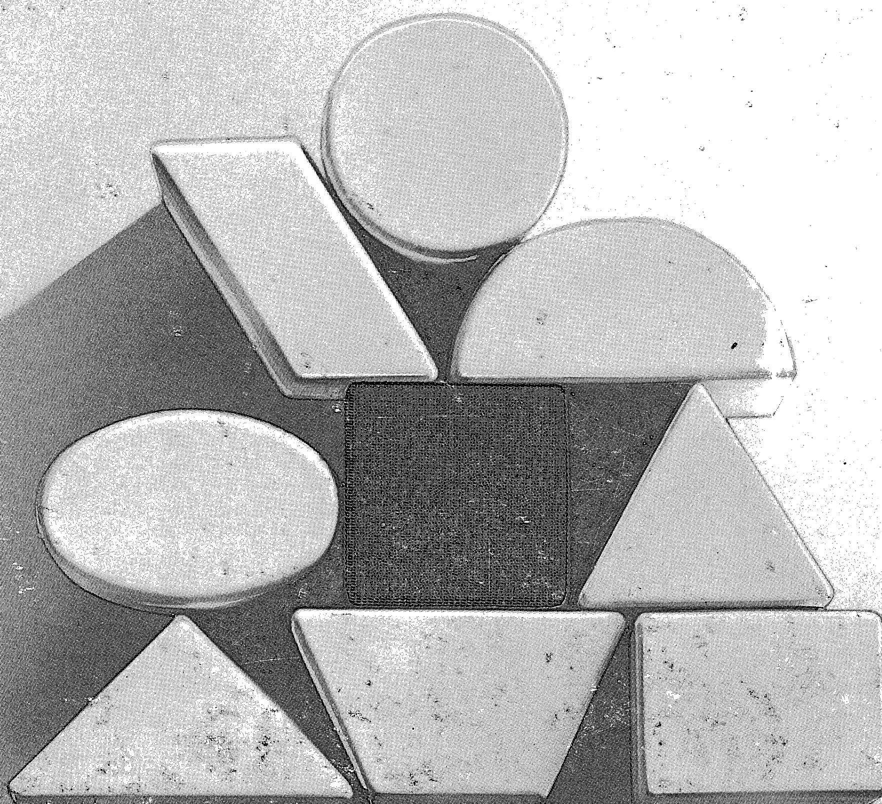
MP/OS Assembly Language Programmer's Reference



Data General

MP/OS Assembly Language Programmers's Reference

Data General
Data General Corporation, Westboro, Massachusetts 01581

NOTICE

FIRST EDITION

(First Printing, July 1979)

Ordering No. 093-400001

Printed in the United States of America
Rev. 00

Preface

Purpose and Organization

This book is intended to serve the needs of experienced assembly language and system programmers who want to know all the details of MP/OS system software.

It is divided into four parts:

Part 1 is the *Assembly Language Programmer's Reference*. This covers the instruction set for MP/Computers. The instructions are arranged in dictionary format for easy reference.

Part 2 is the *Macroassembler Programmer's Reference*. This describes the Macroassembler program, and how you can use it to make your assembly language programs easier to write and more powerful. All the assembly macros are covered in the dictionary at the end of this section.

Part 3 is the *System Reference Manual*. This section describes the MP/OS Operating System in some detail, and tells you how to call system routines from your programs. This section too ends with a dictionary: it describes the system calls in alphabetical order.

These three parts are followed by a series of Appendices, which cover a variety of material you may need to refer to as you write your assembly language programs.

Related Manuals

The list that follows gives a brief description of each of the other manuals which describe Microproducts and MP/OS.

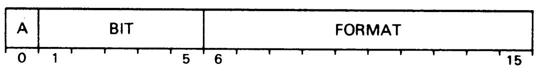
- *An Introduction to Microproducts and MP/OS* (DG No. 069-400000) describes the hardware and software in general terms, to give an overview of your MP/Computer and its capabilities.
- *Microproducts Hardware Systems Reference* (DG No. 014-000636) gives a detailed functional description of the Microproducts line of Microcomputers and related peripherals, board by board.
- *Learning to Use the MP/OS Operating System* (DG No. 093-400000) should be read by anyone who has never used an MP/Computer. It introduces the MP/OS file system and the Command Line Interpreter. A console session gives you step by step hands-on experience with your new MP/Computer.
- *MP/OS Utilities Reference* (DG No. 093-400002) describes the utility programs available with the MP/OS system.
- *MP/Pascal Programmers' Reference* (DG No. 093-400003) describes Data General's extended version of Pascal.
- *MP/Fortran IV Programmers' Reference* (DG No. 093-400004) covers all of the features of this powerful high level language.

Typesetting Conventions

Throughout this manual we use the following conventions to show instruction formats:

COMMAND	We use bold face and uppercase letters to indicate the instruction mnemonics. You code them into your program exactly as they appear.
argument	We use italics and lower case letters to indicate that a particular instruction takes an argument. In your program, you must replace this symbol with the exact code for the argument you need.
[optional]	Brackets denote an optional argument. (Command switches appear in this format as well.) If you decide to use this argument or switch, do not write the brackets into your code: they only set off the choice.
EXAMPLE LINE	We use upper case sans serif for all programming examples.
RESPONSE	If the program can respond to the command in the example, we show the response in uppercase italics.

In addition, we use the following special symbols:



This diagram shows the arrangement of the 16 bits in an instruction. The diagram is always divided into 16 boxes, numbered 0 to 15.

<↓>
represents a New-line character.

CTRL means that you should depress and hold the Control key while you press the character following the CTRL.

Contents

Part 1
Assembly Language Reference

Chapter 1
Methods of Addressing

The various methods of addressing memory locations in MP/Computers give you considerable flexibility when storing and retrieving data or transferring control to a different procedure. This chapter discusses these methods and the conventions governing them. You should familiarize yourself with this material before going on to the rest of the manual.

Addressing Conventions

Each of the 32K words in main memory consists of a 16-bit word. You use a 15-bit address to specify which word will undergo some operation, such as receive or provide data. When you specify the addresses of locations, you will notice that you can address memory in a circular fashion. That is, when you want to address the word succeeding location 77777_8, you address location 0. When you want to address the word preceding location 0 you address location 77777_8. Figure 1.1 shows you a typical format of an instruction used to address memory.

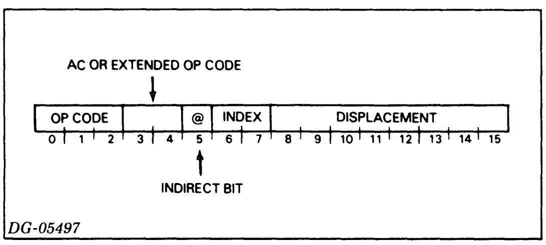


Figure 1.1 Word Addressing Format

Definitions

The following definitions will help you understand the concepts of word addressing.

Addressing modes - three methods of addressing that use a displacement from some reference point to find the desired address. Different modes use different reference points.

Indirect addressing - a method of addressing that uses the first address found as a pointer to another address which, in turn, may be used as a pointer to yet another address, etc. A series of indirect addresses is called an *indirection chain.*

Index bits - bits in the instruction that specify the addressing mode used when executing this instruction.

Indirect bit - a bit in the instruction or address that controls the indirection chain at each step of the addressing process.

Displacement bits - bits in the instruction that specify the displacement distance, in memory locations, between some reference point (determined by the mode) and the desired address.

Effective address calculation - logical process of converting the index, indirect, and displacement bits into an address to be used by the instruction.

Intermediate address - the address obtained by the effective address calculation before testing for indirection.

Page zero - locations 0-377_8 in memory.

When the index bits are 00, the displacement is considered an unsigned integer. When the index bits are 01, 10, or 11, the displacement is considered a signed integer. Table 1.1 shows the range of the displacement field for signed and unsigned integers.

Index Bits	One-word Range	Two-word range*
00	0 to 377_8	0 to 77777_8
01,10,11	-200_8 to 177_8	-40000_8 to 37777_8

Table 1.1 Displacement field ranges

**The Macroassembler will assemble two-word instructions, but will not execute them. That is, you can include 32-bit instructions, or pseudo ops that define 32-bit instructions, in your programs. However, the MP/Computers cannot execute such a program, even though the Macroassembler facility will assemble the correct octal values of the 32-bit instructions.*

Addressing Modes

Word addressing can be done in the following modes:

- Absolute (page zero) addressing
- P.C. (program counter) relative addressing
- Accumulator relative addressing

In addition, direct or indirect addressing can be used in any of these modes. By choosing the proper mode at the appropriate time, you can access any address in the address space.

Figure 1.2 illustrates the three addressing modes.

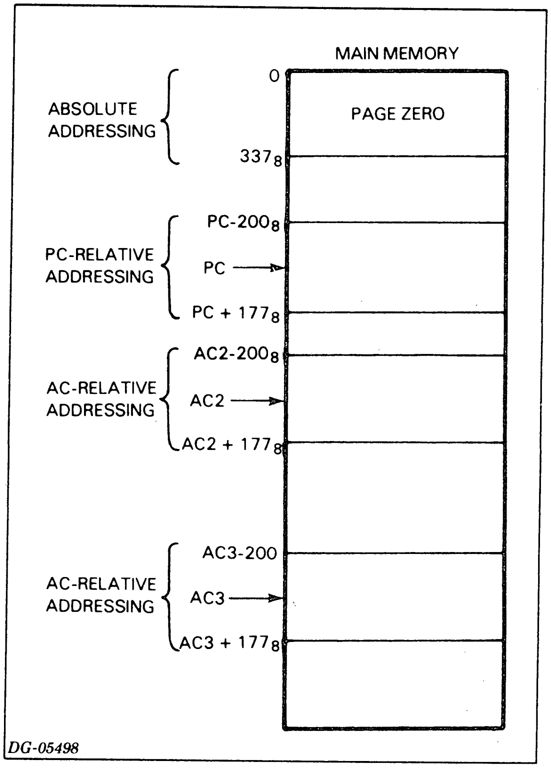


Figure 1.2 Addressing Modes

Absolute Addressing

In absolute addressing mode, the intermediate address is set equal to the unmodified displacement with bits 1-7 set to zero. As a result, instructions specify locations in the range 0-377_8 in the absolute mode.

Page zero thus is very important because any memory-reference instruction can directly address this area. You can use it as a common storage area for items that you frequently reference throughout a program. Note, however, that certain locations are reserved for special purposes. See Table 1.2 for a list of these locations.

P.C. Relative Addressing

In P.C. relative addressing mode, the intermediate address is found by adding the displacement (with bits 1-7 each set to the value of bit 8) to the value of the program counter. The value of the program counter is equal to the address of the word containing the displacement.

Accumulator Relative Addressing

In accumulator relative addressing mode, the intermediate address is found by adding the displacement (with bits 1-7 each set to the value of bit 8) to the contents of bits 1-15 of the accumulator indicated by the index bits. You may use either AC2 or AC3.

Direct and Indirect Addressing

Direct addressing uses the intermediate address without modification.

Indirect addressing uses the intermediate address to obtain an indirect pointer. Bits 1-15 of this pointer are a new intermediate address. If bit 0 of the pointer is 1, then the new intermediate address is used to obtain the next indirect pointer in the indirection chain. If bit 0 of the pointer is 0, then the indirection chain ends and the new intermediate address becomes the effective address.

MP/100 allows up to sixteen levels of indirection in one chain, then halts. MP/200 allows any number of indirection levels.

Auto-Incrementing and Auto-Decrementing

If an intermediate address is in the range $20\text{-}27_8$, and the indirect bit is 1, the contents of the addressed location are incremented by one, and address calculation continues using the *incremented* value of the intermediate address to obtain the next pointer. Locations $20\text{-}27_8$ are called auto-incrementing locations.

If the intermediate address is in the range $30\text{-}37_8$, and the indirect bit is 1, the contents of the addressed location are decremented by one, and address calculation continues using the *decremented* value of the intermediate address to obtain the next pointer. Locations $30\text{-}37_8$ are called auto-decrementing locations.

NOTE: *The state of bit 0* before *the increment or decrement determines whether the indirection chain is continued. For example: Assume an auto-increment location contains* 177777_8 *(all bits = 1 including bit 0), and the location is referenced as part of an indirection chain. After incrementing, the location contains all zeros. However, bit 0 was 1 before the increment, so indirection will continue with the pointer found in location 0.*

Effective Address Calculation

Figure 1.3 illustrates how the processor calculates an effective address. First it determines the addressing mode of the addressing reference, and constructs an intermediate address accordingly. Next it checks for any indirection. If there is no indirection, the effective address takes on the value of the intermediate address. If there is indirection, the processor calculates a new intermediate address, checking for auto-incrementing or -decrementing locations along the way. Once indirection is resolved, the effective address takes on the value of the last-calculated intermediate address.

