in COMMON statements in the subprogram.

3.8.2 USE OF COMMON

To refer to the Common block, other subprograms must also include a COMMON statement. The segment names and lengths may be the same or they may differ. Regardless of the names and lengths specified in the separate subprograms, there is only one Common block for the combined set. It has the same relative origin; the content of the nth byte of Common storage is the same for all subprograms.

EXAMPLE

```
NAME      PROG1
COMMON    AAA,5          ALLOCATE 5 BYTES OF COMMON
COMMON    BBB,10         ALLOCATE 10 BYTES OF COMMON
COMMON    CCC,10         ALLOCATE 10 BYTES OF COMMON
 -
MOVE.B BBB+1,D0          LOAD BYTE 2 OF SEGMENT BBB
 -
END


NAME      PROG2
COMMON    DDD,2          ALLOCATE 2 BYTES OF COMMON
COMMON    EEE,2          ALLOCATE 2 BYTES OF COMMON
COMMON    FFF,1          ALLOCATE 1 BYTE OF COMMON
COMMON    GGG,20         ALLOCATE 20 BYTES OF COMMON
 -
MOVE.B GGG+1,D0          LOAD BYTE 2 OF SEGMENT GGG
 -
END
```

3.8.3 ORGANIZATION OF THE COMMON BLOCK

PROG1	PROG2	COMMON BLOCK
AAA	DDD	BYTE1
		BYTE 2
	EEE	BYTE 3
		BYTE 4
	FFF	BYTE 5
BBB	GGG	BYTE 6
		BYTE 7
		.
		.
		BYTE 15
CCC		BYTE 16
		BYTE 17
		.
		.
		.
		BYTE 25

The MOVE.B instruction in both of the subroutines refer to the same location in Common (Byte 7).

The segment names that appear in the COMMON statements can be used in the operand fields of EQU, DC.L, or any memory reference statement; they may not be used as labels elsewhere in the program. All references to Common are relocatable.

The user establishes the origin of the Common block when the Linking Loader is executed. Two or more subprograms may declare Common blocks that differ in size.

3.8.4 ENTRY

ENTRY defines entry points to the program or subprogram. SYMBOL is an assigned label for some statement in the program. Entry points allow another subprogram to refer to this subprogram. All entry points must be defined in the program. The format of the statement is:

```
ENTRY SYMBOL
```

3.8.5 EXTERNAL

EXTERNAL designates labels in other subprograms that are referenced in this subprogram. The format of the statement is:

```
      EXTERNAL SYMBOL
```

SYMBOL must be defined as an ENTRY in some other subprogram.

Example of Entry and External

```
            NAME PROG1
*
            ENTRY SUB1          DEFINE SUB1 AS AN ENTRY POINT
            ENTRY SUB2          DEFINE SUB2 AS AN ENTRY POINT
*
SUB1        MOVE.W #$1234,D4
             -
            RTS
*
SUB2        MOVE.B #'C',D2
             -
            RTS
            END

            NAME PROG2
            EXTERNAL SUB1       DEFINE SUB1 AS EXTERNAL
            EXTERNAL SUB2       DEFINE SUB2 AS EXTERNAL
*
            JSR SUB1            CALL SUB1 IN PROG1
            JSR SUB2            CALL SUB2 IN PROG2
             -
            END
```

SUB1 and SUB2 are referenced in PROG2 but are actually locations in PROG1.

The COMMON pseudo-op is provided to allow data communication and the EXTERNAL and ENTRY pseudo-ops are provided to allow control communication between separately assembled subprograms that are linked together to form a single program to be executed as a unit.

3.8.6 LISTING OUTPUT FORMAT

Fields of the program are listed in the following format:

COLUMNS	CONTENT
1-4	Line number generated by the assembler
5	Blank
6	Blank ("+" if line is macro expansion)
7-14	Location (32-bit hexadecimal)
15	Blank
16-19	Opcode-word (hexadecimal)
20	Blank

There are seven possible formats for instruction listings:

FORMAT 1.

21-32	Blank
33-	Source line

FORMAT 2.

21-24	Second instruction word.
25-32	Blank
33-	Source line

FORMAT 3.

21-28	32-bit operand
29-30	Blanks
31	Reloc. or Common indicator (R, C)
32	Blank
33-	Source line

FORMAT 4.

21-24	One-word operand
25	Blank
26-29	Second one-word operand
30-32	Blank
33-	Source line

FORMAT 5.

21-24	One-word operand
25	Blank
26-29	Second word operand
new line	
1-20	Blanks
21-28	32-bit operand
29-30	Blanks
31	Relocation or Common indicator (R or C)
32	Blank
33-	Source line

FORMAT 6.

21-28	32-bit operand
29-30	Blanks
31	Relocation or Common indicator (R or C)
new line	
1-20	Blanks
21-24	word operand
25-32	Blanks
33-	Source line

FORMAT 7.

21-28	32-bit operand
29-30	Blanks
31	Relocation or Common indicator (R or C)
new line	
1-20	Blanks
21-28	32-bit operand
29-30	Blanks
31	Relocation or Common indicator (R or C)
32	Blank
33-	Source line

Lines consisting entirely of comments are listed:

1-4	Line number
5	Blank
6	Blank or "+" if line is a macro expansion
7-32	Blanks
33-	Source statement

Pseudo-operations are listed:

1-4	Line number
5	Blank
6	Blank or "+" if line is in macro expansion
7-14	Location (32-bit hexadecimal)
15	Blank

0-word pseudo-ops

16-32	Blanks
33-	Source line

1-word pseudo-ops

16-19	Data word
20-32	Blanks
33-	Source line

2-word pseudo-ops

16-23	Data long-word
24-30	Blanks
31	Relocation, Common, Entry, External Indicator
32	Blank
33-	Source line

3-word pseudo-ops

16-27	3 words of data
28-30	Blanks
31	Relocation, Common, Entry, External Indicator
32	Blank
33-	Source line

A Symbol Table listing has the following format:

COLUMNS	CONTENT
1-8	SYMBOL
9	Blank
10-17	Value in hex (32-bits)
18	Blank
19-	Symbol Type: R-Relocatable M-Macro N-Entry X-External C-Common

3.8.7 RELOCATABLE FILE FORMATS

The relocatable files are text: that is, they contain only printable ASCII characters and are formed into 70-character lines.

