IBM Instruments Inc.

Computer System

Operating System Reference Manual

Part 1. Operating System

Release 1.1

GC22-9199-1

IBM Instruments Inc.

Computer System

Operating System Reference Manual

Part 1. Operating System

Release 1.1

GC22-9199-1

Second Edition (October 1983)

01101830

PREFACE

This manual describes the IBM Instruments Computer System 9000 Operating System (CSOS). It consists of five chapters and five appendixes.

- Chapter 1 -- "Introduction to the Operating System" -- describes the operating system and the system commands used with it.
- Chapter 2 -- "Full-Screen Editor" -- describes the main features of the text editor, including the various commands that can be used to create and edit source and text 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 and Library Utilities" -- describes the two programs available to the user for the development of modular code.
- Chapter 5 -- "Debug" -- describes the special debugging utility designed to run in the multitasking environment of CSOS.
- Appendix A -- "Error Messages" -- lists and defines the various error messages that may be generated in the operation of CSOS.
- Appendix B -- "Instruction Set Summary" -- lists the mnemonics, a description, and the syntax of every instruction recognized by the 68000 assembler.
- Appendix C -- "Character Set" -- summarizes the characters, numbers, and symbols recognized by the 68000 assembler.
- Appendix D -- "Sampler Assembler Output" -- lists an assembler program.
- Appendix E -- "Examples of Linked Assembly-Language Programs" -- lists an ALINK-screen dialog and an output map.

Related Publications:

Publications that discuss related aspects of the Computer System are:

Computer System Product Description, GC22-9183

Computer System Operating System Reference Manual Part 2: Assembly Programmer's Guide to Logical I/O and System Services, GC22-9200

Computer System BASIC Reference Manual, GC22-9184

Computer System FORTRAN Reference Manual, GC22-9194

Computer System PASCAL Reference Manual, GC22-9190

Computer System Problem Isolation Manual, GC22-9192

Additional References:

A good understanding of assembler language programming is assumed in much of this manual. There are many sources of information on the subject, whether formal classroom education or through use of books and other student material.

A knowledge of the facility of the 68000 microprocessor is also important in much of this manual, especially in understanding the instruction types and notation conventions which apply to this microprocessor. There are several books available on the 68000, two of which are:

- Motorola MC68000
  16-Bit Microprocessor User's Manual
  PRENTICE HALL, Inc., Englewood Cliffs, NJ, 1982

- 68000 Assembler Language Programming
  by Gerry Kane, Doug Hawkins, and Lance Leventhal
  OSBORNE/MCGRAW-HILL
  Berkeley, CA, 1981

CONTENTS

1.0 INTRODUCTION TO THE OPERATING SYSTEM

The IBM Instruments Computer System 9000 (CS 9000) has a disk-based multitasking operating system -- CSOS -- 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 CSOS. 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 CSOS 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). If the file has a ".BIN" extension, the extension need not be typed.

When no command is found resident or transient, the command is assumed to be SUBMIT and the entered data the submit file to be processed.

Where CSOS 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. If the user has a formatted hard-disk as drive 4, then the prompt will be "4>" at power on. Otherwise, the default prompt is "0>" for diskette systems.

1.1.2 NAMING FILES

A fully specified filename consists of three fields: A volume identifier, a filename and an extension. When filenames are specified in system commands, specific delimiters 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:>filename.ext

or

<drive:>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 and is always terminated by a colon. This references the volume mounted in the physical drive number specified. 0 to 3 correspond to #FD00 to #FD03, (diskette drives) and 4 to 7 correspond to #HD00 to #HD03 (hard-disk drives).
filename	is a field of one to eight alphanumeric characters with the leading character alphabetic. It is generally 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.TED

123456:BLUE.SRC   CLYDE:TESTCASE.BIN

VOL7:TERRYPGM.REL   POLLY:TEMPFILE.SRC

1.1.2.2 Entering Filenames

To specify a file, give the volume name or disk drive number, filename, and extension. The following are examples of unique files:

```
    MASTER:INPUT.TXT    1:INPUT.TXT    4:INPUT.HEX    0:INPUT2.TXT
```

The system maintains a default volume/drive. If a file is on the default drive, the volume name or drive number and colon may be omitted from the file specification. If the default drive is set to zero with the volume 'RAM2' mounted in the drive, the following file descriptors would be identical:

```
    0:MELS.BIN    MELS.BIN    RAM2:MELS.BIN
```

Using the SET command, the user may modify the default drive. Any drive in the system may become the default drive. The default volume would become whatever volume is mounted in 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)
JANEPROG	(file extension missing)

1.1.3 WILDCARD FEATURE

CSOS 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 its 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 that would be required 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??
```

NOTE: A trailing character after an asterisk will generate a syntax error.

1.1.4 CTRL-ALT-DEL FUNCTION

There may be times when the user wishes to restart the system without resetting it. CSOS uses a three-key sequence as a "warm start" mechanism: Ctrl-Alt-Del. This set of keystrokes causes a restart that prints the start-up banner, and readies the system for new commands. Any pending SUBMIT file is terminated, attached drives and devices become unattached, and the SPOOL queue becomes empty.

WARNING: This function should be used sparingly. If used while writing to disk, it may result in a corrupted disk.

1.1.5 TASKS AND MULTITASKING

A "task" (sometimes termed "process") is a program that is run under the control of CSOS. In fact, parts of CSOS 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 CSOS to gain access to system resources in a controlled manner. CSOS 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 in Part 2 of this manual (GC22-9200-1).

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 CSOS commands and actually constitutes the "system" with which the user interacts. The SYSTEM task begins running when CSOS is started. The user may issue commands to DELAY or change the priority of the SYSTEM task, but should do so with care.

There is a second predefined task in CSOS: 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 DELAY 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

<u>Each task has a priority value associated with it.</u> The priority is a number between 1 and 127. Within CSOS, 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 default priority of SYSTEM is 64, while the priority of "idle" is 0.

A maximum of thirteen tasks in addition to the SYSTEM task is allowed.

<u>Each task has an associated status byte.</u> 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
2 - Task is delayed on a timeout
3 - Task is blocked on I/O

4 - Task is suspended pending completion of an asynchronous
operation.
5 - Task is undergoing termination

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 CSOS 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 DELAY command and the DELAY system-call provide this facility. A DELAYed 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 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 RUNTASK command with an initial priority. The task priority can be changed at any time by using the PRIority command.

1.1.5.3 Multitasking in a Real-time Operating System

The attached figure shows a very simplified schematic of interactions between the Operating System, Input/Output device interrupts, and multiple tasks (either system utilities or user-written).

A 'task' can be thought of as a program **in memory** which is associated with a major 'job' being done. Tasks can be creating a display, moving data from memory to disk, doing some calculations, using the communication port, etc. A task may call in other programs as part of its own operation; may remove itself, may suspend itself for awhile, or start up another task. An application 'package', such as Chromatography, is a set of tasks, and are all loaded into memory (from disk) when the application is called for.

Each task that is to be a part of a multitask operation is given a loading memory address and a priority by the author; however, this priority can be changed while the application is running. It can even be assigned a priority when the task is started from the keyboard. Tasks that are to run on the computer simultaneously must be in different memory areas to avoid an overlay of programs, and possible system crash. (See the RUNTASK command and the GETPCB System Call (Part 2, System Calls)).

Each task in memory with the same highest priority is given equal time (50 milliseconds) by the operating system. This is accomplished by the SYSTEM CLOCK actually interrupting the processor every 50 milliseconds. The interrupt service routine gives control to the task switch routine in the operating system. This routine looks at the current priorities of all active tasks and gives control to the one with the highest priority (even if it was the one that was just interrupted). If several tasks have the same high priority as the interrupted one, this one is put on the bottom of the list, and the next task in the list is given control.

It would appear that a lower priority task would never get a chance, and that is so. However, if the higher priority tasks are 'waiting' for some input/output request to be finished, then these tasks are now on the 'wait' list, so the operating system gives control to the lower priority task.

Since almost all programs use Input or Output (to the disk, printer, RS-232, etc.) eventually all tasks get completed, even ones with low priority.

Be aware, however, that if you have a task that does no I/O (sometimes referred to as "CPU bound"), then you should not run it at a priority higher than the SYSTEM task, because the SYSTEM task will be locked out and you will not be able to issue commands (like KILL or PRI) to correct the situation.

The set of programs for handling hardware interrupts have a priority higher than that of any task. Hardware interrupts signal the occurrence of an event or the completion of a data transfer.

When a hardware interrupt occurs it forces the processor to stop whatever task is executing and to branch to one of 256 possible addresses (or programs). Interrupts can be 'masked' so that they are ignored until 'armed'. The interrupt programs are generally very short so that they are executed quickly and return control to the program which was interrupted (see diagram on page 1-8).

```
    Hardware
    Interrupts    (I/O Devices)
     |  |  |
     v  v  v
    ___________                                          ______
  |            |                                           |
->|  BRANCH    |-----------------0                          |
| |  TABLE     |        |  |  |                             |
| |____________|___     |  |  |                             |
|   |              |    |  |  |                           MEMORY
|   | INTERRUPT    |  <-   |  |                          (SYSTEM)
|  _| HANDLER &    |  <----   |                             |
| | | ROUTINES     |          |                             |
| | |______________|___       v                             |
| |   |               |                                     |
| |   |  OPERATING    |                                     |
|  -->|  SYSTEM       |                                     |
|     |  -----------  |                                     |
|     |  Task Switch  |-------------------0                 |
|     | (by priority) |           |  |  |                   |
|     |  -----------  |           |  |  |                   |
| --->|  I/O Request  |           |  |  |                   |
| |   |  -----------  |           |  |  |                   |
| |   |  I/O Drivers  |-----------|--|--|--> DEVICES        |
| |   |_______________|_____      |  |  |                   |
| |     |                   |  <--   |  |                   |
| |     | TASK 0/ <Priority>|        |  |                   |
| |     | System task       |        |  |                   |
| |     |                   |        |  |                   |
| |<----|                   |        |  |                   |
| |     |___________________|___     |  |                 ----
| |        |                    | <--   |                 MEMORY
| |        | TASK 1/ <Priority> |       |                 (User)
| |        | User task          |       |                   |
| |        |                    |       |                   |
| |<-------|                    |       v                   |
| |        |____________________|                           :
| |<-----------  __:_____:____:_______                      :
|   _________   |                    |                      :
|__|         |  | TASK N/ <Priority> |                      |
   |         |  | User task          |                      |
   | clock   |  |                    |                      |
   |_________|  |                    |                      V
                |____________________|                    ______
```

These hardware interrupts (if not masked) are also prioritized, so that even an interrupt service program can be interrupted by a higher priority interrupt. This does not happen too often, but the assignment of input/output interrupts takes some care.

The keyboard handling is an example of the relationship between interrupts and tasks. When a key, or a combination of keys, is depressed, the processor is interrupted and the interrupt service routine for the keyboard transfers the data from the keyboard to a RAM 'keyboard buffer' area used by the operating system. The interrupt service routine then returns to the system. A task, perhaps a system or a user task, waiting for some keyboard input, when next given a time period for execution, will then examine this buffer for the presence of characters. It will take appropriate action depending on the character(s) found there. The usual thing is to display what was typed, and then to execute the appropriate section of the program in memory.

1.1.6 SYSTEM MEMORY POOL

Most operating system functions require some amount of memory to temporarily save information. Memory for this purpose is drawn from a reserved system memory pool. If the system memory pool were not used, then each system function would have to reserve its temporary memory whether it was being used or not. In practice this means that large amounts of memory would be wasted. Since all temporary memory is generally not used at the same time, the reserved pool is made small so that more memory is available for the user. In some cases many system functions are required at the same time and the system memory pool is not large enough to accommodate all temporary memory requests. At such a time the user will typically receive the error message:

```
MEMORY NOT AVAILABLE
```

The recommended action is for you to increase the size of the system memory pool with the SET SM command (see Appendix F: SYSTEM MEMORY CONSUMPTION).

1.2 SYSTEM COMMANDS

1.2.1 USER TRANSIENTS

Any binary file (type 01) 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 followed by a JUMP 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 INPUT.TXT,MYVOL:OUTPUT.TXT
```

invokes the COPY program from drive 1 and specifies the input and output files to be used.

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 name of the device to be attached. The first character must be a '#' to indicate a device name. The next three characters must match the existing driver name. An error will occur if the driver does not support additional devices.

ERROR MESSAGES

```
SYNTAX ERROR
```

The device driver may produce unique error messages. See device driver documentation.

1.2.3 ATCHDRV (RESIDENT)

The ATCHDRV command "attaches" a device driver to the system by loading a binary image file (type 01) from disk and executing it. The file is assumed to contain device driver code. The command examines register D7.W on return for an indication of the driver's success at initialization. A nonzero value indicates failure. The format of the command is:

```
     ATCHDRV filename.ext
```

where 'filename.ext' is the name of the file containing the driver code. If no extension is specified in the filename, the command assumes '.DRV'.

ERROR MESSAGES

```
     SYNTAX ERROR
     INVALID LOAD FILE FORMAT
     LOAD ADDRESS TOO LOW
     NOT ENOUGH MEMORY TO LOAD
     NO FILE TRANSFER ADDRESS
     DUPLICATE PDB NAME
     DUPLICATE EDB NAME
```

The device driver being attached may produce unique error messages. See device driver documentation.

1.2.4 CLS (RESIDENT)

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

```
CLS
```

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 (see Part 2, Section 1.2.5):

```
#device
```

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

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

Directly accessing the disk drives through their device names (#FD00 through #FD03 or #HD00 through #HD03) is not accepted by the COPY command. Also, do not attempt to copy between two disks with the same volume identifiers. The operating system does not support duplicate volume identifiers mounted at the same time.

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 specified, the user is prompted before each file is copied. If both source and destination are files, the following options may be chosen:

C -- compare files. A byte-by-byte comparison is made between the two files specified. If a mismatch is found, the format of the output is:

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

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

D -- delete files without prompting. If a file already exists on the destination disk, it will be automatically overwritten.

M -- specify buffer size (M=XX). The default buffer size used is from the buffer start to the end of user memory (APPEND). The M option specifies the number of 1K-byte (i.e., 1024-byte) blocks of memory to be used. The buffer start address is $2000 past the program load address. (This can be found by typing LOAD COPY.) If COPY loads at $E000, then the buffer start address is $10000. If the user typed option M=50, then the buffer end address would be $1C800 (50 x 1024 = 51200 = $C800).

S -- create contiguous file. The output file created in a file to file copy will be a contiguous file.

V -- verify destination file. File sectors are read in from the source disk and then written to the destination disk. To verify the file copy, the sectors are read back from the destination file, and compared with the original file sector data that has been preserved in memory. If a mismatch is encountered, an error message is printed, and the destination file is deleted.

Y -- copy files without prompting. If copying using wildcards, all files will be copied without selective user interaction. If the file already exists on the destination disk, the user will still be prompted, unless the 'D' option was also specified.

The following options are used only when outputting to a device.

L -- append linefeed. If a record has a carriage return as a delimiter, a linefeed will be written following it.

T -- truncate record. Maximum record lengths of 132 characters, will be truncated at 79 characters followed by a carriage return.

When the source of data is a device, the following keyboard control character is accepted:

Ctrl/D -- end of input

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.

The following short-hand notations are accepted by the COPY command for device names:

'#' as a source name gets input from the '#CON' device.
'#' as a destination name outputs to the '#SCRN0' device.

Note: Do not use '#CNSLO' as a destination device. This is a three-line window at the bottom of the screen used for echoing command input and displaying error messages.

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
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 the user for each file with the extension .SMP on the default volume (here it was RAMVOL). When the user replies 'Y', the file is copied to the destination volume (TEST), with the extension .SRC. If the file already exists on the destination disk, the user is asked whether it should be overwritten. In this example, each file is verified after it has been copied. The abbreviations used are: Y = Yes, N = No, Q = Quit.

EXAMPLES

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

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

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

transfers characters typed in from the keyboard, to the device connected to the serial port #SER00. The terminating character is CTRL-D.

```
COPY #,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 (#CON), until an end-of-file character (CTRL-D) is typed.

ERROR MESSAGES

SYNTAX ERROR
ILLEGAL DEVICE NAME

ILLEGAL FILE SPECIFICATION	
CANNOT HAVE AMBIGUOUS FILENAME	- Ambiguous names only accepted in a file to file copy
FILENAMES MUST BE SAME FORMAT	- Ambiguous characters must be in the same positions in the filenames specified
ILLEGAL DEVICE FOR COPY PROGRAM	- Cannot use disk device names
SOURCE MUST BE TEXT FILE	- Only text files can be copied to a device
END OF MEMORY ADDRESS ERROR	- An error occurred in accessing the end of user memory, which is used to calculate available buffer space.
SOURCE AND DESTINATION ARE THE SAME	- the destination file would overwrite the source file. This is assumed to be a user error since it serves no useful purpose.
SOURCE FILE IS EMPTY	
ERROR IN VERIFY - FILE NOT COPIED	
FILE SIZES ARE NOT THE SAME	
COMPARE ABORTED AFTER 10 MISMATCHES	

1.2.6 DELAY (RESIDENT)

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

```
DELAY taskname[,time slices]
```

where 'taskname' is the name of a task currently in the system. The 'time slice' parameter is optional; its default value is $FFFFFFFF. (This is a delay measured in years, effectively "forever.") The time slice count may be any numeric value. The duration of a time slice is normally 50 milliseconds but shortens with increased I/O activity. Hence, DELAY is not reliable as a timing mechanism. The Real-Time Manager should be used for applications requiring a truly accurate timer mechanism. (see Part 2 of the Operating System Reference Manual.)

EXAMPLES

```
DELAY SYSTEM,200        (200 time slices)
DELAY TASK1             ("forever")
```

See the RESUME command.

ERROR MESSAGES

```
SYNTAX ERROR
NO SUCH TASK
```

1.2.7 DELETE (RESIDENT)

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

where 'volume' is the disk drive number or the volume identifier. If omitted, the default is used. This command will only delete those files whose access code indicate they are deletable. (See the SECURE command for access code information. The filename and extension fields may contain wildcard characters.

Following a DIR command, the system will "ask" the user to confirm the command file by file. A 'Y' response deletes the file. The 'Q' response ends the delete prompting. Any other input is interpreted as a NO, and the file is not deleted.

<u>EXAMPLE SEQUENCE</u>

```
DELETE 1:TEST?.TXT

 DELETE (Y/N/Q) MYVOL:TEST1.TXT ? N

 DELETE (Y/N/Q) MYVOL:TEST2.TXT ? Y

 DELETE (Y/N/Q) MYVOL:TEST3.TXT ? Y
```

drive 1 has three TXT files named TEST1, TEST2, and TEST3. The following command sequence removes files TEST2.TXT and TEST3.TXT but not TEST1.TXT.

<u>ERROR MESSAGES</u>

```
SYNTAX ERROR
FILE NOT FOUND
```

1.2.8 DIAG (RESIDENT)

The DIAG command invokes the diagnostics of the Computer System. To use the diagnostics refer to "PROBLEM ISOLATION MANUAL" (GC22-9192).

1.2.9 DIR (RESIDENT/TRANSIENT)

The DIR command provides a list of the files on a specified volume. The format of the command is:

```
DIR [[volume:][filename.ext[,device]][;options]
```

where 'volume' is the disk drive number or the volume identifier. If omitted, the default is used. The 'filename' and 'ext' fields may contain wildcard characters. The 'device' may be any legal device name except for any disk devices. The default device is the page 0 screen (#SCRNO). In order to use options with the DIR command, the file DIR.BIN must be on the disk in the default drive. The following options are accepted:

E - extended directory information
S - alphabetically sorted directory listing

The directory listing has the following format:

```
NAME     NAME     NAME     NAME
```

unless the 'E' option is requested. For the 'E' option, the directory has the following format:

```
NAME     TYPE CODE     ACCESS CODE     LENGTH     CREATED     REVISED
```

The type and access codes are in hexadecimal. The length is the number of bytes in the file in decimal. Created and revised are the dates the file was created and last revised.

Note: You can stop the displayed directory from scrolling by pressing Ctrl-Numlock. Press any other key to resume. To quit the listing press Ctrl-Break.

<u>EXAMPLES</u>

```
DIR 1:
```

will list the entire directory of the volume in drive 1.

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

will list on the console all files from drive 1 that have the extension HEX.

```
DIR TEST?.*,#PR
```

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

ERROR MESSAGES

```
SYNTAX ERROR
FILE NOT FOUND
DIR.BIN NEEDED ON DEFAULT DRIVE TO PROCESS OPTIONS
E OPTION NOT SUPPORTED FOR OLD-STRUCTURE DISK
```

1.2.10 DISKCOPY (TRANSIENT)

The DISKCOPY program copies the contents of the source disk to the destination disk. The sectors are copied, starting with logical sector 0, and copying through the total number of sectors per disk, including the volume identifier. Both source and destination disk must have the same physical format, (e.g., sector size and number of sectors per disk). The format of the command is:

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

where source and destination are drive numbers (0-3 correspond to #FD00-#FD03 and 4-7 correspond to #HD00-#HD03), or they may be omitted. In this case the user is prompted for source, destination, and desired options. Legal options are:

C -- compare disk. A byte-by-byte comparison is made between the two disk devices specified. If a mismatch is found, the format of the output is:

```
LOGICAL                 DISK 1    DISK 2
SECTOR     OFFSET        BYTE      BYTE
------     ------       ------    ------
xxxxxxxx    xxxx          xx        xx
```

All values printed are in hexadecimal. If ten or more mismatches are encountered, an error message is printed and the disk comparison is terminated.

V -- verify destination disk. Sectors are read from the source disk and then written to the destination disk. To verify the disk copy, the sectors are read back from the destination disk, and compared with the original sector data that has been preserved in memory. If a mismatch is encountered, an error message is printed and the disk copy/verify is terminated.

R -- recover sector. If a CRC error is found on the source disk, whatever data that was read in, is copied to the destination disk. This allows the capability to recover partial data from bad sectors. The format of the output messages is:

```
CRC READ ERROR IN LOGICAL SECTOR $********
```

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

```
DISKCOPY
ENTER SOURCE DRIVE NUMBER: 0
ENTER DESTINATION DRIVE NUMBER: 1
ENTER OPTIONS: V
WARNING:  DESTINATION DISK CONTENTS
          WILL BE DESTROYED
COPY FROM DRIVE 0 TO DRIVE 1 ? Y
CURRENT DESTINATION VOLUME IDENTIFIER IS:  OPSYS
DO YOU WANT TO CHANGE THE VOLUME IDENTIFIER (Y/N)?  Y
ENTER NEW VOLUME IDENTIFIER:  MASTER
DISKCOPY COMPLETED
```

After the disk copy has completed successfully, the user is given the opportunity to change the destination disk volume identifier, (this is only available for diskette copies). If the volume identifiers of both the source and destination diskettes remain the same, a warning message is printed indicating that one diskette should be removed. The operating system does not support duplicate volume identifiers mounted at the same time.

For a system with only one diskette drive, DISKCOPY prompts the user to change from source diskette to destination diskette in the drive specified. The program utilizes the maximum amount of user 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.

ERROR MESSAGES

```
SYNTAX ERROR
DISK FORMATS DO NOT MATCH
END OF MEMORY ADDRESS ERROR
INVALID DRIVE NUMBER
CHANGE VOLUME IDENTIFIER ERROR
VERIFY ERROR:  LOGICAL SECTOR ********
COMPARE ABORTED AFTER 10 MISMATCHES
```

1.2.11 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.

<u>ERROR MESSAGES</u>

SYNTAX ERROR

The device driver may produce unique error messages. See device driver documentation.

1.2.12 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 device name of the driver. All control blocks associated with the device driver are disabled.

If the driver supports multiple devices, the devices should be detached via the DTCHDEV command before attempting to detach the driver.

<u>ERROR MESSAGES</u>

SYNTAX ERROR

The device driver may produce unique error messages. See device driver documentation.

1.2.13 FORMAT (TRANSIENT)

The FORMAT program is used to format and initialize a diskette or hard disk, so it may be used to store and access files with the Computer System operating system. The format of the command is:

```
FORMAT
```

The user is prompted for all required information. A carriage return may be entered where default values are specified.

PROGRAM FUNCTIONS:

(1) Format and initialize floppy disk.

(2) Format and initialize 10MB hard disk.

(3) Change Volume Identifier (floppy disk only).

(Q) Quit

DRIVE NUMBER:

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

Hard Disk drive numbers are: 4,5,6, or 7

FORMATS SUPPORTED:

10MB Hard Disk, Sector Size 256
8 inch, Double sided, double density diskette, sector size 256
5¼-inch, Double sided, double density diskette, sector size 256

Note: The drive type is determined by the operating system.

VOLUME IDENTIFIER:

One to six digits or letters without imbedded blanks; e.g., TEST5

Notes:
1. Prohibited volume names: '0' through '9'
2. Volume Identifier must be unique, it should not already exist on another disk.

INTERLEAVE FACTOR:

This is used as a fixed value when the physical sector numbers are incremented on the disk. The disk performance is a function of the interleave value. The defaults specified reflect the optimum sector interleave. Accepted values are:

10MB hard disk -- 2 to 32 (default=9)

8-inch diskette -- 1 to 13 (default=13)
5 1/4-inch diskette -- 1 to 8 (default=3)

BAD SECTOR LIST:	(Hard disk only). Each hard disk comes with a list of known bad sectors from the manufacturer. These logical sector numbers should be entered one at a time. They are stored in a bad sector table on the hard disk, and are locked out of the sector allocation scheme used by the file structure.

The user will be prompted before the format begins. For a diskette, a 'Y' (yes) or 'N' (no) will be accepted. Since a large amount of data could be lost by unintentionally formatting a hard disk, the user must type 'YES' twice before the disk will be formatted.

The FORMAT program executes two passes that are visible to the user. Each pass displays the logical sector number being accessed. (To maintain readability, the numbers are only displayed after a multiple number of sectors have been processed.) The first pass formats all tracks of the disk, and initializes sectors with information necessary for the file structure. The second pass reads all sectors on the disk as a form of verifying the integrity of the disk. If an error occurs in reading a sector, a retry is performed. If the read fails again, an error message is printed and the process is terminated for diskette formatting. With the hard disk, the sector number is put into the bad sector table that is stored on disk, and sector verification continues to completion. However, if an error occurs in logical sector 0, the disk cannot be used. This sector contains descriptive information about the disk, and must be error free.

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 disk was last formatted.

ERROR MESSAGES

OPEN ERROR
CONFIGURATION ERROR
WRITE SECTOR ERROR
HARD DISK CONTROLLER NOT PRESENT
DRIVE NOT READY
RESTORE ERROR
SEEK TRACK ERROR
FORMAT TRACK ERROR
LSN 0 IS DEFECTIVE
BAD SECTOR TABLE FULL
STATUS =$**** (**** = error code in hexadecimal)

1.2.14 FREE (TRANSIENT)

The FREE program displays the status of disk storage (e.g., available, used). The format of the command is:

```
          FREE [drive number]
```

where drive number is 0-7. If omitted, the default drive is used.

<u>EXAMPLES</u>

```
FREE 1
Volume:  RAMVOL
Total Free                 Sectors: 1967   Bytes: 503552   ( 51%)
Largest Free Contiguous    Sectors: 1958   Bytes: 501248   ( 51%)
Total Used                 Sectors: 1881   Bytes: 481536   ( 49%)
Total on Volume            Sectors: 3848   Bytes: 985088
```

<u>ERROR MESSAGES</u>

```
INVALID DRIVE NUMBER
VOLUME NOT FILE SYSTEM LEVEL 2
ERROR OPENING ______.  ERROR CODE = ______
ERROR READING VOLUME LABEL.  ERROR CODE = ______
ERROR READING BITMAP.  ERROR CODE = ______
```

1.2.15 HELP (TRANSIENT)

The HELP program displays a list of CSOS commands on the console screen. The format of the command is:

```
HELP
```

1.2.16 JUMP (RESIDENT)

The JUMP command allows the user to leave CSOS and go to any arbitrary absolute address. This command should be used with care; ANY number will be accepted by the command.

EXAMPLES

```
        JUMP $E112
```

will go to the address E112 (hexadecimal).

```
        JUMP 256
```

will go to the address 256 (decimal).

ERROR MESSAGES

BAD PARAMETER

1.2.17 KILL (RESIDENT)

The KILL command removes tasks from the system. The format of the command is:

```
        KILL taskname
```

where 'taskname' is the name of a task currently in the system (except SYSTEM). If the SYSTEM task were to be KILLed, none of the CSOS commands described in this manual would be available.

ERROR MESSAGES

```
SYSTEM TASK CANNOT BE KILLED
NO SUCH TASK.
```

1.2.18 LIST (SUBMIT FILE)

The LIST command will display an ASCII text file on the screen. This command is a submit file that calls the file COPY.BIN from the default volume. The format of the command is:

```
LIST filename.ext
```

where 'filename.ext' is the name of a text file. If the user has a common extension name that he wishes to default to, he could edit the file LIST.SUB. For example, the following submit file assumes the extension .SRC for all text files to be listed:

```
COPY &0.SRC,#;T
N
```

1.2.19 LISTDEV (RESIDENT)

The LISTDEV command lists all devices currently known to the system. The format of the command is:

```
        LISTDEV
```

If an alternate device identifier is supported (e.g. volume identifier), its six-character name is displayed, along with the device attribute byte in hexadecimal.

Device Attributes:

```
  bit 7:  1=Non sharable device/only 1 outstanding OPEN
      6:  1=driver supports asynchronous EVENT Posting
      5:  1=driver supports Byte I/O requests
      4:  1=driver supports Asynchronous requests (AREAD, AWRITE)
      3:  1=driver supports Alternate Device ID (Volume Identifier)
      2:  Reserved
      1:  1=driver supports multiple devices
      0:  1=driver is reentrant
```

Note: Device attributes of hexadecimal 21 is bits 0, 4 and 5 on.

<u>EXAMPLES</u>

```
LISTDEV
      #SCRN0          21      #SCRN1          21      #CNSL0          21
      #CON            30      #KPD            30      #PR             B0
      #GR             01      #SER00          30      #SER01          30
      #SER02          30      #PPU            B0      #FD02           18
      #FD00   TEST    18      #FD01   RAM404  18
      #FD03           18
```

1.2.20 LOAD (RESIDENT)

The LOAD command transfers a program from disk to the RAM area and prints out the transfer address. The program is not executed; control returns to CSOS command level. LOAD requires that the file be binary type (00 or 01 type). The format of the command is:

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

where 'volume' is the disk drive number or the volume identifier. If omitted, the default drive is used. No wildcard characters are permitted in the filename or extension. If the extension is omitted, the default '.BIN' is used. When a program has successfully been loaded, the transfer address will be displayed, if file type 01.