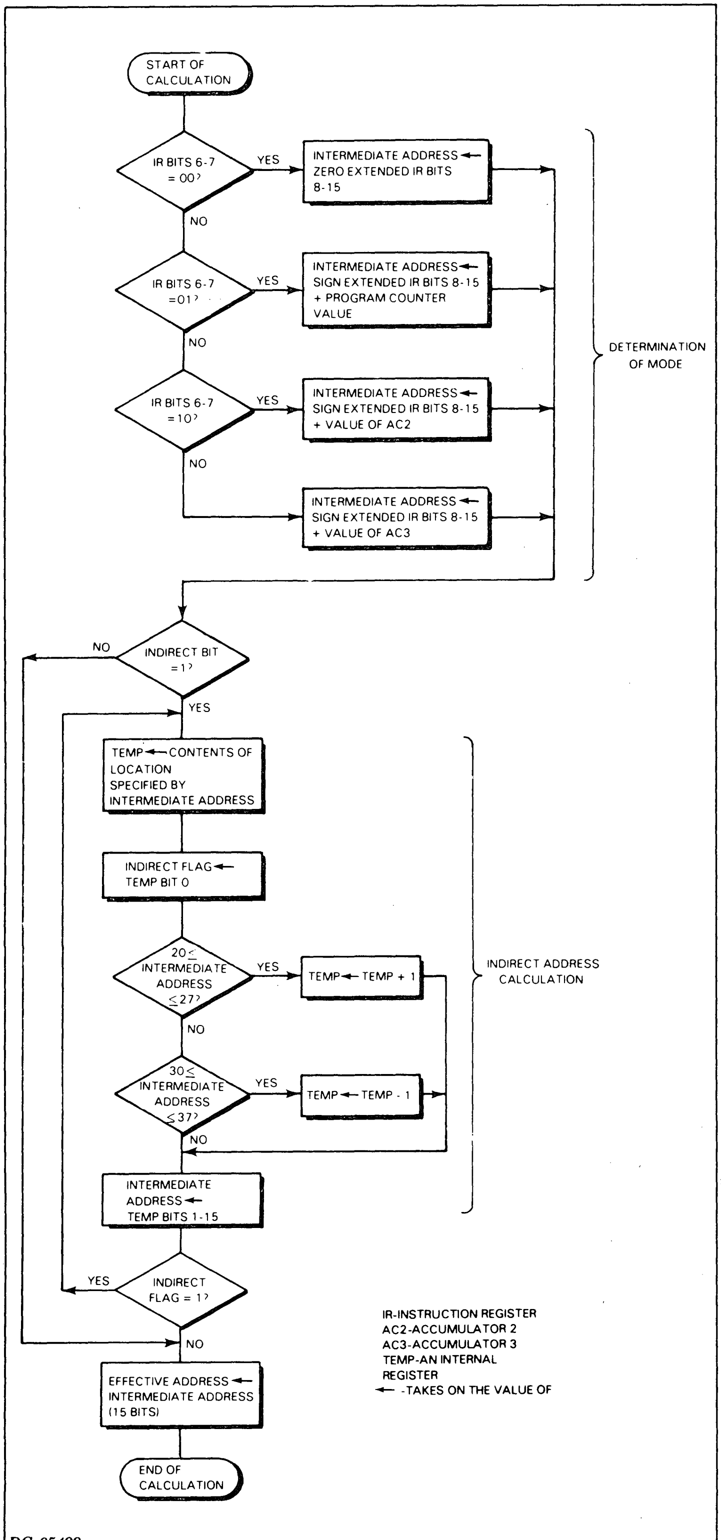


Figure 1.3 Memory address calculation

Allocation of Storage Locations

Table 1.2 describes how page zero memory locations are allotted.

Address	Function
0-15	Reserved for system use
16	Unique Storage Position - assigned on a per task basis
17	Reserved for system use
20-37	Auto-incrementing and -decrementing locations (user accessible)
40-41	Reserved
42	Stack limit for MP/200 **SAVE** and **PSHO** instructions
43-47	Reserved
50-377	User accessible locations
400+	MP/OS may use some of these locations - In general, these locations are user accessible

Table 1.2 Storage allocation

Chapter 2
Data Operations

The following abbreviations are used in this chapter and throughout this book:

ACm - accumulator (m=0, 1, 2, or 3)
ACS - source accumulator
ACD - destination accumulator
PC - program counter

The term *skip* is also used throughout the book. *Skip* means that the contents of the PC are incremented by 1 and the instruction at that location will be the next instruction to be executed.

Byte Manipulation

Bytes are represented as 8-bit unsigned binary integers. A byte in memory is selected by a 16-bit *byte pointer*. Bits 0-14 of the byte pointer contain the memory address of a 2-byte word. Bit 15 (the *byte indicator*) indicates which byte of the addressed location will be used. If bit 15 is 0, the high-order byte (bits 0-7) will be used. If bit 15 is 1, the low-order byte (bits 8-15) will be used. See Figure 2.1.

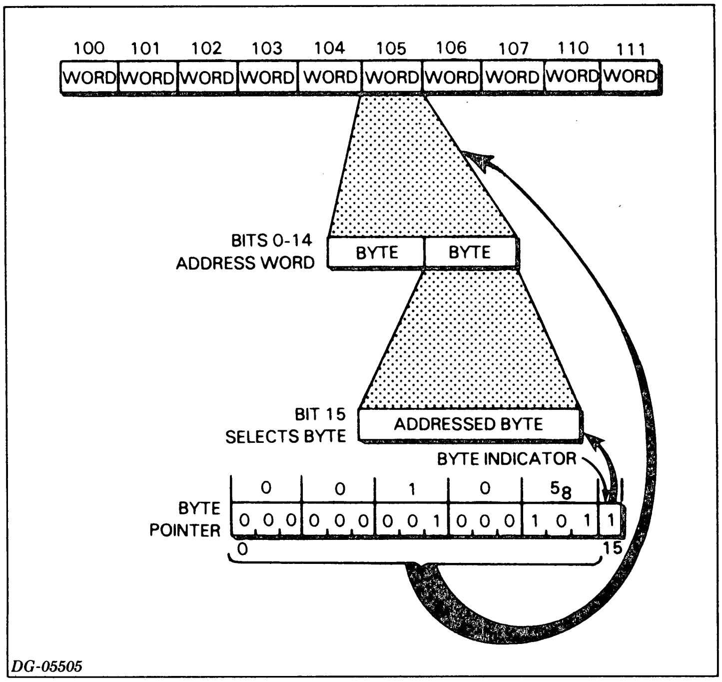


Figure 2.1

Table 2.1 lists the byte manipulation instructions. Note that **LDB** moves a byte from memory to the lower half of a destination accumulator, clearing the high-order half of the destination accumulator. When **STB** moves a byte from an accumulator to memory, it does not change the other byte contained in that word of memory.

Mnem	Instructions	Action
LDB*	Load Byte	Places a byte of information into an accumulator.
STB*	Store Byte	Stores the right byte of an accumulator into a byte of memory.

Table 2.1 Byte manipulation instructions

**This instruction is not supported by microNOVA or MP/100.*

Integer Arithmetic Operations

A signed integer is represented by a two's-complement number in one or more 16-bit words. The sign of the number is positive if bit 0 of the first word is 0 and negative if that bit is 1.

An unsigned integer is represented by using all the bits of one or more 16-bit words to represent the magnitude.

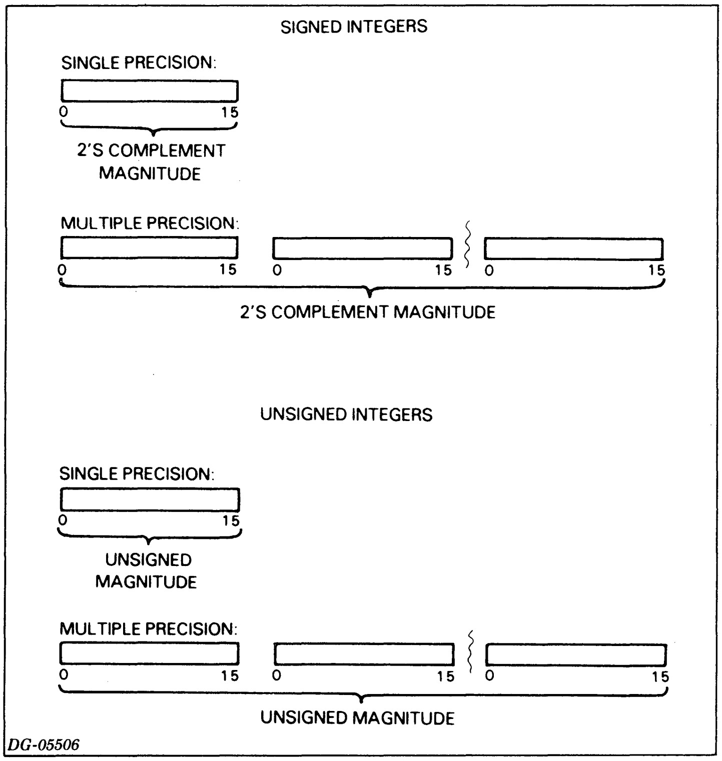


Figure 2.2 Representation of signed and unsigned integers

Single precision integers are one word (16 bits) long and can be manipulated by MP/Computers. You can assemble multiple precision integers which are two or more words long, but you will need software routines to execute programs containing multiple precision integers. Table 2.2 shows the range of single precision numbers MP/Computers can represent, as well as the range of two word multiple precision (double precision) words you can assemble.

Type	Single Precision	Double Precision
Unsigned	0 to 65,535	0 to 4,294,967,295
Signed	-32,768 to +32,767	-2,147,483,648 to +2,147,483,647

Table 2.2 Range of single and double precision integers

The instructions appearing in Table 2.3 perform integer arithmetic (often referred to as fixed point arithmetic). These operations include:

- Moving data between a memory location and an accumulator
- Performing binary arithmetic on values in accumulators
- Incrementing or decrementing a value in memory.

Table 2.3 also gives a brief description of the action of each integer arithmetic instruction.

Mnem	Instructions	Action
ADC	Add Complement	Adds the complement of the unsigned integer contained in ACS to the unsigned integer contained in ACD.
ADD	Add	Adds the contents of ACS to the the contents of ACD.
DIV	Unsigned Divide	Divides the unsigned 32-bit integer contained in AC0 and AC1 by the unsigned integer contained in AC2.
DIVS*	Signed Divide	Divides the signed 32-bit integer contained in AC0 and AC1 by the signed contents of AC2.
DSZ	Decrement and Skip if Zero	Decrements the contents of the addressed word, then skips the next word if the decremented value is zero.
INC	Increment	Increments the contents of ACS.
ISZ	Increment and Skip if Zero	Increments the contents of the addressed word, then skips the next word if the incremented value is zero.
LDA	Load Accumulator	Moves data from the addressed memory location to the specified accumulator.
MOV	Move	Moves the contents of ACS through the arithmetic logic unit (ALU) to ACD.
MUL	Unsigned Multiply	Multiplies the unsigned contents of AC1 and AC2 and adds the results to the unsigned contents of AC0.
MULS*	Signed Multiply	Multiplies the signed contents of AC1 and AC2 and adds the results to the signed contents of AC0.
NEG	Negate	Forms the two's complement of the contents of ACS.
STA	Store Accumulator	Moves data from the specified accumulator to the addressed memory location.
SUB	Subtract	Subtracts the contents of ACS from the contents of ACD.

Table 2.3 Integer Arithmetic Instructions

**This instruction is not supported by microNOVA or MP/100.*

Note that the results of some of the integer arithmetic instructions can affect the value of carry. Overflow conditions complement this value. For specifics, refer to the descriptions of individual instructions in Chapter 4.

Logical Operations

Logical entities are represented as individual bits in a 16-bit word. Each bit is treated as a separate binary value. When two words are involved (logical AND) only corresponding bits of each word interact.

Table 2.4 lists the logical operation instructions and describes each one briefly.

Mnem	Instructions	Action
AND	And	Forms the logical AND of ACS and ACD, puts result in ACD.
COM	Complement	Forms the logical complement of the contents of ACS.

Table 2.4 Logical Operation Instructions

Memory Reference Operations

Memory reference instructions perform one or more of the following operations:

- Load data from memory into a machine register
- Store data from a machine register to memory
- Alter the contents of memory.

Table 2.5 lists the memory reference instructions and describes each one.