Binary bytes are encoded as two ASCII characters each. As an example, the byte 39 hex would be encoded as ASCII "3" followed by ASCII "9." Object code is simply a stream of such ASCII-encoded bytes.

A relocatable address is flagged with the ASCII "R" character. The "R" indicates that the preceding 8 characters represented an address which is to be relocated using the base address.

An address in COMMON is flagged with the ASCII "M" character. The "M" indicates that the preceding 8 characters represented an address which is to be relocated using the COMMON starting address.

An external reference consists of 16 characters which are the encoded form of the eight-character name of the reference followed by the ASCII "X" character.

An entry point consists of 16 characters which are the encoded form of the eight-character name of the entry, a relocatable address, and the ASCII "N" character. The relocated address becomes the address of the entry point which is used to resolve all references to the entry.

A program name consists of an encoded address followed by the ASCII "P" character followed by an entry-point which includes the name and relocatable address. The first address is the COMMON length of the program.

4.0 LINKER, LOCATER, AND LIBRARY MANAGER

CS-OS software development products provide the user with several software tools for the modular development of code. Modular software development makes programs that are lengthy or complex much easier to code and maintain. The tools that CS-OS provides for such development are a linker (LINK), a locator (LOCATE) and a library management program (LIB).

The linker program enables the user to combine two or more modules into a single one. The locator binds the relocatable addresses in a linked program to absolute memory locations. The library management program allows the user to form a library that can be accessed by the linker. Together, these programs comprise a powerful and versatile module-management package.

4.1 SECTION I. LINK

4.1.1 DESCRIPTION

The linking program, LINK, accepts assembler or language translator output as input. This input contains entry points, relocatable addresses, common addresses, and possible references of external symbols.

4.1.2 INPUT TYPES

LINK recognizes three input types:

1. Simple relocatable modules produced by an assembler or language translator.
2. Specific modules in a library.
3. An entire library.

In this manual the term "specific module" always means input types 1 and 2. The linker combines all specific modules in the order in which it receives them as input. Output from a previous execution of LINK may also be used as input to LINK, at a later time, if additional modules are to be added.

4.1.3 RELOCATION

The linker combines modules by relocating them with respect to the previous modules processed. The relocatable addresses of the first module remain unchanged, but the subsequent modules are relocated to reflect their relationship to the previous modules. Thus the length of the first module is added to the relocatable addresses of the second module. The combined length of the first and second modules are added to the relocatable addresses of the third module, and so on.

4.1.4 EXTERNAL REFERENCES

RESOLVED

When an external reference is encountered, the linker searches the contents of all the modules it has already processed for an entry point that matches the external reference. If it does not find any, the linker searches for it in the modules not yet processed. If unsatisfied external references remain after all specific modules have been processed by LINK, the program will check any general libraries that have been given as inputs. These are searched for any modules having an entry point satisfying any resolved external reference. Library modules that have the matching entry points are then linked in with the specific modules already linked. When a library module that is added contains unresolved external references, another search of the libraries occurs.

UNRESOLVED

If unresolved externals still remain after searches of the libraries, a WARNING message is produced by LINK. This message does not necessarily indicate an error condition. Both entry points and unresolved external references are left in the linker output to allow the option of future linking. Programs can then be developed and linked together in stages. Other modules can be linked to satisfy these unresolved externals at a later time.

4.1.5 MODULE AND LIBRARY MAXIMUMS

LINK allows a maximum of twenty (20) specific modules as input at one time. Up to ten (10) general libraries are allowed at one time. These limits apply to each execution only. Multiple executions of LINK provide the means for exceeding these maximum values. For example, the linker could be run once with twenty specific modules and then rerun with the previous linker output and nineteen specific modules. This process allows a virtually infinite linking capability.

4.1.6 OPERATING INSTRUCTIONS FOR LINK

1. To invoke LINK, type

```
LINK
```

The system will produce the following prompt:

```
ENTER FILES TO BE LINKED
INPUT FILE:
```

The user should respond with one of the three file types:

```
name.ext

name.LIB (module name)

name.LIB
```

A carriage return will enter the filename and generate a repetition of the INPUT FILE: prompt. Another filename can now be typed and entered. A carriage return without a filename will terminate the input process.

2. The system now produces the following prompt:

```
DO YOU WANT A LINK MAP?
```

Answer YES or NO for a LINK map. (The letter "Y" is sufficient for specifying a YES answer. Any word beginning with a letter other than "Y" will be interpreted as a NO answer.)

The LINK map is printed on the line printer by default. If the Computer System does not have a line printer attached, the answer must be no, to avoid "hanging up" the system.

3. The system next prompts with

```
DO YOU WANT AN OUTPUT MODULE?
```

Answer YES or NO for linker output. If YES, the system will generate the following prompt:

```
OUTPUT FILE:
```

(The letter "Y" is sufficient for specifying a YES answer. Any word beginning with a letter other than "Y" will be interpreted as a NO answer.)

Respond with

```
name.ext.
```

Note: It is suggested that you adopt the convention of using "LNK" as the extension for all LINK output filenames.

4.1.7 EXAMPLES OF LINKER USE

The following examples illustrate the use of LINK to perform common operations.

EXAMPLE I

This first example demonstrates a simple linking procedure: the linking of three subroutines (named as external) to a main program. The name of the program is MAIN.REL. The names of the three subroutines are CON.REL, CONSTR.REL, and CENT.REL.

First, invoke the linker by typing

```
LINK.BIN
```

which generates the first prompt:

```
ENTER FILES TO BE LINKED
INPUT FILE:

(respond) MAIN.REL (carriage return)
```

which enters the name of the main program. If files to be linked are implemented as subroutines, as they are in this case, it is important for the calling program to be listed first because the linker links files in the order they are received.

```
INPUT FILE:

(respond) CONV.REL (carriage return)
```

which enters the name of the first module.

```
INPUT FILE:

(respond) CONSTR.REL (carriage return)

INPUT FILE:

(respond) CENT.REL (carriage return)

INPUT FILE:

(carriage return)
```

which terminates the input process. All the modules have been input.

```
DO YOU WANT A LINK MAP?

(respond) YES
```

Sample LINK map:

```
LINK MAP

MAIN        00000000
CONV        000002AB
CONSTR      000002F0
CENT        00000310
```

The address following the global symbol is the location that the module was relocated to.

```
DO YOU WANT AN OUTPUT MODULE:

(Respond) YES
```

If the user responds YES, an output module containing the relocated linked modules will be created.

```
OUTPUT FILE:
```

This prompt asks for the name of a new file. (It is suggested that you adopt the convention of naming linker output files with "LNK" as the extension.)