```
LOAD 1:PROG1
```

loads PROG1.BIN from drive 1, into user memory.

<u>ERROR MESSAGES</u>

```
FILE NOT FOUND
INVALID LOAD FILE FORMAT
LOAD ADDRESS TOO LOW
NOT ENOUGH MEMORY FOR LOAD
```

1.2.21 PAGE (RESIDENT)

The PAGE command switches (toggles) the displayed pages of graphics memory. If Page 0 is displayed, then the PAGE command will cause Page 1 to be displayed and visa versa. The format of the command is:

```
PAGE
```

1.2.22 PRI (RESIDENT)

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

```
       PRI taskname,priority
```

where 'taskname' is the name of a task currently in the system, and 'priority' is the new priority. Priorities can be in the range decimal 1 to 127. See section on multitasking (section 1.1.5.3) for a description of priority. (The SYSTEM tasks priority can be changed, too).

<u>ERROR MESSAGES</u>

SYNTAX ERROR
INVALID PRIORITY
NO SUCH TASK

1.2.23 RAMDIAGS (TRANSIENT)

The RAMDIAGS program invokes the diagnostics of the Computer System. To use the diagnostics refer to PROBLEM ISOLATION MANUAL (GC22-9192).

1.2.24 RENAME (RESIDENT)

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

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

where 'volume' is the disk drive number or the volume identifier. If omitted, the default drive is used. No wildcard characters are permitted in either the new or old names or extensions. The access code of the file must be one that allows renaming (see the SECURE command for access-code information). Also, newname.newext cannot be the name of a file already on the disk.

EXAMPLES

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

This renames the file DAVES.OLD to DAVES.NEW.

ERROR MESSAGES

DUPLICATE NAME
SYNTAX ERROR
FILE PROTECTED

1.2.25 REPAIR (TRANSIENT)

The REPAIR program enables the user to display the contents of a disk sector, and edit the data in the sector using a full-screen format. This utility will only work on disks formatted under the Release 1.1 Operating System. The format of the command is:

```
REPAIR
```

The following sub-commands may be given:

#	is the number of a disk sector. If the number is preceded by a '$', it is interpreted as a hexadecimal number, otherwise it is interpreted as a decimal number. NOTE: For information on disk structure, see Appendix E of the Operating System Reference Manual, Part 2 (GC22-9200).
E	enables the user to edit the sector currently being displayed. The cursor moves to the data area. The cursor keys (up, down, left, right) may be used to move around the hex and ASCII data areas. If changes are made, the corresponding data in a buffer of the sector is changed. Pressing Return will update the ASCII area to reflect changes made in the hex area, and update the hex area to update changes made in the ASCII area. If the user is entering data and reaches the end of a hex or ASCII data line, the cursor will wrap around to the beginning of the next line of the same data type. Pressing the ESC key terminates editing the sector. If any changes were made, the user is prompted to update the sector on disk or ignore the changes.
F	displays the next sector of the disk.
B	displays the previous sector of the disk.
Q	quits program.
H	prints the sub-commands available.

To understand the data fields within a sector that you may wish to observe or edit, see Part 2, Appendix E.

ERROR MESSAGES

```
ERROR OPENING ****** (device name)
SECTOR NUMBER OUT OF BOUNDS
ERROR IN READING SECTOR *****
ERROR IN WRITING SECTOR *****
STATUS = $**** (error code in hexadecimal)
```

1.2.26 RESUME (RESIDENT)

The RESUME command immediately places a DELAYed task on 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 a task currently in the system.

<u>ERROR MESSAGES</u>

NO SUCH TASK

1.2.27 RUNTASK (RESIDENT)

The RUNTASK command starts a task, from the command line. The format of the command is:

```
        RUNTASK filename.ext,priority
```

where 'filename.ext' is the name of a binary program that has a transfer address (type 01). If no extension is given, the default '.BIN.' is used. 'Priority' is the number indicating the position of the task on the READY queue. It can range from 1 to 127. After the task is loaded in memory (see Note 1) and has been placed on the READY queue, control returns to the system. A maximum of 13 tasks, in addition to the SYSTEM task, is allowed.

<u>ERROR MESSAGES</u>

```
SYNTAX ERROR
INVALID PRIORITY
NO TRANSFER ADDRESS
NO MORE TASKS MAY BE STARTED
DUPLICATE TASK NAME
```

The '.BIN' files supplied on the system diskette are not to be executed via RUNTASK. They are to be executed from the SYSTEM task by simply typing their names: COPY, ASM, FORMAT, BASIC, ... etc.

NOTES:

1. Tasks that are to run on the computer simultaneously must be in different memory areas to avoid an overlay of programs, and possible system crash.

2. While tasks are being run by RUNTASK, no transient commands (e.g.,FORMAT, COPY) can be executed. These commands load a binary file into memory that may overlay a running tasks program.

3. Programs written in assembler must use the EXIT system call (see Operating System Reference Manual Part 2) instead of the RTS instruction to exit.

1.2.28 SAVE (RESIDENT)

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

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

where 'volume' is the disk drive number or the volume identifier. If omitted, the default is used. The filetype of the save file will be 00 if no transfer address is present, and 01 if a transfer address is supplied. No wildcard characters are permitted in the filename or the extension. If the file already exists on the disk, it will be overwritten without warning. If the extension specified is .SYS the file created will be contiguous rather than the default of extendable. All addresses can be entered in decimal or hexadecimal notation.

EXAMPLES

```
SAVE 1:USER.BIN,$9000,$97FF,$9100
```

this saves 2048 bytes of memory as a binary file to be entered at the address 9100 hexadecimal.

```
SAVE BASEPAGE.SAV,0,255
```

saves the first 256 bytes of memory.

ERROR MESSAGES

SYNTAX ERROR
INVALID FILE SPECIFICATION
INVALID TRANSFER ADDRESS - the address specified is not on an even boundary or it is not between the start and end addresses specified.
ENDING ADDRESS IS TOO LOW OR TOO HIGH - the address specified is below the starting address or it is beyond the end of system memory.

1.2.29 SECURE (RESIDENT)

The SECURE command changes a files access code (see below), which determines the files security. The code permits protection of certain files from change, deletion, or renaming. The format of the command is:

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

where the 'volume' is the disk drive number or the volume identifier. If omitted, the default drive is used. No wildcard characters are permitted in either the filename or extension. The access codes defined in CSOS 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.

To determine a files current access code, use the DIR command with E option.

EXAMPLES

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

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

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

protects the file INIT.CMD from being renamed.

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

allows INIT.CMD to be renamed but not deleted.

ERROR MESSAGES

```
SYNTAX ERROR
INVALID FILE SPECIFICATION
DIR.DIR FILE ACCESS CANNOT BE CHANGED
VALUE RANGE ERROR
```

1.2.30 SET (RESIDENT)

The SET command allows the user to control certain system characteristics. The format of the command is:

```
        SET parameter[=value]
```

where 'parameter' is one of the two-letter mnemonics shown below. If a 'value' is specified, the system parameter will be set to it, otherwise the current system value will be displayed in decimal. Parameters accept values specified as either decimal or hexadecimal.

DD -- disk drive	-- sets or displays the physical drive number to be used as the default when the drive number or volume identifier are omitted in a file specification. A formatted disk must be mounted in the drive. If 'DD' is set to drive 0, the default volume identifier will always be updated to whatever volume is mounted in drive 0. The 'SET DD' command accepts a drive number or a volume identifier as input, however the default volume is not maintained if the disk is moved to another drive. The default volume always remains associated with the physical drive.
EC -- error code	-- sets or displays the mode of the status code in error messages printed by the system calls PRTERR and DPRTERR (see Part 2). If set to 'Y' (Yes), the error code will always be printed. If set to 'N' (No), the error code is only printed when there is no message text.
LD -- line depth	-- sets or displays the number of lines per page for the line printer. The default value is 60 lines. Accepted values are 1 through 32767.
LW -- line width	-- sets or displays the number of characters per line for the line printer. The default value is 80 characters. The maximum value is 132.
SM -- system memory	-- sets the amount of memory to be added to (value is positive), or removed from (value is negative) the system memory pool. After it has been successfully set, or if no value is specified, the amount of system memory is displayed in 1K byte blocks. (See Section 1.1.6, System Memory Consumption.)

TS -- tab settings -- sets or displays the tab stops to be used for the console. The default value is 10 columns. The maximum value is 79.

EXAMPLES

SET DD = 2	set default to drive 2, update default volume identifier
SET DD	displays default drive number and volume identifier
SET EC = Y	include error code in error messages
SET LW = 132	set printer line width to 132 characters
SET SM = 10	increase system memory pool by 10 pages -- (10 x 1024 bytes)

ERROR MESSAGES

SYNTAX ERROR
BAD PARAMETER
ILLEGAL VALUE - a nonnumeric or invalid value was entered.

1.2.31 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
```

where 'taskname' is the name of a task in the system. The system searches the list of open devices for that taskname, and prints out the logical unit number and device name for each entry in the list.

EXAMPLES

SHOW SYSTEM

```
   00246 #CON        00249 #SCRN0        00250 #CNSL0        00251 #SCRN1
```

shows the opened devices for the system task.

ERROR MESSAGES

NO SUCH TASK

1.2.32 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 does not exist, an error will result. The format of the command is:

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

where 'volume' is the disk drive number or the volume identifier. If omitted, the default is used. No wildcard characters are permitted in the filename or extension.

EXAMPLES

```
SPOOL SYSVOL:TEXTFILE.TXT
```

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 invoked 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).

Notes: 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 100 ($64).

1.2.33 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
```

For more information on spooler commands, see SPOOL.

1.2.34 SPOOLQ (RESIDENT)

The SPOOLQ command displays status information about the spooling task on the console screen. The format of the command is:

```
SPOOLQ
```

The information displayed includes:

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

For more information on spooler commands, see SPOOL.

1.2.35 SUBMIT (RESIDENT)

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

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

where the 'volume' is the disk drive number or volume identifier. If omitted the default drive is used. All commands from the file will attempt to execute regardless of previous command errors. However, a submit file can be aborted by pressing Ctrl-Break (the next submit file commands will not be executed).

No wildcard characters are permitted in the filename or extension of the SUBMIT file. If the extension is omitted, the default '.SUB' is used. 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, following the first one. 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. A command line, with parameters substituted, must be 80 characters or less.

ERROR MESSAGE

```
SYNTAX ERROR
SUBMIT FILE ERROR
WRONG FILE TYPE
```

1.2.36 SYSLEVEL (TRANSIENT)

The SYSLEVEL program displays the release level and version number of the operating system. The format of the command is:

```
     SYSLEVEL
```

<u>EXAMPLES</u>

```
SYSLEVEL

RELEASE LEVEL 01 VERSION 01 INTERNAL LEVEL 10
```

1.2.37 SYSMAP (TRANSIENT)

The SYSMAP program displays the current bounds of available memory on your machine. The format of the command is:

```
SYSMAP
```

- "Start of User Memory" is the lowest address available for you to load programs.
- "End of User Memory" is the highest RAM address available for your programs to use.
- "End of Ram" is the highest RAM address available on your machine.
- "Total System Memory Pool" is the number of bytes in the system memory pool for use by device drivers and the file system.
- "Total Free Memory" is the current amount of memory from the system memory pool that is not in use.
- "Largest Contiguous Block" is the current largest available contiguous block of memory in the system memory pool.

<u>EXAMPLES</u>

```
SYSMAP

START OF USER MEMORY  $00028800  TOTAL SYSTEM MEMORY POOL  $00005000
END OF USER MEMORY    $0011ABFE  TOTAL FREE MEMORY         $00004D28
END OF RAM            $0011FFFE  LARGEST CONTIGUOUS BLOCK  $00004D28
```

1.2.38 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. There will be a single line of console output for each task in the system (except "idle"). The task name, priority, time of generation and status will be printed. The priority is output in decimal notation. The status of a task in the system can be one of the following:

Ready - task is eligible to run
Delayed - task is delayed for a certain number of time slices
Blocked - task is waiting for a device to service an I/O request
Suspended - task is waiting for completion of asynchronous I/O
Terminating - task is undergoing termination

EXAMPLES

```
     TASKS

     NAME     PRIORITY       CREATE TIME          STATUS

   SYSTEM        064      07 NOV 83 01:30:04      READY
   TASK1         010      07 NOV 83 01:44:03      BLOCKED
```

1.2.39 TIME (RESIDENT)

The TIME command displays or sets the date and current time of day. The format of the command is:

```
TIME[=dwk dm mon yr hh:mm:ss]
```

where 'TIME' alone displays the time, and when followed by parameters, sets the time. Time of day is in 24-hour notation. The day of the week (dwk) and month (mon) are represented by three-letter abbreviations. The other time elements are represented by numbers. If an incorrect syntax or an invalid TIME parameter is encountered, the current TIME will be printed, along with the appropriate error message.

The TIME value is preset upon delivery. The system keeps proper elapsed time by use of a battery-operated clock. When files are created under CSOS, the current time value is stored.

<u>EXAMPLES</u>

```
TIME
SAT 25 DEC 82 12:00:00
```

is the time at noon on Christmas day 1982.

<u>ERROR MESSAGES</u>

BAD PARAMETER

1.2.40 WHEREIS (TRANSIENT)

The WHEREIS program displays the load address of a file that has a .BIN extension. The format of the command is:

```
     WHEREIS filename
```

Where 'filename' is the name of a binary program that has a transfer address (type 01). The extension of the file is assumed to be .BIN.

<u>EXAMPLE</u>

```
     WHEREIS COPY

Load address is $00E000
```

<u>ERROR MESSAGES</u>

```
CAN'T OPEN FILE 'filename'.BIN
CAN'T READ FILE
```

2.0 COMPUTER SYSTEM TEXT EDITOR

2.1 INTRODUCTION

ED is a full-screen editor for the IBM Instruments Computer System. It enables you to create and edit source and text files. The editing features include global and interactive search and replace commands, block functions, and a file merging command.

ED runs on the Computer System with at least 128K of RAM. ED can only edit text files created under the Computer System operating system. Files with OBJ or BIN extensions cannot be edited. The maximum line length for a file line is 133 characters and the maximum number of lines is 1000.

2.2 GENERAL INFORMATION

The cursor used by ED is a solid rectangle. It is always displayed in the reverse video mode (inverse of normal) from the video mode of the line in which it is positioned.

Marked lines and the command line are displayed in inverse video. All other lines are displayed in normal video.

Commands are entered on the command line and may be executed from the command line or from the data area. Functions are assigned to individual keys and are executed by pressing the key. Some functions are not applicable to the command line, while other functions have different effects if executed when the cursor is on the command line or in the data area.

2.3 INVOKING AND EXITING ED

You invoke ED by typing ED on the Computer System command line. Once ED has been loaded, it issues the prompt "enter a filename or q to quit:". At this point, the diskette containing ED can be removed from its drive.

To edit a file, type in the filename. ED will search for the file. If it is found, the first 1000 lines of the file are read into storage and the first 22 lines are displayed. If ED cannot find the file, it assumes that a new file is to be created and displays an empty screen. In either case, you can now edit the file.

When you are done editing the file, the File and Quit commands will bring back the initial filename prompt. You can edit a new file (or the same file again) by entering its name or type "q" or "Q" to exit from ED.

2.4 STORAGE MANAGEMENT

Storage is restored before a file is edited. While a file is being edited, ED attempts to conserve and reuse storage, but it is possible to deplete storage. ED constantly monitors the available storage and issues a warning when it is low. At this point you must File or Quit to recover storage. Commands and functions that require storage (such as Get) might not work or might leave partial results. If you do not File or Quit, ED will eventually Quit for you to avoid a heap overflow condition. Any changes in the file since the last File or Save will be lost.

2.5 SCREEN FORMAT

ED divides the screen into four areas:

- Data area (lines 1-22)

  ED displays the current window into the file in the data area. Each line of the data area displays data from a line of the file. ED uses the entire width of the screen (either 40 or 80 characters) for data display.

  If a file line is longer than the screen width, only a segment is displayed. Long file lines do not wrap across more than one line in the data area. You must move the viewing window to see different segments of long file lines. The window may be shifted up, down, left, or right by functions described later.

- Command line (line 23, inverse video)

  You enter ED commands by moving the cursor down to the command line (see command toggle function) and typing the command. The editing functions described later may be used to edit commands on the command line.

- Status area (lines 24 and 25)

  The status area contains several fields. ED uses the first line of the status area to give you information about the file. On this line are the filename, the current position in the file, and the current value for the insert toggle.

  There are a pair of numbers giving the coordinates of the current position in the file. The first number gives the line and the second number gives the displacement from the beginning of the line, starting at one for the first position of the line. ED indicates the current position in the file in another way as well. If the cursor is in the data area, it is always at the current position in the current file. If the cursor is on the command line, the current position in the file is displayed in the reverse video mode of the line that it is in. The highlighted character is called the toggle character.

  The possible values for the insert toggle are "Insert" and "Replace." The setting of the insert toggle affects what happens when text from the keyboard is entered into the data area or the command line of the display.

  The second line of the status area is used for messages that ED sends to you as a result of commands and functions. This line is also used to display internal error codes when ED detects an error in itself. To see how internal errors are displayed, invoke the [error] function by typing A-E (ALT E).

2.6 KEYBOARD LAYOUT

ED attaches special significance to many of the 83 keys on the keyboard. The names of these keys as referred to in this document and their locations on the keyboard are given below.

- Shift keys. The SHIFT, ALT, and CTRL keys together comprise the shift keys. The description of a shifted version of a key will be prefixed by a one letter code followed by a dash. The possible prefixes are "S-" for SHIFT, "A-" for ALT, and "C-" for CTRL. For example, to indicate that the "TAB" key should be depressed while holding down the alphabetic shift key, we will say "S-TAB".

- Numeric keypad keys. The keys on the numeric keypad will always be used for their control functions, not as numbers. The names of these keys (and their numeric labels) are: INS (0), END (1), DOWN (2), PGDN (3), LEFT (4), RIGHT (6), HOME (7), UP (8), and PGUP (9).

- Delete key. The key to the right of INS is the delete key, written as DEL.
- Return key. The return key is written as RET.
- Backspace key. The key above RET labeled with a back arrow is the backspace key, written as BACKSPACE.
- Tab key. The key to the left of Q is the tab key, written as TAB.
- Escape key. The escape key is located above TAB and is written as ESC.
- Function keys. The ten keys along the left edge of the keyboard are the function keys. They are referred to as F1, F2, ..., F10.

2.7 FUNCTIONS

Most of the text manipulation that you will do using ED will involve use of the functions described in this chapter. Except as noted, functions may be used to edit the command line as well as text in the data area of the display.

Functions' names are lowercase character strings, possibly containing blanks, enclosed in square brackets ([]).

Functions can be divided into several categories, depending on how they influence the file and the display.

Cursor movement functions change only the display; they do not change the file being edited in any way. If the action indicated by the description of a cursor movement function would make the cursor move off the screen, ED will redraw the screen in a function dependent way so that the requested position in the file or command line is displayed. No function can ever take the cursor outside of the bounds of the file. Attempts to do so will move the cursor as far as possible in the requested direction.

Status change functions alter controlling parameters of ED. All of the status change functions affect the setting of the insert toggle.

File modification functions change the line containing the cursor in some way. If you use any of them while the cursor is in the data area of the file, the file becomes "dirty," and the Quit command will not allow you to discard the contents of the file without confirmation. ED considers the file dirty even if a change is subsequently undone. Changing a command on the command line does not cause the current file to become dirty.

Block functions in ED use the concept of marks and operations on a marked block of a file. There can be at most one **marked block.** A marked block is defined by issuing one or two **mark** line functions. Either the beginning or ending mark of a block may be entered first.

The presence of a marked block does not influence the operation of any non-block function or command. In particular, you may add or delete characters or lines inside of a marked block. All of the block functions in ED operate only on the data area of the display; most of them do nothing if invoked while the cursor is on the command line.

BACKTAB WORD FUNCTION

Key: S-TAB

Purpose: The [backtab word] function moves the cursor to the left, stopping at the first character in the previous word.

A word is defined to be a group of non-blank characters separated by blanks. If necessary, ED redraws the display so that the new position of the cursor is on the screen.

See also [tab word].

BEGIN LINE FUNCTION

Key: HOME

Purpose: The [begin line] function moves the cursor to the first column of the current line.

If the first column of the current line is not on the display, ED redraws the screen or command line so that the first column of the line appears in the first column of the screen. If the cursor is already in the first column of the current line, [begin line] does nothing.

See also [end line].

BEGIN MARK FUNCTION

Key: A-1

Purpose: The [begin mark] function moves the cursor to the beginning of the marked block.

ED redraws the screen if necessary so that the beginning of the marked block is on the display. [begin mark] will not change the cursor column; it just jumps vertically to the row containing the first line of the marked block, in the same column as the cursor or the toggle character (if the cursor was on the command line).

See also [end mark].

BOTTOM FUNCTION

Key: C-END

Purpose: The [bottom] function moves the cursor to the last line of the file, without changing its column.

When [bottom] is invoked, ED redraws the display so that the last line of the file is displayed on the last row of the data area of the screen.

If the last line of the file is already on the last row of the data area, only the cursor is moved.

See also [top].

BOTTOM EDGE FUNCTION

Key: C-PGDN

Purpose: The [bottom edge] function moves the cursor to the bottom row of the data area.

The cursor moves straight down and the contents of the screen do not change.

This function has no effect if the cursor is on the command line.

See also [top edge].

CENTER LINE FUNCTION

Key: F5

Purpose: The [center line] function redraws the screen with the line containing the cursor at the center line of the screen.

This function has no effect if the cursor is on the command line.

COMMAND TOGGLE FUNCTION

Key: ESC

Purpose: The [command toggle] function swaps the cursor from the data area to the command line or vice versa.

If the cursor was on the command line when [command toggle] is entered, it moves to the toggle character position in the data area. If the cursor was in the data area, it moves to the command line and ED highlights its former position in the data area by displaying the character in that position in the reverse video mode from its current video mode.

This is the only way to move the cursor onto the command line.

CONFIRM CHANGE FUNCTION

Key: C-BACKSPACE

Purpose: The [confirm change] function replaces the target string of an interactive Change command with its replacement string.

This function is only meaningful in the context of an interactive Change command. ED will display an error message if you try to use it at any time other than immediately after a successful interactive Change command.

See also the Change command in the next chapter.

COPY MARK FUNCTION

Key: A-Z

Purpose: The [copy mark] function copies a marked block to the position given by the cursor.

ED inserts a copy of the marked lines after the line containing the cursor. This function has no effect if the cursor is on the command line.

It is illegal to copy a marked area onto or into itself, and ED will not perform the attempted operation.

ED has no block move function. It may be simulated by a [copy mark] followed by a [delete mark].

See also [mark line].

DELETE CHAR FUNCTION

Key: DEL

Purpose: The [delete char] function deletes the character at the cursor and shifts the remaining characters on the line one position to the left to close up the gap.

You will probably find [delete char] useful mostly for fixing errors in existing text. You should use the [rubout] function for correcting mistakes immediately after text has been entered.

See also [insert toggle] and [rubout].

DELETE LINE FUNCTION

Key: S-F8

Purpose: The [delete line] function deletes the line containing the cursor.

ED shifts the bottom part of the data area up by one line to fill in the gap created. The cursor remains stationary.

This function has no effect if the cursor is on the command line.

See also [insert line].

DELETE MARK FUNCTION

Key: A-D

Purpose: The [delete mark] function deletes the lines in the marked block and unmarks the file. ED redraws the screen and relocates the cursor if part of the block was displayed.

See also [unmark] and [mark line].

DOWN FUNCTION

Key: DOWN

Purpose: The [down] function moves the cursor one position towards the bottom of the screen, without changing its column.

If you invoke [down] when the cursor is positioned on the last line of the data area, ED scrolls the screen up by one line, bringing a new line into view.

When the cursor is on the command line, [down] has a slightly different meaning. In this context, [down] scrolls the display up by one line, then moves the cursor to the new line of data uncovered. You must use the [command toggle] function to return the cursor to the command line.

If the cursor is already on the last line of the file when [down] is invoked, nothing happens.

See also [up] and [down4].

DOWN4 FUNCTION

Key: F2

Purpose: The [down4] function moves the cursor four positions towards the bottom of the screen, redrawing the screen if necessary.

If [down4] is invoked when the cursor is positioned in the bottom three lines of the data area of the screen or on the command line, ED moves the cursor to the bottom line of the data area and redraws the data so that the line number of the bottom line in the data area is four greater than the previous contents of that screen line. If the cursor is within three lines of the bottom of the file when [down4] is invoked, ED moves the cursor to the bottom line of the file, redrawing the screen as necessary.

See also [up4] and [down].

END LINE FUNCTION

Key: END

Purpose: The [end line] function moves the cursor to the end of the line.

If the last column of the current line is not on the display, ED redraws the file or command line so that the last column of the line is shown in approximately the middle of the screen. If the cursor is already at the end of the current line, [end line] does nothing.

Since the cursor may be past the end of the line when [end line] is invoked, it is possible that [end line] will move the cursor to the left.

See also [begin line].

END MARK FUNCTION

Key: A-2

Purpose: The [end mark] function moves the cursor to the end of the marked block.

ED redraws the screen if necessary so that the end of the marked block is on the display. [end mark] does not change the cursor column; it just jumps vertically to the row containing the last line of the marked block in the same column as the cursor or toggle character (if the cursor was on the command line).

See also [begin mark].

ERASE END LINE FUNCTION

Key: F6

Purpose: The [erase end line] function erases characters rightward from the cursor to the end of the line.

The new length of the line will be one less than the column number of the cursor.

The cursor does not move.

ERROR FUNCTION

Key: A-E

Purpose: The [error] function gives an example of the message ED produces when it detects an error in its own code.

If you get such an internal error message while you are using ED, try scrolling to a different page and then back to the point of the error, then retry the operation which caused the error message. If the problem persists, try to figure out the sequence of steps that is required to force ED to fail, then send a documented description of the error circumstances to IBM Instruments.

EXECUTE FUNCTION

Key: C-RET

Purpose: The [execute] function performs the command on the command line, even if the cursor is not currently on the command line.

The RET key always performs the [execute] function when the cursor is on the command line. The [execute] function is very useful in conjunction with the Change and Locate commands, to repeat the previous operation.

INSERT LINE FUNCTION

Key: F9

Purpose: The [insert line] function adds a new line to the current file following the line containing the cursor.

ED positions the cursor in the first column of the new line and shifts lines following the new line down to make room for the new line.

The only way to insert a line before the beginning of the file is to perform [split] when the cursor is positioned on the first character of the file.

This function has no effect if the cursor is on the command line.

See also [delete line] and [return].

INSERT TOGGLE FUNCTION

Key: INS

Purpose: The [insert toggle] function changes the setting of the insert toggle from Insert mode to Replace mode, or vice versa.

ED displays the current setting of the insert toggle as a field in the status area. The possible values for the insert toggle setting in the status area are 'Insert' and 'Replace'.

If the insert toggle is set to 'Replace,' the character you type replaces the character at the current cursor position, no other characters of the line are affected, and ED moves the cursor one position to the right.

If the insert toggle is set to 'Insert,' ED shifts the characters of the current line from the cursor to the end of the line right by one position, places the new character in the space this created at the cursor position, and moves the cursor one position to the right. Characters at the right edge of the screen are shifted off the screen.

Note: if the line containing the cursor is 133 characters long (maximum length), inserting a character will cause the last character in the line to be deleted.

The [return] function is also affected by the insert toggle.

See also [return].

JOIN FUNCTION

Key: A-J

Purpose: The [join] function appends the successor of the line containing the cursor to the cursor line.

The cursor may be located anywhere in the data area and does not move after you perform [join]. ED redraws the data area of the screen after [join] to close up the gap created when the next line is moved up onto the cursor line.

If you execute [join] immediately after [split], the display will not change.

If the combined length of the cursor line and its successor line is greater than 133 (maximum line length), [join] is not done. This could be due to blanks at the end of the lines.

This function has no effect if the cursor is on the command line or is on the last line of the file.

See also [split].

LEFT FUNCTION

Key: LEFT

Purpose: The [left] function moves the cursor one position to the left.

If the requested move would cause the cursor to move off the screen, ED redraws the display so that the new cursor position and some surrounding context appear on the screen. If the cursor is already in column one, [left] has no effect.

See also [right], [left8], and [left40].

LEFT8 FUNCTION

Key: F3

Purpose: The [left8] function moves the cursor eight positions to the left.

This function allows faster horizontal scrolling than the [left] function.

If the requested move would cause the cursor to move off the screen, ED redraws the display so that the new cursor position and some surrounding context appear on the screen. If the cursor is already in the first seven columns of the line, [left8] moves the cursor to column 1.

See also [right8] and [left].

LEFT40 FUNCTION

Key: C-LEFT

Purpose: The [left40] function moves the cursor forty positions to the left.

This function allows faster horizontal scrolling than the [left] and [left8] functions.

If the requested move would cause the cursor to move off the screen, ED redraws the display so that the new cursor position and some surrounding context appear on the screen. If the cursor is already in the first forty columns of the line, [left40] moves the cursor to column 1.

See also [right40], [left8], and [left].

MARK LINE FUNCTION

Key: A-L

Purpose: The [mark line] function sets a mark on the line containing the cursor.

If there are no marks in the file, ED redisplays the current line in inverse video. If this is the second line mark in the file, ED displays the cursor line and all lines between it and the other marked line in inverse video. One or more marked lines forms a marked block.

The cursor must be in the data area when you invoke [mark line]. It is an error to invoke [mark line] if there are already two marks outstanding.

The highlighting associated with line marks always spans an entire row of the screen. Thus, even though a line may be only 10 characters long, if the line is part of a block, ED will highlight the entire screen row used to display the line.

Blocks can be used to copy, delete, shift, and save sections of a file.

See also [unmark], [delete mark], [copy mark], [shift right], [shift left], and the Save command in the next section.

PAGE DOWN FUNCTION

Key: PGDN

Purpose: The [page down] function scrolls the display down to the next page.

ED redraws the display so that the 20 lines below the current screen contents (higher line numbers) are brought into view. If there are less than 20 lines below the line currently on the bottom row of the data area, ED displays the last 22 lines of the file.

The cursor is normally not moved by [page down]. This implies that the sequence [page down] followed by [page up] usually leaves the cursor in its original position.

See also [page up].

PAGE UP FUNCTION

Key: PGUP

Purpose: The [page up] function scrolls the display up to the previous page.

ED redraws the display so that the 20 lines above the current screen contents (lower line numbers) are brought into view. If there are less than 20 lines above the line currently on the top row of the data area, then ED displays the first 22 lines of the file.

The cursor is normally not moved by [page up]. This implies that the sequence [page up] followed by [page down] usually leaves the cursor in its original position.

See also [page down].

RETURN FUNCTION

Key: RET

Purpose: The [return] function executes a command if the cursor is on the command line and positions the cursor on the next line if the cursor is in the data area.

When the cursor is on the command line, [return] causes ED to interpret and perform the command on the command line. If the command is invalid, the cursor will not move; otherwise the effect depends upon the command. See the command section for information on specific commands.

When the cursor is in the data area, the effect of [return] depends upon the insert toggle. In "Replace" mode, [return] sets the cursor in column one of the line after the cursor line. ED will redraw the screen if this line or column is not on the screen. In "Insert" mode, [return] has the following result: a new line is inserted into the file after the cursor line and the cursor is moved under the first non-blank character in the cursor line.

RIGHT FUNCTION

Key: RIGHT

Purpose: The [right] function moves the cursor one position to the right.

If the requested move would cause the cursor to move off the screen, ED redraws the display so that the new cursor position and some surrounding context appear on the screen. If the cursor is already in column 133, [right] has no effect.

See also [left] and [right8].

RIGHT8 FUNCTION

Key: F4

Purpose: The [right8] function moves the cursor eight positions to the right.

This function allows faster horizontal scrolling than the [right] function.

If the requested move would cause the cursor to move off the screen, ED redraws the display so that the new cursor position and some surrounding context appear on the screen. If the cursor is near the maximum length of a line (columns 125 through 133) [right8] moves the cursor to column 133, the maximum allowable column.

See also [left8] and [right].

RIGHT40 FUNCTION

Key: C-RIGHT

Purpose: The [right40] function moves the cursor forty positions to the right.

This function allows faster horizontal scrolling than the [right] and [right8] functions.

If the requested move would cause the cursor to move off the screen, ED redraws the display so that the new cursor position and some surrounding context appear on the screen. If the cursor is near the maximum length of a line (columns 93 through 133) [right40] moves the cursor to column 133, the maximum allowable column.

See also [left40], [right8], and [right].

RUBOUT FUNCTION

Key: BACKSPACE

Purpose: The [rubout] function deletes the character to the left of the cursor and shifts the remaining characters on the current line one position to the left to close up the gap.

If the cursor is in column one, [rubout] will delete the first character on the line and shift the remaining characters left by one position.

You will probably find [rubout] useful mainly for correcting mistakes immediately after they have been typed. You should use the [delete char] function for correcting existing data.

See also [insert toggle] character and [delete char].

SHIFT LEFT FUNCTION

Key: F7

Purpose: The [shift left] function shifts the text in a marked block to the left.

ED writes the prompt "shift left - how many columns ? (0-133)". When you respond with the size of the shift, ED deletes that many characters from the beginning of each line in the marked block.

This function has no effect if the cursor is on the command line.

See also [shift right].

SHIFT RIGHT FUNCTION

Key: F8

Purpose: The [shift right] function shifts the text in a marked block to the right.

ED writes the prompt "shift right - how many columns? (0-133)". When you respond with the size of the shift, ED inserts that many blanks into each line of the marked block before the first character on the line.

This function is most useful for indenting program text.

Note that characters shifted past column 133 (maximum line length) will be deleted.

This function has no effect if the cursor is on the command line.

See also [shift left].

SPLIT FUNCTION

Key: A-S

Purpose: The [split] function divides the line containing the cursor at the cursor.

ED moves text at or following the cursor on the current line to the beginning of a new line which ED inserts into the file following the cursor line. The cursor does not move.

This function has no effect if the cursor is on the command line.

The only way to insert a line before the beginning of the file is to perform [split] when the cursor is positioned on the first character of the file.

See also [join].

TAB WORD FUNCTION

Key: TAB

Purpose: The [tab word] function moves the cursor to the right, stopping at the first character in the next word.

A word is defined to be a group of non-blank characters separated by blanks. If the cursor is at or beyond the end of the line, [tab word] does not move the cursor. If necessary, ED redraws the display so that the new position of the cursor is on the screen.

See also [backtab word].

TOP FUNCTION

key: C-HOME

Purpose: The [top] function moves the cursor to the first line of the file, without changing its column.

When [top] is invoked, ED redraws the display so that the first line of the file is displayed on the first row of the data area of the screen.

If the first line of the file is already on the first row of the data area, only the cursor is moved.

See also [bottom].

TOP EDGE FUNCTION

Key: C-PGUP

Purpose: The [top edge] function moves the cursor to the top row of the data area.

The cursor moves straight up and the contents of the screen do not change.

This function has no effect if the cursor is on the command line.

See also [bottom edge].

UNMARK FUNCTION

Key: A-U

Purpose: The [unmark] function removes any existing marks.

The text inside of the marked block is not affected by [unmark].

If a marked block was on the screen, ED removes the inverse video highlighting of the area.

You may execute [unmark] no matter where the cursor is located.

See also [mark line].

UP FUNCTION

Key: UP

Purpose: The [up] function moves the cursor one position towards the top of the screen.

Invoking [up] when the cursor is positioned at the top of the screen causes the cursor to remain stationary and the data on the screen to shift down by one line, bringing a new line into view. If the cursor is already at the top of the file when [up] is invoked, nothing happens.

If the cursor is on the command line when you invoke [up], it moves to the bottom line of the data area. You must use the [command toggle] function to return the cursor to the command line.

See also [down] and [up4].

UP4 FUNCTION

Key: F1

Purpose: The [up4] function moves the cursor four positions towards the top of the screen, redrawing the screen if necessary.

If [up4] is invoked when the cursor is positioned in the top three lines of the data area of the screen, ED moves the cursor to the top line of the data area and redraws the data on the display so that the line number of the top line in the data area is four less than the previous contents of that line. If the cursor is within three lines of the top of the file when [up4] is invoked, ED moves the cursor to the top line of the file, redrawing the screen as necessary.

See also [down4] and [up].

2.8 COMMAND REFERENCE

Commands are entered on the command line and may be up to 133 characters long. The first character of the command can be preceded by any number of blanks. Commands can be entered in upper or lower case. One or more blanks are required to separate the name of a command from any argument, except for the Change and Locate commands.

In all commands, extraneous information at the end of a command line is ignored.

In the command reference section that follows, the ED commands appear alphabetically. The description of each command includes its name, its format, a short description, then a more detailed description explaining the operation of the command and its optional parameters, as well as references to related commands.

In contrast to the function reference section of the last chapter, square brackets ([]) are used in this section to indicate optional arguments to commands.

CHANGE COMMAND

Format: C /pattern/newstring/[-][*]

Purpose: The Change command changes occurrences of a pattern string to a replacement string, either one occurrence at a time interactively or globally, starting next to the cursor position or the toggle character (if the cursor is on the command line).

For a global change, use the star (*) at the end of the command. Interactive change commands do not end with star. Global changes alter every occurrence of the target string. Interactive changes are more complicated.

When ED performs an interactive Change command, it locates the next occurrence of the target string and moves the cursor to it, but does not yet make the change. To actually change the target string to the replacement string, you must use the [confirm change] function (C-BACKSPACE). If you do not wish to change the particular occurrence of the target string that was found, just enter anything other than [confirm change].

The interactive Change command remains on the command line after it is executed, so that you may perform an [execute] function (C-RET) to retry the Change command and locate the next occurrence of the target string.

The character used to separate the pattern and replacement strings is the first non-blank one after the command name, as for the Locate command. You must enter all three occurrences of this delimiter.

The search always starts in the column next to the cursor position or toggle character. The optional minus sign in the Change command controls the direction of the search. If it is absent, the search direction is forward (towards higher numbered lines), while if the minus sign is present the search works backwards towards the top of the file.

ED ignores the case of the pattern and of the text in the file when searching. Thus, the pattern /MiXeD/ will find any of the strings "mixed", "MIXED", or "mIxEd". ED preserves the case of characters in the replacement string.

Note: If the first non-blank character on the command line is a 'C' or a 'c', ED will attempt to perform a change. Anything after the command is ignored.

Each entry in the following table is a legal Change command:

Command	Comments
c/xyz/abc/	Changes "xyz" to "abc" interactively with a forward search.
caxyzaa*1?	Deletes all occurrences of "xyz" (by changing them to the empty string) from the current file position to the end of the file. 1? is ignored.
C/xyz/XYZ/-*	Capitalizes all occurrences of "xyz" between the beginning of the file and the current position in the file.
c'/*'(*'*	Changes all occurrences of "/*" that occur between the current position in the file and the end of the file to "(*".

See also the Locate command.

FILE COMMAND

Format: File

Purpose: The File command writes a copy of the current file to disk using the name in the status area and removes it from memory.

See also the Name and Save commands.

GET COMMAND

Format: Get filename [(L1,L2)]

Purpose: The Get command copies part or all of another file into the file being edited. ED searches for the file with the specified filename. If the file is found, ED inserts it after the line containing the cursor or the toggle character (if the cursor is on the command line).

To get only part of a file, use the (L1, L2) option, replacing L1 with the line number of the first line to be copied and L2 with the line number of the last line to be copied.

If the specified file (or the requested part of it) is too long to fit into the file being edited, <u>none</u> of the file is inserted.

LOCATE COMMAND

Format: [L] /pattern[/[-]]

Purpose: The Locate command finds the next (or previous) occurrence of a pattern string in the current file, starting next to the cursor position or the toggle character (if the cursor is on the command line).

ED searches for the specified pattern and moves the cursor to the first place it is found, redrawing the display if necessary. The Locate command remains on the command line after it is executed, so that the [execute] function (C-RET) may be used to repeat the search without retyping the command.

You need to specify the name of the Locate command only if the search pattern contains the slash character (/). In this case, you must use some character other than slash to delimit the pattern. ED takes the first non-blank character after the command name to be the pattern delimiter. The second occurrence of this character terminates the pattern. The closing delimiter is optional unless the minus sign is specified or L is specified.

The search starts in the column next to the cursor position or toggle character. The optional minus sign in the Locate command controls the direction of the search. If it is absent, the search direction is forward (towards higher numbered lines), while if the minus sign is present the search works backwards towards the top of the file.

ED ignores the case of the pattern and of the text in the file when performing the search. Thus, the pattern /MiXeD/ will find any of the strings "mixed", "MIXED", or "mIxEd".

Note: If the first non-blank character on the command line is an 'L', 'l', or '/' ED will attempt to do a locate. Anything after the locate command is ignored if the command begins with 'L', else it could be part of the pattern.

Each entry in the following table is a legal Locate command:

Command	Comments
/xyz	Searches forward for "xyz" (assuming z is the end of the line)
/xyz/-	Searches backward for "xyz".
L'xyz'	Searches forward for "xyz".
L2/*2	Searches forward for "/*".
/xyz abc/	Searches forward for "xyz abc".
L-xyz--	Searches backward for "xyz".
L, '9', abc	Searches forward for " '9'", ignoring abc
/ '9' abc	Searches forward for " '9' abc" (and anything else after abc)

See also the Change command.

NAME COMMAND

Format: Name filename

Purpose: The NAME command assigns a new name to a file; that is, it saves an existing file under a new name. The "old" file is not lost, however, and can be retrieved under its original name. ED changes the filename given in the status area.

ED does no checking of the specified filename. If an invalid filename is assigned to a file with the Name command, the error will not be detected until an attempt is made to write out the file.

The Name command sets the indicator that records that the file has been changed.

See also the File and Save commands.

QUIT COMMAND

Format: Quit

Purpose: The Quit command terminates editing of a file and removes it from memory without writing it to disk.

If you have modified the file or changed its name since the last time it was saved to disk, ED writes the prompt "The file has been modified. Do you want to quit? (y/n)" in the status area. To confirm the quit, type "y" or "Y". To abort the quit, type "n" or "N".

If the file had not been modified, or after quit confirmation, ED destroys the copy of the file in memory and redraws the display to prompt for another file to edit.

SAVE COMMAND

Format: Save [filename] [(block)]

Purpose: The Save command writes a copy of the file to disk.

If you do not specify a filename, the Save command writes out the file to disk using the name in the status area. If you do give a filename, the file is written using that name. In either case, the file is not removed from memory.

If you specify the (block) option, the Save command writes out the lines in the current marked block (if there is one). The (block) option cannot be specified without a filename.

To obtain a printed listing of the block or file, you may "Save" it to the file #pr as:

```
save #pr [(block)]
```

The file modification indicator is not reset by the Save command unless Save is entered without a filename and without the (block) option. This is to prevent a later Quit command from releasing a file that has been "Saved" only to the printer or only partially saved.

See also the File and Name commands.

You should not save a file to the current screen (#SCRN0).

The following table lists all of the functions of ED, their default keys, and whether they can be invoked from the data area, command line, or both.

Function name	Where invoked	Default key
backtab word	D, C	S-TAB
begin line	D, C	HOME
begin mark	D, C	A-1
bottom	D, C	C-END
bottom edge	D	C-PGDN
command toggle	D, C	ESC
confirm change	D	C-BACKSPACE
copy mark	D	A-Z
delete char	D, C	DEL
delete line	D	S-F8
delete mark	D, C	A-D
down	D, C	DOWN
down4	D, C	F2
end line	D, C	END
end mark	D, C	A-2

D - can be invoked from data area
C - can be invoked from the command line

erase end line	D, C	F6
error	D, C	A-E
execute	D, C	C-RET
insert line	D	F9
insert toggle	D, C	INS (F10*)
join	D	A-J
left	D, C	LEFT
left8	D, C	F3
left40	D, C	C-LEFT
mark line	D	A-L
page down	D, C	PGDN
page up	D, C	PGUP
return	D, C	RET
right	D, C	RIGHT
right8	D, C	F4
right40	D, C	C-RIGHT
rubout	D, C	BACKSPACE
shift left	D	F7
shift right	D	F8

split	D	A-S
tab word	D, C	TAB
top	D, C	C-HOME
top edge	D	C-PGUP
unmark	D, C	A-U
up	D, C	UP
up4	D, C	F1

Commands:

```
C /pattern/newstring/[-][]*]
File
Get filename [(L1,L2)]
L /pattern/[-]
Name filename
Quit
Save [filename][(block)]
```

3.0 COMPUTER SYSTEM MACRO ASSEMBLER

3.1 SCOPE

The intent of this chapter is to provide sufficient information to develop 68000 assembly language programs which may be run on the Computer System. The information herein pertains to the elements of the assembler. Detailed information pertaining to the 68000 microprocessor is provided in various generally available publications.

3.1.1 INTRODUCTION

The Computer System Assembler is used to translate assembler source programs into a relocatable form suitable for further processing with the linker, ALINK. The assembler runs on the IBM Instruments Computer System Operating System.

The assembler includes the following features:

- Relocatable code generation
- Complex expressions
- Symbol table listing
- Macros
- Conditional assembly
- Structured syntax
- Cross-reference

3.1.2 ASSEMBLY LANGUAGE

The symbolic language used to code source programs for processing by the assembler is called 68000 assembly language. This language is composed of the following symbolic elements:

a. Symbolic names or labels, which represent instruction, directive, and register mnemonics, as well as user-defined memory labels and macros.

b. Numbers, which may be represented in binary, octal, decimal, or hexadecimal notation.

c. Arithmetic and logical operators, which are employed in complex expressions.

d. Special-purpose characters, which are used to denote certain operand syntax rules, macro functions, source line fields, and numeric bases.

3.1.2.1 Machine-Instruction Operation Codes

Appendix B summarizes that part of the assembly language that provides mnemonic machine-instruction operation codes for the 68000 machine instructions.

3.1.2.2 Directives

The assembly language contains mnemonic directives which specify auxiliary actions to be performed by the assembler. Directives are not always translated to machine language.

Assembler directives assist the programmer in controlling the assembler output, in defining data and symbols, and in allocating storage.

3.1.3 68000 ASSEMBLER

The assembler translates source statements written in the 68000 assembly language into relocatable object code, assigns storage locations to instructions and data, and performs auxiliary assembler actions designated by the programmer. Object modules produced by the assembler are compatible with the Computer System linker ALINK, referred to as the "linkage editor" or "linker."

The assembler includes macro and conditional assembly capabilities, and implements certain "structured" programming control constructs. The assembler generates relocatable code which may then be linked into a memory image format.

3.1.3.1 Assembler Purposes

The two basic purposes of the assembler are to:

- Provide the programmer with the means to translate source statements into relocatable object code -- that is, to the format required by the linker.

- Provide a printed listing containing the source language input, assembler object code, and additional information (such as error codes, if any) useful to the programmer.

3.1.3.2 Assembler Processing

Assembly is a two-pass process. During the first pass, the assembler develops a symbol table, associating user-defined labels with values and addresses. During the second pass, the translation from source language to machine language takes place, using the symbol table developed during pass 1. In pass 2, as each source line is processed in turn, the assembler generates appropriate object code and the assembly listing.

3.1.4 RELOCATION AND LINKAGE

"Relocation" refers to the process of binding a program to a set of memory locations at a time other than during the assembly process. For example, if subroutine "ABC" is to be used by many different programs, it is desirable to allow the subroutine to reside in any area of memory. ABC is assembled once, producing an object module which contains enough information so that another program (the linker) can easily assign a new set of memory locations to the module.

In addition to program relocation, the linkage editor must also resolve inter-program references. For example, the other programs that are to use subroutine ABC must contain a jump-to-subroutine instruction to ABC. However, since ABC is not assembled at the same time as the calling program, the assembler cannot put the address of the subroutine into the operand field of the subroutine call. The linkage editor, however, will know where the calling program resides and, therefore, can resolve the reference to the call to ABC. This process of resolving inter-program references is called "linking." An example of linking two object modules is shown in Appendix E.

3.1.5 NOTATION

Commands and other input/output (I/O) are presented in this manual in a modified Backus-Naur Form (BNF) syntax. Certain symbols in the syntax, where noted, are used in the real I/O; however, others are meta-symbols whose usage is restricted to the syntactic structure. These meta-symbols and their meanings are as follows:

< >	The angular brackets enclose a symbol, known as a syntactic variable, that is replaced in a command line by one of a class of symbols it represents.
|	This symbol indicates that a choice is to be made. One of several symbols, separated by this symbol, should be selected.
[ ]	Square brackets enclose a symbol that is optional. The enclosed symbol may occur zero or one time.
[ ]...	Square brackets followed by periods enclose a symbol that is optional/repetitive. The symbol may appear zero or more times.

Operator entries are to be followed by a carriage return.

3.2 SOURCE PROGRAM CODING

3.2.1 INTRODUCTION

A source program is a sequence of source statements arranged in a logical way to perform a predetermined task. Each source statement occupies a line of printable text, where each line may be one of the following:

a. Comment

b. Executable instruction

c. Assembler directive

d. Macro invocation

3.2.2 COMMENTS

Comments are strings, composed of any ASCII characters (refer to Appendix C), which are inserted into a program to identify or clarify the individual statements or program flow. Comments are included in the assembly listing but, otherwise, are ignored by the assembler.

A comment may be inserted in one of two ways:

1. At the beginning of a line, starting in column one, where an asterisk (*) is the first character in the line. The entire line is a comment, and an instruction or directive would not be recognized.

2. Following the operation and operand fields of an assembler instruction or directive, where it is preceded by at least one space.

EXAMPLES:

```
* THIS ENTIRE LINE IS A COMMENT.

BRA LAB2  THIS COMMENT FOLLOWS AN INSTRUCTION.
```

3.2.3 EXECUTABLE INSTRUCTION FORMAT

68000 assembly language programs are translated by the assembler into relocatable object code. This object code may contain executable instructions, data structures, and relocation information. This translation process begins with symbolic assembly language source code, which employs reserved mnemonics, special symbols, and user-defined labels. 68000 assembly language is line-oriented.

3.2.3.1 Source Line Format

Each source statement has an overall format that is some combination of the following four fields:

1. label

2. operation

3. operand

4. comment

The statement lines in the source file must not be numbered. The assembler will prefix each line with a sequential number, up to four decimal digits.

The format of each line of source code is described in the following paragraphs.

3.2.3.2 Label Field

The label field is the first field in the source line. A label which begins in the first column of the line may be terminated by either a space or a colon. A label may be preceded by one or more spaces, provided it is then terminated by a colon. In neither case is the colon a part of the label.

Labels are allowed on all instructions and assembler directives which define data structures. For such operations, the label is defined with a value equal to the location counter for the instruction or directive, including a designation for the program section in which the definition appears.

Labels are required on the assembler directives which define symbol values (SET, EQU, REG). For these directives, the label is defined with a value (and for SET and EQU, a program section designation) corresponding to the expression in the operand field.

Labels on MACRO definitions are saved as the mnemonic by which that macro is subsequently invoked. No memory address is associated with such labels. A label is also required on the IDNT directive. This label is passed on to the relocatable object module; it has no associated internal value.

No other directives allow labels.

Labels which are the only field in the source line will be defined equal to the current location counter value and program section.

3.2.3.3 Operation Field

The operation field follows the label field and is separated from it by at least one space. Entries in the field would fall under one of the following categories:

a. Instruction mnemonics - which correspond to the 68000 instruction set.

b. Directive mnemonics - pseudo-operation codes for controlling the assembly process.

c. Macro calls - invocations of previously-described macros.

The size of the data field affected by an instruction is determined by the data size code. Some instructions and directives can operate on more than one data size. For these operations, the data size code must be specified or a default size of word (16-bit data) will be assumed. The size code need not be specified if only one data size is permitted by the operation. The data size code is specified by appending a period (.) to the operation field, followed by B, W, or L, where:

B = Byte (8-bit data)
W = Word (the default size; 16-bit data)
L = Long word (32-bit data)

The data size code is not permitted, however, when the instruction or directive does not have a data size attribute.

Examples (legal):

LEA	2(A0),A1	Long word size is assumed (.B,.W not allowed); this instruction loads effective address of first operand into A1.
ADD.B	ADDR,D0	This instruction adds bytes whose address is ADDR to low order byte in D0.
ADD	D1,D2	This instruction adds low order word of D1 to low order word of D2. (W is the default size code.)
ADD.L	A3,D3	This instruction adds entire 32-bit (long word) contents of A3 to D3.

Example (illegal):

SUBA.B #5,A1	Illegal size specification (.B not allowed on SUBA). This instruction would have attempted to subtract the value 5 from the low order byte of A1; byte operations on address registers are not allowed.

3.2.3.4 Operand Field

If present, the operand field follows the operation field and is separated from the operation field by at least one space. When two or more operand subfields appear within a statement, they must be separated by a comma but may not contain embedded spaces; e.g., D1, D2 is illegal. In an instruction like ' ADD D1,D2' the first subfield (D1) is generally applied to the second subfield (D2) and the results placed in the second subfield. Thus, the contents of D1 are added to the contents of D2 and the result is saved in register D2. In the instruction ' MOVE D1,D2' the first subfield (D1) is the sending field and the second subfield (D2) is the receiving field. In other words, for most two-operand instructions, the general format 'opcode source, destination' applies.

3.2.3.5 Comment Field

The last field of a source statement is an optional comment field. This field is ignored by the assembler except for being included in the listing. The comment field is separated from the operand field (or the operation field, if there is no operand) by one or more spaces, and may consist of any ASCII characters. This field is important in documenting the operation of a program.

3.2.4 ARITHMETIC OPERATIONS

The 68000 instruction set includes the operations of add, subtract, multiply, and divide. Add and subtract are available for all data operand sizes. Multiply and divide may be signed or unsigned. Operations on decimal data (BCD) include add, subtract, and negate. The general form is:

```
    <operation>.<size>    <source>,<destination>
```

Examples:

ADD.W	D1,D2	Adds low order word of D1 to low order word of D2.
SUB.B	#5,(A1)	Subtracts value 5 from byte whose address is contained in A1.

3.2.5 MOVE INSTRUCTION

The MOVE instruction is used to move data between registers and/or memory. The general form is:

```
    MOVE.<size>     <source>,<destination>
```

Examples:

MOVE	D1,D2	Moves low order word of D1 into low order word of D2.
MOVE.L	XYZ,DEF	Moves long word addressed by XYZ into long word addressed by DEF.
MOVE.W	#'A',ABC	Moves word with value of $4100 into word addressed by ABC.
MOVE	ADDR,A3	Moves word addressed by ADDR into low order word of A3.

3.2.6 COMPARE AND CHECK INSTRUCTIONS

The general formats of the compare and check instructions are:

```
    CMP.<size>    <operand 1 >,<operand 2 >

    CHK          <bounds>,<register>
```

where operand 1 is compared to operand 2 by the subtraction of operand 1 from operand 2 without altering operand 1 or operand 2.

Condition codes resulting from the execution of the compare instruction are set so that a "less than" condition means that operand 2 is less than operand 1, and "greater than" means that operand 2 is greater than operand 1.

The CHK instruction will cause a system trap if the register contents are less than zero or greater than the value specified by "bounds."

Examples:

CMP.L	ADDR,D1	Compares long word at location ADDR with contents of D1, setting condition codes accordingly.
CHK	(A0),D3	Compares word whose address is in A0 with low order word of D3; if check fails (see text), a system trap is initiated.

3.2.7 LOGICAL OPERATIONS

Logical operations include AND, OR, EXCLUSIVE OR, NOT, and two logical test operations. These functions may be done between registers, between registers and memory, or with immediate source operands. The general form is:

```
<operation>.<size>    <source>,<destination>
```

Example:

AND	D1,D2	Low order word of D2 received logical 'and' of low order words in D1 and D2.

The destination may also be the status register (SR).

3.2.8 SHIFT OPERATIONS

Shift operations include arithmetic and logical shifts, as well as rotate and rotate with extend. All shift operations may be either fixed with the shift count in an immediate field or variable with the count in a register. Shifts in memory of a single bit position left or right may also be done. The general form is:

```
<operation>.<size>    <count>,<operand>
```

Examples:

LSL.W	#5,D3	Performs a left, logical shift of low order word of D3 by 5 bits; .W is optional (default).
ASR	(A2)	Performs a right, arithmetic shift of word whose address is contained in A2; since this is a memory operand, the shift is only 1 bit.
ROXL.B	D3,D2	Performs a left rotation with extend bit of low order byte of D2; shift count is contained in D3.

3.2.9 BIT OPERATIONS

Bit operations allow test and modify combinations for single bits in either an 8-bit operand for memory destinations or a 32-bit operand for data register destinations. The bit number may be fixed or variable. The general form is:

```
    <operation>     <bitno>,<operand>
```

Examples:

BCLR	#3,XYZ(A3)	clears bit number 3 in byte whose address is given by address in A3 plus displacement of XYZ.
BCHG	D1,D2	Tests a bit in D2, reflects its value in condition code Z, and then changes value of that bit; bit number is specified in D1.

3.2.10 CONDITIONAL OPERATIONS

Condition codes can be used to set and clear data bytes. The general form is:

```
    Scc     <location>
```

where "cc" may be one of the following condition codes:

CC or HS	GE	LS	PL
CS or LO	GT	LT	T
EQ	HI	MI	VC
F	LE	NE	VS

Example:

```
    SNE     (A5)+        If condition code "NE" (not equal) is true, then
                         set byte whose address is in A5 to 1's;
                         otherwise, set that byte to 0's; increment A5 by
                         1.
```

3.2.11 BRANCH OPERATIONS

Branch operations include an unconditional branch, a branch to subroutine, and 14 conditional branch instructions. The general form is:

```
    <operation>.<extent>    <location>
```

Examples:

```
    BRA     TAG          Unconditional branch to the address TAG.

    BSR     SUBDO        Branch to subroutine SUBDO.

    Bcc.S   NEXT         Short branch to NEXT, on condition "cc," which
                         may be one of the following condition codes
                         (note that T and F are not valid condition codes
                         for conditional branch):

                  CC or HS  GT     LT     VC
                  CS or LO  HI     MI     VS
                  EQ        LE     NE
                  GE        LS     PL
```

All conditional branch instructions are PC-relative addressing only, and may be either one- or two-word instructions. The corresponding displacement ranges are:

```
    one-word   -128...+127 bytes      (8-bit displacement)
    two-word   -32768...+32767 bytes  (16-bit displacement)
```

Forward references in branch instructions will use the longer format by default (OPT BRL). The default may be changes to the shorter format by specifying OPT BRS. The default extent may be overridden for a single branch operation by appending an "S" or "L" extent code to the instruction -- for example:

```
    BRA.L    LAB
```

A branch instruction with a byte displacement must not reference the statement which immediately follows it. This would result in an 8-bit displacement value of 0, which is recognized by the assembler as an error condition.

Example (illegal):

```
          BEQ.S  LAB1    LAB1 is the next memory word and, thus generates
     LAB1 MOVE    1,D0,   an error.
```

3.2.12 JUMP OPERATIONS

Jump operations include a jump to subroutine and an unconditional jump. The general form is:

```
     <operation>.<extent>    <location>
```

Examples:

```
JMP      4(A7)       Unconditional jump to the location 4 bytes beyond the
                     address in A7.

     JMP.L    NEXT       Long (absolute) jump to the address NEXT.

     JSR      SUBDO      Jump to subroutine SUBDO.
```

Forward references to a label will use the long absolute address format by default (OPT FRL). The default may be changed to the shorter format by specifying OPT FRS. The default extent may be overridden on a single jump operation to a label by appending "S" or "L" as an extent code for the instruction.

3.2.13 DBCC INSTRUCTION

This instruction is a looping primitive of three parameters: condition, data register and label. The instruction first tests the condition to determine if the termination condition for the loop has been met and, if so, no operation is performed. If the termination condition is not true, the data register is decremented by one. If the result is -1, execution continues with the next instruction. If the result is not equal to -1, execution continues at the indicated location. Label must be within 16-bit displacement. The general format of the instruction is:

```
    DBcc    <data register>,<label>
```

where "cc" may be one of the following condition codes:

CC or HS	GE	LS	PL
CS or LO	GT	LT	T
EQ	HI	MI	VC
F	LE	NE	VS

Examples:

```
    LAB1    NOP
            DBGT  D0,LAB1
            DBLE  D1,LAB2
            DBT   D2,LAB1
            DBF   D3,LAB2
    LAB2
```

3.2.14 LOAD/STORE MULTIPLE REGISTERS

This instruction allows the loading and storing of multiple registers. Its general format is:

```
    MOVEM.<size> <registers>, <location> (register to memory)

    MOVEM.<size> <location>, <registers> (memory to register)
```

where size may be either W (default) or L.

The <registers> operand may assume any combination of the following:

R1/R2/R3, etc., means R1 and R2 and R3

R1-R3, etc., means R1 through R3

When specifying a register range, A and D registers cannot be mixed; e.g., A0-A5 is legal, but A0-D0 is not.

The order in which the registers are processed is independent of the order in which they are specified in the source line; rather, the order of register processing is fixed by the instruction format.

Examples:

MOVEM (A6)+,D1/D5/D7	Load registers D1, D5 and D7 from three consecutive (sign-extended)

	words in memory, the first of which is given by the address in A6; A6 is incremented by 2 after each transfer.
MOVEM.L A2-A6,-(A7)	Store registers A2 through A6 in five consecutive long words in memory; A7 is decremented by 4 (because of .L0; A6 is stored at the address in A7; A7 is decremented by 4; A5 is stored at the address in A7, etc.
MOVEM (A7)+,A1-A3/D1-D3	Loads registers D1, D2, D3, A1, A2, A3 in order from the siz consecutive (sign-extended) words in memory, starting with address in A7 and incrementing A7 by 2 at each step.
MOVEM.L A1/A2/A3,REGSAVE	Store registers A1, A2, A3 in three consecutive long words starting with the location labeled REGSAVE.

3.2.15 LOAD EFFECTIVE ADDRESS

This instruction allows computation and loading of the effective address into an address register. The general format is:

```
    LEA    <operand>,<register>
```

Example:

LEA XYZ(A2,D5),A1	Load A1 with effective address specified by first operand; see later explanation of addressing mode "address register indirect with index".

3.3 SYMBOLS AND EXPRESSIONS

3.3.1 SYMBOLS

Symbols recognized by the assembler consist of one or more valid characters (see Appendix B), the first eight of which are significant. The first character must be an uppercase letter (A-Z) or a period(.). Each remaining character may be an uppercase letter, a digit (0-9), a dollar sign ($), a period (.), or an underscore (_).

Numbers recognized by the assembler include decimal, hexadecimal, octal, and binary values. Decimal numbers are specified by a string of decimal digits (0-9); hexadecimal numbers are specified by a dollar sign ($) followed by a string of hexadecimal digits (0-9, A-F); octal numbers are specified by an "at" sign (@) followed by a string of octal digits (0-7); binary numbers are specified by a percent sign (%) followed by a string of binary digits (0-1).

Examples:

Octal	-	An "at" sign followed by a string of octal digits Example: @12345
Binary	-	A percent sign followed by a string of binary digits Example: %10111
Decimal	-	A string of decimal digits Example: 12345
Hexadecimal	-	A dollar sign ($) followed by a string of hexadecimal digits. Example: $12345

One or more ASCII characters enclosed by apostrophes (') constitute an ASCII string. ASCII strings are left-justified and zero-filled (if necessary), whether stored or used as immediate operands. This left justification will be to a word boundary if one or two characters are specified, or to a long word boundary if the string contains more than two characters. (In order to specify an apostrophe within a literal or string, two successive apostrophes must appear where the single apostrophe is intended to appear.)