Mnem	Instructions	Action
DSZ	Decrement and Skip if Zero	Decrements the contents of the addressed word; then skips the next instruction if the resulting value is zero.
ISZ	Increment and Skip if Zero	Increments the contents of the addressed word; then skips the next instruction if the resulting value is zero.
LDA	Load Accumulator	Moves a word out of memory and into an accumulator.
LDB*	Load Byte	Moves a byte from memory to an accumulator.
STA	Store Accumulator	Stores the contents of an accumulator into a memory location.
STB*	Store Byte	Moves the right byte of one accumulator to a byte in memory.

Table 2.5 Memory Reference Instructions

******This is an MP/200 instruction.*

ALC MANIPULATION

ALC Format

Each of the eight Arithmetic/Logic Class (ALC) instructions performs a specific function upon the contents of one or two accumulators and the carry bit. The eight functions are *Add, Subtract, Negate, Add Complement, Move, And, Complement,* and *Increment.* The instructions are identified by the mnemonics of the eight functions, which are **ADD, SUB, NEG, ADC, MOV, AND, COM,** and **INC.**

In addition to the specific functions performed by an individual instruction, there is a group of general functions all ALC instructions can perform. These general functions include shift operations, which rotate the data left or right, or swap the bytes. Also included are various tests that can be performed on the data. With each test the instructions can check the data for some condition and skip or not skip the next sequential word, depending on the outcome of the test. Finally, the instructions can load or not load the results of the specific and general functions into the destination accumulator and carry. The diagram below shows the format of the ALC instructions.

1	ACS	ACD	OP CODE	SH	C	#	SKIP
0	1 2	3 4	5 7	8 9	10 11	12	13 15

Table 2.6 lists the ALC instructions and briefly describes each one.

Mnem	Instructions	Action
ADC	Add Complement	Adds the complement of the unsigned number contained in ACS to the unsigned number contained in ACD.
ADD	Add	Adds contents of ACS to the contents of ACD.
AND	AND	Forms the logical AND of the contents of ACS and ACD.
COM	Complement	Forms the logical complement of the contents of ACS.
INC	Increment	Increments the contents of ACS
MOV	Move	Moves the contents of ACS through the ALU.
NEG	Negate	Forms the two's complement of the contents of ACS.
SUB	Subtract	Subtracts contents of ACS from the contents of ACD.

Table 2.6 ALC instructions

ALC Instruction Execution

The ALC instructions use an Arithmetic Logic Unit (ALU) to process data. Figure 2.3 illustrates the logical organization of the ALU.

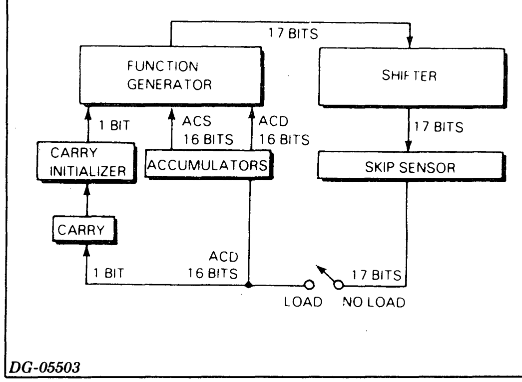


Figure 2.3 Logical organization of the ALU

When an ALC instruction begins execution, it loads the contents of carry and the contents of the accumulator(s) to be processed into the ALU. There are five distinct stages of ALU operation. We will discuss these stages separately.

Carry

The ALU begins its manipulation of the data by determining a new value for carry. This new value is based upon three things: the old value of carry, bits 10-11 of the ALC instruction, and the ALC instruction being executed. The ALU first determines the effect of the instruction bits 10-11 on the old value of carry. Table 2.7 shows each of the mnemonics that can be appended to the instruction mnemonic, the value of bits 10-11 for each choice, and the action each one takes.

Mnemonic	Value of bits 10-11	Action
[c] omitted	00	Leave Carry unchanged
[c]=**Z**	01	Initialize Carry to 0
[c]=**O**	10	Initialize Carry to 1
[c]=**C**	11	Complement Carry

Table 2.7 Carry Mnemonics

Function

The ALU next evaluates the effect of the specific function (bits 5-7) upon the data. For the instructions *Move, AND,* and *Complement,* the ALU performs the function on the data word(s) and saves the result. The value of carry is as it was calculated above. For the instructions *Add, Add Complement, Subtract, Negate,* and *Increment,* the result of the function's action upon the data word(s) may be larger than 2^{16} - 1. An overflow results. In this situation, the ALU saves the low-order 16 bits of the function result, but it complements the value of carry calculated above.

> **NOTE:** *At this stage of operation, the ALU does not load either the saved value of the function result into the destination accumulator, or the calculated value of carry into carry.*

Shift Operations

Next the ALU performs any specified shift operation on the 17-bit output from the function generator (16 bits of data plus the calculated value of the carry bit). Depending on which shift operation is specified in the instruction, the function generator output can be rotated left or right one bit, or have its bytes swapped. Table 2.8 shows the different shift operations that can be performed, the value of bits 8-9 for each choice, and the action each choice takes. Figure 2.4 shows how each shift operation works.

Mnemonic	Value of bits 8-9	Action
[sh] omitted	00	Do not shift the result of the ALC operation.
[sh]=**L**	01	Rotate left the 17-bit combination of carry bit and ALC operation result.
[sh]=**R**	10	Rotate right the 17-bit combination of carry bit and ALC operation result.
[sh]=**S**	11	Swap the two 8-bit halves of the ALC operation result without affecting carry bit.

Table 2.8 Shift Mnemonics

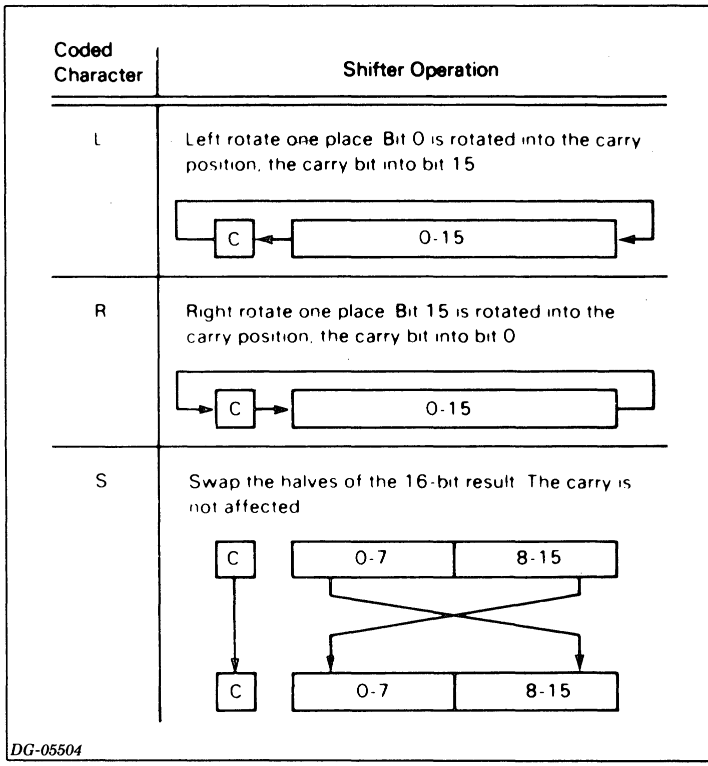


Figure 2.4 Shift operations

Skip Tests

The ALU can test the result of the shift operation for one of a variety of conditions, and skip or not skip the next location depending upon the result of the test. Table 2.9 shows the tests that can be performed, the value of bits 13-15 for each choice, and the action each choice takes.

Mnemonic	Value of bits 13-15	Action
[skip] omitted	000	No skip.
[skip]=**SKP**	001	Skip unconditionally.
[skip]=**SZC**	010	Skip if carry is zero.
[skip]=**SNC**	011	Skip if carry is nonzero
[skip]=**SZR**	100	Skip if ALC result is zero.
[skip]=**SNR**	101	Skip if ALC result is nonzero.
[skip]=**SEZ**	110	Skip if either ALC result or Carry bit is zero.
[skip]=**SBN**	111	Skip if both ALC result and carry are nonzero.

Table 2.9 Skip Test Mnemonic

Load/No-Load

If the no-load bit (bit 12) is 0, the ALU loads the result of the shift operation into the destination accumulator, and loads the new value of carry into carry. If the no-load bit is 1, then the ALU does not load the result of the shift operation into the destination accumulator, and does not load the new value of carry into carry, but all other operations, such as skip tests, take place. This no-load option is particularly convenient to use when you want to test for some condition without destroying the contents of the destination accumulator. Table 2.10 shows how to code the load/no-load operation.

Symbol	Value of bit 12	Action
# omitted	0	Load the result of the shift operation into ACD and carry.
#	1	Do not load the ALC operation result into ACD; retain the initial value of carry from the start of this instruction.

Table 2.10 Load/No Load Symbols

NOTE: *ALC instructions must not have both the* No-Load *and the* Never-Skip *options specified at the same time. This bit combination (bits 12-15=1000) is used to specify other non-ALC instructions.*

Stack Operations

The stack is a series of consecutive locations in memory. In their simplest form, stack instructions add items in sequential order to the top of the stack and retrieve them in the reverse order. You can define several stack areas in your program, but you can directly use only one stack at any time. The MP/Computers use the push-down stack concept to provide easily accessible temporary storage of data, variables, return addresses, etc.

The stack is made up of an area in memory, defined by the stack control registers, and referenced by the stack instructions. Since the stack locations are not reserved in memory, you can define them to be convenient to your current use. Figure 2.5 shows a typical stack area in memory.

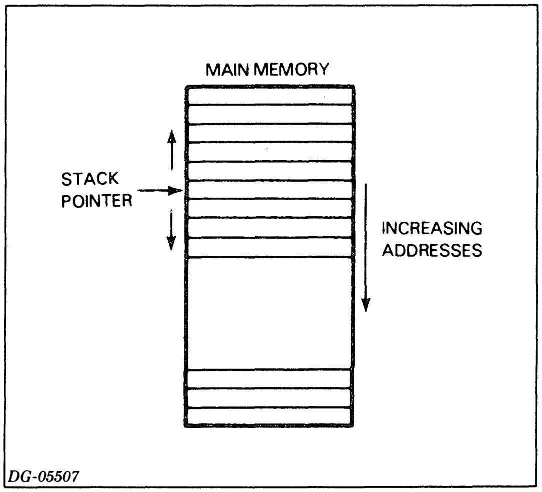


Figure 2.5 The stack

The simplest use of the stack is for temporary storage of single data words. You can also use the stack to store a *return block*, which greatly simplifies the process of entering and returning from subroutines.

The return block consists of five words. These words contain the contents of the accumulators (AC0, AC1 and AC2), the carry bit, and the frame pointer. There are no fixed upper or lower limits for the stack, but if the stack crosses a specified boundary an overflow interrupt will result (see "Stack Overflow Protection" below).

Stack Control Registers

The stack pointer and the frame pointer are two 15-bit registers that control access to the stack.

Stack Pointer

When the stack is set up, the value of the stack pointer is initially set to one less than the address of the first location in the stack. After that, each time you push a word onto the stack the stack pointer increments by one. When you pop the top word from the stack, the stack pointer decrements by one. If you push or pop a five-word return block, the stack pointer increments or decrements by five. This means that the stack pointer always points to the last element on the stack.

Frame Pointer

When you enter a subroutine which begins with a **SAV** or **SAVE***n* instruction, the frame pointer is set to the same value as the stack pointer. However, the frame pointer does not automatically change its value when you push or pop words, although you can use an instruction to alter the value. This makes it a useful reference pointer, because it allows direct access to temporary data locations in the stack "frame".

When a program calls another routine, a return block is pushed upon the stack. This return block enables the state of the machine to be restored when the routine uses the **RET** instruction to return control to the calling program. Since the frame pointer automatically changes upon entering a new routine, it will point to the last word in the return block. This means you can use the frame pointer as a reference to the beginning of an area for local use by the called routine.

Stack Overflow Protection

There are two kinds of overflow protection used on MP/Computers. The **SAV** and **PSHA** instructions provide overflow protection against crossing a *page boundary*. A page boundary is a memory location whose address is an integral multiple of 400_8. If, during the execution of one of these instructions, the stack pointer crosses a page boundary, and interrupts are enabled, then a stack overflow interrupt occurs. If you cross a page boundary when interrupts are not enabled, then an interrupt will occur as soon as you enable interrupts.

The **SAVE** and the **PSHO** instructions provide interrupt protection against exceeding the *stack limit*. Location 42_8 contains the stack limit. If, during the execution of one of these instructions, the contents of the stack pointer are greater than the contents of location 42_8, then a stack overflow interrupt occurs. If the contents of the stack pointer are less than or equal to the contents of location 42_8, then no interrupt occurs.