EXAMPLE II

This second example assumes that the three subroutines from the first example have been stored in a library (by means of the library program). The library is called UTIL.LIB. The main program is still MAIN.REL and still needs the same three routines.

Invoke the linker by typing LINK.

```
ENTER FILES TO BE LINKED.
INPUT FILE:

(respond) MAIN.REL (carriage return)
```

which enters the name of the main file to be linked.

```
INPUT FILE:

(Respond) UTIL.LIB (carriage return)
```

Since the main program needs three external routines that are known to be in UTIL.LIB, that library name can be given as input to LINK. The linker will search all of the specific modules first. If unresolved external references are found, the linker will then check any general libraries given. The modules will be found in the library in this case.

```
INPUT FILE:

(respond) (carriage return)

.
.
.
```

The rest of the session will be identical to the corresponding portion of Example I.

4.2 SECTION II. LOCATE

4.2.1 DESCRIPTION

LOCATE produces an absolute, binary module from the relative, ASCII output of LINK. LOCATE will also accept, as input, single modules of assembler output. LOCATE adds a base address to each of the relative addresses, and a common base address to each of the common addresses. This binary module is created on disk (not in memory), thereby allowing greater versatility.

If a module sent to LOCATE contains unresolved external references, no error condition occurs, but the occurrence itself generates a warning message, and the address is replaced by zero. This allows the user to substitute an address by hand or to enter a breakpoint before the instruction is executed.

4.2.2 OPERATING INSTRUCTIONS FOR LOCATE

Note: LOCATE cannot be used unless the Computer System has a line printer attached.

1. To invoke LOCATE, type in LOCATE.

2. Enter the name of the file to be bound to absolute addressing.

3. Enter the base address.

4. Enter the common address. (Note: If you enter only a carriage return , the common address will be set to the base address.)

5. Enter the transfer address. (Note: If you enter only a carriage return, the transfer address will be set to the base address.)

Note: The user has the option of specifying the base, common, and transfer input addresses in either hexadecimal or decimal notation. If hex is chosen, the number must be preceded by a dollar sign.

4.2.3 OUTPUT FILENAME

The output of LOCATE is created in a file with the same filename as the input file; however, the extension of the LOCATE output file will be "BIN."

WARNING -- If a file already exists with the LOCATE output filename, it will automatically be deleted by LOCATE before the output is formed.

4.2.4 FORMAT OF CS-OS BINARY FILES

Binary files under CS-OS are stored as segments of memory images. This class of file includes system utilities, nonresident commands, SAVE files, and output from LOCATE.

There are two types of data in a binary file: transfer address and memory image data. Each type of data is stored in a block of up to 256 bytes.

The format of a transfer address is:

```
BYTE 1   transfer address mark  $16
BYTE 2-5 transfer address - 4 bytes/32 bits
```

The format of memory data is:

```
BYTE 1    memory data mark  $02
BYTE 2-5  memory address  4 bytes/32 bits
BYTE 6    count of data bytes
BYTE 7-n  data bytes exactly as in memory
```

If there is more than one transfer address, then the last one encountered is used. The last sector is padded with $00. This has no effect on memory loading.

4.3 SECTION III. LIB

4.3.1 DESCRIPTION

The library management program (commonly referred to as "LIB") allows the user to manipulate libraries quickly, easily, and powerfully. The LIB utility is especially useful for creating a library that contains frequently used routines.
These routines are listed as an external in the program that uses them. For example, one library could contain scientific subroutines, another could be created to contain programs of special interest to the user.

4.3.2 LIB COMMANDS

The LIB program has five commands:

1. CREATE
2. ADD
3. DELETE
4. LIST
5. EXIT

- CREATE or CR

  The CREATE command formats a new library file. All libraries must be named with the extension "LIB" or they will not be recognized as libraries by the LINK program.

- ADD or AD

  The ADD command adds a module to a previously existing library or forms a new one following the use of the CREATE command.

- LIST or LI

  The LIST command lists the names of all modules contained in a given library. The LIST command also provides the option of listing all of the entry points contained in each module as well as the module names.

- DELETE or DE

  The DELETE command removes a module from a given library.

  Note: When using either the ADD or DELETE commands, it is important to remember that a file with the same name as the library being modified, but with the extension "TMP," is used and then deleted. If the user has a file with that name on the disk, it will be destroyed.

- EXIT or EX

  The EXIT command returns control to the operating system.

4.3.3 OPERATING INSTRUCTIONS FOR LIB

To invoke LIB, type in LIB. The resulting prompt will be a dollar sign ($).

Then enter any legal command.

The legal command formats are as follows:

CREATE <name.LIB>

where <name.LIB> is the name of a new library

LIST <name.LIB> E

where <name.LIB> is an existing library. If the library is empty, it will be noted. If the E option is not specified, only the names of the the modules themselves will be listed. If the E option is specified, the names of each module will be listed, along with their entry points.

ADD <name.ext> TO <name.LIB>

adds a new module to an existing library. The

same module cannot occur more than once in the same library.
Note: the module name in the library will be the name within the file. It is recommended that the name within the file and the filename be the same.

DELETE <module name> FROM <name.LIB>

where <module name> is the name of the module to be deleted from the given library.
Note: <module name> is not a filename. To find the name of the modules in a given library, use the LIST command before using DELETE.

EXIT

returns the user to system mode.

4.3.4 EXAMPLES OF LIB USE

The following are some typical examples of how the library management program can be used. All examples assume that the library program has been invoked by typing LIB.BIN. The dollar sign ($) on the left is the prompt generated by the library and is not to be typed in by the user.

EXAMPLE I

This example presupposes that the user wishes to develop a software library of scientific functions that can be conveniently Linked to other modules.

- Assume that one of these functions is a subroutine contained in a file named SQRT.REL. This subroutine performs a square root operation with certain calling conventions.
- The library is to be called SCI.LIB.
- The contents of the library will be listed after it is created.