```
Examples: DC.L   'ABCD'
          DC.L   '''79'
          DC.W   '*'
          DC.L   'I''M'
```

3.3.2 SYMBOL DEFINITION CLASSES

Symbols may be differentiated by usage into two general classes. Class 1 symbols are used in the operation field of the instruction (see paragraph 3.2.4 for field definitions); Class 2 symbols occur in the label and operand fields of the instruction. Assembler directives, instruction mnemonics, and macro names comprise Class 1 symbols; user-defined labels and register mnemonics are included in Class 2 symbols.

A Class 1 symbol may be redefined and used independently as a Class 2 symbol, and vice versa. As long as each symbol is used correctly, no conflict will result from the existence of two symbols of different classes with the same name. For example, the following is a legal instruction sequence:

```
        ADD   D1,ADD
         .
         .
         .
  ADD   DS     2
```

By its use as a Class 1 symbol, the first "ADD" is recognized as an instruction mnemonic; likewise, the second ADD is recognized as a Class 2 symbol identifying a reserved storage area. The assembler differentiates a Class 1 symbol from a Class 2 symbol with the same name, thereby allowing two symbol table entries with the same name but different class.

Macro labels are a special case because the same symbol will appear as the label (Class 2) in the MACRO definition and, subsequently, as an operation code mnemonic (Class 1) in invocation of that same macro. Macro labels are defined to be Class 1 symbols; their presence in the label field of a MACRO directive is ignored as a Class 2 symbol. Therefore, macro names may be redefined as Class 2 symbols without conflict.

A symbol may not be redefined within the same class. For example, ADD (reserved Class 1 symbol) may not be redefined as a macro label (also Class 1), nor may "A5" (reserved Class 2 symbol) be redefined as a statement or storage location label (also Class 2). A reserved symbol may be used only within its own class.

3.3.3 USER-DEFINED LABELS

Labels are defined by the user to identify memory locations in program or data areas of the assembly module. Each label has two attributes: the program section in which the memory location resides, and the offset from the beginning of that program section.

Labels may be defined in the label field of an executable instruction or a data definition directive source line. It is also possible to SET or EQU a label to either an absolute or a relocatable value.

3.3.4 EXPRESSIONS

Expressions are composed of one or more symbols, which may be combined with unary or binary operations. Legal symbols in expressions include:

a. User-defined labels and their associated absolute or relocatable values.

b. Numbers and their absolute values.

c. The special symbol "*", which identifies the present location counter value, which may be either absolute or relocatable.

3.3.5 OPERATOR PRECEDENCE

Operators recognized by the assembler include the following:

a. Arithmetic operators:

addition	(+)	
subtraction	(-)	
multiplication	(*)	
division	(/)	-- produces a truncated integer result
unary minus	(-)	

b. Shift operators (binary):

shift right	(>>)	-- the left operand is shifted to the right (and zero-filled) by the number of bits specified by the right operand.
shift left	(<<)	-- analogous to >>

c. Logical operators (binary):

and	(&)
or	(!)

Expressions are evaluated with the following operator precedence:

1. parenthetical expression (innermost first)
2. unary minus
3. shift
4. and, or
5. multiplication, division
6. addition, subtraction

Operators of the same precedence are evaluated left to right. All results (including intermediate) of expression evaluation are 32-bit, truncated integers. Valid operands include numeric constants, ASCII literals, absolute symbols, and relocatable symbols (with "+" and "-" only).

3.4 REGISTERS

The 68000 has sixteen 32-bit registers (D0-D7, A0-A7) in addition to a 24-bit program counter and 16-bit status register.

Registers D0-D7 are used as data registers for byte, word, and long word operations. Registers A0-A7 are used as software stack pointers and base address registers; they may also be used for word and long word data operations. All 16 registers may be used as index registers.

Register A7 is used as the system stack pointer.

The following register mnemonics are recognized by the assembler:

D0-D7	Data registers.
A0-A7	Address registers.

A7,SP	Either mnemonic represents the system stack pointer of the active system state.
USP	User stack pointer.
CCR	Condition code register (low 8 bits of SR).
SR	Status register. All 16 bits may be modified in the supervisor state. Only low 8 bits (CCR) may be modified in user state.
PC	Program counter. Used only in forcing program counter-relative addressing (see page 3-30).

3.4.1 VARIANTS ON INSTRUCTION TYPES

Certain instructions allow a "quick" and/or an "immediate" form when immediate data within a restricted size range appears as an operand. These abbreviated forms are normally chosen by the assembler, when appropriate. However, it is possible for the programmer to "force" such a form by appending a "Q" or "I" to the mnemonic opcode (to indicate "quick" or "immediate", respectively) on instructions for which such forms exist. If the specified quick or immediate form does not exist, or if the immediate data does not conform to the size requirements of the abbreviated form, an error will be generated.

Some instructions also have "address" variant forms (which refer to address registers as destinations); these variants append an "A" to the instruction mnemonic (e.g., ADDA, CMPA). This variant will be chosen by the assembler without programmer specification, when appropriate to do so; the programmer need specify only the general instruction mnemonic. However, the programmer may "force" or specify such a variant form by appending the "A". If the specified variant does not exist or is not appropriate with the given operands, an error will be generated.

The CMP instruction also has a variant form (CMPM) in which both operands are a special class of memory references. The CMPM instruction requires postincrement addressing of both operands. The CMPM instruction will be selected by the assembler, or it may be specified by the programmer.

The variations -- A, Q, I, and M -- must conform to the following restrictions:

A   Must specify an address register as a destination, and cannot specify a byte size code (.B).

Q Requires immediate operand be in a certain size range. MOVEQ also requires longword data size.

I The size of immediate data is adjusted to match size code of operation.

M Both operands must be postincrement addresses.

For example, the instruction

```
ADDQ   #9,D0      Attempts to add value 9 to D0
```

will cause an assembly error, because the immediate operand is not in the valid size range (1 through 8).

Although the assembler will choose the appropriate opcode variation -- A, Q, I, or M -- when the suffix is not specified, the explicit encoding of the suffix with the basic opcode is recommended for the following purposes:

a. For documentation, to make clear in the source language the instruction form that was assembled.

b. To force a format other than that which the assembler would choose. For example, the assembler would choose the quick (Q) form for the instruction

```
ADD    #1,D4      Adds the value 1 to D4 via an ADDQ (2-byte)
                  instruction.
```

If the immediate (I) form was desired, the programmer would need to declare it explicitly, as follows:

```
ADDI   #1,D4      Adds the value 1 to D4 via an ADDI (4-byte)
                  instruction.
```

c. To generate invariant code when using variant immediate data (separate assemblies).

3.4.2 ADDRESSING MODES

Effective address modes, combined with operation codes, define the particular function to be performed by a given instruction.

References to data addresses may be odd only if a byte is referenced. Data references involving words or long words must be even. Likewise, instructions must begin on an even byte boundary.

Individual bits within a byte (operand for memory destinations) or long words (operands for D register destinations) may be addressed with the bit manipulation instructions (paragraph 3.2.4.10). Bits for a byte are numbered 7 to 0, with 7 being the most significant bit position and 0 the least significant. Bits for a word are numbered 15 to 0, with 15 being the most significant bit and 0 the least significant. Bits for a long word are numbered from 31 to 0, with 31 being the most significant bit position and 0 the least significant bit position.

Table 3-1 summarizes the addressing modes defined for the 68000, their invocations, and significant constraints.

Table 3-1. Address Modes

MODE	INVOCATION	COMMENTS
1) Register direct	An Dn	
2) Memory address		
a) Simple indirect	(An)	
b) Predecrement	-(An)	
c) Postincrement	(An)+	
d) Indirect with displacement (16-bit)	<absolute>(An) <complex>(An)	
e) Indirect with index (16- or 32-bit) plus displacement (8-bit)	<absolute>(An,Ri)	

Table 3-1. Address Modes (continued)

3) Special address		
a) PC with displacement (16-bit)	<simple>	Expression must be backward, within current relocatable section.
	<aboslute>(PC) <simple>(PC) <complex>(PC)	Forced PC-relative. Must fit within 16-bit signed field; resolved at assembly or link time.
b) PC with index (16- or 32-bit) plus displacement (8-bit)	<simple>(PC)	Expression must be backward, within current relocatable section.
	<absolute>(PC,Ri) <simple>(PC,Ri)	Forced PC-relative; expression must be within current program section.
c) Absolute (16- or 32-bit)	<absolute> <complex> <simple>	Expression must be forward reference or not in current program section.
d) Immediate (8-, 16-, or 32-bit)	#<absolute> #<simple> #<complex>	
4) Implicit PC reference		Invoked by conditional branch (Bcc) or DBcc instruction; the effective address is a displacement from the PC; the displacement is either 8 or 16 bits, depending on OPT BRS, OPT BRL, and whether these options are overridden on the current instruction.

Table 3-2 provides a cross reference of operand formats and addressing modes. Given an operator of the format shown in the first column, the other columns show which addressing mode is indicated, depending on whether the expression is absolute, simple relocatable, or complex relocatable.

Table 3-2. Cross-Reference: Effective Addressing Mode, Given Operand Format and <expr> Type

	EFFECTIVE ADDRESSING MODE		
PERAND FORMAT	ABSOLUTE <expr>	IMPLE RELOCATABLE <expr>	COMPLEX RELOCATABLE <expr>
<expr>(An)	d(An)	d(PC,An)	d(An)
<expr>(Dn)	invalid	d(Pc<Dn) *	invalid
<expr>(An,Ri)	d(An,Ri)	invalid	invalid
<expr>	absolute (W,L)	d(PC)* or absolute (W,L)	absolute (W,L)
<expr>(PC)	d(PC)	d(PC)	d(PC)
<expr>(PC,Ri)	d(PC,Ri) *	d(PC,Ri) *	invalid
#<expr>	immediate(B,W,L)	immediate (W,L)	immediate (W,L)

*Must be within current program section.

Listed below are definitions of the symbols used in Tables 3-1 and 3-2, and throughout the remainder of this section:

An	Address register number "n" (0-7).
Dn	Data register number "n" (0-7).
Ri	Index register number "i"; may be any address (An) or data (Dn) register with optional ".W" or ".L" size designation (16 vs 32 bits).
B,W,L	Byte, word, long word data sizes.
d(An)	Address register indirect with displacement (d).
d(An,Ri)	Address register indirect with index (Ri) plus displacement (d).
d(PC)	Program counter with displacement (d).
d(PC,Ri)	Program counter with index (Ri) plus displacement (d).

<absolute>	Absolute expression.
<simple>	Simple relocatable expression.
<complex>	Complex relocatable expression.

3.4.2.1 Register Direct Modes

These effective addressing modes specify that the operand is in one of the 16 multifunction registers (eight data and eight address registers). The operation is performed directly on the actual contents of the register

```
Notations:  An
            Dn      where n is between 0 and 7

Examples:  CLR.L   D1              Clear all 32 bits of D1.

           ADD     A1,A2           Add low order word of A1 to low order
                                   word of A2.
```

3.4.2.2 Memory Address

The following effective addressing modes specify that the operand is in memory and provide the specific address of the operand.

ADDRESS REGISTER INDIRECT The address of the operand is in the address register specified by the register field.

```
Notation:  (An)

Examples:  MOVE    #5,(A5)         Move value 5 to word whose address is
                                   contained in A5.

           SUB.L   (A1),D0         Subtract from D0 the value in the long
                                   word whose address is contained in A1.
```

ADDRESS REGISTER INDIRECT WITH POSTINCREMENT The address of the operand is in the address register specified by the register field. After the operand address is used, it is incremented by one, two or four, depending upon whether the size of the operand is byte (.B), word (.W), or long (.L).

Notation: (An)+

Examples:	MOVE.B (A2)+,D2	Move byte whose address is in A2 to low order byte of D2; increment A2 by 1.
	MOVE.L (A4)+,D3	Move long word whose address is in A4 to D3; increment A4 by 4.

ADDRESS REGISTER INDIRECT WITH PREDECREMENT The address of the operand is in the address register specified by the register field. Before the operand address is used, it is decremented by one, two, or four, depending upon whether the operand size is byte (.B), word (.W), or long (.L).

Notation: -(An)

Examples:	CLR -(A2)	Subtract 2 from A2; clear word whose address is now in A2.
	CMP.L -(A0),D0	Subtract 4 from A0; compare long word whose address is now in A0 with contents of D0.

ADDRESS REGISTER INDIRECT WITH DISPLACEMENT The address of the operand is the sum of the address in the address register and the sign-extended displacement.

Notation: d(An)

Examples:	AVAL	EQU	5	AVAL is equated to 5 (for use in next instruction).
		CLR.B	AVAL(A0)	Clear byte whose address is given by adding value of AVAL (=5) to contents of A0.
		MOVE	#2,10(A2)	Move value 2 to word whose address is given by adding 10 to contents of A2.

ADDRESS REGISTER INDIRECT WITH INDEX The address of the operand is the sum of the address in the address register, the sign-extended displacement, and the contents of the index (A or D) register.

Notations:	d(an,Ri)	Specifies low order word of index register.
	d(An,Ri.W)	
	d(An,Ri.L)	Specifies entire contents of index register.

Examples:	ADD	AVAL(A1,D2),D5	Add to low order word of D5 the word whose address is given by addition of contents of A1, the low order word of index register (D2), and the displacement (AVAL).
	MOVE.L	D5,$20(A2,A3.L)	Move entire contents of D5 to long word whose address is given by addition of contents of A2, contents of entire index register (A3), and the displacement ($20).

3.4.2.3 Special Address Modes

Special address modes use the effective address register field to specify the special addressing mode instead of a register number. The following table provides the ranges for absolute short and long addresses.

32-bit address	16-bit representation of 32-bit address
00000000 . 00007FFF	0000 . Absolute short 7FFF
00008000 . . FFFF7FFF	(No representation in 16 bits; must be absolute long)
FFFF8000 . . FFFFFFFF	8000 . Absolute short . FFFF

ABSOLUTE SHORT ADDRESS The 16-bit address of the operand is sign extended before it is used. Therefore, the useful address range is 0 through $7FFF and $FFFF8000 through $FFFFFFFF.

Notation: XXX

Example:	JMP	$400	Jump to hex address 400

ABSOLUTE LONG ADDRESS The address of the operand is the 32-bit value specified.

Notation: XXX

Exmaple:	JMP	$12000	Jump to hex address 12000

PROGRAM COUNTER WITH DISPLACEMENT The address of the operand is the sum of the address in the program counter and the sign-extended displacement integer. The assembler calculates this sign-extended displacement by subtracting the address of displacement word from the value of the operand field.

Notation:	<expression>(PC)		Forced program counter-relative.
Example:	JMP	TAG(PC)	Force the evaluation of 'TAG' to be program counter-relative.

PROGRAM COUNTER WITH INDEX The address is the sum of the address in the program counter, the sign-extended displacement value, and the contents of the index (A or D) register.

Notations:	<expression>(Ri.W)		Specifies low order word of index register. .W is optional (default).
	<expression>(Ri.L)		Specifies entire contents of index register.
	<expression>(PC,Ri)		Forced program counter-relative. Ri.W or Ri.L legal.
Examples:	MOVE	T(D2),TABLE	Moves word at location (T plus contents of D2) to word location defined by TABLE. T must be a relocatable symbol.
	JMP	TABLE(A2.W)	Transfers control to location defined by TABLE plus the lower 16-bit content

		of A2 with sign extension. TABLE must be a relocatable symbol.
JMP	TAG(PC,A2.W)	Forces evaluation of 'TAG' to be program counter-relative with index.

IMMEDIATE DATA An absolute number may be specified as an operand by immediately preceding a number or expression with a '#' character. The immediate character (#) is used to designate and absolute number other than a displacement or an absolute address.

Notation: #XXX

Examples:	MOVE	#1,D0	Move value 1 to low order word of D0.
	SUB.L	#1,D0	Subtract value 1 from the entire contents of D0.

3.4.3 NOTES ON ADDRESSING OPTIONS

By default, the assembler will resolve all forward references by using the longer form of the effective address in the operand reference. The programmer may override this default by specifying OPT FRS, which designates that forward absolute references should be short, or OPT BRS, designating that forward relative branches should use the shorted (8-bit) displacement format.

On an instruction which does not allow a size code, the current forward reference default format may be overridden (for that instruction only) by appending .S (short) or .L (long) to the instruction mnemonic. A similar override may be performed in the structured syntax control directives via the extent codes (see paragraph 3.6.3 for further explanation). No override is possible on instructions with size code specification. Notably, this override procedure is possible on branch and jump instructions.

The shorter form of the effective address for relative branch instructions is an 8-bit displacement; the longer format is a 16-bit displacement. For absolute jumps, the shorter effective address is the 16-bit absolute short; the longer format is the 32-bit absolute long mode. In the case of forward references in either relative branches or absolute jumps, if the shorter format is directed and the longer format is later found necessary when the reference is resolved, an error will occur.

References to symbols already defined, whether absolute or relative, are resolved by the assembler into the appropriate effective address, unless .S or .L is forced on the instruction.

A short form may be forced by following the instruction mnemonic with .S.

Example:

BEQ.S	LOOP1	If condition code 'EQ' (equal) is true, then branch to LOOP1 (using the short form of the instruction).

In this case, the instruction size is forced to one word. An error will be printed if the operand field is not in the range of an 8-bit displacement.

Since 8-bit value fields are not relocated, a Bcc.S instruction which branches to an XREF or other expression-required location is not allowed. Such an instruction format will result in an assembler error. A relative branch to a symbol known to be an XREF will employ the longer (16-bit) displacement, with resolution by the linkage editor.

Default actions of the assembler have been chosen to minimize two common address mode errors:

a. Displacement range violations

   Relative branch instructions (Bcc, BRA, BSR) allow either 8-bit or 16-bit displacements from the PC. On forward references in such instructions, the default action is to assume the 16-bit displacement (OPT BRL), which also allows resolution by the linkage editor, should that prove necessary.

b. Inappropriate absolute short address

   Absolute addresses may be short (16-bit) or long (32-bit). On forward references with absolute effective address, the default action is to assume the long format (OPT FRL).

Default conditions have been chosen to prevent errors by using addressing formats which ensure address resolution in the broadest range of conditions, at the expense of code efficiency. Each default may be overridden to improve efficiency or to create position independent code. Also, the current address size defaults (options BRL, BRS, FRL, FRS) may be overridden in certain cases on specific instructions which do not allow size codes by appending .S or .L, as in Bcc,S and JMP.L (Bcc, BSR, JMP, JSR only).

3.5 ASSEMBLER DIRECTIVES

3.5.1 INTRODUCTION

All assembler directives (pseudo-ops), with the exception of "DC" and "DCB", are instructions to the assembler rather than instructions to be translated into object code. This chapter contains descriptions and examples of the basic forms of the most frequently used assembler directives. Directives controlling the macro and conditional assembly capabilities are described in Section 3.5. Directives used in structured syntax are described in Section 3.6. The most commonly used directives supported by the assembler are grouped by function in Table 3-3.

Table 3-3. 68000 Assembler Directives

DIRECTIVE	FUNCTION
ASSEMBLY CONTROL	
INCLUDE	Include second file
OFFSET	Define offsets
END	Program end
SYMBOL DEFINITION	
EQU*	Assign permanent value
SET*	Assign temporary value
REG*	Define register list

Table 3-3. 68000 Assembler Directives (cont'd)

DIRECTIVE	FUNCTION
DATA DEFINITION STORAGE ALLOCATION	
DC**	Define constants
DS**	Define storage
DCB**	Define constant block
LISTING CONTROL AND OPTIONS	
PAGE	Top of page
LIST	Enable the listing
NOLIST or NOL	Disable the listing
FORMAT	Enable the automatic formatting
NOFORMAT	Disable the automatic formatting
SPC n	Skip n lines
NOPAGE	Disable paging
LLEN n	Set line lengths 72≤ n ≤132
TTL	Up to 60 characters of title
NOOBJ	Disable object output
OPT	Assembler options
FAIL	Programmer-generated ERROR
LINKAGE EDITOR CONTROL	
IDNT*	Relocatable identification record
XDEF	External symbol definition
XREF	External symbol reference
** Label optional.	

3.5.2 ASSEMBLY CONTROL

3.5.2.1 END - Program End

FORMAT: END [<start address>]

DESCRIPTION: END directive indicates to the assembler that the source is finished. Subsequent source statements are ignored. The END directive encountered at the end of the first pass through the source program causes the assembler to start the second pass. The start address should be specified unless it is external to the module. If no start address is specified, it is still possible to include a comment field, provided the comment field is set off by an exclamation point(!). This syntax indicates to the assembler that the operand field is null, but that a comment field follows.

3.5.2.2 OFFSET - Define Offsets

FORMAT: OFFSET <expression>

DESCRIPTION: The OFFSET directive is used to define a table of offsets via the Define Storage (DS) directive without passing these storage definitions on to the linkage editor, in effect creating a dummy section. Symbols defined in a OFFSET table are kept internally, but no code-producing instructions or directives may appear. SET, EQU, REG, XDEF, and XREF directives are allowed.

<expression> is the value at which the offset table is to begin. The expression must be absolute and may not contain forward, undefined, or external references.

OFFSET is terminated by an ORG, OFFSET, SECTION, or END directive.

3.5.2.3 INCLUDE - Include Secondary File

FORMAT: INCLUDE <file spec>

DESCRIPTION: This directive is inserted in the source program at any point where a secondary file is to be included in the source input stream.

3.5.3 SYMBOL DEFINITION

Symbol definition directives EQU, REG, and SET provide the only method by which a symbol appearing in the label field may be assigned a 'value' other than that corresponding to the current location counter.

3.5.3.1 EQU - Equate Symbol Value

FORMAT: <label> EQU <expression> [<comments>]

DESCRIPTION: EQU directive assigns the value of the expression in the operand field to the symbol in the label field. The label and expression follow the rules given in Section 3.2 . The label and operand fields are both required and the label cannot be defined anywhere else in the program.

The expression in the operand field of an EQU cannot include a symbol that is undefined or not yet defined (no forward references are allowed).

3.5.3.2 SET - Set Symbol Value

FORMAT: <label> SET <expression> [<comments>]

DESCRIPTION: SET directive assigns the value of the expression in the operand field to the symbol in the label field. Thus, the SET directive is similar to the EQU directive. However, the SET directive allows the symbol in the label field to be redefined by other SET directives in the program. The label and operand fields are both required.

The expression in the operand field of a SET cannot include a symbol that is undefined or not yet defined (no forward references are allowed).

3.5.3.3 REG - Define Register List

FORMAT:	<label> REG <reg list> [<comment>]
DESCRIPTION:	REG directive assigns a value to <label> that can be translated into the register list mask format used in the MOVEM instruction. The label cannot be redefined as a Class 2 symbol anywhere else in the program. <reg list> is of the form:

```
R1[-R2][/R3[-R4]]...
```

Example: A1-A5/D0/D2-D4/D7

3.5.4 DATA DEFINITION AND STORAGE ALLOCATION

The directives in this section provide the only means by which object code may begin or end on odd byte boundaries. All instructions and all word or long word-sized data must begin and end on even byte boundaries. Odd byte alignment is allowed only for the DC.B, DS.B, DCB.B, and COMLINE directives. All other operations which generate relocatable object code will be preceded by a zero fill byte if word boundary alignment is required.

3.5.4.1 DC - Define Constant

FORMAT: [<label>] DC.B <operand(s)> Define constant in bytes
DC.W <operand(s)> Define constant in words (default)
DC.L <operand(s)> Define constant in long words

DESCRIPTION: The function of the DC directive is to define a constant in memory. The DC directive may have one operand, or multiple operands which are separated by commas. The operand field may contain the actual value (decimal, hexadecimal, or ASCII). Alternatively, the operand may be a symbol or expression which can be assigned a numeric value by the assembler. The constant is aligned on a word boundary if word (.W) or long word (.L) is specified, or a byte boundary if byte (.B) is specified. Only word (.W) and long word (.L) constants may be relocated.

The following rules apply to size specifications of DC directives with ASCII string as operands:

DC.B	One byte is allocated per ASCII character.
DC.W	The string will begin on a word boundary. If the string address contains an odd number of characters, a zero fill byte will follow the last character.
DC.L	The string will begin on a word boundary. If the string length is not a multiple of four bytes, the last long word will be zero filled.

Unless option CEX is in effect, a maximum of six bytes of constants will be displayed on the assembly listing.

EXAMPLES OF ASCII STRINGS

DC.B 'ABCDEFGHI'	Memory would have nine contiguous bytes with the ASCII characters A through I.
DC.B 'E' DC.B 'J'	Memory will have characters "EJ" ($454A) in contiguous bytes.
DC.B 'E' DC.W 'E'	Memory will have $45004500 in contiguous bytes, the first zero byte being an odd byte fill as outlined above.
DC 'X'	Memory will have $5800 in contiguous bytes.
DC.L '12345'	Memory will have $3132333435000000 in contiguous bytes.

EXAMPLES OF NUMERIC CONSTANTS

DC.B 10,5,7	Memory would have three contiguous bytes with the decimal values 10, 5, and 7 in their respective bytes.
DC.W 10,5,7	Each operand is contained in a word. The value 10 is contained in the first word, right justified. The value 5 is in the second word, and the value 7 is in the third word.

DC.L	10,5,7	Each operand in a long word. The value 10 is contained in the first long word (4 bytes) right justified. The value 5 is in the second long word, and the value 7 is in the third long word.
DC	LABEL+1	The generated value will be the address of LABEL plus 1 in a word size operand.
DC	$FF,$10,$AE	Rules for hexadecimal are same as decimal.

If the resulting value in an operand expression exceeds the size of the operand, an error is generated. For example,

DC.B	$FFF	This will cause an error because $FFF cannot be represented in 8 bits.
DC	$FFF6F	This will cause an error because $FFF6F cannot be represented in 16 bits.

3.5.4.2 DS - Define Storage

FORMAT: [<label>] DS.B <operand> Define storage in bytes
DS.W <operand> Define storage in words (default)
DS.L <operand> Define storage in long words

DESCRIPTION: DS directive is used to reserve memory locations. The contents of the memory reserved is not initialized in any way.

EXAMPLES:

```
        DS.B   10      Define 10 contiguous bytes in memory
        DS     10      Define 10 contiguous words in memory
PT1     DS    $10      Define 16 contiguous words in memory
PT2     DS.L  100      Define 100 contiguous long words in memory
```

The label will reference the lowest address of the defined storage area. If word or long-word mode is specified, the storage area is aligned on a word boundary. If it is desired to force alignment on a word boundary, the directive DS 0 may be used.