Note that stack interrupts have priority over I/O interrupts.

When a stack overflow interrupt occurs, the processor:

- Disables interrupts
- Stores the contents of the program counter into memory location 0
- Performs a jump indirect through location 3.

Location 3 must contain a pointer to a stack overflow interrupt handler.

Note that the MP/OS operating system automatically handles stack overflow interrupts for you.

Table 2.11 lists the stack instructions and briefly describes each one.

Mnem	Instructions	Action
MFFP	Move From Frame Pointer	Places the contents of the frame pointer in bits 1-15 of the specified accumulator. Bit 0 contains 0.
MFSP	Move From Stack Pointer	Places the contents of the stack pointer in bits 1-15 of the specified accumulator. Bit 0 contains 0.
MTFP	Move To Frame Pointer	Places bits 1-15 of the specified accumulator in the frame pointer.
MTSP	Move To Stack Pointer	Places bits 1-15 of the specified accumulator in the stack pointer.
POPA	Pop Accumulator	Pops the top word off the stack and places it in the specified accumulator. Decrements stack pointer by 1.
PSHA	Push Accumulator	Increments stack pointer by 1; then pushes the contents of the specified accumulator onto the top of the stack. Checks for page boundary overflow.
PSHO*	Push Accumulator	Increments stack pointer by 1; then pushes the contents of the specified accumulator onto the top of the stack. Checks for stack limit overflow.
RET	Return	Places the contents of the frame pointer in the stack pointer; then pops the top five words off the stack and places them in the accumulators, carry bit, and the program counter. Restores the frame pointer and loads this value into AC3.
SAV	Save	Pushes a return block onto the stack. Checks for page boundary overflow. Designed for use after a **JSR** instruction.
SAVE*	Save	Pushes a return block onto the stack. Allocates a frame by adding a constant to the stack pointer. Checks for stack limit overflow. Designed for use after a **JSR** instruction.

Table 2.11 Stack Instructions

**This instruction is not supported by microNOVA or MP/100.*

Program Flow Alteration

Normally, program execution is sequential. That is, the CPU processes instructions from sequential memory locations. You can alter this sequential flow by using one of the program flow alteration instructions. Each of these instructions loads a new value into the program counter; the CPU continues execution at that new address and continues with the instructions immediately following.

Table 2.12 lists the program flow alteration instructions and gives a brief description of each.

Mnem	Instructions	Action
DSZ	Decrement And Skip If Zero	Decrements the addressed word; then skips if the decremented value is zero.
ISZ	Increment And Skip If Zero	Increments the addressed word; then skips if the incremented value is zero.
JMP	Jump	Places the effective address in the program counter.
JSR	Jump To Subroutine	Increments program counter and stores incremented value in AC3; then places the effective address in the program counter.
RET	Return	Returns control from a subroutine containing a **SAVE** or **SAV** instruction.
TRAP	Trap	Places the address of this instruction in location 46_8 and jumps indirect through location 47_8.

Table 2.12 Program Flow Alteration Instructions

Chapter 3
Input/Output

Since there are many different I/O devices available for MP/Computers, the following discussion is necessarily general. For programming considerations of a specific I/O device, or for a description of the I/O bus and its protocol, refer to the *Microproducts Hardware Systems Reference* (DGC No. 014-000636).

Device Codes

MP/Computers have a 6-bit I/O device selection network, corresponding to bits 10-15 in the I/O instruction format. The devices are connected to this network in such a way that each device responds only to commands sent with its own device code. With a 6-bit device code, you can specify 64 device codes. Some of these device codes are reserved for the CPU and certain processor options, but the remaining codes are available for referencing I/O devices. The assembler recognizes mnemonics for those devices assigned a code by Data General. A complete list of these appears in Appendix C of this manual.

Programmed I/O

Programmed I/O transfers data one word at a time under direct program control. For relatively slow devices, such as terminals, which transfer one character at a time and require an immediate echo, programmed I/O is the most efficient method of I/O operation. Programmed I/O is also used to specify parameters for data channel operations or for obtaining system status information.

Data Channel I/O

MP/Computers have two data channels. The standard data channel transfers data between a device and memory via the CPU. The high speed data channel transfers data directly between a device and memory. Both types of transfer are transparent to the user.

Both standard and high speed data channels permit data to be transferred in blocks of words, with program control necessary only at the start of the operation. The CPU stops during each word transfer made by either data channel, although the length of time it stops is much shorter for the high speed data channel. In general, data channel I/O is a very efficient method of transferring large blocks of data between memory and a fast I/O device. When single words or bytes are needed, however, programmed I/O is more appropriate.

In general, data channel devices are controlled in two phases. Phase I specifies the starting location in memory for the first word to be transferred, the number of words to be transferred, and the direction of the transfer. This phase is done with programmed I/O instructions. During phase II the information transfers take place between device and memory.

When a data channel device is ready to send or receive data, it issues a data channel request to the processor. At the beginning of every memory cycle, the processor synchronizes any requests that are then being made. The MP/100 CPU pauses at the end of an instruction cycle to honor all previously synchronized requests. (The MP/200 CPU can honor standard requests at the end of any instruction cycle. High-speed requests may occur at any time, though they block out CPU memory references.) When a request is honored, a word is transferred via the data channel between the device and memory. No software control is necessary.

All requests are honored according to the position of the simultaneously requesting devices on the data channel priority chain. The requesting device with the highest priority is serviced first; the device with the next highest priority is serviced next, and so on, until all requests have been honored. The synchronization of new requests occurs concurrently with the honoring of other requests. If a device continually requests the data channel, that device can prevent all devices with lower priority from gaining access to the channel.

After handling all data channel requests, the processor then handles all outstanding I/O interrupt requests. Only then does program execution continue.

I/O Instructions

Table 3.1 lists the standard I/O instructions and briefly describes each one.

Mnem	Instructions	Action
DIA	Data In A	Transfers data from the A buffer of an I/O device to an accumulator.
DIB	Data In B	Transfers data from the B buffer of an I/O device to an accumulator.
DIC	Data In C	Transfers data from the C buffer of an I/O device to an accumulator.
DOA	Data Out A	Transfers data from an accumulator to the A buffer of an I/O device.
DOB	Data Out B	Transfers data from an accumulator to the B buffer of an I/O device.
DOC	Data Out C	Transfers data from an accumulator to the C buffer of an I/O device.
NIO	No I/O Transfer	Changes a flag without causing the transfer of data.
SKP*t*	I/O Skip	Tests a flag and skips the next sequential word if the test condition is true.

Table 3.1 Standard I/O instructions

Program Interrupt Facility

This section discusses priority interrupts and priority interrupt systems. If you are running programs under the MP/OS operating system, then the operating system provides drivers that use the priority system for you. If you are not running under the MP/OS operating system, then you will find the information in this section useful. For more information, refer to Part III of this manual.

Your programs must be able to tell when data transfers are complete. You can do this by repeatedly testing the Done flag; however, this method is wasteful if the processor could be doing more useful work elsewhere. The program interrupt facility provides peripheral devices with a convenient means of alerting the CPU for service.

The following aspects of the program interrupt facility are discussed below:

- The interrupt request line
- Control flags
- Initiating an interrupt
- Servicing an interrupt
- Dismissing an interrupt.

The Interrupt Request Line

All peripherals that use the program interrupt facility have access to a direct line to the CPU. This line, called the interrupt request line, transmits requests for service from the devices to the CPU.

Control Flags

The Interrupt On (ION) flag in the CPU and the Interrupt Disable flags of the devices allow you to control which devices can request interrupts. When the ION flag is set to 1, the interrupt facility is enabled and the CPU can respond to an interrupt request. When the ION flag is set to 0, the interrupt facility is disabled and the CPU ignores all interrupt requests.

You can set the ION flag to 1 by issuing a Start command in any non-skip I/O instruction to the CPU. You can set the ION flag to 0 by issuing a Clear command in any non-skip I/O instruction to the CPU. (See the CPU instructions in Chapter 5.)

Each device using the interrupt facility has an Interrupt Disable flag in its controller. You manipulate these flags independently to disable interrupts on an individual device level. If you set a device's Interrupt Disable flag to 1, then that device cannot request an interrupt.

You can simultaneously set multiple Interrupt Disable flags by using the *Mask Out* instruction. (See the *Mask Out* instruction in Chapter 5, and "Priority Interrupts", below.)

Initiating an Interrupt and CPU Response

When a device requires service, it sets its Done flag to 1, which initiates a program interrupt request. If the device's Interrupt Disable flag is 0, the CPU receives the request. If the Interrupt Disable flag is 1, the CPU will ignore the interrupt request until the Interrupt Disable flag is reset.

If interrupts are enabled (ION flag is 1), the CPU will service a program interrupt upon completion of an instruction or a data channel request, if no other data channel requests are pending (data channel requests have a higher priority than program interrupts).

The CPU responds to a program interrupt request by first setting the ION flag to 0 so that no other interrupts can be started. Next the CPU stores the contents of the program counter in location 0. This allows the CPU to resume the interrupted program after servicing the interrupt. Finally, the CPU simulates a jump indirect through one of three locations to the interrupt service handler. Table 3.2 shows the possible pointer locations and when each is used.

Location	When Used
1	For device interrupt requests
2*	For real-time clock interrupt requests
3	For stack interrupt requests

Table 3.2 Interrupt handler pointers

**MP/100 and microNOVA only.*

The address in any of these locations can point directly to the interrupt service handler, or it can be the first address of an indirection chain pointing to the handler.

Servicing an Interrupt

An interrupt service handler should save the state of the CPU, identify the device that requested the interrupt, and transfer control to the appropriate peripheral service routine. Saving the state means saving the contents of the accumulators, carry, and the stack registers so that, when the interrupt service is complete, the CPU can resume execution of the interrupted program as if no interrupt occurred.

To identify the device requesting the interrupt, the handler can issue the *Interrupt Acknowledge* instruction. This instruction loads the device code of the highest priority interrupting device into an accumulator. Another method would be to use a series of *I/O Skip* instructions which test the Done flags of the devices. However, with this method a masked out device can be identified as requesting an interrupt, if the device's Done flag has been set to 1.

After the handler identifies the device requesting an interrupt, it usually transfers control to a peripheral service routine. This routine performs the required information transfer and then either starts the device on a new operation, or idles it if there are no operations to be performed.

Dismissing an Interrupt

After servicing the interrupting device, either the peripheral service routine or the main interrupt handler should perform the following sequence of events:

- Set the device's Done flag to 0 to dismiss the interrupt just honored. (If you leave this out, the undismissed interrupt will cause another interrupt when you attempt to transfer control back to the interrupted program.)
- Restore the state of the CPU.
- Set the ION flag to 1 to enable interrupts. (Although interrupts are enabled with one instruction the CPU will not respond to an interrupt request until after the *next* instruction executes)
- Return to the interrupted program. (This usually is done by a jump indirect through location 0, since this is where the CPU placed the value of the program counter when it began to service the interrupt.)

Priority Interrupts

The priority interrupt facility allows faster or more important I/O devices to interrupt the service of slower or less important I/O devices. The two basic features of this facility are the interrupt priority mask and the priority interrupt handler.

Interrupt Priority Mask

The interrupt priority mask is a 16-bit word. Each I/O device in a system is assigned a mask bit that governs the device's Interrupt Disable flag. When a particular bit in the mask is set to 1, the Interrupt Disable flag in the corresponding device is set to 1 and the device is masked out, or disabled. This means it has a lower priority than the device currently being serviced. Those devices whose corresponding mask bits are 0 have a higher priority than the device being serviced. The CPU interrupts service to the lower priority device to honor an interrupt request from a higher priority device.

The mask bits are assigned to the devices on the hardware level, so you cannot alter them. You can, however, control the order of priority of these bits. In your program, you can use the priority mask to rank your I/O devices in any order. Note,however that certain I/O devices which operate at roughly the same speed are assigned the same mask bit; these devices will always have the same priority.

Priority Interrupt Handler

If you decide to use a priority interrupt structure in your system, the interrupt handler must be re-entrant. This means that if a device service routine is interrupted by a higher priority device, there will be no loss of the information the handler needs to restore the state of the machine. For a handler to be re-entrant, it must be able to save the contents of location 0 (the return address) and the current priority mask each time it is entered at a higher level. It should also be able to perform the following sequence of operations:

- Save the state of the processor (accumulators, carry, stack registers, contents of location 0, and the current priority mask)
- Identify the device requesting the interrupt
- Transfer control to the interrupting device's service routine
- Establish and store a new priority mask
- Enable interrupts
- Service the device
- Disable interrupts
- Restore the state of the processor
- Enable interrupts
- Transfer control to the return address saved from location 0.