$CREATE SCI.LIB

This command formats a new library called SCI.LIB (as long as a library by that name does not already exist). The library manager returns the message "LIBRARY CREATED" to confirm that the

command has been executed.

```
$ADD SQRT.REL TO SCI.LIB
```

The LIST command will list all module names contained in the library. In our example, LIST will respond with:

```
MODULE NAME: SQRT
```

assuming the module name and filename were both SQRT.

```
$EXIT
```

This command returns control to the operating system.

EXAMPLE II

In this example, two modules are deleted from an existing library. The library will be listed and the modules deleted.

- Assume WORDP.LIB contains some word-processing functions called STRINGL, STRINGR, STRINGC, and STREND.

The object is to delete STREND and STRINGR from the library.

```
$LIST WORDP.LIB
```

The LIST command lists the contents of the library. In this example the output would be:

```
MODULE NAME: STRINGL
MODULE NAME: STRINGR
MODULE NAME: STRINGC
MODULE NAME: STREND
```

The next step is to delete the two modules modules STREND and STRINGR from the library by using the DELETE command:

```
$DELETE STREND FROM WORDP.LIB
$DELETE STRINGR FROM WORDP.LIB
```

The revised library can then be listed:

```
$LIST WORDP.LIB
```

which would produce:

```
MODULE NAME: STRINGL
MODULE NAME: STRINGC
```

```
$EXIT
```

returns control to the operating system.

EXAMPLE III

The third example demonstrates the replacement of an existing library module with a new version.

- Assume that SQRT is a module contained in SCI.LIB. and that a faster method of computing square roots is implemented in a new version of SQRT.REL.

  The new program routine is to be incorporated in SCI.LIB as a replacement for the outdated version.

- Assume that SCI.LIB contains SQRT, COS, and SIN.

```
$LIST SCI.LIB
```

lists the contents of the library:

```
MODULE NAME: SQRT
MODULE NAME: COS
MODULE NAME: SIN
```

Library member names are listed in the order in which they were added to the library.

```
$DELETE SQRT FROM SCI.LIB
```

Delete the old version of SQRT.

```
$LIST SCI.LIB
```

The library now contains:

```
MODULE NAME: COS
```

```
                    MODULE NAME: SIN.

$ADD SQRT.REL TO SCI.LIB
```

The library now contains the new version of SQRT.

```
$LIST SCI.LIB
```

will produce:

```
MODULE NAME: COS
MODULE NAME: SIN
MODULE NAME: SQRT
```

Notice that the order of the modules in the library is now different. SQRT was the last library member added and this is reflected in the output produced by LIST.

```
$EXIT
```

returns control to the system.

5.0 CS-DEBUG

5.1 INTRODUCTION

CS-Debug is a debugging utility designed to run in the multitasking environment of CS-OS.

5.2 OPERATING CS-DEBUG

DEBUG is located to operate in high memory, out of the way of user programs.

First LOAD the program to be debugged, then type DEBUG.

To begin testing a program, the programmer sets breakpoints (optional) and uses the GO command to jump execution to the first statement of the program. This address is contained in relocation register R0.

To terminate the debugging (after the program releases any resources it may have acquired), the programmer places a breakpoint where the program KILLs or terminates itself. When this breakpoint is hit, the EX command is used to leave the debugging process. If any channel other than the EX command is used to return to the system, performance becomes unpredictable and hangups may occur.

Debugging Modular Software

There is a fundamental problem in debugging a program consisting of modules that have been linked after being assembled independently. While the assembler listing for each module displays relocatable addresses relative to zero, the user must do some arithmetic to locate an address or label within a module after it has been linked into a larger program. This procedure is not only inconvenient but time-consuming and vulnerable to error as well.

Relocation Registers

CS-Debug contains a set of eight additional registers not found in the computer hardware. These are the eight relocation registers, implemented in software. The relocation registers are displayed, set and changed in the same manner as the address (A) and data (D) registers.

Although in certain cases the relocation registers may be legally used in place of hex values, their primary purpose is to aid in converting the zero-based addresses of assembler listings to the absolute run-time addresses needed to debug software.

When the debugger module is invoked, R0 will contain the base address of the debugger code itself. R1 and the PC should be set to the base address of the first program module. These addresses alone suffice for the debugging of a single-module program. Any relative address in the module can be translated to an absolute address with the command

<relative address> R1

5.2.1 DEBUGGING MULTIPLE-MODULE PROGRAMS

CS-Debug's relocation registers greatly facilitate debugging of multiple-module programs. The base address of each module is contained in one of therelocation registers, ready for use whenever an address in that module is referenced.

Assigning the base addresses of modules to the relocation registers requires the use of the LOCATE map. First, the debugger sets the base address of itself and assigns it to R0. Then the user sets the base address of the first module by assigning to R1, the value corresponding to that module on the LOCATE map. Additional modules are set in the same manner.

Debugging can begin after each module's base address has been set and assigned to a relocation register. Any relative address from the assembler listing of any module can be accessed with the command

<relative address> R#

where "#" is the number of the relocation register to which that module has been changed.

5.2.2 EXAMPLE OF SETTING UP CS-DEBUG FOR A MULTIPLE-MODULE PROGRAM

For purpose of this example consider a software package consisting of three modules: MOD1, MOD2, and MOD3. Assume that this software does not work, so needs debugging. Use the linker (LINK) to link together the MOD1, MOD2 and MOD3. Note: In order to avoid confusion it is recommended that the user adopt a convention in naming debug modules. A suggestion is to name the linker output with the original module's name plus an appended letter (e.g."D").

Save the LOCATE map, which in this example looks like the following:

```
MOD1        10E00
MOD2        1124E
MOD3        118EC
```

Now form a loadable program by using the LOCATE utility. Bring up the debugger by typing

DEBUG <parameters>

When the debugger comes up, R0 will contain the base address of CS-Debug. To set R1, R2, and R3 to MOD1, MOD2, and MOD3 respectively, type

```
R1   10E00
R2   1124E
R3   118EC
```

where the values have been obtained from the LOCATE map.

In order to access a value in any module, simply type in the relative address found on the assembly listing and then the number of the relocation register (e.g., R2) containing the base address of the module. The address for debugging will then be computed for you.