```
EXAMPLE:   DS.B  1   RESERVE ONE BYTE
           DS    0   SET LOCATION COUNTER TO EVEN BOUNDARY
```

The operand must not include a forward reference (to an undefined symbol).

3.5.4.3 DCB - Define Constant Block

FORMAT: [<label>] DCB[.<size code>] <length>,<value> [<comment>]

DESCRIPTION: DCB directive causes the assembler to allocate a block of bytes, words, or long words, depending upon the <size code> specified. If <size code> is omitted, word (.W) is the default size. The block length is specified by the absolute expression <length>, which may not contain undefined, forward, or external references. The initial value of each storage unit allocated will be the sign-extended expression <value>, which may contain forward references. <length> must be greater than zero. <value> may be relocatable unless byte size (.B) is specified.

3.5.5 LISTING CONTROL

3.5.5.1 PAGE - Top of Page

FORMAT: PAGE

DESCRIPTION: Advance the paper to the top of the next page. The PAGE directive does not appear on the program listing. No label or operand is used, an no machine code results.

3.5.5.2 Listing Output Options

LIST - LIST THE ASSEMBLY

FORMAT: LIST

DESCRIPTION: Print the assembly listing on the output device. This option is selected by default. The source text following the LIST directive is printed until an END or NOLIST directive is encountered.

NOLIST - DO NOT LIST THE ASSEMBLY

FORMAT: NOLIST or NOL

DESCRIPTION: Suppress the printing of the assembly listing until a LIST directive is encountered.

FORMAT - FORMAT THE SOURCE LISTING

FORMAT: FORMAT

DESCRIPTION: Format the source listing, including column alignment (see Table 3-4) and structured syntax indentation. This option is selected by default.

NOFORMAT - DO NOT FORMAT THE SOURCE LISTING

FORMAT: NOFORMAT

DESCRIPTION: The source listing will have the same format as the source input file.

SPC - SPACE BETWEEN SOURCE LINES

FORMAT: SPC n

DESCRIPTION: Output n blank lines on the assembly listing. This has the same effect as inputting n blank lines in the assembly source. A blank line is defined by the assembler to be a line with only a carriage return.

NOPAGE - DO NOT PAGE SOURCE OUTPUT

FORMAT: NOPAGE

DESCRIPTION: Suppress paging to the output device. Output lines are printed continuously with no page headings or top and bottom margins.

LLEN - LINE LENGTH

FORMAT: LLEN n

DESCRIPTION: Set the number of columns to be output to n. The minimum value of n is 72 and the maximum 132. The default value for n is 132 columns.

TTL - TITLE

FORMAT:	TTL <title string>
DESCRIPTION:	Print the <title string> at the top of each page. A title consists of up to 60 characters. The same title will appear at the top of all successive pages until another TTl directive is encountered. In order to print a title on the first listing page, the TTl directive must precede the first source line which will appear on the listing.

NOOBJ - NO OBJECT

FORMAT:	NOOBJ
DESCRIPTION:	Suppress the generation of object code.

OPT - ASSEMBLER OUTPUT OPTIONS

FORMAT:	OPT <option>[,<option>]... [<comment>]
DESCRIPTION:	Follows the command format.

OPTIONS:

A	Absolute address. All non-indexed operands which reference either labels or the current assembler location counter (*) will be resolved as absolute addresses.
NOA	Disable A (default).
BRL	Forward branch long (default). Forward references in relative branch instructions (Bcc, BRA, BSR) will assume the longer form (16-bit displacement, yielding a 4-byte instruction).
BRS	Forward branch short. As with BRL, but using the shorter form (8-bit displacement, yielding a 2-byte instruction).
CEX	Print DC expansions.
NOCEX	Opposite of CEX (default).
CL	Print conditional assembly directives (default).
NOCL	Opposite of CL.
CRE	Print cross-reference table at end of source listing. This option must precede first symbol in

	source program. If this option is not in effect, only the symbol table will be printed.
D	Debug option (output symbol table to file with the same name as the object code file, but with an extension of ".RS").
FRL	Forward reference long (default). Forward references in the absolute format will assume absolute long mode (32-bit).
FRS	Forward reference short. Forward references in the absolute format will assume absolute short mode (16-bit).
MC	Print macro calls (default).
NOMC	Opposite of MC.
MD	Print macro definitions (default).
NOMD	Opposite of MD.
MEX	Print macro expansions.
NOMEX	Opposite of MEX (default).
O	Create output module (default).
NOO	Opposite of O.
PCO	PC relative addressing within ORG. Employ relative addressing when possible on backward references occurring in an ORG section.
NOPCO	Disable PCO (default).
PCS	Force PC relative addressing. This option may be used to force position independent code (see Section 3.7); however, this option does not force PC relative addressing of unknown forward references.
NOPCS	Disable PCS (default).

3.5.6 FAIL - PROGRAMMER GENERATED ERROR

FORMAT: FAIL <expression>

DESCRIPTION: The FAIL directive will cause an error or warning message to be printed by the assembler. The total error count or warning count will be incremented as with any other error or warning. The FAIL directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the error has been printed. The <expression> is evaluated and printed as the error or warning number on the assembly listing. Errors are numbered 0-499; warnings are numbered 500 and above.

3.5.7 LINKAGE EDITOR CONTROL

3.5.7.1 IDNT - Relocatable Identification Record

FORMAT: <module name> IDNT <version>,<revision> [<descr>]

DESCRIPTION: Every relocatable object module must contain an identification record as a means of identifying the module at link time. The module name is specified in the label field or the IDNT directive, while the version and revision numbers are specified as the first and second operands, respectively. The comment field of the IDNT directive is also passed on the linkage editor as a description of the module.

3.5.7.2 XDEF - External Symbol Definition

FORMAT: XDEF <symbol>[,<symbol>]... [<comment>]

DESCRIPTION: This directive specifies symbols defined in the current module that are passed on to the linkage editor as symbols which may be referenced by other modules linked to the current module.

3.5.7.3 XREF - External Symbol Reference

FORMAT:

```
XREF[.S]  <symbol> [,<symbol>]...
           [,<symbol> [,<symbol>]...]...
```

DESCRIPTION: This directive specifies symbols referenced in the current module but defined in other modules. This list is passed on to the linkage editor.

".S" indicates the XREF symbols will be linked into low address memory so that direct addressing of these symbols may be accomplished through absolute short mode.

EXAMPLE: XREF AA,A2,A3,B3,C3

3.6 INVOKING THE ASSEMBLER

3.6.1 COMMAND LINE FORMAT

```
ASM <sourcefile> [,option] ...
```

sourcefile	: Required. The default extension is ASM and will replace any other extension.
options	: Can be specified in upper or lower case and in any order. If the same option is specified more than once, the last specification holds. Options must not be preceded by a blank.
+C[filename]	: Produce object code (default). If a filename is specified without an extension, the default extension .OBJ is added. If a filename is not specified, the output file created is sourcefilename.OBJ
-C	: Inhibit production of object code.
+L[filename]	: Produce a listing file. If a filename is specified without an extension, the default extension .LST is added (unless the name begins with a #). If a filename is not specified, the listing file created is sourcefilename.LST

We suggest that you issue the commands SET LW=132 and SET LD=65 before listing assembler output on the printer.

-L	: Inhibit production of a listing file (default).
+M	: List macro expansions (if +L specified)
-M	: Inhibit listing of macro expansions (default).
+R	: Produce a cross-reference (if +L specified).
-R	: Inhibit production of a cross-reference (default)
+S	: List structured control statement expansions (if +L specified).
-S	: Inhibit listing of structured control statement expansions (default)
+W	: Enable warning messages during assembly (default).
-W	: Disable warning messages during assembly.

3.6.2 ASSEMBLER OUTPUT

Assembler outputs include an assembly listing, a symbol table, and an object program file.

The assembly listing includes the source program, as well as additional information generated by the assembler. Most lines in the listing correspond directly to a source statement. Lines which do not correspond directly to a source line include:

- Page header and title
- Error and warning lines
- Expansion lines for instructions over three words in length.

The assembly listing format is shown in Table 3-4. The label, operation, and operand fields may be extended if the source field does not fit into the designated output field.

The last page of the assembly listing is the symbol table. Symbols are listed in alphabetical order, along with their values and an indication of the relocatable section in which they occur (if any). Symbols that are XDEF, XREF, REG, in named common, or multiply defined are flagged. If option CRE has been specified in the program, the cross-reference listing will identify the source lines on which the symbol was defined or referenced (definitions appear first, flagged with a "-").

An example of assembler output is provided in Appendix D.

TABLE 3-4. Standard Listing Format

COLUMN	CONTENTS	EXPLANATION
1-4	Source line number	4-digit decimal counter
6	Section number	1-digit hex section number (blank indicates location counter is absolute)
8-15	Location counter value	In hex
17-20	Operand word	In hex
21-24	First extension word	In hex
25-28	Second extension word	In hex; any additional extension words appear on the next line
30-37	Label field	
39-46	Operation field	
48-67	Operand field	
70-N	Comment field	

3.6.3 ASSEMBLER RUNTIME ERRORS

During runtime, the assembler may generate its own error messages. These are listed in Appendix A.

Any assembly instruction which may generate six or more bytes of code, and that is found to have an operand error, will generate six bytes of object code. The code for instruction, however, will be \$4AFB, which is an illegal opcode, and the extension word(s) will be \$4E71, which is a NOP. These six bytes allow more instructions to be patched in place, or a jump to be inserted to a patch area anywhere in the address space.

Instructions which generate only two or four bytes will continue to generate a 2- or 4-byte length instruction, respectively, whenever an operand is in error. The instruction word, however, will be illegal and the extension will be a NOP.

Undefined operations will generate six bytes of code with an illegal opcode and NOP extensions.

3.7 MACRO OPERATIONS AND CONDITIONAL ASSEMBLY

3.7.1 INTRODUCTION

This chapter describes the macro (paragraph 3.5.2) and the conditional assembly (paragraph 3.5.3) capabilities of the assembler. These features can be used in any program.

3.7.2 MACRO OPERATIONS

Programming applications frequently involve the coding of a repeated pattern of instructions that within themselves contain variable entries at each iteration of the pattern, or basic coding patterns subject to conditional assembly at each occurrence. In either case, macros provide a shorthand notation for handling these patterns. Having determined the iterated pattern, the programmer can, within the macro, designate fields of any statement as variable. Thereafter, by invoking a macro, the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

Macro usage can be divided into two basic parts -- definition and expansion.

When the pattern is defined it is given a name. This name becomes the mnemonic by which the macro is subsequently invoked (called). The name of a macro definition should not be the same as an existing instruction mnemonic, or an assembler directive.

Expansion occurs when the previously defined macro is called (invoked). The macro call causes source statements to be generated. The generated statements may contain substitutable arguments. The statements that may be generated by a macro call are relatively unrestricted as to type. They can be any processor instruction, almost any assembler directive, or any previously defined macro. Source statements generated by a macro call are subject to the same conditions and restrictions to which programmer generate statements are subject.

To invoke a macro, the macro name must appear in the operation field as a source statement. Most arguments are placed in the operand field. By suitably selecting the arguments in relation to their use as indicated by the macro definition, the programmer causes the assembler to produce in-line coding variations of the macro definition.

The effect of a macro call is the same as an open subroutine in that it produces in-line code to perform a predefined function. The in-line code is inserted in the normal flow of the program so that the generated instructions are executed in-line with the rest of the program each time the macro is called.

3.7.2.1 Macro Definition

The definition of a macro consists of three parts:

a. The header: label MACRO

   The label of the MACRO statement is the "name" by which the macro is later invoked. This name must be a unique class 1 symbol. A macro name may not have a period (.) as any character other than the first.

b. The body

   The body of a macro is a sequence of standard source statements. Macro parameters are defined by the appearance of argument designators within these source statements. Legal macro-generated statements include the set of 68000 assembly language instructions, assembler directives, structured syntax statements, and calls to other, previously defined macros.

However, macro definitions may not be nested. When macro text lines are saved for later expansion, all spaces in the source line are compressed. This space compression will be noticed only if the listing is unformatted, or if the macro text includes literal strings with multiple spaces (which would not expand correctly). Macro expansion lines which contain more than 80 characters are truncated at 80 characters, which is the maximum length of an assembler input line.

c. The terminator: ENDM

3.7.2.2 Macro Invocation

The form of a macro call is: [label] name[.qualifier] [parameter list]

Although a macro may be referenced by another macro prior to its definition in the source module, the macro must be defined before its first in-line expansion. The name of the called macro must appear in the operation field of the source statement; parameters may appear as a qualifier to the macro name and/or in the operand field of the source statement, separated by commas.

The macro call produces in-line code at the location of the invocation, according to the macro definition and the parameters specified in the macro call. The source statements so generated are then assembled, subject to the same conditions and restrictions affecting any source statement. Nested macro calls are also expanded at this time.

3.7.2.3 Macro Parameter Definition and Use

Up to thirty-six different, substitutable arguments may appear in the source statements which constitute the body of a macro. These arguments are replaced by the corresponding parameters in a subsequent call to that macro.

Arguments are designated by a backslash character (\), followed by a digit (0 through 9) or an upper case letter (A through Z). Argument designator \0 refers to the qualifier appended to the macro name; parameters in the operand field of the macro call refer to argument designations \1 through \9 and \A through \Z, in that order.

The parameter list (operand field) of a macro call may be extended onto additional lines if necessary. The line to be extended must end with a comma separating two parameters, and the subsequent extension line must begin with an ampersand (&) in column 1. The extension of the parameter list will begin with the first non-blank characters following the ampersand. No other source lines may occur within an extended parameter call, and no comment field may occur except after the last parameter on the last extension line.

Argument substitution at the time of a macro call in handled as a literal (string) substitution. The string corresponding to a given parameter is substituted literally wherever that argument designator occurs in a source statement as the macro is expanded. Each statement generated in this expansion is assembled in-line. (Note that argument \0 begins with the first character following the period which separates the qualifier from the macro name, if a qualifier is present.)

It is possible to specify a null argument in a macro call by an empty string (not a blank); it must still be separated from other parameters by a comma (except for \0). In the case of a null argument referenced as a size code, the default size code (W) is implied; when a null argument itself is passed as an argument in a nested macro call, a null argument is passed. All parameters have a default value of null at the time of a macro call.

If an argument has multiple parts or contains commas or blanks, the entire argument must be enclosed within angle brackets (< and >). Such arguments must still be separated from other arguments by commas. A bracketed argument with no intervening character (<>) will be treated as a null argument. Embedded brackets must occur in pairs. Parameter \0 may not be bracketed and, hence, may not contain blanks (although commas are legal). Note that a macro argument may not contain the characters "<" or ">" unless they occur as part of the argument bracketing.

3.7.2.4 Labels Within Macros

To avoid the problem of multiply defined labels resulting from multiple calls to a macro which employs labels in its source statements, the programmer may direct the assembler to generate unique labels on each call to a macro.

Assembler-generated labels include a string of the form .nnn, where nn is a three-digit decimal number. The programmer may request an assembler-generated label by specifying \@ in a label field within a macro body. Each successive label definition which specifies a \@ directive will generate successive values of .nnn, thereby creating unique labels on repeated macro calls. Note that \@ may be preceded or succeeded by additional characters for additional clarity and to prevent ambiguity (more than four preceding characters may introduce a problem with non-uniqueness of symbols).

References to an assembler-generated label always refer to the label of the given form defined in the current level of macro expansion. Such a label is referenced as an operand by specifying the same character string as that which defines the label.

3.7.2.5 The MEXIT Directive

The MEXIT directive terminates the macro source statement generation during expansion. It may be used within a conditional assembly structure (see paragraph 3.5.3) to skip any remaining source lines up to the ENDM directive. All conditional assembly structures pending within the macro currently being expanded are also terminated by the MEXIT directive.

Example:

```
SAV2     MACRO
         MOVE.L   \1,SAVET       SAVE 1ST ARGUMENT
         MOVE.L   \2,SAVET+4     SAVE 2ND ARGUMENT
         IFEQ     '\3',''        IS THERE A 3RD ARGUMENT?
         FAIL     1000           DID ASSEMBLER GO THRU HERE?
         MEXIT                   NO, EXIT FROM MACRO
         ENDC
         MOVE.L   \3,SAVET+8     SAVE 3RD ARGUMENT
         ENDM
```

3.7.2.6 NARG Symbol

The symbol NARG is a special symbol when referenced within a macro expansion. The value assigned to NARG is the index of the last argument passed to the macros in the parameter list (even if nulls). NARG is undefined outside of macro expansion, and may be referenced as a Class 1 or 2 user-defined symbol outside of a macro expansion.

3.7.2.7 Implementation of Macro Definition

When the sequence of source statements:

```
             .
             .
             .
    MAC1     MACRO
             stmt1
             stmt2
             .
             .
             .
             stmtn
             ENDM
             .
             .
             .
```

is encountered in a source program, the following actions are performed:

a. The symbol table is checked for a Class 1 symbol entry of 'MAC1'. If such an entry is already present, a redefined symbol error (231) is generated; if no such entry exists, an entry is placed in the symbol table, identifying MAC1 as a macro.

b. Starting with the line following the MACRO directive, each line of the macro body is saved in a character sequence identified with MAC1. In the example, stmt1 through stmtn are saved in this manner. No object code is produced at this time. A check is made for missing parameter references in the macro text (e.g., parameter \1, \2, and \4 are referenced, but \3 is not).

c. Normal processing resumes with the line following the ENDM directive.

3.7.2.8 Implementation of Macro Expansion

When the statement:

```
     MAC1.qualifier    param1.parm2,...,paramn
```

is encountered in a source program calling the previously defined macro MAC1 (above), the following actions are performed:

a. Since the label field is blank, the string 'MAC1' is recognized as the operation code of the instruction. The symbol table is consulted for a Class 1 symbol entry with this name. If no such entry exists, an undefined symbol error (238) is generated. In this case, the entry indicates that the symbol identifies a macro.

b. The rest of the line is scanned for parameters which are saved as literals of null values, one such value in each of the thirty-six parameter fields. If the source line ends with a comma, the next line is checked for an extension of the parameter list. A cross-check is made with the macro definition for the number of parameters in the call. No object code is produced.

c. Macro expansion consists of the retrieval of the source lines which comprise the macro body. Each line is retrieved in turn, with special character pairs replaced by parameter strings or assembler-generated label strings.

   If a backslash character (\) is followed by either a digit (0 through 9) or an uppercase letter (A through Z), the two characters are replaced by the literal string which corresponds to that parameter on the macro invocation lines(s).

   A character sequence which includes "\@" is replaced by an assembler-generated label, as defined in paragraph 3.5.2.4. An assembler-generated label is uniquely identified by the characters preceding and/or appended to the "\@" sequence and the macro invocation in which the reference occurs. Such labels may appear anywhere in the source line and will always refer to the current macro expansion.

   NOTE: Space compression is automatically done within macros. For example, the instruction DC.B '   ' becomes DC.B ' '.

d. When a line has been completely expanded, the line assembled as any other source input line. At this time, any errors in the syntax of the expanded assembly code are found. Expanded lines longer than 80 characters are truncated and an error code is generated.

   If a nested macro call is encountered, the nested macro expansion takes place recursively. There is no set limit to the depth of macro call nesting.

3.7.3 CONDITIONAL ASSEMBLY

Conditional assembly allows the programmer to write a comprehensive source program that can cover many conditions. Assembly conditions may be specified through the use of arguments in the case of macros, and through definition of symbols via the SET and EQU directives. Variations of parameters can then cause assembly of only those parts necessary for the specified conditions.

The I/O section of a program, for example, will vary, depending on whether the program is used in a disk environment or in a paper tape environment. Conditional assembly directives can include or exclude an I/O section, based on a flag set at the beginning of the assembly.

3.7.3.1 Conditional Assembly Structure

The conditional assembly structure consists of three parts:

a. The header

   There are two conditional clauses recognized by the assembler. The first form compares the equality of two strings:

   ```
   IFxx    '<string>','<string2>'
   ```

   "xx" specifies either the string compare (C) condition or the string not compare (NC) condition, representing string equality and inequality, respectively. The result of the string comparison, along with the 'xx' condition, determines whether the body of the conditional structure will be assembled. Either string may contain embedded commas or spaces. An apostrophe that occurs within a string must be specified by double apostrophes.

   The second form of the conditional clause compares an expression against zero:

   ```
   IFxx    expression
   ```

   "xx" specifies a conditional relation between the expression and the value zero. The result of this comparison at assembly time determines whether the body of the conditional structure will be assembled. Valid conditional relation codes include:

   ```
   EQ  :  expression =  0
   ```

```
NE  :  expression <> 0
LT  :  expression <  0
LE  :  expression <= 0
GT  :  expression >  0
GE  :  expression >= 0
```

Because of the nature of this comparison, the expression must be absolute. No forward references are allowed.

b. The body

The body of the conditional assembly structure consists of a sequence of standard source statements. There is no set limit to the depth of conditional assembly nesting; if such nesting occurs, a terminator must be specified for each structure.

c. The terminator: ENDC

When an IFxx directive is encountered, the specified condition is evaluated. If the condition is true, the statements constituting the body of the conditional assembly structure are each assembled in turn. If the relation is false, the entire conditional assembly structure is ignored; the ignored lines are not included in the assembly listing. By specifying the OPT NOCL option (paragraph 3.5.2.10), the header and terminator lines will be ignored for listing purposes.

IFxx and ENDC directives may not be labeled.

Testing for null parameters may be done via the string compare form of the conditional assembly. To assemble conditionally if parameter 1 is null, either of the following directives would be correct:

```
IFxx  '','\1'

    or

IFxx  '\1',''
```

To assemble conditionally if a parameter is present would use either of the IFNC formats analogous to the above two.

A conditional assembly structure is also terminated by a MEXIT directive, as explained in paragraph 3.5.2.5. All conditional assembly structures which originate in a macro are terminated at the exit from that macro (if not before). Only conditional assembly structures which originated within a given macro may be terminated within that macro. These two rules are necessary for the consistent implementation of conditional assembly.

3.7.3.2 Example of Macro and Conditional Assembly Usage

The following example illustrates most of the features of macros and conditional assembly structures. The assembly code is shown as it would appear without line numbers or object code.

```
          .
          .
          .
MACO      MACRO
          MOVE.\0      \1
          CLR.L        \2
          ENDM
          .
          .
          .
MAC1      MACRO
          MOVE.\0      #\1,D\2
          IF\3         \1              CONDITIONAL
          ADD.\0       #1,D\2
          IF\3         \1-5            NESTED CONDITIONAL
          ADD.\0       #2,D\2          \4
          ENDC                         END NESTED CONDITIONAL
          ENDC                         END CONDITIONAL
LAB\@     CLR.L        D1
          MOVE.\0      D\2,(A0)+
          B\3          \@END
          BRA          LAB\@
\@END     \5.\0        #1,D\2
          IFLE         \1
          MACO.\0      <D\2,(A0)>,A\2  NESTED MACRO CALL
          ENDC
          ENDM
          .
          .
          .
          OPT          MEX,NOCL
          MAC1.L       7,3,GT,<TEST PASSES>,ADD
          MOVE.L       #7,D3
          ADD.L        #1,D3
          ADD.L        #2,D3           TEST PASSES
LAB.001   CLR.L        D1
          MOVE.L       D3,(A0)+
          BGT          .002END
          BRA          LAB.001
.002END   ADD.L        #1,D3
          .
          .
          .
          MAC1        0,6,NE,<ERROR HERE>,SUB
          MOVE.       #0,D6
LAB.003   CLR.L       D1
          MOVE.       D6,(A0)+
          BNE         .004END
          BRA         LAB.003
.004END   SUB.        #1,D6
          MACO.       <D6,(A0).,A6       NESTED MACRO CALL
          MOVE.       D6,(A0)
          CLR.L       A6
          .
          .
          .
```

3.8 STRUCTURED CONTROL STATEMENTS

3.8.1 INTRODUCTION

An assembly language provides an instruction set for performing certain rudimentary operations. These operations, in turn, may be combined into control structures -- such as loops, (for, repeat, while) or conditional branches (if-then, if-then-else). The assembler, however, accepts formal, highlevel directives that specify these control structures, generating, in turn, the appropriate assembly language instructions for their efficient implementation. This use of structured control statement directives improves the readability of assembly language programs, without comprimising the desirable aspects of programming in an assembly language.

3.8.2 KEYWORD SYMBOLS

The following Class 1 symbols, used in the structured syntax, are reserved keywords (directives):

ELSE	ENDW	REPEAT
ENDF	FOR	UNTIL
ENDI	IF	WHILE

The following symbols are required in the structured syntax, but are nonreserved keywords:

AND	DOWNTO	TO
BY	OR	
DO	THEN	

Note that AND and OR are reserved instruction mnemonics, however.

3.8.3 SYNTAX

The formats for the IF, FOR, REPEAT, and WHILE statements are found in paragraphs 3.6.3.1 through 3.6.3.4. They are spaced to show the line separations required for Class 1 symbol usage (paragraph 3.6.5.1). Syntactic variables used in the formats area as follows:

<expression>	A simple or compound expression (paragraph 3.6.4).
<stmtlist>	Zero or more assembler directive (Section 3.3) occurring within a structured control statement is examined once - at assembly time. Thus, the presence of a directive within a FOR, REPEAT, or WHILE statement does not imply repeated occurrence of an assembler directive; nor does the presence of a directive within an IF-THEN-ELSE statement imply a conditional assembly structure (Section 3).
<size>	The value B, W, or L, indicating a data size of byte, word, or long, respectively. With the keyword FOR, <size> is a single code applying to <op1>, <op2>, <op3>, and <op4>. With the keywords IF, UNTIL, and WHILE, <size> indicates the size of the operand comparison in the subsequent simple expression (see paragraph 3.6.4.2 for a compound expression).
<extent>	The value S or L, indicating that the branch extent is short or long, respectively. This is appended to the keywords THEN, ELSE, and DO, to force the appropriate extent of the forward branch over the subsequent <stmtlist>. The default extent is determined by the option directive (OPT BRS or OPT BRL) currently in effect.
<op1>	A user-defined operand whose memory-register location will hold the FOR-counter. The effective address must be an alterable mode.

<op2>	The initial value of the FOR-counter. The effective address may be any mode.
<op3>	The terminating value for the FOR-counter. The effective address must be any mode.
<op4>	The step (increment/decrement) for the FOR-counter each through the loop. If not specified, it defaults to a value of #1. The effective address may be any mode.

3.8.3.1 IF Statement

SYNTAX:

```
IF[.<size>]  <expression> THEN[.<extent>]
  <stmtlist>
ENDI

     or

IF[.<size>]  <expression> THEN[.<extent>]
  <stmtlist>
ELSE[.<extent>]
  <stmtlist>
ENDI
```

FUNCTION: If <expression> is true, execute the <stmtlist> following THEN; if <expression> is false, execute the <stmtlist> following ELSE, if present, or advance to next instruction.

NOTES: a. If an operand comparison <expression is specified, the condition codes are set and tested before execution of the <stmtlist>.

b. In the case of nested IF-THEN-ELSE statements, each ELSE will refer to the closest IF-THEN.

3.8.3.2 FOR Statement

SYNTAX:

```
FOR[.<size>]  <op1>  =  <op2>  TO  <op3>  [BY  <op4>]
DO[.<extent>]
  <stmtlist>
ENDF
```

or

```
FOR[.<size>] <opl> = <op2> DOWNTO <op3> [BY <op3>]
DO[.<extent>]
  <stmtlist>
ENDF
```

FUNCTION: These counting loops utilize a user-defined operand, <opl>, for the loop counter. FOR-TO allows counting upward, while FOR-DOWNTO allows counting downward. In both loops, the user may specify the step size, <op4>, or elect the default step size of]1. The FOR-To loop is not executed if <op2> is greater than <op3> upon entry. Similarly, the FOR-DOWNTO loop is not executed if <op2> is less than <op3>.

NOTES: a. The condition codes are set and tested before each execution of the <stmtlist>. This happens even if <stmtlist> is not executed.

b. A step size of]1 may not be meaningful if the counter, <opl>, is used to index through word or longword-sized data.

c. Each immediate operand must be preceded by a "#" sign. For example, the following would loop ten times by steps of four.

```
FOR COUNT = #4 TO #40 BY #4 DO ...
```

d. The FOR structure generates a move, a compare and either an add or subtract. Therefore, if any of the four operands is an A register, <size> may not be B (byte).

3.8.3.3 REPEAT Statement

SYNTAX:

```
REPEAT
  <stmtlist>
UNTIL[.<size>] <expression>
```

FUNCTION: <stmtlist> is executed at least once, even if <expression> is true.

NOTES: a. The <stmtlist> is executed at least once, even if <expression> is true upon entry.

b. If an operand comparison <expression> is specified, the condition codes are set and tested following each execution of the <stmtlist>.

3.8.3.4 WHILE Statement

SYNTAX:

```
WHILE[.<size>] <expression> DO[.<extent>]
  <stmtlist>
ENDW
```

FUNCTION: The <expression> is tested before execution of the <stmtlist>. While the <expression> is true, the <stmtlist> is executed repeatedly.

NOTES: a. If the <expression> is false upon entry, <stmtlist> is not executed.

b. If an operand comparison <expression> is specified, the condition codes are set and tested before each execution of the <stmtlist>. The condition codes are set and tested even if the <stmtlist> is not executed.

3.8.4 SIMPLE AND COMPOUND EXPRESSIONS

Expressions are an integral part of IF, REPEAT, and WHILE statements. An expression may be simple or compound. A compound expression consists of no more than two simple expressions joined by AND or OR.

3.8.4.1 Simple Expressions

Simple expressions are concerned with the bits of the Condition Code Register (CCR). These expressions are of two types. The first type merely tests conditions currently specified by the contents of the CCR. The second type sets up a comparison of two operands to set the condition codes, and afterwards tests the codes.

CONDITION CODE EXPRESSIONS

Fourteen tests (identical to those in the Bcc instruction) may be performed, based on the CCR condition codes. The condition codes, in this case, are preset by either a user-generated instruction or a structured operand-comparison expression. Each test is expressed in the structured control statement by a mnemonic enclosed in angle brackets (< >), as follows:

```
              <CC>
              <CS>
              <EQ>
              <GE>
              <GT>
              <HI>
              <LE>
              <LS>
              <LT>
              <MI>
              <NE>
              <PL>
              <VC>
              <VS>
```

For example:

```
     IF        <EQ>  THEN
       CLR.L     D2
     ENDI

     REPEAT
       SUB
                 D4,D3
     UNTIL     <LT>
```

OPERAND COMPARISON EXPRESSIONS

Two operands may be compared in a simple expression with subsequent transfer of control based on that comparison. Such a comparison takes the form:

```
     <opl> <cc> <op2>
```

where <cc> is a condition mnemonic enclosed in angle brackets (as described in paragraph 3.6.4.1.1), specifying the relation to be tested between <opl> and <op2>. When processed by the assembler, this expression translates to a compare instruction - for example:

```
     CMP    <opl>,<op2>
```

followed by a branch instruction (Bcc) which tests the relation specified. <op1> is normally assigned to the first (leftmost) operand and <op2> to the second (rightmost) operand of the compare instruction.

A size may be specified for the comparison by appending a data size code (B, W, or L) to the directive, with W being the default. The only restriction is that a byte size code (B) may not be used in conjunction with an address register direct operand.

Compare instructions require certain effective addressing modes for their operands. These modes are listed in Table 3-5. However, if the operands, <op1> and <op2>, are not listed in an order that generated a legal compare instruction (Table 3-5), but that will generate a legal compare if the operand order is reversed, the assembler will reverse the operands when expanding the expression. To maintain the nature of the relation specified, the condition operator will also be adjusted, if necessary. For example, "D2 <GT> #5" would be adjusted by the assembler to the equivalent of "#5 <LT> D2"; likewise, "A2 <EQ> (A5)" would be adjusted to the equivalent of "(A5) <EQ> A2". This processing allows the user flexibility of specifying the more meaningful operand order in the expression.

TABLE 3-5. Effective Addressing Modes for Compare Instructions

COMPARE INSTRUCTIONS	EFFECTIVE ADDRESSING MODES FOR:	
	FIRST OPERAND	SECOND OPERAND
CMP	(All)	Data register direct
CMPA	(All)	Address register direct
CMPI	Immediate	(Data alterable)
CMPM	Postincrement register indirect	Postincrement register indirect

If the operands, either as stated or reversed, do not yield a legal compare instruction, an error will result. For example, the statement

```
     IF      (A1) <NE> (A2) THEN
```

would result in an "ERROR 213" message - illegal address mode - during expansion. To avoid this error, a MOVE would be required to effect a legal operand, such as:

```
     MOVE     (A2),D2
     IF       (A1) <NE> D2 THEN
```

Examples:

```
     WHILE.B   (A3) <NE> D2 DO           THIS EXPRESSION IS LEGAL AS STATED.
       MOVE.B    (A5)+,D2
     ENDW

     IF        D7 <LT> #10  THEN         THIS EXPRESSION WILL BE REVERSED.
       BRS       SUBR1
     ELSE
       MULS      #2,D7
     ENDI
```

3.8.4.2 Compound Expressions

A compound expression consists of two simple expressions (paragraph 3.6.4.1) joined by a logical operator. The Boolean value of the compound expression is determined by the Boolean values of the simple expressions and the nature of the logical operator (AND or OR).

The two simple expressions are evaluated in the order in which they are given. However, if an AND separates the expressions and the first expression is false, the second expression will not be evaluated. Likewise, if an OR separates the first expression is true, the second expression will not be evaluated. In these cases, the compound expression is either false or true, respectively, and the condition codes reflect the result of only the first simple expression.

A size may be specified for each operand comparison expression. The size of the comparison for the first expression may be appended to the directive, while the size of the comparison for the second expression may be appended to the keyword AND or OR. For example, in the statement

```
     IF.L   D3 <GT> (A0)  OR.B  #'Q' <EQ> BUFFER1
```

the first comparison is a longword comparison, and the second is a byte comparison.

3.8.5 SOURCE LINE FORMATTING

3.8.5.1 Class 1 Symbol Usage

Class 1 symbols, as described in paragraph 3.6.2, are the assembler directives (including macro names), instructions mnemonics and the structured control directives. Only one of these is recognized on each source line. Thus, each directive (reserved keyword) of a structured control statement and each executable instruction generated by the programmer must be written on a separate source line. The following source line, for example, is in error:

```
     REPEAT  MOVE -(A5),D2  UNTIL <EQ>
```

because the MOVE and UNTIL symbols and their operands are not recognized, but are treated as part of the comment field of the REPEAT directive. Likewise, the following lines are in error:

```
     IF      <VS> THEN JSR OVERFLOW
     ELSE    JMP (A3) ENDI
```

because the JSR, JMP, and ENDI symbols and their operands are not recognized. The correct format for these lines would be as follows:

```
     REPEAT
       MOVE      -(A5),D2
     UNTIL     <EQ>
```

and

```
     IF        <VS> THEN
       JSR       OVERFLOW
     ELSE
       JMP       (A3)
     ENDI
```

3.8.6 LIMITED FREE-FORMATTING

To improve readability, limited free-formatting allows the operand field of the IF, UNTIL, WHILE, and FOR directives to be extended onto additional consecutive lines.

For example:

```
    IF      #15 <LT> D7
              AND
            (A3) <NE> D3 THEN

    UNTIL   (A7)+ <EQ> D2 OR
                  <VS>

    FOR     D1 = #1 to #5
                 BY #1 DO
```

3.8.6.1 Nesting of Structured Statements

Structured statements may be nested as desired to create multi-level control structures. An example of such nesting is the following:

```
    IF        <EQ>  THEN
      REPEAT
        MOVE      D0,(A5)+
        ADDQ      #4,D0
        MOVE.L    A4,(A4)+
      UNTIL.L   A5 <LE> A4
    ELSE.L
      FOR       D2 = #10  TO #20  BY #2  DO
        WHILE     D4 <LT> D2  AND  D4 <LT> #100  DO
          MOVE.L    10(A3,D4.W),(A5)+
          ADDQ      #2,D4
        ENDW
      ENDF
    ENDI
```

3.8.6.2 Assembly Listing Format

By default (FORMAT directive), the assembly listings are formatted according to Table 3-4. In addition, the operation and operand fields of source lines in structures syntax are indented two columns for each nested level of operation. This automatic formatting may be turned off by using the NOFORMAT directive.

The assembly language code generated for the structured syntax is included in the listing when the S option is specified in the ASM command line.

3.8.7 EFFECTS ON THE USER'S ENVIRONMENT

If the S option is specified in the ASM command line (paragraph 3.4.1), the generated code of the structured control expansions is listed. There may be three items found in this code that will affect the user's environment:

a. During assembly, local labels beginning with "Z_L" are generated. These labels use the same increment counter (.nnn) as local labels in macros (see paragraph 3.5.2.4). They are stored in the symbol table and should not be duplicated in user-defined labels.

b. In the FOR loop, <op1> is a user-defined symbol. When exiting the loop, the memory/register assigned to this symbol contains the value which caused the exit from the loop.

c. Compare instructions (see Table 3-5) are generated by the assembler whenever two operands are tested relationally in a structured statement. During runtime, however, these assembler-generated instructions set the condition codes of the CCR (in the case of a loop, the condition codes are set repeatedly). Any user-written code, therefore, either within or following a structured statement, that references the CCR should be attentive to the effect of these instructions.

3.9 GENERATING POSITION INDEPENDENT CODE

3.9.1 FORCING POSITION INDEPENDENCE

When creating a relocatable program module, it is often desirable to ensure that all references to operands in relocatable sections are position independent effective addressed -- i.e., that no absolute addresses occur as effective addresses for such references. To avoid absolute effective address formats, it is necessary to ensure that all memory operand references are resolved by the assembler (or by the linkage editor at the assembler's direction) into one of the program counter relative or address register indirect addressing modes. Avoiding ORG directives is not sufficient to ensure position independence, since it is possible for the assembler to produce absolute effective address formats even when no absolute symbols have been defined.

For example, if an instruction references a symbol that is not yet defined, or is defined either in another section or as an XREF in an unspecified section, the default action of the assembler is to direct the linkage editor to resolve the reference by supplying the absolute address of the symbol. By specifying OPT PCS, all references known to be in a relocatable section will be resolved as a Program Counter (PC) relative address. However, this does not solve the problem of forward references, which would still default to absolute format. To override an absolute address mode when resolving the effective address format of an operand, the following formats may be used to force program counter relative addressing:

a. Forcing program counter with displacement

   An operand of the form: LABEL(PC)

   will be resolved as a PC with displacement effective address, either by the assembler or by linkage editor (at the assembler's direction). If LABEL cannot be resolved into a 16-bit displacement from the program counter, an error will be generated.

b. Forcing PC with index plus displacement

   An operand of the form: LABEL(PC,Rn)

   will be resolved as a PC with index plus displacement effective address by the assembler. Since the displacement in this mode is 8 bits, the reference must be resolvable by the assembler. If LABEL cannot be resolved by the assembler into an 8-bit displacement from the program counter, an error will be generated.

3.9.2 BASE-DISPLACEMENT ADDRESSING

Although PC relative addresses have the advantage of position independence, such address formats are often not the most meaningful to the programmer when debugging an assembled module. There are many times when a programmer would prefer to see an address relative to a specified base - i.e., in a base-displacement format. This is especially true when addressing tables, arrays, and other data structures. Base-displacement references to a given location are "base relative" and, therefore, fixed with respect to a given base address; PC relative references to that same location are different in each instruction.

Base-displacement addressing must be handled explicitly by the programmer. For example, if the following data area is declared:

```
TEMP        DS        $40
CONST       DC        $10
ARRAY1      DS.L      $10
ARRAY2      DS.L      $10
RESULT      DS.L      $10
```

the programmer may choose to load A6 with the address of TEMP and make references to the other data locations as displacements from this base address. For example, to move the first element of ARRAY1 to D1, the programmer may specify:

```
MOVE.L    ARRAY1-TEMP(A6),D1
```

Indexing with the low order contents of D0 may be added (as the array index):

```
MOVE.L    ARRAY1-TEMP(A6,D0),D1
```

3.9.3 BASE-DISPLACEMENT IN CONJUNCTION WITH FORCED POSITION INDEPENDENCE

Complete code position independence can be achieved by using base-displacement addressing in conjunction with the PCS option and the forced PC relative addressing scheme outlined in Paragraph 3.7.1. Although these techniques can be used to avoid all undesired absolute address formats, there are significant limitations of PC relative addressing in a position independent program, as noted below:

a. PC with displacement

   Pc with displacement effective addresses are only restricted by the 16-bit displacement field. A displacement greater than 32K bytes from the current PC cannot be resolved in this format.

b. PC with index plus displacement

   The displacement field here is restricted to 8 bits, limiting the range of this format to a 128-byte diplacement from the current PC. This 8-bit displacement is not relocatable. Therefore, only symbols with a known displacement from the program counter may be resolved in a PC with index plus displacement format.

c. Operands in the alterable addressing category

Neither PC relative mode is allowed as an alterable operand. This is a significant limitation in instructions which require an alterable operand, such as the destination operand in a MOVE instruction.

By appropriate use of base registers, these limitations can be overcome.

4.0 LINKER AND LIBRARY UTILITIES

4.1 INTRODUCTION

The Linker (ALINK) and Library (LIBRARY) utilities are a pair of complementary programs that aid in the process of generating executable programs under the Computer System operating system.

The Linker links or binds relocatable object-code modules, and optional modules from libraries, to form a program that is executable.

The Library utility builds a library by combining one or more relocatable object-code modules. Such a library may contain frequently used procedures (such as the mathematical functions of FORTRAN) which can be used in subsequent link processes.

4.1.1 BUILDING AN EXECUTABLE PROGRAM

To get from the source file of a program to an executable object code file, the user must proceed as follows:

1. The source file is compiled or assembled without errors. The result of compiling or assembling is a relocatable object-code file of type '.OBJ'.

2. The relocatable object-code is linked, and may include run-time support libraries. The result is an executable code file of type .BIN.

3. The program can then be run (executed) on the Computer System simply by typing its filename.

The following sections in this manual describe the Linker and Librarian object-code management system.

4.2 LINKER

The Linker is a utility which accepts files of relocatable object-code generated by the compilers and assembler, plus library files generated by the Library utility, and links or binds those into a form suitable for execution.

As well as binding together relocatable modules from various language processors, the Linker can search libraries of commonly used functions, (such as the PASCAL run time environment), and link only those modules that are referenced into the final loadable output file.

In order to link relocatable modules into an executable object-code file, the Linker requires the following pieces of information:

- The optional name of the listing file where the Linker messages and memory map information is to be listed. If no listing file name is given, no memory map information is generated, and messages appear on the user's console.
- The name of the object-code file on which to write the final linked output.
- The name(s) of the file(s) from which the relocatable object-code is read.
- A list of one or more libraries which are to be used to satisfy external references within the object-code file.

When linking a mixture of Pascal or Fortran object modules and Assembler object modules, the main program must be in Pascal or Fortran. Note that the linker produces a jump to an external reference when this reference is defined in a different segment, so that the only communication between Assembler and high level language modules should be calls to entry points of routines.

4.2.1 INVOKING ALINK

The command line format for ALINK is:

```
ALINK maininfile[,infile][,+L=listfile][,+L][,+P=commandfile][,+P][,-P]
     [,+S=startadr][,+M=memsize][,+O=outfile]
```

Where:

maininfile	The name of an object code file to which control will pass when the finished program is executed. The default extension for this file is .OBJ. The default extension is used if the user does not supply an extension. If no main input file is provided, then ALINK will use <outfile.OBJ> by default.
infile	The name of an object code file to be linked. The default extension for this file is .OBJ. The default extension is used if the user does not supply an extension. Up to 50 input files and options may be specified.
+L=listfile	Commands ALINK to place its listing file on file <listfile>. The default extension for this file is .MAP. The default extension is used if the user does not supply an extension. If no +L= or +L option is given, no listing output is generated.
+L	Commands ALINK to place its listing file on file <mainfile.LST>. If no +L= or +L option is given, no listing output is generated.
+P=commandfile	Commands ALINK to take further directives (options and input files) from <commandfile>. The default extension for this file is .CMD. The default extension is used if the user does not supply an extension.
+P	Commands ALINK to take further directives (options) from file <maininfile.CMD>, one to a line.
-P	Use no command file.
+S=startadr	Sets startadr as the address at which the linked output module will be loaded when it is executed. To specify startadr in hexadecimal, begin the number with a $. The default load address is $E000. If you specify an illegal number, the default value will be used.
+M=memsize	Makes memsize bytes of stack/heap available to the linked output module when it is executed. To specify memsize in hexadecimal, begin the number with a $. The default stack/heap size is 32767 bytes and the minimum stack/heap size is 4096 bytes. If you specify 0, all RAM available at run time will be used. If you specify an illegal number, the default value will be used. If a nonzero memsize is given, and that amount of RAM is not available at run time,

a message 'Not enough stack/heap space' will be printed at run time.

outfile	The name of a file which will be output from ALINK. The default extension for this file is .BIN. The default extension is used if the user does not supply an extension. If no +O= option is used, the file <maininfile.BIN> is used.

4.2.2 LINKER ERROR MESSAGES

The Linker can display various error messages in the course of its operation. The error messages are self-explanatory. There are three grades of error messages, with different outcomes:

Warnings	are correctable errors. The error can be corrected and the link proceeds. For example, misspelling a filename will result in a message to the effect that the file cannot be opened, at which point the filename can be retyped.
Errors	are correctable in that the user can proceed with the link process, but the generated object-code file is not created properly.
Fatal errors	are those from which the Linker cannot correct or recover. In those cases the linker returns to the system, and a prompt appears on the command line.

4.3 LIBRARY UTILITY

The Library Program binds compiled or assembled relocatable object-code modules (.OBJ) into a collection called a library. The purpose of a library is to provide a repository for commonly used object modules that have to be present when linking (see the Linker description), such that the common modules end up bound together into the final executable code module.

The library utility typically requires the following pieces of information from the user:

- The name of the file which is to receive the listing (results and log) of the library process.

- The name of the file which is to contain the generated library when the library generation process is complete.
- The name(s) of file(s) (with the .obj) suffix, which contain the constituent parts of the library to be generated.

4.3.1 INVOKING THE LIBRARY PROGRAM.

The command line format for the library program is:

```
LIBRARY outfile,infile[,infile]...[,infile][,+L=listfile][,+L][+P=promptfile][,+P]
```

where:

outfile	The name of the file that LIBRARY will use for its output. The default extension for this file is .OBJ. The default extension is used if the user does not supply an extension.
infile	The name of a file which will be input to LIBRARY. The default extension for this file is .OBJ. The default extension is used if the user does not supply an extension.
+P=promptfile	Commands LIBRARY to take further directives (input files or options) from file <promptfile>. The default extension for this file is .CMD. The default extension is used if the user does not supply an extension.
+P	Commands LIBRARY to take further directives (input files or options) from file <outfile.CMD>, one to a line.
+L=listfile	Commands LIBRARY to place its listing file on file <listfile>. The default extension for this file is .LST. The default extension is used if the user does not supply an extension. If no +L= or +L option is given, no listing output is generated.
+L	Commands LIBRARY to place its listing file on file <outfile.LST>. If no +L= or +L option is given, no listing output is generated.

If the Librarian cannot find the specified input file it issues a message to the effect:

The file 'whatever.obj' can't be opened

4.4 OBJECT FILE FORMATS

This chapter describes the layout of the object-code files that the Linker and Librarian can process.

4.4.1 NOTATION USED TO DESCRIBE OBJECT FILE FORMATS

The symbol "::=" is read as "defined to be". Where a whole list of objects appear to the right of a "pile" of "::=" signs, it implies a choice of any of the objects.

Objects enclosed in "angle brackets", "<" and ">" are syntactic objects which are defined in terms of other objects.

An object followed by an asterisk sign, "*", can be repeated "zero to many times" (the list of objects can be empty).

An object followed by a plus sign, "+", can be repeated "one to many times" (there must be at least one of that object).

4.4.2 LINKER FILE LAYOUT

This section is a description of the Linker File at the "top level".

```
<Link File>       ::= <Module File>
                  ::=   <Library>
                  ::=     <Unit File>
                  ::=       <Execute File>

<Module File>     ::= <Module>* EOF mark

<Library File>    ::= <Library Module Block>+ <Library Entry Block>+
                      <Module>+  <Text Block>*  EOF Mark
<Unit File>       ::= <Unit Block> <Module>+ <Text Block> EOF Mark

<Execute File>    ::= <Executable Block> <Module>*
                  ::=   <Quick Load Block>

<Module>          ::= <Module Name Block> <Other Block>+ <End Block>

<Other Block>     ::=  Entry Block
                  ::=  External Block
::=  Start Block
::=  Code Block
::=  Relocation Block
::=  Common Relocation Block
::=  Common Definition Block
::=  Short External Block
::=  Data Initialization Block
::=  FORTRAN data area definition block
::=  FORTRAN data area Initialization Block
::=  FORTRAN Data Area Reference Block
::=  Executable Data Area Initialization Block
::=  FORTRAN Executable Data Area Reference Block
```

4.4.3 BYTE-LEVEL DESCRIPTION OF LINKER BLOCKS

All Linker and Librarian object-code blocks start with a single "identifier byte." This block identifier takes values from 80 (base 16) upwards.

4.4.3.1 #80 - Module Name Block

byte -->	0	80	size (3 bytes)
	4	module name (8 bytes)	
	12	segment name (8 bytes)	
	20	csize (4 bytes)	
	24	comments (24 .. size-1 bytes) ...	

80	Hexadecimal 80 indicates a Module Name Block.
size	Number of bytes in this block.
module name	Blank padded ASCII name of module.
segment name	ASCII name of segment in which this module will reside.
csize	Number of bytes in the code block for this module.
comments	Arbitrary information - ignored by the Linker.

4.4.3.2 #81 - End Block

byte -->	0	81	size (3 bytes)
	4	csize (4 bytes)	

81	Hexadecimal 81 indicates this is an End Block.
size	Number of bytes in this block - it is always 000008.
csize	Number of bytes in the code block for this module.

4.4.3.3 #82 - Entry Point Block

byte -->		
0	82	size (3 bytes)
4 8	link name (8 bytes)	
12	user name (8 bytes)	
20	loc (4 bytes)	
24	comments (24 .. size-1 bytes) ...	

82	Hexadecimal 82 indicates this is an Entry Point Block
size	Number of bytes in this block.
link name	Blank padded ASCII Linker name of entry point.
user name	Blank Padded ASCII user name of entry point.
loc	Location of entry point relative to this module.
comments	Arbitrary information - ignored by the Linker.

4.4.3.4 #83 - External Reference Block

```
             ┌─────────┬──────────────────────────┐
byte -->   0 │   83    │     size (3 bytes)       │
             ├─────────┴──────────────────────────┤
           4 │            link name               │
           8 │            (8 bytes)               │
             ├────────────────────────────────────┤
          12 │            user name               │
             │            (8 bytes)               │
             ├────────────────────────────────────┤
          20 │         ref 1 (4 bytes)            │
             ├────────────────────────────────────┤
          24 │         ref 2 (4 bytes)            │
             ├────────────────────────────────────┤
             │              . . .                 │
             ├────────────────────────────────────┤
             │ each reference consumes 4 bytes    │
             ├────────────────────────────────────┤
             │              . . .                 │
             ├────────────────────────────────────┤
      16+4*n │         ref n (4 bytes)            │
             └────────────────────────────────────┘
```

83	Hexadecimal 83 indicates this is an External Reference
size	Number of bytes in this block.
link name	Blank padded ASCII Linker name of external reference.
user name	Blank padded ASCII user name of external reference.
ref 1	Location of first reference relative to this module.
ref 2	Location of second reference relative to this module.
. . .	Other references.
ref n	Location of last reference relative to this module.

4.4.3.5 #84 - Starting Address Block

```
              ┌───────┬──────────────────────────┐
byte -->   0  │  84   │     size (3 bytes)       │
              ├───────┴──────────────────────────┤
           4  │         start (4 bytes)          │
              ├──────────────────────────────────┤
           8  │         gsize (4 bytes)          │
              ├──────────────────────────────────┤
          12  │ comments (12 .. size-1 bytes) ...│
              └──────────────────────────────────┘
```

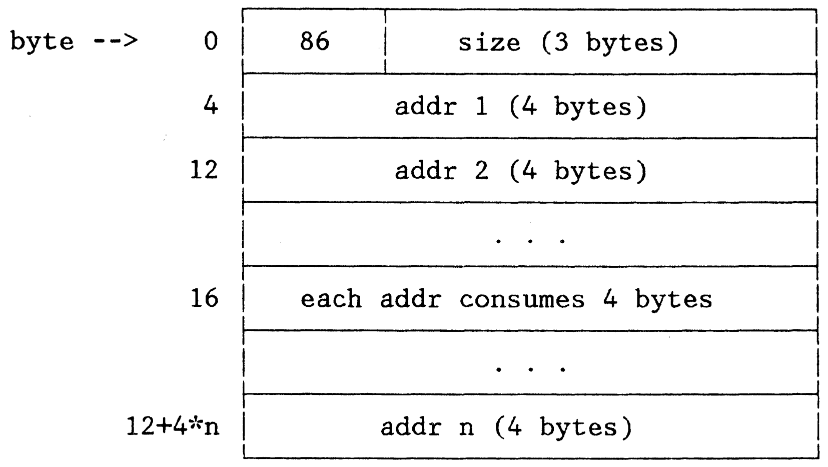
84	Hexadecimal 84 indicates this is a Starting Address Block.
size	Number of bytes in this block.
start	Starting address relative to this module.
gsize	Number of bytes in the global data area.
comments	Arbitrary information - ignored by the Linker.

4.4.3.6 #85 - Code Block

```
              ┌───────┬──────────────────────────┐
byte -->   0  │  85   │     size (3 bytes)       │
              ├───────┴──────────────────────────┤
           4  │          addr (4 bytes)          │
              ├──────────────────────────────────┤
           8  │ object-code (8..size-1 bytes) ...│
              └──────────────────────────────────┘
```

85	Hexadecimal 85 indicates this is a Code Block.
size	Number of bytes in this block.
addr	Module-relative address of first code byte.
object-code	The object-code - always an even number of bytes.

4.4.3.7 #86 - 32-Bit Relocation



86	Hexadecimal 86 indicates this is a 32-bit Relocation Block.
size	Number of bytes in this block.
addr 1	Location of first address to relocate.
addr 2	Location of second address to relocate.
. . .	Locations of other addresses to relocate.
addr n	Location of last address to relocate.

4.4.3.8 #87 - Common Block Reference

```
             ________________________________
byte -->  0 |  87   |     size (3 bytes)     |
            |_______|________________________|
          4 |          common name           |
            |           (8 bytes)            |
            |________________________________|
         12 |        ref 1 (4 bytes)         |
            |________________________________|
         16 |        ref 2 (4 bytes)         |
            |________________________________|
         20 |             . . .              |
            |________________________________|
            |each reference consumes 4 bytes |
            |________________________________|
            |             . . .              |
            |________________________________|
      8+4*n |        ref n (4 bytes)         |
            |________________________________|
```

87	Hexadecimal 87 indicates this is a Common Block Reference.
size	Number of bytes in this block.
common name	Blank padded ASCII common block name.
ref 1	Location of first reference relative to this module
ref 2	Location of second reference relative to this module.
. . .	Other references relative to this module.
ref n	Location of last reference relative to this module.

4.4.3.9 #88 - Common Block Definition

byte -->	0	88	size (3 bytes)
	4	common name (8 bytes)	
	12	dsize (4 bytes)	
	16	comments (16 .. size-1 bytes) ...	

88	Hexadecimal 88 indicates this is a Common Block Definition
size	Number of bytes in this block.
common name	Blank padded ASCII common data area name.
dsize	Number of bytes in this common data area.
comments	Arbitrary information - ignored by the Linker.

4.4.3.10 #89 - Short External Reference Block

byte -->	0	89	size (3 bytes)
	4	link name (8 bytes)	
	12	user name (8 bytes)	
	20	ref 1 (2 bytes)	ref 2 (2 bytes)
	18+2*n	. . .	ref n (2 bytes)

89	Hexadecimal 89 indicates this is a Short External Reference Block.
size	Number of bytes in this block.
link name	Blank padded ASCII Linker name of external reference.
user name	Blank padded ASCII user name of external reference.
ref 1	Location of first reference relative to this module.
ref 2	Location of second reference relative to this module
. . .	Location of other references relative to this module.
ref n	Location of last reference relative to this module.

4.4.3.11 #8A - FORTRAN Data Area Definition Block

```
                 ┌────────┬──────────────────────────┐
byte -->      0  │   8A   │     size (3 bytes)       │
                 ├────────┴──────────────────────────┤
              4  │          data area name           │
                 │            (8 bytes)              │
                 ├───────────────────────────────────┤
             12  │          dsize (4 bytes)          │
                 └───────────────────────────────────┘
```

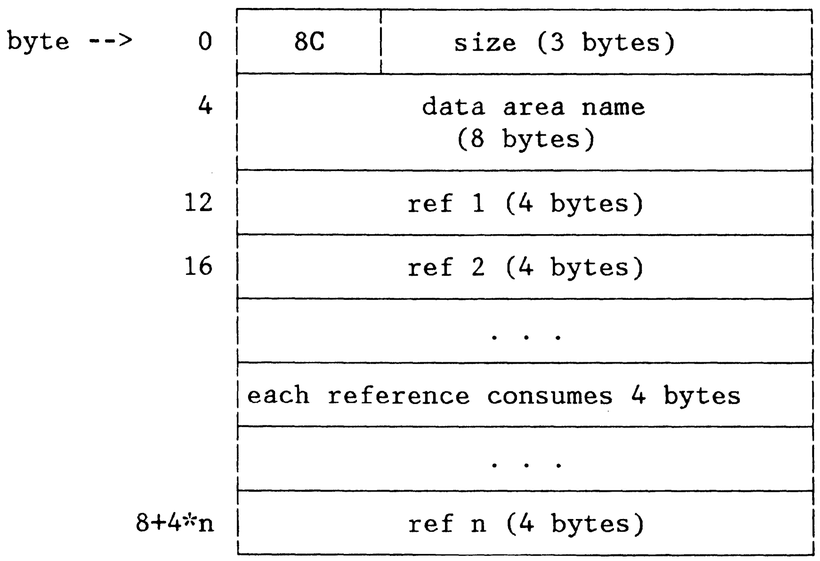
8A	Hexadecimal 8A indicates this is a FORTRAN Data Area Definition Block.
size	Number of bytes in this block.
data area name	Blank padded ASCII name of FORTRAN fixed data area.
dsize	Size of this data area.

4.4.3.12 #8B - FORTRAN Data Area Initialization Block

```
                ┌─────────┬─────────────────────────┐
byte -->     0  │   8B    │     size (3 bytes)      │
                ├─────────┴─────────────────────────┤
             4  │          data area name           │
                │            (8 bytes)              │
                ├───────────────────────────────────┤
            12  │          daddr (4 bytes)          │
                ├───────────────────────────────────┤
            16  │ data occupies bytes 16 .. size-1  │
                │ in the rest of the block    00 *  │
                └───────────────────────────────────┘
```

8B	Hexadecimal 8B indicates this is a FORTRAN Data Area Initialization Block.
size	Number of bytes in this block.
data area name	Blank padded ASCII name of FORTRAN fixed data area.
daddr	Starting address of this data.
data	The initialization data.
00 *	If the size of the data block is odd, there is one byte of 00 added to make the block an even number of bytes in size.

4.4.3.13 #8C - FORTRAN Data Reference Block



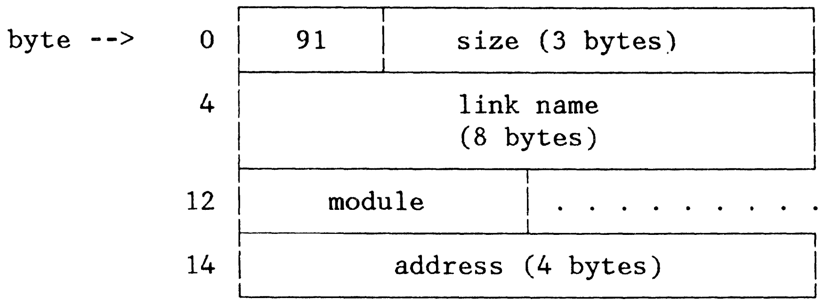
8C	Hexadecimal 8C indicates this is a FORTRAN Data Area Reference Block.
size	Number of bytes in this block.
data area name	Blank padded ASCII name of FORTRAN fixed data area.
ref 1	Location of first reference.
ref 2	Location of second reference.
. . .	Location of other references.
ref n	Location of last reference.

4.4.3.14 #90 - Library Module Block

byte -->		
0	90	size (3 bytes)
4	module name (8 bytes)	
12	msize (4 bytes)	
16	caddr (4 bytes)	
20	taddr (4 bytes)	
28	module count	module 1
32	module 2	. . .
	module n-1	module n

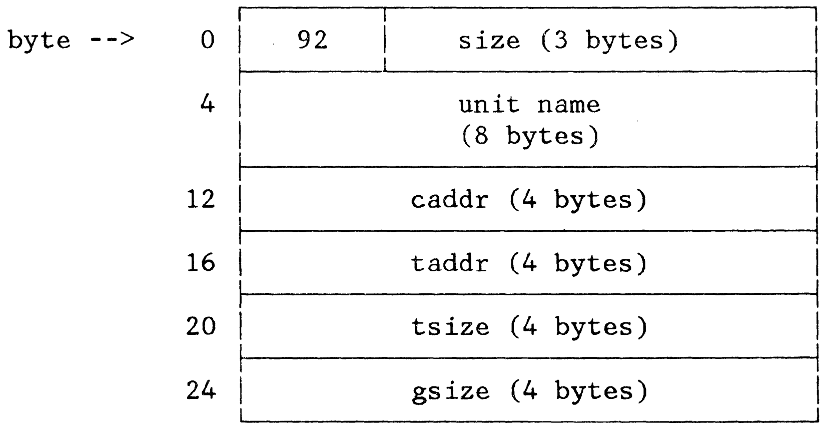
90	Hexadecimal 90 indicates this is a Library Module Block.
size	Number of bytes in this block.
module name	Name of this module.
msize	Number of bytes of code in this module.
caddr	Disk address of module.
taddr	If non-zero, is the disk address of the text block. If zero, there is no text block.
tsize	Size of text block.
module count	Number of other modules that this module references.
module 1	Number of the first module referenced.
module 2	Number of the second module referenced.
. . .	Numbers of other modules referenced.
module n	Number of the last module referenced.

4.4.3.15 #91 - Library Entry Block



91	Hexadecimal 91 indicates that this is a library entry block
size	Number of bytes in this block
link name	Blank-padded ASCII linker name of this entry point
module	Number of the module in which this entry appears
address	Address within this module of this entry point

4.4.3.16 #92 - Unit Block



92	Hexadecimal 92 indicates that this is a Unit Block.
size	Number of bytes in this block - always 0001C.

unit name	Name of this unit.
caddr	Disk address of module.
taddr	Disk address of text block.
tsize	Size of text block.
gsize	Number of bytes of globals in this unit.

4.4.3.17 #93 - FORTRAN Executable Data Area Reference Block

```
                 ┌─────────┬─────────────────────┐
byte -->      0  │   93    │    size (3 bytes)   │
                 ├─────────┴─────────┬───────────┘
              4  │   area number     │ . . . . . . . . . .
                 ├───────────────────┴─────────┐
              6  │       ref 1 (4 bytes)       │
                 ├─────────────────────────────┤
             10  │       ref 2 (4 bytes)       │
                 ├─────────────────────────────┤
                 │            . . .            │
                 ├─────────────────────────────┤
                 │ each reference consumes 4 bytes │
                 ├─────────────────────────────┤
                 │            . . .            │
                 ├─────────────────────────────┤
          2+4*n  │       ref n (4 bytes)       │
                 └─────────────────────────────┘
```

93	Hexadecimal 93 indicates this is a FORTRAN Executable Data Area Reference Block.
size	Number of bytes in this block.
area number	Data area number.
ref 1	Address of first reference.
ref 2	Address of second reference.
. . .	Addresses of other references.
ref n	Address of last reference.

4.4.3.18 #94 - FORTRAN Executable Data Area Initialization Block

```
               ┌─────────┬──────────────────────┐
byte -->    0  │   94    │    size (3 bytes)    │
               ├─────────┴─────┬────────────────┘
            4  │ data area number│ . . . . . . . . .
               ├───────────────┴────────────────┐
            6  │         daddr (4 bytes)        │
               ├────────────────────────────────┤
           10  │ initialization data . . . . . .│
               ├────────────────────────────────┤
               │ . . . . . . . . . . . . . . . .│
               ├──────────────────────┬─────────┤
               │ . . . . . . . . . . .│   00    │
               └──────────────────────┴─────────┘
```

94	Hexadecimal 94 indicates this is a FORTRAN Executable Data Area Initialization Block.
size	Number of bytes in this block.
data area number	Number of the FORTRAN Data Area.
daddr	Starting address for this data.
initialization data	The data to fill the block with.
00	If the size of the initialization data is an odd number of bytes, a filler of 00 is appended to make it an even number of bytes.

4.4.3.19 Text Block

```
              ┌──────────────────────────────┐
byte -->    0 │  Textual data .............  │
              ├──────────────────────────────┤
            4 │............................  │
              └──────────────────────────────┘
```

The format of a text block is operating system dependent. The current version uses the UCSD text file format, excluding the two initial header blocks.

A text block is always stored aligned on a disk block boundary.

4.4.3.20 EOF Mark

```
              ┌──────┬──────┐
byte -->    0 │  00  │  00  │
              └──────┴──────┘
```

An EOF (end of file) mark is indicated by a block containing two bytes of zero.

4.4.4 EXECUTABLE BLOCK DETAILS

This section describes the layout of an executable block. It includes details of the jump table and segment tables.

4.4.4.1 Layout of an Executable Block

```
                  ┌─────────┬──────────────────────────┐
bytes -->     0   │   8F    │      size (3 bytes)      │
                  ├─────────┴──────────────────────────┤
              4   │   Jump Table Address (4 bytes)     │
                  ├────────────────────────────────────┤
              8   │    Jump Table Size (4 bytes)       │
                  ├────────────────────────────────────┤
             12   │        Data Size (4 bytes)         │
                  ├───────────────────┬────────┬───────┤
             16   │        Num        │   00   │  00   │
                  ├─────────┬─────────┼────────┼───────┤
             20   │   00    │   00    │   00   │  00   │
                  ├─────────┴─────────┴────────┴───────┤
             24   │         Size 1 (4 bytes)           │
                  ├────────────────────────────────────┤
             28   │         Size 2 (4 bytes)           │
                  ├────────────────────────────────────┤
                  │              . . .                 │
                  ├────────────────────────────────────┤
        20+4*n    │         Size n (4 bytes)           │
                  ├────────────────────────────────────┤
        24+4*n    │ Jump Table (... size-1 bytes) ...  │
                  └────────────────────────────────────┘
```

8F	Hexadecimal 8F indicates this is an Executable Block Definition.
size	Number of bytes in this block.
jump table address	Absolute load address of jump table.
jump table size	Number of bytes in the jump table.
data size	Total number of bytes in global common data areas.
num	Number of FORTRAN Data Areas.

00 00 00 00 00 00	six bytes of zero filler.
size 1	Size of first FORTRAN Data Area.
size 2	Size of second FORTRAN Data Area.
. . .	Sizes of other FORTRAN Data Areas.
size n	Size of last FORTRAN Data Area.
jump table	The jump table itself, including the executable code for the loader.

If any FORTRAN Executable Data Area Initialization Blocks are present, they must immediately follow the executable block.

4.4.4.2 Format of the Jump Table

A4 --> $$TOP	Number of Segments (2 bytes)	
+2	Main Segment Table (32 bytes)	
+34	Segment Table #2 (32 bytes) . . . . Segment Table #n (32 bytes)	
2+n*32	Dummy Table #n+1 (4 bytes)	
	$_START Descriptor (10 bytes)	
	Segment #1 P#2 Descriptor . . . . Segment #1 P#n Descriptor	
	Segment #2 P#1 Descriptor . . . . Segment #2 P#n Descriptor	All segment descriptors are 10 bytes.
	Segment #3 P#1 Descriptor	
	. . .	
	Seg. #m P#n Descriptor (10 bytes)	
-20	Address pf REMOVE1 (4 bytes)	
-16	Address of Buffer (4 bytes)	
-12	Address of Code File (4 bytes)	
-8	Active Segment List (4 bytes)	
-4	Address of $$TOP (4 bytes)	
$$LOADIT	Object-code necessary to load and execute a segment.	

4.4.4.3 Layout of a Segment Table

A Segment Table consists of eight 32-bit values:

byte -->	0	Address of first descriptor
	4	File Address of Segment
	8	Size of code in bytes
	12	Actual Address in Memory
	16	Scratch Return Address
	20	Segment Reference Count
	24	Active Segment-list link
	28	. . . Reserved . . .

4.5 LOAD MODULE FILE FORMAT

Binary files contain 1-byte markers that describe the record following it. The following are defined for the Computer System binary files created by the ALINK program or the SAVE command:

16(hex)	Transfer Address Marker. The next 4 bytes are the file's transfer address.
03	Data Block Marker. The following 4 bytes are the address of this block, followed by 2 bytes containing the block length.
02	Data Block Marker. The following 4 bytes are the address of this block, followed by 1 byte containing the block length. Note: This marker was used in creating binary files under the 1.0 operating system. It is supported as read only under the 1.1 operating system.

5.0 DEBUG

5.1 INTRODUCTION

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

5.2 INITIAL SETUP

The DEBUG program is shipped with your system in relocatable format under the name DEBUG.OBJ. It is necessary for you to determine a memory location for running DEBUG and then to locate DEBUG at that address as a binary file.

To determine the desired location, start the system and issue SYSMAP, which will display addresses of the system. Pick a location that will be outside the program(s) you want to test and still have room for DEBUG (1C00 hex bytes long). Then execute ALINK to locate it at that location.

You now will have a version of DEBUG called DEBUG.BIN which will operate in an unused part of memory in your machine.

5.3 OPERATING DEBUG

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 (usually the address at which the program was located).

To terminate the debugging (after the program releases any resources it may have acquired), the programmer places a breakpoint where the program terminates itself. When this breakpoint is hit, the E command is used to leave the debugging process. If any channel other than the E 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

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.

A relocation register, say R1, and the PC could be set to the base address of the program module to be tested. 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
```

5.3.1 EXAMPLE OF SETTING UP 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 (ALINK) 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 LINK map, which in this example looks like the following:

MOD1	10E00
MOD2	1124E
MOD3	118EC

Bring up the debugger by typing

```
DEBUG <parameters for program under test>
```

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 LINK 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.