To set up a system of priorities, place a *Mask Out* instruction in the interrupt service handler for each device. This instruction changes the priority mask, thus controlling which devices can interrupt. Devices that should not interrupt the device being serviced are masked out (prevented from requesting an interrupt) if their mask bits are 1. In addition, all pending interrupt requests from devices controlled by that bit are disabled. The other mask bits, corresponding to devices that can interrupt, are set to 0.

When each interrupt service handler uses the *Mask Out* instruction as described above, masking out is a dynamic process, changing each time a different device is serviced. The system of priorities allows the device with the highest priority to interrupt all other devices, and the device with the lowest priority to be interrupted by all other devices.

Figure 3.1 shows a simplified version of program flow in a priority interrupt system.

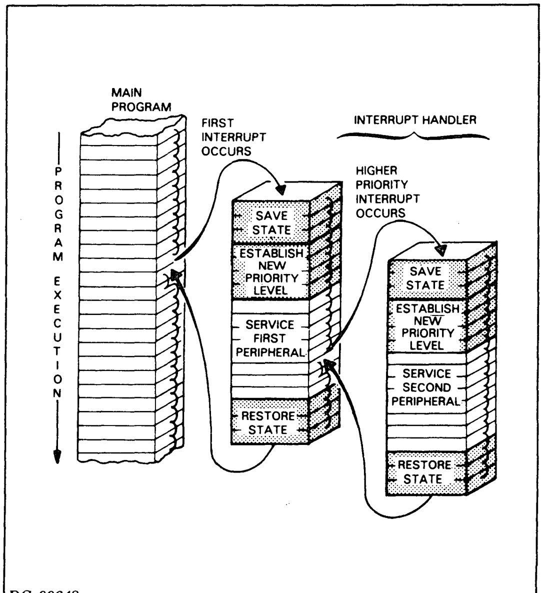


Figure 3.1 Program flow in a priority interrupt system

CPU I/O Instructions

Table 3.3 lists the I/O instructions dealing with the CPU.

Mnem	Instructions	Action
HALT (DOC, CPU)	Halt	Stops the processor.
INTA (DIB, CPU)	Interrupt Acknowledge	Returns the device code of an interrupting device.
INTDS (NIOC, CPU)	Interrupt Disable	Sets Interrupt On flag to 0.
INTEN (NIOS, CPU)	Interrupt Enable	Sets Interrupt On flag to to 1.
IORST (DOAC, CPU)	Reset	Sets all Busy, Done and Interrupt Disable flags to 0.
MSKO (DOB, CPU)	Mask Out	Changes the interrupt disable flags according to the priority mask.
SKP*t*, CPU	CPU Skip	Tests the Interrupt On flag and flag and skips the next sequential word if the test condition is true.

Table 3.3 CPU I/O instructions

Most of the instructions in Table 3.3 have two forms. If you use the standard form of the instruction, **DOB**, for example, then you can specify a function *f* to act upon the ION flag. If you use the special form of the instruction, **MSKO**, you cannot specify any such function. The functions you can specify with the standard form of the instructions are shown in Table 3.4. Refer to the specific instruction entries in Chapter 5 for more information.

Mnemonic	Sets bits 8-9 to	Action
---	00	Does not alter ION flag.
S	01	Sets ION flag to 1.
C	10	Clears ION flag to 0.
P	11	Leaves ION flag unchanged.

Table 3.4 Optional Mnemonics

The *Skip* instruction can perform tests on the ION and Power Fail flags. Table 3.5 lists the possible tests and the mnemonics for each. Refer to the **SKP*t*, CPU** entry in Chapter 5 for more information.

Memonic	Action
SKPBN	SKIP if interrupts are disabled.
SKPBZ	SKIP if interrupts are enabled.

Table 3.5 Skip tests

Real-Time Clock

The real-time clock generates periodic interrupts when you enable it. The MP/200's basic controller has a standard real-time clock (device code 14).

The MP/100 has an integral real-time clock that you can configure to interrupt at intervals of either 1.9 ms, or intervals defined by the line frequency.

When the CPU receives an interrupt, it sets the Interrupt On flag to 0, places the updated program counter in location 0, and simulates a jump indirect through location 2.

Table 3.6 describes the two (MP/100 and microNOVA) instructions that allow you to enable or disable the real-time clock. Note that the instructions have two forms. The **DOA** form allows you to set the Interrupt On flag. If you use the **RTC** form of the instructions, you cannot set the Interrupt On flag.

Mnem	Instructions	Action
RTCEN **DOA 2,CPU**	Real-Time Clock Enable	Enables the real-time clock. The instruction **DOAS 2,CPU** sets ION to 1
RTCDS **DOA 2,CPU**	Real-Time Clock Disable	Disables the real-time clock. The instruction **DOAS 2,CPU** sets ION to 1.

Table 3.6 Real-Time clock instructions (MP/100 and microNOVA only

Power Up Sequence

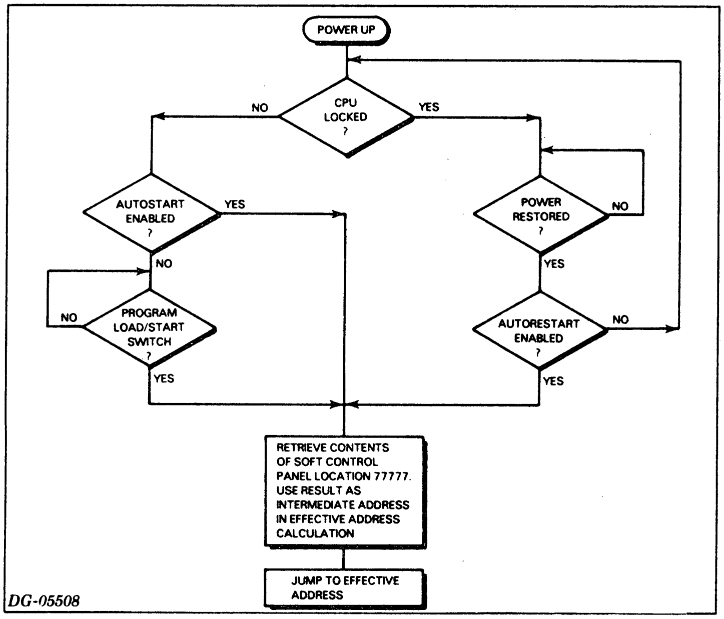


Figure 3.2 Power up sequence for MP/100 CPU

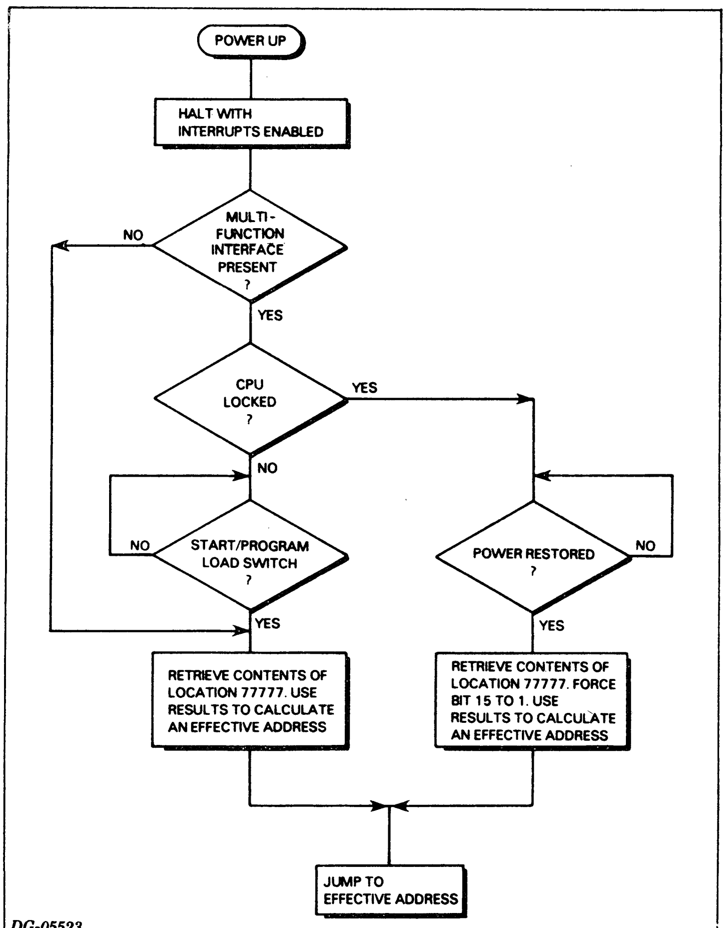


Figure 3.3 Power-up sequence for the MP/200 SPU

After power is applied to the SPU, the CPU is initialized and enters the *Halt* state. If the control panel is unlocked, CPU response depends upon whether the auto-restart feature is enabled. If auto-restart is enabled, the CPU makes an effective address calculation using the contents of soft control panel location 77777_8 as an intermediate address. The CPU sets bit 15 of location 77777_8 when it reads that location. When the effective address is resolved, the CPU jumps to the effective address location and begins execution.

If the auto-restart feature is not enabled, the CPU remains halted until the user depresses the Program Load/Start switch. Then the CPU jumps through the soft control panel location 77777_8, as described above.

In the MP/200, if the multi-function controller is not present, the CPU does not check for CPU lock, or for auto-restart, and it does not stay in the Halt state until the Program Load/Start switch is depressed. Instead, the CPU jumps immediately to the address pointed to by memory location 77777_8.

For more information about the power up sequence, refer to the *Microproducts Hardware System Reference,* DGC No. 014-000636.

Chapter 4
Instruction Dictionary

This chapter lists all the instructions for MP/Computers except for I/O instructions. The instructions are arranged in alphabetical order according to instruction mnemonic.

Each instruction entry includes:

- The mnemonic recognized by the assembler
- The bit format
- The number and format of any arguments
- A description of how the instruction works.

ADC

Add Complement

ADC*[c][sh][#] acs,acd[,skip]*

Valid for: MP/200, MP/100

1	ACS	ACD	1	0	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Adds an unsigned integer to the logical complement of another unsigned integer.

Sets carry to the specified value. Adds the logical complement of the unsigned, 16-bit integer in ACS to the unsigned, 16-bit integer in ACD. If the addition produces a result greater than 2^{16} -1, then the value of carry is complemented. Places the 17-bit result (carry and function result) in the shifter. Performs the specified shift operation, and loads the result of the shift into carry and ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

NOTE: *If the number in ACS is less than the number in ACD, the instruction complements carry.*

ADD

Add

ADD*[c][sh][#] acs,acd[,skip]*

Valid for: MP/200, MP/100

1	ACS	ACD	1	1	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Performs unsigned integer addition and complements the value of carry if appropriate.

Sets the value of carry to the specified value. Adds the unsigned, 16-bit number in ACS to the unsigned, 16-bit number in ACD. If the result is greater than 2^{16}-1, then the value of carry is complemented. Places the 17-bit result (carry and function result) in the shifter. Performs the specified shift operation and places the result of the shift in carry and ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

AND

AND

AND*[c][sh][#]acs,acd[,skip]*

Valid for: MP/200, MP/100

1	ACS	ACD	1	1	1	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Forms the logical AND of the contents of two accumulators.

Set the value of carry to the specified value. Places the carry and the logical AND of ACS and ACD in the shifter. Each bit placed in the shifter is 1 only if the corresponding bit in both ACS and ACD is 1; otherwise the resulting bit is 0. The instruction then performs the specified shift operation and places the result in the carry and ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

COM

Complement

COM*[c][sh][#]acs,acd[,skip]*

Valid for: MP/200, MP/100

1	ACS	ACD	0	0	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Forms the logical complement of the contents of an accumulator.

Sets the value of carry to the specified value. Forms the logical complement of the number in ACS, and places the 17-bit value (carry and function result) in the shifter. Performs the specified shift operation and places the result in carry and ADC if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

DIV

Unsigned Divide

DIV

Valid for: MP/200, MP/100

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	1	0	1	1	1	1	1	0	0	1	0	0	0

Divides the unsigned 32-bit integer in two accumulators by the unsigned contents of a third accumulator. The quotient and remainder each occupy one accumulator.

Divides the unsigned 32-bit number contained in AC0 and AC1 by the unsigned, 16-bit number in AC2. The remainder and quotient are unsigned, 16-bit numbers and are placed in AC0 and AC1, respectively. The carry bit is set to 0. The contents of AC2 remain unchanged.