5.3 CS-DEBUG COMMANDS -- SYNTAX AND DEFINITIONS

The syntax notation used with CS-Debug is:

Parentheses -- () -- indicate that the enclosed parameter is required. These symbols are used to define the command format. They are not part of the command itself and are not to be entered.

(num) is a value expression of the form:

```
(hex value)
(hex value) + R#
(hex value) R#
 R#
```

where R# is a relocation register.

reg	any address, data, or relocation register; program counter, status register, or system stack
class	defines a register as address (A), data (D), or relocation (R)
#	values 0 through 7, specifying a register
hex value	hexadecimal numeric expression
hex exp	a value expression of the form: <hex value> <hex value> + R# <hex value> R# R#
<count>,<end>,<start>	are hexexps
Address Registers:	A0, A1, A2, A3, A4, A5, A6, A7
Data Registers	D0, D1, D2, D3, D4, D5, D6, D7
Relocation Registers:	R0, R1, R2, R3, R4, R5, R6, R7

5.4 SUMMARY OF CS-DEBUG COMMANDS

Command	Description
A	Prints contents of all address registers
A:	Prints and prompts for change on all address registers
A#	Prints contents of an address register
A#: hexexp	Prints and prompts for change of an address register
A# hexexp	Sets an address register to hexexp
BR	Prints all breakpoints set
BR hexexp	Sets a breakpoint at hexexp
BR -hexexp	Removes a breakpoint at hexexp
BR CLEAR	Clears all breakpoints set
D	Prints contents of all data registers
D:	Prints and prompts for change on all data registers
D#	Prints contents of a data register
D#:	Prints and prompts for change on a data register
D# hexexp	Sets a data register to hexexp
DR	Display all registers
DM hexexp hexexp	If second hexexp >= first hexexp, then display memory between first and second hexexps. If second hexexp < first hexexp, then display second-hexexp bytes starting at first hexexp.
DM hexexp	Display 16 bytes starting at hexexp
EX	Release console and kill debug task Note: Program being debugged must not hold any resources when this CS-Debug command is executed
G hexexp	Begin execution at hexexp
G	Begin execution at location contained in PC
OP hexecp	Change memory starting at hexexp
PC	Print contents of PC
P:	Print and prompt for change on PC
PC hexexp	Set PC to hexexp
SS	Print contents of system stack
SS:	Print and prompt for change on system stack
SS hexexp	Set system stack to hexexp
SR	Print contents of status register
SR:	Print and prompt for change in status register
SR hexexp	Set status register to hexexp
R	Print contents of all relocation registers
R:	Print and prompt for change on all relocation registers
R#	Print a relocation register
R#:	Print and prompt for change on relocation register
R#	Set relocation register to hexexp

Note: In the list above, several commands and their initial operand are separated by a blank space. In practice, this space is not required.

5.5 REGISTER DISPLAY

Format	Description
reg# hexexp	Puts value of hexexp in reg#
reg#:	Prints original value and prompts for a new hexexp
reg#	Prints register value
class	Prints the value of all registers in class
class:	Cycles through all the registers in the class, printing old values and prompting for new ones
DR	Display values of all registers

5.5.1 REGISTER DISPLAY EXAMPLES

```
Command                          Description

A0 8004                          ASSIGN A VALUE TO A0
A1:                              DISPLAY A1 AND PROMPT FOR CHANGE
A1=00000000 ?824
A1                               DISPLAY A1
A1=00000824
A0                               DISPLAY A0
A0=00008004
A                                DISPLAY ALL A REGISTERS
A0=00008004 A1=00000824 A2=00000000 A3=00000000
A4=00000000 A5=00000000 A6=00000000 A7=0001C5EE
R2 600                           SET R2
R0 5280                          SET R0
R                                DISPLAY ALL R REGISTERS
R0=00005280 R1=0000B076 R2=00000600 R3=00000000
R4=00000000 R5=00000000 R6=00000000 R7=00000000
D:                               PRINT AND PROMPT FOR CHANGE ON ALL D REGISTERS
D0=00000000 ?32                  STORE 32 IN D0
D1=00000000 ?                    NO CHANGE
D2=00000000 ?23
D3=00000000 ?43
D4=00000000 ?
D5=00000000 ?
D6=00000000 ?42
D7=00000000 ?66
D                                DISPLAY ALL D REGISTERS
D0=00000032 D1=00000000 D2=00000023 D3=00000043
D4=00000000 D5=00000000 D6=00000042 D7=00000066
A:                               PRINT AND PROMPT FOR CHANGE ON ALL A REGISTERS
A0=00008004 ?                    HITTING CR ALONE GIVES NO CHANGE
A1=00000824 ?
A2=00000000 ?630
A3=00000000 ?
A4=00000000 ?
A5=00000000 ?
A6=00000000 ?
A7=0001C5EE ?
DR                               DISPLAY ALL REGISTERS
PC=0000B076 SR=2000 S6=0001656E
D0=00000032 D1=00000000 D2=00000023 D3=00000043
D4=00000000 D5=00000000 D6=00000042 D7=00000066
A0=00008004 A1=00000824 A2=00000630 A3=00000000
A4=00000000 A5=00000000 A6=00000000 A7=0001C5EE
R0=00005280 R1=0000B076 R2=00000600 R3=00000000
```

5.6 MEMORY DISPLAY

Format	Description
DM <start> <end>	Display memory in hex and ASCII <start> must be less than <end>
DM <start> [count]	Display [count] bytes beginning at <start>. [count] must be less than <start>. Default [count] is 16.

All displays are done in 16-byte units, with the number of bytes displayed rounded to the next highest multiple of 16.