5.4 DEBUG COMMANDS -- SYNTAX AND DEFINITIONS

The syntax notation used with DEBUG is:

```
CM EXP
```

CM is a command that directs the next action of DEBUG. Section 5.5 contains a list of commands.

EXP is a numeric expression which consists of numbers, relocation registers, and qualifiers. An EXP ultimately reduces to a single numeric value, or number.

Numeric expressions (EXP) are of the following form:

VAL	32 bit value
VAL.B	8 bit value
VAL.W	16 bit value
VAL.L	32 bit value

The optional two-character extension to the value (.B,.W,.L) is a length modifier which masks the value as indicated.

VAL is a value of the form:

NUM	number value
R#	relocation register value),
NUM R#	number plus relocation register value
NUM+R#	" " " " "

NUM is a number of the forms:

hex	hex number (default)
$hex	hex number
-hex	negative hex number
-$hex	negative hex number
&dec	decimal number
-&dec	negative decimal number

5.4.1 EXPRESSION EXAMPLES

The following table shows examples of typical expression usage, and the Hex and decimal values of the expression. (NOTE: assume R3 contains 20000 hex.)

EXPRESSION	HEX VALUE	DECIMAL VALUE
8EFC3	8EFC3	585667
$8EFC3	8EFC3	585667
&25R3.W	19	25
$FFE2.L	FFFF001E	-65506
-1.B	FF	255
10+R3	20010	131088

The HEX VALUE is unsigned 32 bit. The DECIMAL VALUE is signed 32 bit.

5.5 SUMMARY OF DEBUG COMMANDS

Command	Description
A	Displays contents of all address registers
A:	Displays and prompts for change of all address registers
A#	Displays contents of an address register
A#: exp	Displays and prompts for change of an address register
A# exp	Sets an address register to hexexp
BR	Displays all breakpoints set
BR exp	Sets a breakpoint at address exp
BR -exp	Removes a breakpoint at address exp
BR CLEAR	Clears all breakpoints set
D	Displays contents of all data registers
D:	Displays and prompts for change on all data registers
D#	Displays contents of a data register
D#:	Displays and prompts for change of a data register
D# exp	Sets a data register to exp
DR	Displays all registers
DM exp	Displays memory at starting address. Returns to system
DM exp exp	Displays memory between two addresses. If second hexexp < first hexexp, then displays second-hexexp bytes starting at first hexexp.
G	Begins execution at address contained in PC
G exp	Begins execution at address exp
OP exp	Changes memory starting at exp
PC	Displays contents of PC
P:	Displays and prompts for change of PC
PC exp	Sets PC to exp
SS	Displays contents of system stack
SS:	Displays and prompts for change in system stack
SS exp	Sets system stack to exp
SR	Displays contents of status register
SR:	Displays and prompts for change in status register
SR exp	Sets status register to exp
R	Displays contents of all relocation registers
R:	Displays and prompts for change of all relocation registers
R#	Displays a relocation register
R#:	Displays and prompts for change of relocation register
R# exp	Sets relocation register to exp
T	Traces single instruction execution
US	Displays contents of user stack
US:	Displays and prompts for change of user stack
US exp	Sets user stack to exp

5.6 REGISTER DISPLAY

Register commands are used to display or change address, data, relocation registers, along with system stack, user stack, status, and program counter registers.

5.6.1 REGISTER DISPLAY EXAMPLES

```
Command                   Description

DR                          Display all registers
PC=00010842 SR=2700       SS=00010406 US=0000CD8C
-------------------------------------------------
D0=00000000 D1=00000001 D2=00000002 D3=00000000
D4=00000000 D5=00000000 D6=00000000 D7=00000000
-------------------------------------------------
A0=00000000 A1=00000000 A2=00000000 A3=00000000
A4=00000000 A5=00000000 A6=00000000 A7=00010406
-------------------------------------------------
R0=00040000 R1=00041B42 R2=00000000 R3=00000000
R4=00000000 R5=00000000 R6=00000000 R7=00000000
-------------------------------------------------


D6                        Display register D6
D6=00000000

D6 6                      Change contents of D6 to 6


D6                        Display D6
D6=00000006


D:                        Prompt for change of all data registers
D0=00000000 ?             Don't change D0
D1=00000001               Don't change D1
D2=00000002               Don't change D200000066
D3=00000003 ?3            Change D3 to 3
D4=00000004               Change D4 to 4
D5=00000005 ?5            Change D5 to 5
D6=00000006 ?             Don't change
D7=00000000 ?7            Change D7 to 7
```

```
D                        Display all data registers
D0=00000000 D1=00000001 D2=00000002 D3=00000003
D4=00000004 D5=00000005 D6=00000006 D7=00000007


SS                       Display system stack
SS=00010406


US                       Display user stack
US=0000CD8C


US:                      Change user stack
US:0000CD8C?E000


US                       Display user stack
US=0000E000


SR                       Display status register
SR=2700


R                        Display relocation registers
R0=00040000 R1=00041B42 R2=00000000 R3=00000000
R4=00000000 R5=00000000 R6=00000000 R7=00000000


PC R1                    Set PC to relocation register


PC                       Display PC
PC=00041B42


A0                       Display address register ad
A0=00000000


A0 1000+R1               Change A0


A0                       Display A0
A0=00042B42

D2                       Display D2
D2=00000002
```

```
D2 -1                       Change D2 to -1 Hex


D2                          Display D2
D2=FFFFFFFF


D2 -1 W                     Change D2 to -1 hex, word length


D2                          Display D2
D2=0000FFFF


D                           Display data registers
D0=00000000 D1=00000001 D2=0000FFFF D3=00000003
D4=00000004 D5=00000005 D6=00000000 D7=00010406


A                           Display address registers
A0=00042B42 A1=00000000 A2=00000000 A3=00000000
A4=00000000 A5=00000000 A6=00000000 A7=00010406
```

5.7 MEMORY DISPLAY

The DM command displays memory bytes in both hexadecimal and ASCII.

Format	Description
DM start	Displays 16 bytes of memory beginning at start address.
DM start end	Displays memory bytes from start address to end address; end must be greater than or equal to start.
DM start count	Displays 'count' bytes beginning at start address; 'count' must be less than start.

NOTE: 'start', 'end', and 'count' are all numeric expressions.

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

Pressing Ctrl-Break on the keyboard will terminate the display.

5.7.1 MEMORY DISPLAY EXAMPLES

Command	Description

```
DM 10000                    Display memory at 10000 Hex
010000  4E F9 00 01 08 30 4A FC 4A FC 4A FC 4A FC 4A FC  N....0J.J.J.J.J.


DM10000 10020               Display 10000 to 10020 Hex
010000  4E F9 00 01 08 30 4A FC 4A FC 4A FC 4A FC 4A FC  N....0J.J.J.J.J.
010010  4A FC 4A FC 4A FC 4A FC 4A FC 4A FC 4A FC 4A FC  J.J.J.J.J.J.J.J.
010020  4A FC 4A FC 4A FC 4A FC 4A FC 4A FC 4A FC 4A FC  J.J.J.J.J.J.J.J.


DM 10000 20                 Display 20 hex bytes starting at 10000 hex
010000  4E F9 00 01 08 30 4A FC 4A FC 4A FC 4A FC 4A FCN....0J.J.J.J.J.
010010  4A FC 4A FC 4A FC 4A FC 4A FC 4A FC 4A FC 4A FC  J.J.J.J.J.J.J.J.
010020  4A FC 4A FC 4A FC 4A FC 4A FC 4A FC 4A FC 4A FC  J.J.J.J.J.J.J.J.
```

5.8 MEMORY CHANGE

The OP command can be used to display or change memory a byte at a time.

Format	Description
OP exp	Opens memory at address specified in exp. Enters the open subcommand mode.

5.8.1 OPEN SUBCOMMAND MODE

In the open subcommand mode, the address contained in exp 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 numeric expression.

Format	Description
exp <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:

exp <CR>	Set the value at the address to exp, then go to next address. <CR> is carriage return.
<CR>	Go to next address
=<CR>	Stay at current address
<CR>	Go to previous address
.<CR>	Without [hexexp], terminate subcommand mode

5.8.2 MEMORY CHANGE EXAMPLES

Command	Description
OP 1000	Begin memory change at 10000 hex
010000 4E ?	Don't change, go to next address
010001 F9 ?	Don't change, go to next address
010002 00 ?	Don't change, go to previous address
010001 F9 ?&64	Change location to 64 decimal go to next
010002 00 ?	Don't change, go to previous address
010001 40 ?	Notice change to 40 hex (64 dec), go to next
010002 00 ?	Go to next
010002 01 ?=	Stay at same address
010003 01 ?=	This is useful for
010003 01 ?=	Polling an I/O to next address
010003 01 ?=	Location
010003 01 ?	Go to next
010002 01 ? FE8A	Change to FE8A Hex
VALUE TOO BIG	Can't enter a word into a byte
010004 08 ? FE8A.B	Specify byte length
010005 30 ?	Backup
010004 8A ?	Verify lower byte of word
010005 30 ?.	Terminate the open subcommand mode

5.9 EXECUTION CONTROL

Format	Description
BR	Displays all breakpoints
BR exp	Sets breakpoint at exp
BR -exp	Removes a breakpoint from exp
BR CLEAR	Clears all breakpoints
E	Returns control to system
G	Begins execution at address in PC
G exp	Begins execution at exp
T	Traces the execution of one instruction.

5.9.1 EXAMPLES OF EXECUTION CONTROL

Command	Description
BR Breakpoint(s)=	Display Breakpoints None is set
R0 10000	Examine relocation register
BR R0	Set breakpoint at R0
BR	
Breakpoint(s): 010000	Display breakpoints
G OR0	Begin program execution at 10000 hex

```
                          Breakpoint hit
PC=00010000 SR=2000       SS=0000CD8C US=0000CD8C
-----------------------------------------------
D0=00000000 D1=00000000 D2=00000000 D3=00000000
D4=00000000 D5=00000000 D6=00000000 D7=00000000
-----------------------------------------------
A0=00000000 A1=00000000 A2=00000000 A3=00000000
A4=00000000 A5=00000000 A6=00000000 A7=0000CD8C
```

```
T                        Trace one instruction
PC=00010830 SR=2000      SS=0000CD8C US=0000CD8C
-----------------------------------------------
D0=00000000 D1=00000000 D2=00000000 D3=00000000
D4=00000000 D5=00000000 D6=00000000 D7=00000000
-----------------------------------------------
A0=00000000 A1=00000000 A2=00000000 A3=00000000
A4=00000000 A5=00000000 A6=00000000 A7=0000CD8C
-----------------------------------------------
BR -10000                Remove breakpoint at 10000 hex

BR                       Display breakpoints
BREAKPOINT(S):           None set

BR 83A                   Set breakpoint at 83A hex

BR 83ARO                 Set breakpoint at 1083A hex (83A+RO)

BR 84A+RO                Set breakpoint at 1084A hex

BR                       Display breakpoints
BREAKPOINT(S):           00083A 01083A 01084A

BR -83A                  Remove breakpoint (83A hex was a mistake)

BR                       Display breakpoints
BREAKPOINT(S):           01083A 01084A

G                        Continue program execution at PC

                         Breakpoint hit
PC=0001083A SR=2000      SS=0000CD86 US=0000CD8C
--------------------------------------------------------
D0=00000000 D1=00000000 D2=00000000 D3=00000000
D4=00000000 D5=00000000 D6=00000000 D7=00000000
--------------------------------------------------------
A0=00000000 A1=00000000 A2=00000000 A3=00000000
A4=00000000 A5=00000000 A6=00000000 A7=0000CD86

T                        Trace one instruction
PC=0001083E SR2000       SS=00010406 US=0000CD8C
--------------------------------------------------------
D0=00000000 D1=00000000 D2=00000000 D3=00000000
D4=00000000 D5=00000000 D6=00000000 D7=00000000
--------------------------------------------------------
A0=00000000 A1=00000000 A2=00000000 A3=00000000
A4=00000000 A5=00000000 A6=00000000 A7=00010406
```

```
G                        Continue program execution at PC
PC=0001084A SR=2708      SS=00010406 US=0000CD8C
--------------------------------------------------------
D0=00000000 D1=00000000 D2=00000000 D3=00000000
D4=00000000 D5=00000000 D6=00000000 D7=00000000
--------------------------------------------------------
A0=00000000 A1=00000000 A2=00000000 A3=00000000
A4=00000000 A5=00000000 A6=00000000 A7=00010406

BR                       Display breakpoints
BREAKPOINT(S):           01083A 01084A

BR CL                    Clear all breakpoints

BR                       Display breakpoints
BREAKPOINT(S):

E                        Exit debug
```

5.9.2 HARD COPY

The following commands echo CS-Debug commands to the line printer, thus providing a convenient record of a debugging session.

Format	Description
LPT ON	Issues a form feed and begins to send a copy of debug session to line printer.
LPT OFF	Stops sending to line printer.

A.0 APPENDIX: ERROR MESSAGES AND CODES

A.1 ERROR MESSAGES FROM OPERATING SYSTEM COMMANDS

The following messages are those that occur most frequently. For a complete list of errors related to a command, see the command description in Chapter 1.

General Messages

COMMAND FORMAT ERROR	Data entered cannot be processed as a command.
UPPER CASE REQUIRED	Command must be entered in uppercase.
CHECK SPELLING OF COMMAND OR PROGRAM NAME	Command not found.

Messages used by many commands.

SYNTAX ERROR	The command line does not conform to the syntax specified for the command.
BAD PARAMETER	A bad argument in the command line was encountered.

The following messages may occur during program loading --

LOAD ADDRESS TOO LOW	Starting address must be above APPBEGIN.
NOT ENOUGH MEMORY FOR LOAD	Ending address must be below.
INVALID LOAD FILE FORMAT	File type must be binary, type 00 or 01.

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

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 = 02, 03, 06, or 07).

The next error message is from the SET command

ILLEGAL VALUE	Value entered is unreasonable for the type of command

The next 3 messages come from the RUNTASK command:

NO TRANSFER ADDRESS	The binary load module specified in the RUNTASK 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 TASK NAME	There is already a PCB with the name specified in the RUNTASK command.

The next 3 messages are related to commands involving tasks (i.e., PRIority, DELAY WAKEUP, RUNTASK, KILL, DELAY, RESUME, SHOW)

NO SUCH TASK	There is no PCB with a task name that matches the one entered.
INVALID PRIORITY	A priority outside the range 1-27 was used in a RUNTASK or PRIORITY command.
SYSTEM TASK CANNOT BE KILLED	You cannot issue the KILL command for the system task.

The next two messages come from the SAVE command:

INVALID TRANSFER ADDRESS	Transfer address must be between the start address and end address and be on an even boundary.
ENDING ADDRESS IS TOO LOW OR TOO HIGH	Ending address must be between starting address and end of system memory.

The next two messages come from the SECURE command:

DIR.DIR FILE ACCESS CANNOT BE CHANGED	DIR.DIR is the directory file on the disk. Its access code is not to be changed.
VALUE RANGE ERROR	Access code specified is not acceptable.

The next two messagers come from the SUBMIT command:

SUBMIT FILE ERROR	An error occurred while trying to open the requested SUBMIT file or a command line exceeded 80 characters.
WRONG FILE TYPE	A submit file must be text or type 3.

File-dependent messages

WRITE TO OLD-STRUCTURED FILE
FILE ALREADY OPEN FOR WRITE
FILE NOT FOUND
ILLEGAL FILE ACCESS METHOD
INVALID FILENAME
READ BEYOND END OF FILE
BUFFER SIZE INCORRECT
FILE NOT EXTENDABLE
DISK FULL
WRITING TO READY-ONLY FILE
WRONG FILE TYPE FOR ACCESS
INVALID SECTOR SPECIFIED
FILE PROTECTED
NEW FILE BEING READ
BAD FILE -- SYSTEM ERROR
NO SPACE FOR CONTIGUOUS FILE

A.2 COMMON DEVICE AND MANAGER ERROR CODES

A.2.1 MESSAGE FORMAT

The Syntax for messages is:

```
component [ERROR=$NNNN][message text][TASK=taskname]
```

COMPONENT is the name of a device or operating system manager issuing the message.

ERROR=$NNNN is a four digit hexadecimal error code which identifies the error. Driver programs issue error codes in the range $0001 through $00FE. Manager programs issue the remaining error codes. It is possible for a device driver to issue a manager error return code. You can find the reason for this code by looking through the manager error codes section.

This field of the message display is always printed if no message text is available. If message text is available then this field is printed only if you have SET EC=YES.

MESSAGE TEXT is supplied by device drivers and system managers for many of the error codes listed in this section. Message text is printed with the message, if it is available.

TASK=TASKNAME is printed with the error message for all tasks other than the SYSTEM task.

A.2.2 COMMON DEVICE-DRIVER ERROR CODES

$0001	reserved
$0002	reserved
$0003	reserved
$0004	NO CONTROL BLOCK STORAGE
$0005	READ/WRITE ERROR
$0006	INVALID DATA TRANSFER DIRECTION IN DIB
$0007	ILLEGAL BUFFER ADDRESS IN DTCB
$0008	END OF FILE
$0009	RECORD LARGER THAN BUFFER; TRUNCATED - input record too big
$000A	DEVICE NOT READY
$000B	NON-ZERO BYTE I/O WRITE STATUS - buffer full
$000C	NON-ZERO BYTE I/O READ STATUS - buffer empty
$000D	SYSIO REQUEST CANCELLED
$000E	INVALID FUNCTION PACKET CODE AT OFFSET $NNNN
$000F	INVALID TRANSFER MODE IN DIB

$0021 INVALID VALUE FOR SETTRANS FUNCTION PACKET
$0022-$005F have a common meaning for function packet data errors in all drivers. The error code is equal to the function code number plus $0020. The message produced by the system is as follows:

```
ERROR=$NNNN INVALID DATA IN FUNCTION $NNNN AT OFFSET
                          $NNNN.
```

$0060	CANNOT READ - DIBDTD SPECIFIED OUTPUT
$0061	CANCEL FAILED
$0062	READ NOT SUPPORTED - by device driver
$0063	WRITE NOT SUPPORTED - by device driver
$0064	FUNCTION NOT SUPPORTED - by device driver
$0065	BWRITE NOT SUPPORTED - by device driver
$0066	BREAD NOT SUPPORTED - by device driver
$0067	CANNOT WRITE - DIBDTD SPECIFIED INPUT
$0068	TSTBYTE NOT SUPPORTED - by device driver
$0069	DTACHDVR NOT SUPPORTED - by device driver
$006A	ATACHDEV NOT SUPPORTED - by device driver
$006B	DTACHDEV NOT SUPPORTED - by device driver
$006C	reserved
$006D	reserved
$006E	reserved
$006F	reserved

A.2.3 MANAGER ERROR CODES

RTMMGR ERROR CODE

$2000 RTMMGR ERROR

SEMAPHORE ERROR CODES

$3001 DUPLICATE SEMAPHORE NAME
$3002 ALL SEMAPHORES IN USE
$3003 ILLEGAL MAXIMUM COUNT
$3004 COUNT EXCEEDS MAXIMUM COUNT - will be truncated
$3005 ILLEGAL QUEUING MODE
$3006 ILLEGAL SEMAPHORE NAME OR SYSTEM I.D.
$3007 NOT OWNER TASK - cannot detach semaphore
$3008 REQUESTED COUNT EXCEEDS MAXIMUM COUNT
$3009 INSUFFICIENT SYSTEM SPACE TO QUEUE REQUEST
$300A SEMAPHORE HAS BEEN DETACHED - request terminated
$300B INVALID COUNT
$300C SEMAPHORE NOT FOUND
$300D INVALID TIME OUT VALUE
$300E REQUEST TIMED OUT
$300F FUNCTION NOT SUPPORTED
$3010 INITIAL COUNT EXCEEDS MAXIMUM COUNT - will be truncated
$3010 REQUEST CANCELLED

MEMORY MANAGEMENT ERROR CODES

$6001 INVALID NAME IN MPD
$6002 DUPLICATE MEMORY POOL NAME
$6003 NO AVAILABLE FPD BLOCKS - internal storage
$6004 ILLEGAL NAME - passed as function argument
$6005 MEMORY POOL NOT FOUND
$6006 POOL STILL CONTAINS MEMORY - descriptor cannot be removed
$6007 ILLEGAL SUB POOL SIZE - must be greater than zero
$6008 ILLEGAL SUB POOL BOUNDARY - memory must be aligned on page boundary
$6009 SUB POOL OVERLAP - memory overlaps existing sub pool
$600A SUB POOL OVERLAPS SYSTEM AREA
$600B NO AVAILABLE SPD BLOCKS - internal storage
$600C SUB POOL MEMORY NOT FOUND
$600D ILLEGAL DMA MEMORY REQUEST - greater than 64K
$600E reserved
$600F FUNCTION NOT SUPPORTED
$6010 MEMORY NOT AVAILABLE - see Appendix F "System Memory Consumption"
$6011 reserved
$6012 MEMORY CANNOT BE RETURNED

I/O MANAGER ERROR CODES

\$8100	DUPLICATE VOLUME IDENTIFIER
\$8200	INVALID SYSIO CALL
\$8300	LOGICAL UNIT NOT OPENED.
\$8400	NO CONTROL BLOCK STORAGE - inadequate system space, see Appendix F " System Memory Consumption"
\$8500	DUPLICATE LOGICAL UNIT
\$8600	DEVICE NOT FOUND
\$8700	NOT DEVICE OWNER
\$8800	NON SHARABLE DEVICE ALREADY OPEN
\$8900	BYTE I/O NOT SUPPORTED
\$8A00	PROCESSING I/O REQUEST ALREADY
\$8B00	NOT OPENED FOR BYTE I/O
\$8C00	INVALID DIB FIELD (TRN, DTD, or RSO)
\$8D00	EVENT NOT FOUND
\$8E00	EVENT NOT OPENED
\$8F00	ILLEGAL READ BUFFER ADDRESS
\$9000	CONTROL BLOCK NOT WORD-ALIGNED
\$9100	CONTROL BLOCK OR BUFFER OUT OF RANGE

A.2.4 CODES FOR ASYCHRONOUS REQUESTS

SYSIO will return a -1 in register D7.W if the asynchronous operation has started successfully. Completion status is returned in the DTCSTA field of the Data Transfer Control Block. The convention is the same as for register D7 status. A -1 indicates that the operation is not yet complete; a zero indicates complete with no error; and a positive number indicates completion with an error.

A.3 DRIVER ERROR CODES

A.3.1 CRT GRAPHICS DRIVER (#GR) ERROR CODES

$0021-$0041	Data out of limits for a function packet Function number = error number - $0020

A.3.2 CRT DISPLAY DRIVER (#SCRN, #CNSL) ERROR CODES

$000E	INVALID FUNCTION PACKET CODE AT OFFSET $NNNN
$0015	MAXIMUM NUMBER OF WINDOWS OPENED
$0016	ADDRESS BOUNDARY ERROR - OPEN FAIL
$0021 through $004B	DATA OUT OF LIMITS FOR A FUNCTION PACKET (Function number = error number - $0020)
$0069	DTACHDRV NOT SUPPORTED
$006A	ATTACHDEV NOT SUPPORTED
$006B	DTACHDEV NOT SUPPORTED

A.3.3 KEYBOARD (#CON) ERROR CODES

$0009	RECORD LARGER THAN BUFFER; TRUNCATED
$000C	NON-ZERO BYTE I/O READ STATUS
$000D	REQUEST CANCELLED
$000E	INVALID FUNCTION PACKET CODE AT OFFSET $NNNN
$0010	KEYBOARD FUNCTION KEY EXCEPTION
$0021	BAD DATA IN SET TRANSFER MODE FUNCTION PACKET
$002B	INVALID TAB AMOUNT
$002D	COMMAND PARSING NOT ENABLED
$0063	WRITE NOT SUPPORTED
$0065	BWRITE NOT SUPPORTED
$006A	ATCHDEV NOT SUPPORTED
$006B	DTACHDEV NOT SUPPORTED

A.3.4 KEYPAD (#KPD) ERROR CODES

$0009	RECORD LARGER THAN BUFFER; TRUNCATED
$000C	NON-ZERO BYTE I/O READ STATUS
$000D	REQUEST CANCELLED
$000E	INVALID FUNCTION PACKET CODE AT OFFSET $NNNN
$0011	(FPKT 12, 19) SCANCODE ALREADY IN TABLE
$0012	(FPKT 12, 19) NOT ENOUGH SPACE IN TABLE
$0013	(FPKT 13) ERROR IN TABLE STRUCTURE
$0014	(FPKT 13) SCANCODE NOT FOUND IN TABLE
$0015	(FPKT 12, 19, 13) INVALID TABLE NUMBER
$0016	(FPKT 12) ILLOGICAL SCANCODE FOR TABLE
$0017	(FPKT 12) STRING LENGTH <GT> 20
$0018	(FPKT 12, 19) BAD TERMINATOR, NOT $OD OR $04
$0019	(FPKT 12, 19) INVALID BUFFER CODE
$001E	(FPKT 17) TABLES NOT EMPTY, CAN'T REALLOCATE
$001F	(FPKT 17) NOT ENOUGH MEMORY FOR TABLES
$0021	(FPKT 01) DIBTRN MODE NOT 0 OR 1
$0024	(FPKT 03) BAD TIME VALUE (MINUS OR >256)
$0026	(FPKT 06) INVALID LED NUMBER
$0027	(FPKT 07) INVALID LED NUMBER
$002E	(FPKT 14) BAD TABLE #, MUST BE 1-4
$002F	(FPKT 15) INVALID BUFFER CODE
$0031	(FPKT 17) REQUESTED TABLE SIZE NEGATIVE
$0032	(FPKT 19) STRING TOO LONG OR BAD TERMINATOR
$0035	(FPKT 21) INVALID ENTER/SHIFT KEY
$0063	WRITE NOT SUPPORTED
$0065	BWRITE NOT SUPPORTED
$006A	ATCHDEV NOT SUPPORTED
$006B	DTACHDEV NOT SUPPORTED

A.3.5 PRINTER DRIVER (#PR) ERROR CODES

$0006	INVALID DATA TRANSFER DIRECTION IN DIB
$0009	RECORD LARGER THAN BUFFER; TRUNCATED
$000A	DEVICE NOT READY
$000B	NON-ZERO BYTE I/O - WRITE STATUS
$000D	REQUEST CANCELLED
$000E	INVALID FUNCTION PACKET CODE AT OFFSET $NNNN
$000F	INVALID TRANSFER MODE IN DIB
$0010	PRINTER DRIVER COLDSTART FAILED
$0012	PARAMETER ERROR IN DTCB
$0014	ONLY FIXED LENGTH TRANSFER ALLOWED IN GRAPHICS
$0019	DETACH DRIVER FAILED
$0021-$0045	DATA OUT OF LIMITS FOR A FUNCTION PACKET FUNCTION NUMBER = ERROR NUMBER -$0020.
$0061	CANCEL FAILED

A.3.6 RS-232 (#SER) STATUS CODES

$0020	DTCB BUFFER FULL BEFORE READ
$0021	FCN CODE _____RETURNED ERROR
$0022	INTMGR ERROR DURING DMA OPERATION
$0030	DATA SUSPECT: PARITY ERROR DETECTED
$0031	DATA SUSPECT: FRAMING ERROR DETECTED
$0032	DATA LOST: CIRCULAR BUFFER OVERRUN
$0033	DATA LOST: HARDWARE OVERRUN
$0034	BREAK RECEIVED