> **NOTE:** *Before the divide operation takes place, the number in AC0 is compared to the number in AC2. (This is an unsigned compare.) If the contents of AC0 are greater than or equal to the contents of AC2, an overflow results. Carry is set to 1, and the operation is terminated. All operands remain unchanged.*

DIVS

Signed Divide

DIVS

Valid for: MP/200

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1

Divides the signed 32-bit integer in AC0 and AC1 by the signed contents of AC2. The quotient and remainder occupy AC1 and AC0 respectively.

Divides the signed, 32-bit two's complement number contained in AC0 and AC1 by the signed, 16-bit two's complement number in AC2. Places the signed 16-bit quotient in AC1 and the signed, 16-bit remainder in AC0. The rules of algebra determine the sign of the quotient. The sign of the remainder is always the same as the sign of the dividend, except that a zero quotient or a zero remainder is always positive. Sets the carry bit to 0. Leaves the contents of AC2 unchanged.

> **NOTE:** *If the quotient is too large to fit into AC1, an overflow results. The processor sets carry to 1 and terminates the instruction operation. The contents of AC0 and AC1 are unpredictable.*

DSZ

Decrement And Skip If Zero

DSZ *[@]displacement [,index]*

Valid for: MP/200, MP/100

0	0	0	1	1	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Decrements the addressed word; then skips if the decremented value is zero.

Computes the effective address, *E*. Decrements by one the word addressed by *E* and writes the result back into that location. If the updated value of the location is zero, the instruction skips the next sequential word.

INC

Increment

INC*[c][sh][#] acs,acd[,skip]*

Valid for: MP/200, MP/100

1	ACS	ACD	0	1	1	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Increments the contents of an accumulator.

Sets the value of carry to the specified value. Increments the unsigned, 16-bit number in ACS by one. If the incremented value is greater than 2^{16}-1, then the value of carry is complemented. Places the 17-bit value (carry and function result) in the shifter. Performs the specified shift operation, and loads the result of the shift into carry and ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

ISZ

Increment And Skip If Zero

ISZ *[@]displacement [,index]*

Valid for: MP/200, MP/100

0	0	0	1	0	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Increments the addressed word, then skips if the incremented value is zero.

Computes the effective address, *E* . Increments the word addressed by *E* and writes the result back into memory at that location. If the updated value of the location is zero, the instruction skips the next sequential word.

JMP

Jump

JMP *displacement*

Valid for: MP/200, MP/100

0	0	0	0	0	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6 7	8 15

Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

JSR

Jump To Subroutine

JSR *[@]displacement [,index]*

Valid for: MP/200, MP/100

0	0	0	0	1	@	INDEX	DISPLACEMENT
0	1	2	3	4	5	6–7	8–15

Increments and stores the value of the program counter in AC3, and then places a new address in the program counter.

Computes the effective address, *E*. Places the 15-bit address of the next sequential word in AC3. Bit 0 of AC3 is 0. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

> **NOTE:** *The instruction computes* E *before it places the incremented program counter in AC3.*

LDA

Load Accumulator

LDA *ac,[@]displacement [,index]*

Valid for: MP/200, MP/100

0	0	1	AC	@	INDEX	DISPLACEMENT
0	1	2	3–4	5	6–7	8–15

Copies a word from memory to an accumulator.

Computes the effective address, *E*. Places the word addressed by *E* in the specified accumulator. The previous contents of the location addressed by *E* remain unchanged.

LDB

Load Byte

LDB *acs,acd*

Valid for: MP/200

0	1	1	ACD		0	0	1	ACS		0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Uses a byte pointer contained in ACS to load a byte from memory into bits 8-15 of ACD. Sets bits 0-7 of ACD to 0. The contents of ACS remain unchanged, unless ACS and ACD are the same accumulator.

MFFP

Move From Frame Pointer

MFFP *ac*

Valid for: MP/200, MP/100

0	1	1	AC		0	0	0	1	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Places the contents of the frame pointer in bits 1-15 of the specified accumulator. Sets bit 0 of the specified accumulator to 0. Leaves the contents of the frame pointer unchanged.

MFSP

Move From Stack Pointer

MFSP *ac*

Valid for: MP/200, MP/100

0	1	1	AC	0	1	0	1	0	0	0	0	0	0	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Places the contents of the stack pointer in bits 1-15 of the specified accumulator. Sets bit 0 of the specified accumulator to 0. Leaves the contents of the stack pointer unchanged.

MOV

Move

MOV*[c][sh][#]acs,acd[,skip]*

Valid for: MP/200, MP/100

1	ACS	ACD	0	1	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Moves the contents of an accumulator through the arithmetic logic unit (ALU).

Sets the value of carry to the specified value. Places the contents of carry and ACS in the shifter. Performs the specified shift operation and loads the result of the shift into carry and ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

MTFP

Move To Frame Pointer

MTFP *ac*

Valid for: MP/200, MP/100

0	1	2	3-4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC	0	1	0	0	0	0	0	0	0	0	1

Places bits 1-15 of the specified accumulator in the frame pointer. The contents of the accumulator remain unchanged.

MTSP

Move To Stack Pointer

MTSP *ac*

Valid for: MP/200, MP/100

0	1	2	3-4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC	0	1	0	0	0	0	0	0	0	0	1

Places the contents of the specified accumulator in the stack pointer. Leaves the contents of the specified accumulator unchanged.

MUL

Unsigned Multiply

MUL

Valid for: MP/200, MP/100

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	0	1	1	1	1	1	0	0	1	0	0	0

Multiplies the unsigned integers contained in AC1 and AC2. Adds the unsigned contents of AC0 to the product. AC0 and AC1 contain the resulting 32-bit integer.

Multiplies the unsigned, 16-bit number in AC1 by the unsigned, 16-bit number in AC2 to yield an unsigned, 32-bit product. Adds the unsigned, 16-bit number in AC0 to the product. The final result is an unsigned, 32-bit number and occupies AC0 and AC1. Bit 0 of AC0 is the high-order bit of the result and bit 15 of AC1 is the low-order bit. The contents of AC2 and carry remain unchanged. Because the result is a double-length number, overflow cannot occur.

MULS

Signed Multiply

MULS

Valid for: MP/200

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	1

Multiplies the integers in AC1 and AC2. Adds an integer contained in AC0 to the product. AC0 and AC1 contain the result.

Multiplies the signed, 16-bit two's complement number in AC1 by the signed, 16-bit two's complement number in AC2 to yield a signed, 32-bit two's complement product. Adds the signed, 16-bit two's complement number in AC0 to the product. The final result is a signed, 32-bit two's complement number which occupies AC0 and AC1. Bit 0 of AC0 is the sign bit of the result and bit 15 of AC1 is the low-order bit. The contents of AC2 and carry remain unchanged. Because the result is a double-length number, overflow cannot occur.

NEG

Negate

NEG*[c][sh][#] acs,acd[,skip]*

Valid for: MP/200, MP/100

0	1–2	3–4	5	6	7	8–9	10–11	12	13–15
1	ACS	ACD	0	0	1	SH	C	#	SKIP

Forms the two's complement of the contents of an accumulator.

Sets the value of carry to the specified value. Takes the two's complement of the unsigned, 16-bit number in ACS. Places the 17-bit value (carry and the function result) in the shifter. Performs the specified shift operation and places the result in carry and ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

> **NOTE:** *If ACS contains 0, the instruction complements carry.*

POPA

Pop Accumulator

POPA *ac*

Valid for: MP/200, MP/100

0	1	2	3–4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	AC	0	1	1	1	0	0	0	0	0	0	1

Pops the top word off the stack and loads it into the specified accumulator. Decrements the stack pointer by 1.

PSHA

Push Accumulator

PSHA *ac*

Valid for: MP/200, MP/100

0	1	1	AC		0	1	1	0	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Increments the stack pointer by 1; then pushes the contents of the specified accumulator onto the stack at the address specified by the stack pointer (the top of the stack). Leaves the contents of the specified accumulator unchanged.

If bits 8-15 of the stack pointer are 00000000 after the stack pointer is incremented, then a 256-word page boundary has been crossed. This will result in a stack overflow, a condition that will be indicated at the end of the instruction.

PSHO

Push Accumulator

PSHO *ac*

Valid for: MP/200

0	1	1	AC		0	1	1	0	1	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Increments the stack pointer by 1; then pushes the contents of the specified accumulator onto the stack at the address specified by the stack pointer (the top of the stack). Leaves the contents of the specified accumulator unchanged. Checks for stack overflow by comparing the contents of the stack pointer with the contents of location 42_8 (the stack limit). If the contents of the stack pointer exceed the contents of the stack limit, then an overflow occurs. If the contents of the stack limit is less than or equal to the contents of the stack limit, no overflow occurs.

RET

Return

RET

Valid for: MP/200, MP/100

0	1	1	0	0	1	0	1	1	0	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pops a return block off the stack.

The following table shows how information popped off the stack is handled during instruction execution. In this table FP refers to the frame pointer before execution of the *RET* instruction.

Word Popped	Stack Address	Destination
1	FP-0	Bit 0 to carry bit: bits 1-15 to PC.
2	FP-1	AC3
3	FP-2	AC2
4	FP-3	AC1
5	FP-4	AC0

The contents of the frame pointer are loaded into the stack pointer.

Bit 0 of the location now addressed (FP-0) is loaded into carry, and bits 1-15 into the program counter.

The stack pointer is decremented. The contents of the addressed location (FP-1) are loaded into the frame pointer.

The stack pointer is decremented a second time. The contents of the addressed location (FP-2) are loaded into AC2.

The stack pointer is decremented a third time. The contents of the addressed location (FP-3) are loaded into AC1.

The stack pointer is decremented a fourth time. The contents of the addressed location (FP-4) are loaded into AC0.

The stack pointer is decremented a fifth time, and now points to the new top of the stack.

At the end of the *Return* instruction, the new contents of AC3 are loaded into the frame pointer.

There are no checks for underflow in the *Return* instruction.

SAV

Save

SAV

Valid for: MP/200, MP/100

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	1

Pushes a return block onto the stack: then places the value of the stack pointer in the frame pointer and AC3. Leaves the contents of accumulators 0, 1, and 2 unchanged. The table below shows the format of the five words of the return block.

Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	Frame pointer before the save
5	Bit 0 = carry bit
	Bits 1-15 = bits 1-15 of AC3

The first time the stack pointer is incremented the contents of AC0 are written into the location addressed by the stack pointer (SP+1).

The second time the stack pointer is incremented the contents of AC1 are written into the location now addressed by the stack pointer (SP+2).

The third time the stack pointer is incremented the contents of AC2 are written into the location now addressed by the stack pointer (SP+3).

The fourth time the stack pointer is incremented a 0 is written into bit 0 and the contents of the frame pointer are written into bits 1-15 of the location now addressed by the stack pointer (SP+4).

The fifth time the stack pointer is incremented the carry is written into bit 0 and bits 1-15 of AC3 are written into bits 1-15 of the location now addressed by the stack pointer (SP+5).

If bits 8-15 of the stack pointer are ever 00000000 after the stack pointer is incremented, then a 256-word page boundary has been crossed. This will result in a stack overflow, a condition that will be indicated at the end of the instruction.

At the end of the instruction the frame pointer and AC3 are loaded with the new contents of the stack pointer. This means that the stack pointer, the frame pointer, and AC3 all point to the last word pushed onto the stack. Words in the return block can be referenced by a negative displacement from AC3.

Use the **SAV** instruction with the **JSR** instruction, which places the return value of the program counter in AC3. **SAV** then pushes the return value (contents of AC3) into bits 1-15 of the fifth word pushed.

SAVE

Save

SAVE*i*

Valid for: MP/200

1	1	1	0	0	1	0	1	0	1	0	0	0	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

IMMEDIATE
0 … 15

Pushes a return block onto the stack: then places the value of the stack pointer in the frame pointer and AC3. Leaves the contents of accumulators 0, 1, and 2 unchanged. The table below shows the format of the five words of the return block.

Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	Frame pointer before the save
5	Bit 0 = carry bit
	Bits 1-15 = bits 1-15 of AC3

The first time the stack pointer is incremented the contents of AC0 are written into the location addressed by the stack pointer (SP+1).

The second time the stack pointer is incremented the contents of AC1 are written into the location now addressed by the stack pointer (SP+2).

The third time the stack pointer is incremented the contents of AC2 are written into the location now addressed by the stack pointer (SP+3).

The fourth time the stack pointer is incremented a 0 is written into bit 0 and the contents of the frame pointer are written into bits 1-15 of the location now addressed by the stack pointer (SP+4).