5.6.1 MEMORY DISPLAY EXAMPLES

```
Command                    Description

DM 8000 8010               DISPLAY MEMORY BETWEEN 8000 and 8010
008000  4D F9 00 00 82 76 4E 40 00 12 4E 40 00 11 42 00  M....VN ..N ..B.
DM 800  25                 DISPLAY 25 (HEX) BYTES STARTING AT 800
                           NOTICE THAT 30 (HEX) BYTES ARE ACTUALLY PRINTED
                           BECAUSE AN EVEN MULTIPLE OF 16 MULTIPLE BYTES
                           IS ALWAYS PRINTED
000800  4E 4F 57 20 49 53 20 54 48 45 20 54 49 4D 45 20  NOW IS THE TIME*
000810  46 4F 52 20 41 4C 4C 20 47 4F 4F 44 20 4D 45 4E  FOR ALL GOOD MEN
000820  20 54 4F 20 43 4F 4D 45 20 54 4F 20 54 48 45 20  TO COME TO THE
R0 8000                    SET R0
DM 500R0 30                USE R0 TO SPECIFY THE ADDRESS
008500  00 14 00 15 00 00 00 00 00 00 07 C5 07 C3 00 00  ................
008510  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
008520  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
DM 500                     DEFAULT IS ONE LINE (10 HEX) BYTES
000500  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
DM 15
000015  00 12 E2 00 00 13 06 00 00 13 2C 00 00 13 54 00  ................
DM 100 15
000100  00 FE 17 06 00 FE 17 06 00 FE 17 06 00 FE 17 06  ................
000110  00 FE 17 06 00 FE 17 06 00 FE 17 06 00 FE 17 06  ................
```

5.7 MEMORY CHANGE (ENTERING THE SUBCOMMAND MODE)

Format	Description
OP hexexp	Open memory at address specified in hexexp. Enter subcommand mode.

5.7.1 SUBCOMMAND MODE

In subcommand mode, the address contained in hexexp when the OP command was entered is displayed, as well as the value at that address. Then a prompt is issued for a response. A response consists of either a location control character or a value and a location control character. The location control character is required; the value is optional. If a value is specified, the location control character follows it.

Use the escape character to exit the subcommand mode.

Format	Description
[hexexp] <loc ctrl char>	Set the value of the address to [hexexp] if desired. Use the location control character to determine what location to process next.

Location control characters:

<cr>	go to next address
=	stay at current address
^	go to previous address
.	without [hexexp], terminate subcommand mode

5.7.2 MEMORY CHANGE EXAMPLES

Command	Description
OP 8000	Enter address change at 8000
008000 42 ?53	Set value, go to next address
008001 50 ?	Don't change, go to next address
008002 00 ?^	Go to previous address
008001 50 ?45	Set value and stay at same address
008002 45 ?46	" " " " " " "
008001 46 ?	No change, go to next address
008002 00 ?	" " " " " "
008003 00	" " " " " "
008004 82	" " " " " "
008005 76 ?46	Change value; go to next address
008006 4E ?	
008007 40 ?	
008008 00 ?	
008009 12 ?^	Go to previous address, no change
008008 00 ?^	"
009007 40 ?22	Set value
008008 00 ?=	Stay at same address; do not change (could be used to
008008 00 ?=	monitor an I/O port)
008008 00 ?	
008009 12 ?.	Terminate subcommand mode

5.8 EXECUTION CONTROL

Format	Description
BR	Display all breakpoints
BR hexexp	Set at breakpoint at hexexp
BR -hexexp	Remove a breakpoint from hexexp
BR CLEAR	Clear all breakpoints
G	Begin execution at address in PC
G hexexp	Begin execution at hexexp
EX	Return control to system

5.8.1 EXAMPLES OF EXECUTION CONTROL

Command	Description
BR	DISPLAY BREAKPOINTS
BREAKPOINT(S)=	NONE IS SET
BR 8004	SET ONE AT 8004
G 8000	BEGIN EXECUTION AT 8000
PC=00008004 SR=2000 SS=0001C5EE	BREAKPOINT WAS HIT AND CONTROL RETURNED TO CS-Debug
D0=00000032 D1=00000000 D2=00000023 D3=00000043	
D4=00000000 D5=00000000 D6=00000042 D7=00000066	
A0=00008004 A1=00000824 A2=00000630 A3=00000000	
A4=00000000 A5=00000000 A6=00000000 A7=0001C5EE	
BR	BREAKPOINT IS REMOVED AFTER IT IS EXECUTED
BREAKPOINT(S)=	
PC 8200	SET PC
BR 8320	SET BREAKPOINT
BR	DISPLAY IT
BREAKPOINT(S)=008320	
G	BEGIN EXECUTION AT ADDRESS SPECIFIED IN PC
PC=008320 SR=2000 SS=0001C5EE	BREAKPOINT HAS BEEN ENCOUNTERED
D0=00000032 D1=00000000 D2=00000023 D3=00000043	
D4=00000000 D5=00000000 D6=00000042 D7=00000066	
A0=00008004 A1=00000824 A2=00000630 A3=00000000	
A4=00000000 A5=00000000 A6=00000000 A7=0001C5EE	

5.8.2 HARD COPY

Format	Description
LPT ON	Issue a form feed and begin to send a copy of debug session to line printer.
LPT OFF	Stop sending to line printer.

A.0 APPENDIX: ERROR MESSAGES

A.1 PROCESSOR TRAP HANDLING

CS-OS includes a facility for detecting processor TRAPS and providing a display of the pertinent information available as an aid in troubleshooting.

There are two types of TRAPS:

TYPE 1 Standard processor TRAP

includes
- OPCO invalid OP code trap
- DIV0 divide by zero trap
- CHKC check instruction trap
- TRPV Trap V instruction
- PRIV privelege violation

TYPE 2 Extended information TRAPS

- ADDR illegal address trap
- SPUR spurious interrupt trap
- BUS bus error trap
- ABUS address bus error
- DBUS data bus error
- PROT memory protection error
- DTAK missing DTACK error
  (This message can result from an attempt to address memory or devices that are not implemented on your machine. Every access must terminate with "Data Transfer Acknowledge" or DTACK. If it does not, an error message is generated.
- MPAR parity error
- POWR power failure error

TRAP DISPLAY FORMAT

```
FNC=XXXX   ADD=XXXXXXXX   INR=XXXX     } TYPE 2 ONLY

TASK=TASKNAME        XXXX TRAP ERROR

PC=XXXXXX         SR=XXXX          USP=XXXXXX     SSP=XXXXXX
D0=XXXXXXXX       D1=XXXXXXXX      D2=XXXXXXXX    D3=XXXXXXXX
```

```
D4=XXXXXXXX     D5=XXXXXXXX     D6=XXXXXXXX     D7=XXXXXXXX
A0=XXXXXXXX     A1=XXXXXXXX     A2=XXXXXXXX     A3=XXXXXXXX
A4=XXXXXXXX     A5=XXXXXXXX     A6=XXXXXXXX     A7=XXXXXXXX

PRESS ANY KEY TO REBOOT
```

NOTES

The extended information for type 2 traps is:

FNC = processor function code
ADD = access address at time of trap
INR = Instruction register at time of trap

A.2 SYSTEM DEVICE ERROR MESSAGES

Note: You are advised that the information given here will change substantially in future releases.