A.3.7 IEEE-488 (#BUS) STATUS CODES

$0020	CONTROLLER NOT ATTACHED
$0021	CONTROLLER ALREADY ATTACHED
$0022	DUPLICATE OR INVALID BUS ADDRESS
$0023	ATTACH WOULD EXCEED MAXIMUM DEVICE COUNT
$0024	DRIVER CANNOT DETACH: DEVICES ACTIVE
$0028	FUNCTION CODE - RETURNED ERROR
$0030	WRITE NOT STARTED: NOT ADDRESSED AS TALKER
$0031	WRITE ABORTED: TIMEOUT ON DMA DATA WRITE
$0032	WRITE ABORTED: ERROR DURING COMMAND SEQUENCE
$0033	WRITE ABORTED: TIMEOUT ON DATA
$0034	WRITE ABORTED: LOST TALKER STATE
$0035	WRITE NOT STARTED: NULL RECORD
$0040	READ NOT STARTED: NOT ADDRESSED AS LISTENER
$0041	READ ABORTED: TIMEOUT ON DMA DATA READ
$0042	READ ABORTED: ERROR DURING COMMAND SEQUENCE
$0043	READ ABORTED: TIMEOUT ON DATA
$0044	READ ABORTED: LOST LISTENER STATE

A.3.8 ITC (#ITC) STATUS CODES

$0006	INVALID DATA TRANSFER DIRECTION IN DIB
$0009	RECORD LARGER THAN BUFFER; TRUNCATED
$000D	REQUEST CANCELLED
$000E	INVALID FUNCTION PACKET CODE AT OFFSET $NNNN
$000F	INVALID TRANSFER MODE IN DIB
$0010	DUPLICATE ITC IDENTIFIER
$0011	INVALID ITC IDENTIFIER
$0012	INVALID ATTACH/DETACH CODE
$0015	LOWER TRIGGER BYTE EXCEEDS UPPER TRIGGER BYTE

$0016	BUFFER OFFSET EXCEEDS BUFFER LENGTH
$0017	EITHER BUFFER LENGTH OR OFFSET ILLEGAL
$0018	NO AVAILABLE SYSTEM MEMORY
$0019	RETURN OF SYSTEM MEMORY FAILED
$0021	INVALID TRANSFER MODE
$0023	INVALID TIME OUT PARAMETER
$0062	READ NOT SUPPORTED
$0063	WRITE NOT SUPPORTED
$0065	BWRITE NOT SUPPORTED
$0066	BREAD NOT SUPPORTED
$0068	TSTBYTE NOT SUPPORTED
$006A	ATACHDEV NOT SUPPORTED
$006B	DTACHDEV NOT SUPPORTED
$0070	WRITE WAIT FAILED
$0071	READ WAIT FAILED
$0072	READ SIGNAL FAILED
$0073	WRITE SIGNAL FAILED
$0075	REQUEST TIMED OUT
$0076	CHANNEL(S) IN USE DRIVER WILL NOT BE DETACHED
$0077	ERROR IN COLDSTART SEQUENCE
$0078	ERROR IN DETACH SEQUENCE - DETACH INCOMPLETE
$0079	DRIVER NOT ATTACHED
$007A	DRIVER IS ATTACHED

A.3.9 PARALLEL PORT (#PPU) DRIVER ERROR CODES

$0005	READ ERROR - trying to read in printer output mode
$000A	DEVICE NOT READY
$000B	NON-ZERO BYTE I/O WRITE STATUS - buffer full
$000C	NON-ZERO BYTE I/O READ STATUS - buffer empty
$000D	REQUEST CANCELLED
$000E	INVALID FUNCTION PACKET CODE AT OFFSET $NNNN
$000F	INVALID TRANSFER MODE IN DIB
$0021	INVALID VALUE FOR SET TRANSFER MODE DATA
$0023	SET PARALLEL PORT MODE DATA INCORRECT
$0025	SET TIMEOUT DATA INCORRECT
$0027	SET AUTO LINEFEED DATA INCORRECT
$0068	BTEST IS NOT SUPPORTED BY THIS DRIVER
$006A	ATCHDEV NOT SUPPORTED
$006B	DTACHDEV NOT SUPPORTED

A.3.10 DISK (#FD0X OR #HD0X) ERROR CODES

$0005	READ/WRITE ERROR
$0006	INVALID DATA TRANSFER DIRECTION IN DIB

$0007	ILLEGAL BUFFER ADDRESS IN DTCB
$000A	DEVICE NOT READY
$000E	INVALID FUNCTION PACKET CODE AT OFFSET $NNNN
$0010	SEEK TRACK
$0011	LOGICAL SECTOR OR TRACK NUMBER TOO BIG $********
$0012	VOLUME CHANGED
$0013	reserved
$0014	I/O REQUEST TIMED OUT
$0015	reserved
$0016	ILLEGAL BUFFER ADDRESS
$0017	DISK FORMAT NOT RECOGNIZED
$0018	DISK WRITE PROTECTED
$0019	SECTOR BUFFER TOO SMALL
$001A	WRITE FAULT $********
$001B	CRC ERROR $********
$001C	SECTOR NOT FOUND $********
$002F	ILLEGAL VOLUME IDENTIFIER
$0061	CANCEL FAILED
$0065	BWRITE NOT SUPPORTED
$0066	BREAD NOT SUPPORTED
$0067	CANNOT WRITE - DIBDTD SPECIFIED INPUT
$0068	TSTBYT NOT SUPPORTED
$006A	ATCHDEV NOT SUPPORTED
$006B	DTACHDEV NOT SUPPORTED
$0070	DATA ADDRESS MARK NOT FOUND $********
$0071	reserved
$0072	ABORTED COMMAND
$0073	reserved
$0074	reserved
$0075	reserved
$0076	UNCORRECTABLE DISK ERROR $********
$0077	BAD SECTOR DETECTED $********
$0078	HARD DISK CONTROLLER NOT PRESENT

A.3.11 SENSOR I/O ERROR CODES

$0010	DEVICE LOCKED
$0011	DEVICE ALREADY OPEN
$0012	A/D OVERANGE
$0013	CTC TIMER OVERRUN
$0014	ILLEGAL OPEN MODE
$0015	A/D TIME-OVERRUN

A.4 ABNORMAL-TERMINATION SCREEN

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
	DIVO	divide by zero trap
	CHKC	check instruction trap
	TRPV	Trap V instruction
	PRIV	privilege violation
	1010	illegal instruction
	1111	illegal instruction
	TR13	unexpected trap 13
	TR14	unexpected trap 14
	?INT	unexpected miscellaneous trap
	ABRT	Abort button interrupt

TYPE 2 Extended information TRAPS

ADDR	illegal address trap; word operand on odd address
SPUR	spurious interrupt trap
BUS	bus error trap
ABUS	address bus error
DBUS	data bus error
PROT	memory protection error; attempt to store into system memory
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.5 ASSEMBLY ERROR CODES

A.5.1 ERROR MESSAGES

Error messages generated during an assembly may originate from the assembler or from Pascal or the operating system environment. Assembler-generated messages may be of two forms:

1. ``******ERROR xxx -- nnnn``

   where xxx is the number of the error (defined in the list in this appendix), and nnnn is the number of the line where the previous error occurred.

   Errors indicate that the assembler is unable to interpret or implement the intent of a source line.

2. ``******WARNING xxx -- nnnn``

   where xxx is the number of the error (defined in the list in this appendix), and nnnn is the number of the line where the previous error occurred.

   Warnings may indicate possible recoverable errors in the source code, or that a more optimal instruction format is possible.

ERROR CODE	MEANING OF ERROR
	SYNTACTIC ERRORS
200	ILLEGAL CHARACTER (IN CONTEXT)
201	SIZE CODE/EXTENSION IS INVALID
202	SYNTAX ERROR
203	SIZE CODE/EXTENSION NOT ALLOWED
204	LABEL REQUIRED
205	END DIRECTIVE MISSING
206	REGISTER RANGES FOR THE MOVEM INSTRUCTION MUST BE SPECIFIED IN INCREASING ORDER
207	A AND D REGISTERS CAN'T BE INTERMIXED IN A MOVEM REGISTER RANGI
	OPERAND/ADDRESS MODE ERRORS
210	MISSING OPERAND(S)
211	TOO MANY OPERANDS FOR THIS INSTRUCTION
212	IMPROPER TERMINATION OF OPERAND FIELD
213	ILLEGAL ADDRESS MODE FOR THIS OPERAND
214	ILLEGAL FORWARD REFERENCE
215	SYMBOL/EXPRESSION MUST BE ABSOLUTE
216	IMMEDIATE SOURCE OPERAND REQUIRED
217	ILLEGAL REGISTER FOR THIS INSTRUCTION
218	ILLEGAL OPERATION ON A RELATIVE SYMBOL
219	MEMORY SHIFTS MAY ONLY BE SINGLE BIT
220	INVALID SHIFT COUNT
221	INVALID SECTION NUMBER
	SYMBOL DEFINITION
230	ATTEMPT TO REDEFINE A RESERVED SYMBOL
231	ATTEMPT TO REDEFINE A MACRO; NEW DEFINITION IGNORED
232	ATTEMPT TO REDEFINE THE COMMAND LINE LOCATION
233	COMMAND LINE LENGTH MUST BE > 0; IGNORED
234	REDEFINED SYMBOL
235	UNDEFINED SYMBOL
236	PHASING ERROR ON PASS2
237	START ADDRESS MUST BE IN THIS MODULE, IF SPECIFIED
238	UNDEFINED OPERATION (OPCODE)
239	NAMED COMMON SYMBOL MAY NOT BE XDEF
	DATA SIZE RESTRICTIONS
250	DISPLACEMENT SIZE ERROR
251	VALUE TOO LARGE
252	ADDRESS TOO LARGE FOR FORCED ABSOLUTE SHORT
253	BYTE MODE NOT ALLOWED FOR THIS OPCODE

254	MULTIPLICATION OVERFLOW
255	DIVISION BY ZERO

MACRO ERRORS

260	MISPLACED MACRO, MEXIT, OR ENDM DIRECTIVE
261	MACRO DEFINITIONS MAY NOT BE NESTED
262	ILLEGAL PARAMETER DESIGNATION
263	A PERIOD MAY OCCUR ONLY AS THE FIRST CHARACTER IN A MACRO NAME
264	MISSING PARAMETER REFERENCE
265	TOO MANY PARAMETERS IN THIS MACRO CALL
266	REFERENCE PRECEDES MACRO DEFINITION
267	OVERFLOW OF INPUT BUFFER DURING MACRO TEXT EXPANSION

CONDITIONAL ASSEMBLY ERRORS

270	UNEXPECTED 'ENDC'
271	BAD ENDING TO CONDITIONAL ASSEMBLY STRUCTURE (ENDC EXPECTED)

STRUCTURED SYNTAX ERRORS

280	MISPLACED STRUCTURED CONTROL DIRECTIVE (IGNORED)
281	MISSING "ENDI"
282	MISSING "ENDF"
283	MISSING "ENDW"
284	MISSING "UNTIL"
285	UNRESOLVED SYNTAX ERROR IN THE PRECEDING PARAMETERIZED STRUCTURED CONTROL DIRECTIVE; RECOVERY ATTEMPTED WITH THE CURRENT LINE
286	"=" EXPECTED; CHARACTERS UP TO "=" IGNORED
287	"<" EXPECTED; CHARACTERS UP TO "<" IGNORED
288	">" EXPECTED; CHARACTERS UP TO ">" IGNORED
289	"DO" EXPECTED: REMAINDER OF LINE IGNORED
290	"THEN" EXPECTED; REMAINDER OF LINE IGNORED
291	"TO" OR "DOWNTO" EXPECTED; "TO" ASSUMED
292	ILLEGAL CONDITION CODE SPECIFIED

MISCELLANEOUS

300	IMPLEMENTATION RESTRICTION
301	TOO MANY RELOCATABLE SYMBOLS REFERENCED <LINKAGE EDITOR RESTRICTED>
302	RELOCATION OF BYTE FIELD ATTEMPTED
303	ABSOLUTE SECTION OF LENGTH ZERO DEFINED (LINK ERROR)
304	NESTED "INCLUDE" FILES NOT ALLOWED; IGNORED
305	FILE NAME REQUIRED IN OPERAND FIELD

INTERNAL ERRORS

400
•
•
•
499

SOURCE CODE NOT OPTIMAL OR RECOVERABLE ERRORS

500	THIS BYTE WILL BE SIGN-EXTENDED TO 32 BITS
501	MISSING PARAMETER REFERENCE IN MACTRO SOURCE
502	TOO MANY PARAMETERS IN THIS MACRO CALL
550	THIS BRANCH COULD BE SHORT
551	THIS ABSOLUTE ADDRESS COULD BE SHORT

NOTE: If more than 10 errors occur in one line, the message

```
***** TOO MANY ERRORS ON THIS LINE
```

will be generated.

B.0 APPENDIX B: INSTRUCTION SET SUMMARY

This appendix provides a summary of the 68000 instruction set. For detailed information, refer to the 68000 16-Bit Microprocessor User's Manual.

INSTRUCTION SET SUMMARY

MNEMONIC	OPERATION	ASSEMBLER SYNTAX	X	N	Z	V	C
ABCD	Add Decimal With Extend	ABCD Dy,Dx ABCD -(Ay),-(Ax)	*	U	*	U	*
ADD	Add Binary (See NOTE 1.)	ADD <ea>,Dn ADD Dn,<ea>	*	*	*	*	*
ADDA	Add Address	ADDA <ea>,Dn	-	-	-	-	-
ADDI	Add Immediate	DDI #<data>,<ea>	*	*	*	*	*
ADDQ	Add Quick	ADDQ #<data>,<ea>	*	*	*	*	*
ADDX	Add Extended	ADDX Dy,Dx ADDX -(Ay),-(Ax)	*	*	*	*	*
AND	AND Logical	AND <ea>,Dn AND Dn,<ea>	-	*	*	0	0
ANDI	AND Immediate	ANDI #<data>,<ea>	-	*	*	0	0
ASL, ASR	Arithmetic Shift	ASd Dx,Dy ASd #<data>,Dy ASd <ea>	*	*	*	*	*
Bcc	Branch Conditionally	Bcc <label>	-	-	-	-	-
BCHG	Test a Bit and Change	BCHG Dn,<ea> BCHG #<data>,<ea>	-	-	*	-	-
BCLR	Test a Bit and Clear	BCLR Dn,<ea> BCLR #<data>,<ea>	-	-	*	-	-
BRA	Branch Always	BRA <label>	-	-	-	-	-

INSTRUCTION SET SUMMARY (continued)

BSET	Test a Bit and Set	BSET Dn,<ea> BSET #<data>,<ea>	-	-	*	-	-
BSR	Branch to Subroutine	BSR <label>	-	-	-	-	-
BTST	Test a Bit	BTST Dn,<ea> BTST #<data>,<ea>	-	-	*	-	-
CHK	Check Register Against Bounds	CHK <ea>,Dn	-	*	U	U	U
CLR	Clear an Operand	CLR <ea>	-	0	1	0	0
CMP	Arithmetic Compare	CMP <ea>,Dn	-	*	*	*	*
CMPA	Arithmetic Compare Address	CMP <ea>,An	-	*	*	*	*
CMPI	Compare Immediate	CMPI #<data>,<ea>	-	*	*	*	*
CMPM	Compare Memory	CMPM (Ay)+,(Ax)+	-	*	*	*	*
DBcc	Test Condition and Decrement and Branch (See NOTE 2.)	DBcc Dn,<label>	-	-	-	-	-
DIVS	Signed Divide	DIVS <ea>,Dn	-	*	*	*	0
DIVU	Unsigned Divide	DIVU <ea>,Dn	-	*	*	*	0
EOR	Exclusive OR Logical	EOR Dn,<ea>	-	*	*	0	0
EORI	Exclusive OR Immediate	EORI #<data>,<ea>	-	*	*	0	0
EXG	Exchange Registers	EXG Rx,Ry	-	-	-	-	-
EXT	Sign Extend	EXT Dn	-	*	*	0	0
JMP	Jump	JMP <ea>	-	-	-	-	-
JSR	Jump to Subroutine	JSR <ea>	-	-	-	-	-

INSTRUCTION SET SUMMARY (continued)

LEA	Load Effective Address	LEA <ea>,An	-	-	-	-	-
LINK	Link and Allocate	LINK An,#<displacement>	-	-	-	-	-
LLSR,LSR	Logical Shift	LSd Dx,Dy LSd #<data>,Dy LSd <ea>	*	*	*	0	*
MOVE	Move Data from Source to Destination	MOVE <ea>,<ea>	-	*	*	0	0
MOVE to SR	Move to the Status Register	MOVE <ea>,SR	*	*	*	*	*
MOVE from SR	Move from the STATUS Register	MOVE SR,<ea>	-	-	-	-	-
MOVE to CC	Move to Condition Codes	Move <ea>,CCR	*	*	*	*	*
MOVE from CC	Move from Condition Codes (M68010)	MOVE CCR,<ea>	-	-	-	-	-
MOVE USP	Move User Stack Pointer	MOVE USP,An Move An,USP	-	-	-	-	-
MOVEA	Move Address	MOVEA <ea>,An	-	-	-	-	-
MOVEC	Move to/from Control Register (M68010) (See NOTE 3.)	MOVEC Rc,Rn MOVEC Rn,Rc	-	-	-	-	-
MOVEM	Move Multiple Registers (See NOTE 4.)	MOVEM <register list>,<ea> MOVEM <register list>,<ea>	-	-	-	-	-
MOVEP	Move Peripheral Data	MOVEP Dx,d(Ay) MOVEP d(Ay),Dx	-	-	-	-	-
MOVEQ	Move Quick	MOVEQ #<data>,Dn	-	*	*	0	0
MOVES	Move to/from Address (M68010)	MOVES <ea>,Rn MOVES Rn,<ea>	-	-	-	-	-

INSTRUCTION SET SUMMARY (continued)

MULS	Signed Multiply	MULS <ea>,Dn	- * * 0 0
MULU	Unsigned Multiply	MULU <ea>,Dn	- * * 0 0
NBCD	Negate Decimal with Extend	NBCD <ea>	* U * U *
NEG	Two's Complement Negation	NEG <ea>	* * * * *
NEGX	Negate with Extend	NEGX <ea>	* * * * *
NOP	No Operation	NOP	- - - - -
NOT	Logical Complement	NOT <ea>	- * * 0 0
OR	Inclusive OR Logical	OR <ea>,Dn OR Dn,<ea>	- * * 0 0
ORI	Inclusive OR Immediate	ORI #<data>,<ea>	- * * 0 0
PEA	Push Effective Address	PEA <ea>	- - - - -
RESET	Reset External Devices	RESET	- - - - -
ROL,ROR	Rotate without Extend	ROd Dx,Dy ROd #<data>,Dy ROd <ea>	- * * 0 *
ROXL,ROXR	Rotate with Extend	ROXd Dx,Dy ROXd #<data>,Dy ROXd <ea>	* * * 0 *
RTE	Return from Exception	RTE	* * * * *
RTR	Return and Restore Condition Codes	RTR	* * * * *
RTS	Return from Subroutine	RTS	- - - - -
SBCD	Subtract Decimal with Extend	SBCD Dy,Dx SBCD -(Ay),-(Ax)	* U * U *

INSTRUCTION SET SUMMARY (continued)

Scc	Set According to Condition	Scc <ea>	- - - - -
STOP	Stop Program Execution	STOP #<data>	- - - - -
SUB	Subtract Binary	SUB <ea>,Dn SUB Dn,<ea>	* * * * *
SUBA	Subtract Address	SUBA <ea>,An	- - - - -
SUBI	Subtract Immediate	SUBI #<data>,<ea>	* * * * *
SUBQ	Subtract Quick	SUBQ #<data>,<ea>	* * * * *
SUBX	Subtract with Extend	SUBX Dy,Dx SUBX -(Ay),-(Ax)	* * * * *
SWAP	Swap Register Halves	SWAP Dn	- * * 0 0
TAS	Test and Set an Operand	TAS <ea>	- * * 0 0
TRAP	Trap	TRAP #<vector>	- - - - -
TRAPV	Trap on Overflow	TRAPV	- - - - -
TST	Test an Operand	TST <ea>	- * * 0 0
UNLK	Unlink	UNLK An	- - - - -

NOTES:

1. <ea> specifies effective address.

2. The assembler accepts DBRA for the F (never true) condition.

3. Rc specifies control register.

4. <register list> specifies the registers selected for transfer to or from memory. <register list> may be:

   Rn - a single register;
   Rn-Rm - a range of consecutive registers with m being greater than n.

   Any combination of the above, separated by a slash.

C.0 APPENDIX C: CHARACTER SET

C.1 The character set recognized by the 68000 Resident Structured Assembler is a subset of ASCII (American Standard Code for Information Interchange, 1968). The characters listed below are recognized by the assembler, and the ASCII code is shown on the following pages.

1. The uppercase letters A through Z

2. The integers 0 through 9

3. Four arithmetic operators: + - * /

4. The logical operators: >> << & !

5. Parentheses used in expressions ( )

6. Characters used as special prefixes:
   # (pound sign) specifies the immediate mode of addressing
   $ (dollar sign) specifies a hexadecimal number
   @ (commercial "at") specifies an octal number
   % (percent) specifies a binary number
   ' (apostrophe) specifies an ASCII literal character

7. The special characters used in macros: < > \ @

8. Three separating characters:
   SPACE
   , (comma)
   . (period)

9. A comment in a source statement may include any characters with ASCII hexadecimal values from 20 (SP) through 7E (~).

10. Character used as a special suffix:

    : (colon) specifies the end of a label

ASCII Character Set

CHARACTER	COMMENTS	HEX VALUE
NUL	Null or tape feed	00
SOH	Start of Heading	01
STX	Start of Text	02
ETX	End of Text	03
EOT	End of Transmission	04
ENQ	Enquire (who are you, WRU)	05
ACK	Acknowledge	06
BEL	Bell	07
BS	Backspace	08
HT	Horizontal Tab	09
LF	Line Feed	0A
VT	Vertical Tab	0B
FF	Form Feed	0C
RETURN	Carriage Return	0D
SO	Shift Out (to red ribbon)	0E
SI	Shift In (to black ribbon)	0F
DLE	Data Link Escape	10
DC1	Device Control 1	11
DC2	Device Control 2	12
DC3	Device Control 3	13
DC4	Device Control 4	14

ASCII Character Set (continued)

NAK	Negative Acknowledge	15
SYN	Synchronous idle	16
ETB	End of Transmission Block	17
CAN	Cancel	18
EM	End of Medium	19
SUB	Substitute	1A
ESC	Escape, prefix	1B
FS	File Separator	1C
GS	Group Separator	1D
RS	Record Separator	1E
US	Unit Separator	1F
SP	Space or blank	20
!	Exclamation point	21
"	Quotation marks (dieresis)	22
#	Number sign	23
$	Dollar sign	24
%	Percent sign	25
&	Ampersand	26
'	Apostrophe (acute accent, closing single quote)	27
(	Opening parenthesis	28
)	Closing parenthesis	29

ASCII Character Set (continued)

*	Asterisk	2A
+	Plus sign	2B
,	Comma (cedilla)	2C
-	Hyphen (minus)	2D
.	Period (decimal point)	2E
/	Slant	2F
0	Digit 0	30
1	Digit 1	31
2	Digit 2	32
3	Digit 3	33
4	Digit 4	34
5	Digit 5	35
6	Digit 6	36
7	Digit 7	37
8	Digit 8	38
9	Digit 9	39
:	Colon	3A
;	Semicolon	3B
<	Less than	3C
=	Equals	3D
>	Greater than	3E
?	Question mark	3F

ASCII Character Set (continued)

@	Commercial at	40
A	Uppercase letter A	41
B	Uppercase letter B	42
C	Uppercase letter C	43
D	Uppercase letter D	44
E	Uppercase letter E	45
F	Uppercase letter F	46
G	Uppercase letter G	47
H	Uppercase letter H	48
I	Uppercase letter I	49
J	Uppercase letter J	4A
K	Uppercase letter K	4B
L	Uppercase letter L	4C
M	Uppercase letter M	4D
N	Uppercase letter N	4E
O	Uppercase letter O	4F
P	Uppercase letter P	50
Q	Uppercase letter Q	51
R	Uppercase letter R	52
S	Uppercase letter S	53
T	Uppercase letter T	54
U	Uppercase letter U	55

ASCII Character Set (continued)

V	Uppercase letter V	56
W	Uppercase letter W	57
X	Uppercase letter X	58
Y	Uppercase letter Y	59
Z	Uppercase letter Z	5A
[	Opening bracket	5B
\	Reverse slant	5C
]	Closing bracket	5D
	Circumflex	5E
_	Underline	5F
'	Quotation mark	60
a	Lowercase letter a	61
b	Lowercase letter b	62
c	Lowercase letter c	63
d	Lowercase letter d	64
e	Lowercase letter e	65
f	Lowercase letter f	66
g	Lowercase letter g	67
h	Lowercase letter h	68
i	Lowercase letter i	69
j	Lowercase letter j	6A

ASCII Character Set (continued)

k	Lowercase letter k	6B
l	Lowercase letter l	6C
m	Lowercase letter m	6D
n	Lowercase letter n	6E
o	Lowercase letter o	6F
p	Lowercase letter p	70
q	Lowercase letter q	71
r	Lowercase letter r	72
s	Lowercase letter s	73
t	Lowercase letter t	74
u	Lowercase letter u	75
v	Lowercase letter v	76
w	Lowercase letter w	77
x	Lowercase letter x	78
y	Lowercase letter y	79
z	Lowercase letter z	7A
{	Opening brace	7B
|	Vertical line	7C
}	Closing brace	7D
~	Equivalent	7E
DEL	Delete	7F

D.0 APPENDIX D: SAMPLE ASSEMBLER OUTPUT

```
  1                           *****************************************************************************
  2                           *                                                                           *
  3                           *  SAMPLE PROGRAM.  THIS PROGRAM RECEIVES CONTROL FROM THE SYSTEM TASK AND  *
  4                           *                   MAKES USE OF A SEPARATELY ASSEMBLED SUBROUTINE TO PRINT *
  5                           *                   TWO MESSAGES. ON COMPLETION IT RETURNS CONTROL BACK TO  *
  6                           *                   THE SYSTEM TASK VIA EXIT SYSTEM CALL.  THIS EXAMPLE     *
  7                           *                   EXPLICITLY CODES THE SYSTEM CALL TRAP INSTRUCTIONS.     *
  8                           *                   THIS EXAMPLE ALSO MAKES USE OF THE DEFAULT 400 BYTE USER *
  9                           *                   STACK WHICH IS AVAILABLE ONLY FROM THE SYSTEM TASK.     *
 10                           *                                                                           *
 11                           *****************************************************************************
 12                           EXAMMAIN IDNT     1,1
 13                                    XREF     MSGSUB                DEFINE EXTERNAL SUBROUTINE
 14 0 00000000 303C0001                MOVE.W   #1,D0
 15 0 00000004 4EB900000000            JSR      MSGSUB                CALL SUB TO PRINT MESSAGE 1
 16 0 0000000A 303C0002                MOVE.W   #2,D0
 17 0 0000000E 4EB900000000            JSR      MSGSUB                CALL SUB TO PRINT MESSAGE 2
 18 0 00000014
 19 0 00000014 4E40                    TRAP     #0                    EXIT SYSTEM CALL
 20 0 00000016 002B                    DC.W     43
 21                                    END

****** TOTAL ERRORS      0--   0
****** TOTAL WARNINGS    0--   0


SYMBOL TABLE LISTING


SYMBOL NAME     SECT   VALUE       SYMBOL NAME     SECT   VALUE


MSGSUB     XREF   *   00000000
```

```
 1                        ******************************************************************************
 2                        *                                                                            *
 3                        *  SAMPLE SUBROUTINE  THIS PROGRAM PRINTS ONE OF TWO MESSAGES DEPENDING ON   *
 4                        *                     THE CONTENTS OF D0.W.                                  *
 5                        *                     IT USES THE STRUCTURED PROGRAMMING CONSTRUCTS          *
 6                        *                     TO DECIDE WHICH MESSAGE (IF ANY) TO PRINT.             *
 7                        *                                                                            *
 8                        ******************************************************************************
 9                        MSGPROG  IDNT     1,1
10                                 XDEF     MSGSUB                IDENTIFY MSGSUB AS EXTERNAL LABEL
11                        MSGSUB   IF.W     D0 (EQ) #1 THEN.S
12 0 00000006 4DF900000022           LEA      MSG1,A6
13 0 0000000C 4E40                   TRAP     #0                  PRTMSG SYSTEM CALL
14 0 0000000E 0012                   DC.W     18
15                                 ENDI
16                                 IF.W     D0 (EQ) #2 THEN.S
17 0 00000016 4DF90000002E           LEA      MSG2,A6
18 0 0000001C 4E40                   TRAP     #0                  PRTMSG SYSTEM CALL
19 0 0000001E 0012                   DC.W     18
20                                 ENDI
21 0 00000020 4E75                 RTS                            SUBROUTINE RETURN
22 0 00000022
23 0 00000022 4D4553534147 MSG1    DC.B     'MESSAGE ONE',$0D
24 0 0000002E 4D4553534147 MSG2    DC.B     'MESSAGE TWO',$0D
25                                 END

****** TOTAL ERRORS     0--   0
****** TOTAL WARNINGS   0--   0


SYMBOL TABLE LISTING


SYMBOL NAME     SECT   VALUE       SYMBOL NAME     SECT   VALUE


MSG1              0   00000022    Z_L1.000          0   00000010
MSG2              0   0000002E    Z_L1.001          0   00000020
MSGSUB    XDEF    0   00000000
```

E.0 APPENDIX E: EXAMPLES OF LINKED ASSEMBLY-LANGUAGE PROGRAMS

ALINK SCREEN "DIALOG"

```
Enter source file name  : 1:EXAMMAIN
Enter output file name  :
Enter list file name  :
Enter input file, or option, or RETURN to continue :+L
Enter input file, or option, or RETURN to continue :1:MSGPROG
Enter input file, or option, or RETURN to continue :
Output file : 1:EXAMMAIN.BIN
Listing to 1:EXAMMAIN.MAP
Start address is : $  E000

Input file : 1:EXAMMAIN.OBJ
Input file : 1:MSGPROG.OBJ
$
Code Size = 82 bytes
Last address used = $  E103
```

ALINK OUTPUT MAP

```
Linking segment '        ' (82)
  Initial memavail = 790422
  Final memavail = 790394


Memory map for segment '        '

 : MSGSUB   = $  E018 ;


No:  Segment:    Size:
 0. '        '      52

Load Address is : $  E000
Code Size = 82
Last address used = $  E103
```

F.0 APPENDIX F: SYSTEM MEMORY CONSUMPTION

The System Memory Pool is initialized with 20 pages of storage (1 page = 1024 bytes). This is enough memory for normal activity on a system configured with two floppy drives. As additional resources are added to the system, greater demands are made on the system memory pool and additional memory must be transferred to the system memory pool using the "SET SM" command. In order to judge the amount of memory you should tranfer to or from the pool, consider the following rough guidelines:

SYSTEM RESOURCE	SYSTEM POOL MEMORY CONSUMPTION
SENSOR I/O DRIVE	16 PAGES
GPIB (IEEE-488)DRIVER	3 PAGES
ALPHA WINDOWS	1/8 PAGE/WINDOW
GRAPHICS WINDOWS	1/4 PAGE/WINDOW
DISK DRIVES:	
OPEN VOLUME	3 PAGES/OPEN VOLUME
OPEN FILE	1 PAGE/OPEN FILE
HARD DISK DRIVE	2 PAGES/DRIVE
FLOPPY DISK DRIVE	0

As an example, the following command will tranfer 16 pages of memory to the sytem pool.

```
SET SM=16<CR>
```

If it is desirable to invoke the command each time the system is used, the above command may be placed in the AUTOEXEC submit file.

You may issue the SYSMAP command to display the current status of the memory pool.

GC22-9199-1

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.

GC22-9199-1

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

GC22-9199-1

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