The fifth time the stack pointer is incremented the carry is written into bit 0 and bits 1-15 of AC3 are written into bits 1-15 of the location now addressed by the stack pointer (SP+5).

The **SAVE** instruction allocates a portion of the stack for use by the procedure which executed the **SAVE.** The value of the frame size determines the number of words in this stack area. This portion of the stack will not normally be accessed by push and pop operations, but will be used by the procedure for temporary storage of variables, counters, etc. The frame pointer acts as the reference point for this storage area. Accumulator 3 is set to the value of the frame pointer.

If the contents of the stack pointer exceed the contents of the stack limit (location 42_8), then a stack overflow occurs.

Use the **SAVE** instruction with the **JSR** instruction, which places the return value of the program counter in AC3. **SAVE** then pushes the return value (contents of AC3) into bits 1-15 of the fifth word pushed.

STA

Store Accumulator

STA *ac,[@]displacement [,index]*

Valid for: MP/200, MP/100

0	1	0	AC	@	INDEX	DISPLACEMENT
0	1	2	3 4	5	6 7	8 15

Stores the contents of an accumulator into a memory location.

Places the contents of the specified accumulator in the word addressed by the effective address, *E*. The previous contents of the location addressed by *E* are overwritten. The contents of the specified accumulator remain unchanged.

STB

Store Byte

STB *acs,acd*

Valid for: MP/200

0	1	1	ACD	1	0	0	ACS	0	0	0	0	0	1
0	1	2	3 4	5	6	7	8 9	10	11	12	13	14	15

Moves the right byte (bits 8-15) of the destination accumulator to a byte in memory. The source accumulator contains a byte pointer to the storage location. The contents of the source and destination accumulators remain unchanged.

SUB

Subtract

SUB*[c][sh][#] acs,acd[,skip]*

Valid for: MP/200, MP/100

1	ACS	ACD	1	0	1	SH	C	#	SKIP
0	1–2	3–4	5	6	7	8–9	10–11	12	13–15

Performs unsigned integer subtraction and complements carry if appropriate.

Sets the value of carry to the specified value. Subtracts the unsigned, 16-bit number in ACS from the unsigned, 16-bit number in ACD by taking the two's complement of the number in ACS and adding it to the number in ACD. If the result is greater than 2^{16}-1 or less than 0, then the value of carry is complemented. Places the 17-bit result (carry and the function result) in the shifter. Performs the specified shift operation and places the result of the shift in carry and ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

TRAP

Trap

TRAP *acs,acd,trap number*

Valid for: MP/200, MP/100

1	ACS	ACD	TRAP NUMBER	1	0	0	0
0	1–2	3–4	5–11	12	13	14	15

Places the address of this instruction in location 46_8. Sets bit 0 of location 46_8 to 0. The instruction then executes a jump indirect to location 47_8. Leaves the Interrupt On flag unchanged.

Chapter 5
I/O Instruction Dictionary

This chapter lists all the I/O instructions for MP/Computers. The instructions are arranged in alphabetical order according to instruction mnemonic.

Each instruction entry includes:

- The mnemonic recognized by the assembler
- The bit format
- The number and format of any arguments
- A description of how the instruction works.

The table below lists optional mnemonics which you may want to add to some of your I/O instructions. Chapter 3 explains these mnemonics in detail; the instruction entries will tell you whether you can use one of these optional mnemonics with an instruction.

Optional Mnem	Sets Bits 8-9 to	Action
---	00	Leaves device's Busy and Done flags unchanged.
S	01	Sets device's Busy flag to 1, sets Done flag to 0, and starts operation.
C	10	Sets device's Busy and Done flags to 0.
P	11	Depends on device.

Table 5.1 Optional I/O mnemonics

DIA

Data In A

DIA *device*

Valid for: MP/200, MP/100

0	1	1	AC		0	0	1	F		DEVICE CODE					
0	1	2	3	4	5	6	7	8	9	10					15

Transfers data from the A buffer of an I/O device to an accumulator.

Places the contents of the A input buffer in the specified device in the specified AC. After the data transfer, sets the Busy and Done flags according to the function specified by *f*.

The number of data bits moved depends upon the device. Bits in the AC that do not receive data are set to 0. If the specified device does not exist, the AC will contain -1 after the instruction executes.

DIB

Data in B

DIB*[f]* *ac,device*

Valid for: MP/200, MP/100

0	1	1	AC		0	1	1	F		DEVICE CODE					
0	1	2	3	4	5	6	7	8	9	10					15

Transfers data from the B buffer of an I/O device to an accumulator.

Places the contents of the B input buffer in the specified device in the specified AC. After the data transfer, sets the Busy and Done flags according to the function specified by *f*.

The number of data bits moved depends upon the device. Bits in the AC that do not receive data are set to 0. If the specified device does not exist, the AC will contain -1 after the instruction executes.

DIC

Data In C

DIC*[f]* *ac,device*

Valid for: MP/200, MP/100

0	1	1	AC	1	0	1	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from the C buffer of an I/O device to an accumulator.

Places the contents of the C input buffer of the specified device in the specified AC. After the data transfer, sets the Busy and Done flags according to the specified *f*.

The number of data bits moved depends upon the device. Bits in the AC that do not receive data are set to 0. If the specified device does not exist, the AC will contain -1 after the instruction executes.

DOA

Data Out A

DOA*[f]* *ac,device*

Valid for: MP/200, MP/100

0	1	1	AC	0	1	0	F	DEVICE CODE
0	1	2	3 4	5	6	7	8 9	10 15

Transfers data from an accumulator to the A buffer of an I/O device.

Places the contents of the specified AC in the A output buffer of the specified device. After the data transfer, sets the Busy and Done flags according to the function specified by *f*. The contents of the specified AC remain unchanged.

The number of data bits moved depends upon the device.

DOB

Data Out B

DOB*[f] ac,device*

Valid for: MP/200, MP/100

0	1	1	AC	1	0	0	F	DEVICE CODE
0	1	2	3–4	5	6	7	8–9	10–15

Transfers data from an accumulator to the B buffer of an I/O device.

Places the contents of the specified AC in the B output buffer of the specified device. After the data transfer, sets the Busy and Done flags according to the function specified by *f*. The contents of the specified AC remain unchanged.

The number of data bits moved depends upon the device.

DOC

Data Out C

DOC*[f] ac,device*

Valid for: MP/200, MP/100

0	1	1	AC	1	1	0	F	DEVICE CODE
0	1	2	3–4	5	6	7	8–9	10–15

Transfers data from an accumulator to the C buffer of an I/O device.

Places the contents of the specified AC in the C output buffer of the specified device. After the data transfer, sets the Busy and Done flags according to the function specified by *f*. The contents of the specified AC remain unchanged.

The number of data bits moved depends upon the device.

HALT

Halt

HALT

Valid for: MP/200, MP/100

DOC *0,CPU*

Valid for: MP/200, MP/100

0	1	1	AC		1	1	0	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the Interrupt On flag to 1 and stops the processor. While stopped the processor will honor data channel requests and program interrupt requests.

If the console debug feature is part of the system, control is transferred to the appropriate console software immediately after execution of a *Halt* instruction. If this feature is not part of the system, the processor remains halted and waits for an interrupt.

INTA

Interrupt Acknowledge

INTA

Valid for: MP/200, MP/100

DIB*[f]* *ac*,CPU

Valid for: MP/200, MP/100

0	1	1	AC		0	1	1	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Returns device code of an interrupting device.

Places in bits 10-15 of the specified accumulator a 6-bit device code. (The device code corresponds to the device requesting an interrupt that has the highest priority on the priority interrupt chain.) Sets bits 0-9 of the specified accumulator to 0. After the transfer, in the **DIB** format only, the instruction sets the Interrupt On flag according to the function specified by *f*.

INTDS

Interrupt Disable

INTDS

Valid for: MP/200, MP/100

NIOC CPU

Valid for: MP/200, MP/100

0	1	1	0	0	0	0	0	1	0	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets Interrupt On flag to 0. The CPU will not honor interrupts while the Interrupt On flag is 0.

INTEN

Interrupt Enable

INTEN

Valid for: MP/200, MP/100

NIOS CPU

Valid for: MP/200, MP/100

0	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets Interrupt On flag to 1. If the instruction changes the state of the Interrupt On flag (i.e., the Interrupt On flag was previously 0), the CPU allows one more instruction to execute before the first I/O interrupt can occur.

IORST

Reset

IORST

Valid for: MP/200, MP/100

DOA*[f] ac*,CPU

Valid for: MP/200, MP/100

0	1	1	AC		0	1	0	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the Busy and Done flags of all I/O devices to 0. Sets the 16-bit priority mask to 0. Sets the ION flag to 0 (**IORST** format), or to the function specified by *f* (**DOA** format).

> **NOTES:** *The assembler recognizes the mnemonic* **IORST** *as equivalent to the instruction* **DOAC 0,CPU**.
>
> *If you use the mnemonic* **DOA**, *during execution the processor ignores the accumulator field; the contents of the accumulator remain unchanged.*
>
> *At power-up and when you press the RESET switch, the processor performs the equivalent of an* **IORST** *instruction.*

MSKO

Mask Out

MSKO *ac*

Valid for: MP/200, MP/100

DOB*[f] ac*,CPU

Valid for: MP/200, MP/100

0	1	1	AC		1	0	0	F		1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the priority mask.

Sets the Interrupt Disable flags according to the mask in the specified accumulator. The masking out takes effect after the *next two* instructions have executed. In the **DOB** format only, the instruction then sets the Interrupt On flag according to the function specified by *f*. The contents of the specified accumulator remain unchanged.

Each I/O device in the system is assigned a given mask bit. If the corresponding bit in the specified accumulator is 1, then the device's Interrupt Disable flag is set to 1. If the corresponding bit in tthe specified accumulator is 0, then the device's Interrupt Disable flag is set to 0. Refer to Appendix C for a list of standard DGC device/mask bit assignments.

The assembler recognizes the instruction **MSKO** *ac* as equivalent to **DOB** *ac*,**CPU**.

NIO

No I/O Transfer

NIO *[f] device*

Valid for: MP/200, MP/100

0	1	1	0	0	0	0	0	F		DEVICE CODE					
0	1	2	3	4	5	6	7	8	9	10					15

Sets the Busy and Done flags in the controller of the specified device according to the function specified by *f*. If you append the mnemonic *P* onto **NIO**, you will generate an I/O pulse. The effect of this pulse depends on the device.

RTCDS

Real-time Clock Disable

RTCDS

Valid for: MP/100

DOA *[f]* ,1, CPU

Valid for: MP/100

Disables the real-time clock. In the **DOA** format only, the instruction also sets the Interrupt On flag according to the function specified by *f*.

RTCEN

Real-time Clock Enable

RTCEN

Valid for: MP/100

DOA *[f]* ,2, CPU

Valid for: MP/100

Enables the real-time clock. In the **DOA** format only, the instruction then sets the Interrupt On flag according to the function specified by *f*.

SKP

I/O Skip

SKP*t device*

Valid for: MP/200, MP/100

0	1	1	0	0	1	1	1	T	DEVICE CODE
0	1	2	3	4	5	6	7	8 9	10 15

If the test condition specified by *t* is true, the instruction skips the next sequential word. The possible values of *t* are listed below.

Mnemonic	Value	Test
BN	00	Test Busy flag for nonzero.
BZ	01	Test Busy flag for zero.
DN	10	Test Done flag for nonzero.
DZ	11	Test Done flag for zero.

SKP CPU

CPU Skip

SKP*t* CPU

Valid for: MP/200, MP/100

0	1	2	3	4	5	6	7	8-9	10	11	12	13	14	15
0	1	1	0	0	1	1	1	T	1	1	1	1	1	1

Skips the next sequential word if the test condition specified by *t* is true. The possible test conditions are:

Mnem	Sets Bits 8-9 to	Action
BN	00	Skip if interrupts are disabled.
BZ	01	Skip if interrupts are enabled.

Table 5.2 CPU Skip test conditions

Part 2
The Macroassembler

Chapter 1
Introduction to the Macroassembler

The Macroassembler translates instruction codes and symbolic addresses into numeric codes and addresses. When you input symbolic language, the Macroassembler processes it into absolute or relocatable code and outputs an object file or object module. The computer cannot execute your source program, nor can it execute an object file. An object file must be processed by the Binder into a program file before the computer can execute it.

Macroassembler input

The source program you input to the Macroassembler must be made up of ASCII characters. The Macroassembler interprets the characters in two steps called *passes*. On each pass over the source file the Macroassembler

- Reads a line consisting of a character string terminated by a New-line, Carriage Return, Form Feed or End of File.
- Ignores all null characters.