Device errors in CS-OS are reported in the following format:

device-name ERROR: number

where 'device-name' is a three-character logical name and the error number is hex encoded. For example:

```
LPT ERROR: 0A
DSK ERROR: 02
```

are system device error messages. The set of errors defined in CS-OS are:

01 -- file not found
02 -- file already in use
03 -- file already exists
04 -- no such file
05 -- read/write error
06 -- directory overflow
07 -- disk full
08 -- end-file encountered
09 -- bad disk sector, bad media
0A -- device not ready
0D -- illegal use of File Control Block
12 -- illegal operation (write a read file, etc.)

15 -- bad file name
16 -- back-link error (disk corrupted during write)
17 -- missing file extension

A.3 CS-OS SYSTEM ERROR MESSAGES

FORMAT ERROR	The command line does not conform to the syntax specified for the command.
SYNTAX ERROR	The command line does not conform to the syntax specified for the command.
NUMBER ERROR	A bad numeric argument is present. The drive number is out of range or is not followed by a colon.
FILE NOT FOUND	The requested file could not be found.
INVALID SET PARM BAD PARAMETER	These error messages come from the SET command. They indicate that a bad SET command line was encountered.

The following errors come from COPY:

BAD INPUT (OUTPUT)	A device error; usually accompanied by a device-error message.
ILLEGAL INPUT (OUTPUT) DEVICE	Refers to attempts to use a device in an invalid manner, such as reading from a line printer.
BUFFER OVER-RUN	An overly long input line was encountered. The input file is probably the wrong type for the operationdesired.
ILLEGAL SWITCH	Indicates a syntax error in the switch portion of the command line.
READ (WRITE) ERROR	Encountered in disk-to-disk copying; accompanied by a device error message.
DIR ERROR	The directory on a disk could not be read properly. This message is usually accompanied by a device-error message.

Additional errors are:

SUBMIT FILE ERROR	The filename in the SUBMIT command line could not be found or was not a TEXT file.
BAD FILE TYPE	The file specified for LOAD was not a binary file.
UNABLE TO CHAIN: filename.ext	This error message indicates that a CHAIN request was made to the CS-OS system with filename.ext but cannot be executed (no such file, disk read error, file not the right type, etc.).

The next three error messages result from use of the RENAME command:

RENAMING ERROR	Indicates a disk error in accessing the drive containing the old file.
DUPLICATE NAME	Indicates that the new name already exists on the disk.
FILE PROTECTED	Indicates that the old file is protected from renaming (access code = 01 or 02).

The next 3 messages come from the RUN command:

NO TRANSFER ADDRESS	The binary load module specified in the RUN command does not have a transfer address.
NO MORE TASKS MAY BE STARTED	There is no PCB available to start a task with the RUN command. All PCBs are in use.
DUPLICATE PROCESS NAME	There is already a PCB with the menu specified in the RUN command.
SUBMIT FILE ERROR	An error occurred while trying to open the requested SUBMIT file.
COMMAND FORMAT ERROR	The syntax of a system command is incorrect.
#PR ERROR	The printer is not able to perform a DIR /L function.

The following error messages are related to diskette operations.

DISKETTE WRITE PROTECTED	A write operation was attempted to a write protected diskette.
#FD0X NOT READY	The requested drive is not ready. Place a diskette in the drive and close the drive door.
XXXX ERROR: #FD0X, CMD=XX, TRK=XX, SEC=XX, STA=XX, SEL=XX	This is a disk error message RVOL ERROR = Error reading volume label READ ERROR = Error in sector read WRIT ERROR = Error in sector write WTRK ERROR = Error in track write SEEK ERROR = Error performing seek

The next error messages are from the SET command

SYNTAX ERROR
BAD PARAMETER
VALUE TOO LARGE -- Value entered is unreasonable for the type of command
VOLUME IDENTIFIER NOT MOUNTED
SET DEFAULT VOLUME ERROR
DRIVE NOT FOUND
SET DEFAULT DISKETTE ERROR

The next 4 messages are related to commands involving tasks (i.e., PRIority, DELAY, WAKEUP, RUN, KILL, SUSPEND, RESUME, SHOW)

NO SUCH TASK	There is no PCB with a task name that matches the one entered.
NOT ON READY QUEUE	A task cannot be KILLed if it is not on the ready queue
TASK HOLDS RESOURCES	A task cannot be KILLed if it has not released acquired resources.
INVALID PRIORITY	A priority outside the range 1-127 was used in a RUN or PRIORITY command.

A.4 MACRO ASSEMBLER ERROR MESSAGES

NUMBER	TYPE
0202	Opcode or label error
0204	Syntax error
0205	Label error
0206	Redefined symbol
0207	Undefined opcode
0208	Value range error
0210	Byte overflow
0211	Undefined symbol
0213	EQU or SET pseudo-op error
0216	Pseudo-op error
0220	Phasing error
0221	Symbol table overflow
0223	The pseudo-op cannot be labeled
0226	The MACRO pseudo-op is unlabeled
0227	ENDM pseudo-op cannot be labeled
0228	Macro table overflow
0230	Macro expansion line overflow
0251	Macro nesting error
0254	IF Stack overflow/underflow (nesting error)
0260	Missing END statement
0261	Illegal character in line

A.5 LINK ERROR MESSAGES

BAD FILE NAME	The file name given does not match the requirements of the operating system.
EOF OCCURRED IN RIGHT NIBBLE	The format of input files to LINK is ASCII. Each byte of machine executable code is therefore represented by two bytes. This means that somehow a nonhex character is in the second byte of an ASCII pair representing a HEX byte. This is a fatal error. Reassemble the program.
ERROR OCCURRED IN LIBRARY	An error occurred while opening a general library to search for entry points that would resolve external

	references. The name of the library will be printed.
ERROR OCCURRED OPENING FILE FROM FILE TABLE	The names of each file given as input are each opened one at a time for pass one of LINK. These names are saved in a table to prevent having to rename these files for pass two. An error occurred while trying to open one of these files to process it for pass two.
ERROR OCCURRED WHILE ATTEMPTING PASS2 IN LIBRARY	A general library was found to have modules that resolved previously unsatisfied external(s), but the library could not be opened for pass two.
FILE ALREADY EXISTS	The file specified for LINK output already exists.
FILE NOT FOUND	The file given does not exist in the disk directory.
ILLEGAL CHARACTER OCCURRED IN RIGHT NIBBLE	See EOF OCCURRED IN RIGHT NIBBLE
LIBRARY IS EMPTY	The library given contains no modules.
MAXIMUM NUMBER OF INPUT FILES REACHED	Link allows a maximum of twenty specific files to be entered.
MAXIMUM NUMBER OF LIBRARY FILES REACHED	Link allows a maximum of ten general libraries to be entered at one time.
MODULE NAME TOO LONG -- REENTER ENTIRE INPUT	The module name given as a specific module from a library can only be eight characters long. Reenter both library names and symbols.
MODULE NOT IN LIBRARY	A specific module of a library given as an input was not found in the library.
NO FILES GIVEN -- EXECUTION TERMINATED	At least one specific file must be given as an input to LINK.

NONEXISTING LIBRARY	File given as an input has the extension "LIB" but does not exist in the disk directory.
REDEFINED ENTRY POINT	Two or more of the modules given as input share a common entry point. The value of the first one encountered will be retained. Only a warning.
UNRESOLVED EXTERNAL REFERENCE	This is a warning message only. An external reference in one of the modules did not find a matching entry point.

A.6 LOCATE ERROR MESSAGES

BAD NUMBER -- REENTER	The base address, common base address or transfer address that was entered was either illegal hex or illegal decimal.
ERROR OCCURRED FORMATTING THE OUTPUT FILE	LOCATE forms the output file by opening a file with the same filename as the input but with the the extension "BIN". An error occurred while attempting to format or open this file. Note: This is a system failure.
UNRESOLVED EXTERNAL REFERENCE REMAINING:	The file being bound to absolute addresses contains an external that was never resolved. The external is replaced by an address of zero. This is a warning message only.

A.7 LIBRARY ERROR MESSAGES

BAD FILENAME	Filename given does not fulfill the the requirements of the system.

BAD MODULE NAME	The module name given to be deleted is not eight characters long or less, containing an alphabetic character following alphanumeric ones.
DELETE REQUIRES MODULE NAME AND LIBRARY NAME	Illegal syntax in DELETE command. Check command syntax.
DELETE REQUIRES "FROM" AND LIBRARY NAME	Illegal syntax in DELETE command. Check command syntax.
ERROR DELETING OLD VERSION OF LIBRARY	An error occurred while trying to delete the old version. The old version name (name.LIB) should be present as well as the new name (name.TMP). Note: This is a system failure.

Note: ADD and DELETE make temporary copies of the library to be modified. Once the modification is finished, the original version of the library is deleted and the temporary file is renamed to the original name.

FILE ALREADY EXISTS	An attempt was made to CREATE a library that already exists.
FILE NOT FOUND -- "FROM" REQUIRED	File given as input does not exist. LIB expected an already existing file Illegal syntax in DELETE command. Check command syntax
ILLEGAL MODULE NAME	The module name given to be DELETED is not eight characters or less with alphanumeric characters following an alphabetic one.
ILLEGAL SYNTAX -- "TO" REQUIRED	The ADD command requires the keyword "TO".
LIBRARIES MUST BE CREATED BY "CREATE" COMMAND	Attempt to DELETE or ADD to a library that has an illegal internal format. Possibly a library created by editing.
LIBRARY IS EMPTY	Library given as a parameter in a command contains no modules.
LIBRARY MODULE CONTAINS NO NAME	Error in the internal format of the library given. It is not empty,

	but it contains no module name.
MODULE ALREADY EXISTS IN THE LIBRARY	An attempt was made to ADD a module to a library that already contains a module with the same name.
NO FILE NAME GIVEN	No file name given where one was required. Check command syntax formats for required inputs.
NO NAME FOUND IN MODULE TO BE ADDED	The internal format of the module to be added did not contain a module name. Reassemble the source that produced the input.
NO PROGRAM DELIMITED FOUND IN LIBRARY	Library has faulty internal format. Create and modify libraries exclusively through LIB commands.
REQUIRES BOTH A MODULE NAME AND A LIBRARY NAME	Illegal syntax in ADD command. Check command syntax.
REQUIRES LIBRARY NAME	A library name is a required parameter in both ADD and DELETE commands. Check command syntax.

A.8 GRAPHICS ERROR CODES

0	No error
2	Window not open
3	Axis not equal to 1 or 2
4	Mode not supported
5	NPTS exceeds maximum
6	NPTS<=0
7	Device window exceeds device space
8	Window dimensions incorrectly specified
9	WINNUM exceeds limit
10	Device number not supported
12	Font not supported
14	LINSTYLE not supported
15	Text length exceeds range
16	*
17	*
18	*
19	*
20	Line exceeds window; clipping will be performed

21 Text exceeds window; clipping will be performed
22 *
23 NIPTS<0
24 Mark not supported
25 Mark exceeds window; clipping will be performed
26 Fill area exceeds window; clipping will be performed
27 POS exceeds window; clipping will be performed
28 MAG<1
29 Window already open
30 No window control block space remaining
31 Zero block cannot be filled
32 XOR out of range

GC22-9199

READER'S COMMENT FORM

This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate.

IBM Instruments, Inc. shall have the nonexclusive right, in its discretion, to use and distribute all submitted information, in any form, for any and all purposes, without obligation of any kind to the submitter. Your interest is appreciated.

Note: *Copies of IBM Instruments, Inc. publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM Instruments, Inc. product to your IBM Instruments, Inc. representative or to the IBM Instruments, Inc. office serving your locality.*

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefullness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A.

GC22-9199

Reader's Comment Form

Please do not staple — Fold and Tape

First Class
Permit 40
Armonk
New York

Business Reply Mail

No postage stamp necessary if mailed in the U.S.A.

Postage will be paid by:

IBM Instruments, Inc.
P.O. Box 332
Danbury, Ct. 06810

Please do not staple — Fold and tape

IBM Instruments, Inc.
P.O. Box 332
Danbury, Ct. 06810