Macroassembler output

After processing your source file, the Macroassembler will produce some form of output. The Macroassembler can produce four types of output:

- An object file
- An error listing
- A program listing
- A permanent symbol file.

Object File

To produce an object file, the Macroassembler begins by building *atoms* out of the characters in the source file. Atoms are syntactically recognizable numbers, symbols, or terminals. The Macroassembler then acts upon each atom.

Next the Macroassembler translates the atoms built from the source code lines into binary code. Most of the lines translate into 16-bit binary numbers. The Macroassembler gives each binary number an address (which may be relocatable or absolute). The Macroassembler also includes information about these addresses as part of the object file. Later on, the binder will need this information to convert the object file into a program file.

You do not have to output an object file. Unless there are errors in your source file, the Macroassembler will normally output one. If you want to produce an object file unconditionally, you can add the **/R** function switch to your **MASM** command line. If you do not want to produce an object file, add the **/N** function switch to your **MASM** command line.

Error Listing

The error listing shows the title of the source file and lists all lines which were flagged by errors. It does not contain any information in addition to that found in the assembly listing, but it provides you with a short summary of all problem areas.

Program Listing

The information in a program listing shows you how the macroassembler interpreted your input. The listing consists of lines, and the information on each line is separated into various fields. Table

1.1 shows the various fields and the information contained in each one.

Columns	Information Contained
1-3	Up to three error codes if the line contains errors. The first error flag appears in column 3, the second in column 2, and the third in column 1. If the line contains more than three errors, the error flags do not appear here but are included in the total error count. If the line contains no errors, these columns contain a two-digit line number followed by a blank space.
4-8	The location counter (if applicable). Otherwise these columns are left blank.
9	The relocation flag pertaining to location counter.
10-15	Contain the data field; an instruction, an equivalence value, an expression, or a pseudo op argument. Otherwise these columns are left blank.
16	The relocation flag pertaining to the data field.
17...	The source line as written and expanded by macro calls.

Table 1.1 Program Listing Fields

The relocation mode of the address, which appears in column 9, can take on any of the values shown in Table 1.2.

Flag	Meaning
(space)	Absolute
-	Page zero relocatable
'	Impure code
!	Pure code

Table 1.2 Address relocation mode symbols

The relocation mode of the value, which appears in column 16, can take on any of the values shown in Table 1.3.

Flag	Meaning
(space)	Absolute
-	Page zero relocatable
'	Impure code, word relocatable
!	Pure code, word relocatable
"	Impure code, byte relocatable
&	Pure code, byte relocatable
$	Displacement field is externally defined

Table 1.3 Value relocation mode symbols

A program listing always includes a cross-reference listing of the symbols used. This reference normally lists only the user symbols you define within the source file. By using the /P switch in the **MASM** command line, you can include semipermanent symbols in the cross-reference listing.

The cross-reference listing shows the value of a symbol as well as the page and line of the program in which a symbol appears. For example, if the symbol EL1 has the value 73_8 and appears in the first page, seventh line of your program listing, then the cross-reference listing will show 000073 to the right of the symbol EL1, followed by the page/line indicator, 1/07.

In addition to the information described above, the cross-reference listing also identifies the page and line on which a symbol was defined. The Macroassembler signals the defining location by placing a # sign after the appropriate page/line indicator.

The cross-reference listing includes several assignment symbols. Table 1.4 lists these symbols and their meanings.

Symbol	Meaning
(spaces)	Local symbol
EN	Entry (defined in **.ENT** statement)
EO	Overlay entry (defined in **.ENTO** statement)
XD	External displacement (defined in **.EXTD** statement)
XN	External normal (defined in **.EXTN** statement)
NC	Named common (defined in **.COMM** statement)
MC	Macro name

Table 1.4 Assignment symbols

An example of a typical program listing and cross-reference listing is shown in Figure 1.1. The cross reference listing will appear on a separate page from the assembly listing.

```
0001 LOGIT MICRON ASSEMBLER REV 01.00
01
                              .TITLE   LOGIT        ; Write to log file
03
04                            .ENT     .LOGIT
05
06                            .EXTD    LOGCH
07                            .EXTD    PRINT
08
09                            EXTN     SYSER
10
11
12                    ;
13                    ; LOGIT   - Send consersational I/O to the log file
14                    ;           Calling sequence:
15                    ;
16                    ;                        AC1 - Byte pointer to buffer
17                    ;                        AC2 - Maximum byte count
18                    ;
19                    ;
20                    ;                        JSR    @.LOGIT
21                    ;                        <return>
22                    ;
23                    ;                        AC3 - Frame pointer
24                    ;
25
26                            .ZREL
27
28  00000-000000!   .LOGIT:   LOGIT                 ; Subroutine address
29
30
31       000001               .NREL    1
32
33
34  00000!062401    LOGIT:    SAV                   ; Save caller's state
35
36                            See if the log file is turned on
37
38  00001!020000$.            LDA      0,PRINT      ; Get print flag
39                            SNEZ     0            ; Is it set?
40  00003!062601              RET                   ; No - don't log anything
41
42                            See if we have a log file
43
44  00004!020000$             LDA      0,LOGCH      ; Log file channel #
45                            SNEM1    0            ; Is channel open?
46  00006!062601              RET                   ; No - just return
47
48                            Write out the record
49
50                            ?WRITE   DS           ; Write it out
51  00011!006402              JSR      @=SYSER
52
53  00012!062601              RET                   ; Return to caller
54
55
56                            .END                  ; LOGIT.SR
57  00013!000000$

**00000 TOTAL ERRORS, 00000 PASS 1 ERRORS


0002 LOGIT

COM          100000          1/46
JSR          004000          1/51
LDA          020000          1/38      1/44
LOGCH        000003$ XD      1/06      1/44
LOGIT        000000!         1/28      1/34#
MOV          101000          1/40
PRINT        000002$ XD      1/07      1/38
RET          062601          1/40      1/46     1/53
SAV          062401          1/34
SNEM1        000067! MC      1/45
SNEZ         000005! MC      1/39
SNR          000005          1/40      1/46
SYSER        000004 XN       1/09      1/51
U?NPE        000025          1/51
W?RIT        000004          1/51
.LOGI        000000- EN      1/04      1/28#
?DS          002000          1/51
?I           000004          1/51#
?J           002000          1/51#
?K           000002          1/51#
?SYSE        000001$ XD      1/51
?WRIT        002245! MC      1/50
```

DG-05052

Figure 1.1 A typical program listing

Chapter 2
Character Input and Atoms

Introduction

The Macroassembler interprets character strings in two modes: string mode and normal mode. In string mode, the Macroassembler accepts character strings literally. In normal mode, the Macroassembler interprets character strings as series of *atoms* which may have symbolic interpretations. The Macroassembler interprets most input as normal mode input.

An atom is a group of characters that makes up a symbol or number that the Macroassembler can recognize. These groups of characters can represent digits, space characters, symbol names, variables, operations such as addition, relational operators such as "less than", etc.

This chapter describes character input modes, as well as many types of atoms.

Character Input

String Mode

Any ASCII character or group of characters can form a string mode character string. To distinguish such input from normal mode input, you must precede and end string mode input with certain psuedo ops or symbols. There are several different ways to do this.

Comments

A comment begins with a semicolon. The character string which forms the comment is followed by a standard line delimiter (usually a New-line character). An example of a comment is:

```
;This is a comment
```

Macro Definition Strings

A macro definition string begins with the pseudo op **.MACRO** followed by a standard line delimiter (usually a New-line character), and ends with the symbol %. An example of this is:

```
    .MACRO    SAMPLE
    LDA       1,LENGTH
    LDA       2,WIDTH
    MUL
    STA       0,AREA
%
```

Text Strings

A text string begins with a text pseudo op (i.e., **.TXT**, **.TXTE**, **.TXTF**, or **.TXTO**, followed by a space or tab character. Following this is the actual text string enclosed in delimiters. The delimiters here can be any character not used in the text string. Examples of text string lines are:

```
.TXT     /This is a sample text string./
.TXTM    *The width is area/length.*
.TXTE    AThe multiply symbol is *.A
```

Normal Mode

In normal mode, the Macroassembler interprets character strings as groups of certain ASCII characters called atoms. These strings are divided into lines and each line must be followed by a standard delimiter (usually a New-line character). The Macroassembler considers any input that is not in string mode to be in normal mode.

The Macroassembler recognizes the ASCII characters listed in Appendix A as legal elements in normal string input. In general, legal elements can be alphabetics, numerals, relational symbols (see Table 2.1), most conventional punctuation, and certain special characters. During assembly, any character or group of characters not within this subset is flagged with a **B** (bad character) and ignored.

Atoms

Atoms are syntactic units of assembly language and are made up of groups of characters. There are four types of atoms:

- Symbols
- Terminals
- Numbers
- Special atoms

Refer to Chapter 3 for a discussion of symbols.

Terminals

Terminals are made up of either a single character or a pair of characters. They serve primarily to separate numbers and symbols from other numbers and symbols, such as an addition sign or a space. The addition sign is an example of an operator terminal, and the space is an example of a break character terminal.

Operator Terminals

Operator terminals are shift, arithmetic, logical, or relational symbols you use with integers and symbols to form expressions. Table 2.1 lists all operator symbols and their meaning.

Type	Symbol	Meaning
Shift	B	Bit alignment (shift bits)
Arithmetic	+	Addition
Arithmetic	-	Subtraction
Arithmetic	*	Multiplication
Arithmetic	/	Division
Logical	&	Logical AND
Logical	!	Logical OR
Relational	==	Equal
Relational	>=	Greater than or equal
Relational	>	Greater than
Relational	<	Less than
Relational	<=	Less than or equal
Relational	<>	Not equal

Table 2.1 Operator terminals

> **NOTE:** *The Macroassembler distinguishes between the bit shift operator and the ordinary ASCII B in three instances. If you precede the B with an integer, a right parenthesis, or an underscore (i.e.,* **13B,)B,** *or* **_B***), the Macroassembler will interpret this as a bit shift operator.*

Break terminals

This class of terminals serves primarily as separators. Table 2.2 lists all members of this class.

Symbol	Description
□	The class of spaces. Includes a space, comma, tab, or any combination of these.
=	Defines the symbol preceding the = sign.
:	Defines the symbol preceding the : sign.
()	May enclose a symbol or expression.
[]	May enclose the arguments of a macro call or a label used for conditional assembly.
;	Indicates beginning of a comment string.
<↓>	New line character - terminates a line of source code.

Table 2.2 Break terminals

Numbers

The Macroassembler accepts three types of number atoms:

- Single precision integer, stored in one word
- Double precision integer, stored in two words
- Single precision floating point constant, stored in two words

Single precision integers may appear in expressions and in data statements. Double precision integers and floating point constants may appear only in data statements.

Single Precision Integers

Single precision integers are represented as single 16-bit words. You can represent any unsigned integer in the range of 0 to 65535_{10}. Using two's complement representation, you can represent any signed integer in the range of -32,768 to +32,767. See *Assembly Language Reference*, Chapter 2, for a discussion of two's complement representation.

Single precision integers are represented using the format shown below.

[sign] d...d [.] break

where

sign is an optional + or -,

d is a digit in the range of the current input radix,

[.] is an optional decimal point, and

break is a digit outside the current input radix, any character, or a period (.).

Note that the first digit must be in the range 0-9.

If a decimal point precedes the break character, the Macroassembler will evaluate the integer as decimal. If you omit the decimal point, the Macroassembler evaluates the integer in the current input radix. You can set the input radix to be any number between 2 and 20 (see the pseudo op **.RDX**).

When you select a radix of 11 or more, you will, from time to time, use integers containing letters to represent digits, such as B93D. You must precede all integers using a letter as the first digit with a zero, so that the macroassembler can distinguish them from character strings. Some examples of this are:

```
0A45
0E333
0J7
```

You can end an integer with any of the operators listed in Table 2.1, or with one of these four terminals: □,), ;, or <↓>.

However, the bit shift operator, **B**, is an exception. If you are using a radix of 12 or greater, the Macroassembler will ordinarily interpret **B** as a digit. You must precede **B** with an underscore _ for the Macroassembler to interpret **B** as a bit shift operator. The Macroassembler uses the underscore as a sign of where to break the number string, but otherwise ignores it. The example below shows how to use the underscore.

049B13	Macroassembler interprets **B** as a digit
04_B13	Macroassembler interprets **B** as a bit shift operator

Special Integer-Generating Formats

You can use two special integer-generating formats wherever you would use integers. The first of these converts a single ASCII character to its 7-bit octal value. The input format is:

"char

where

char represents any ASCII character except null (000).

Note that the Macroassembler interprets only the character immediately following the quotation mark in this format. Some examples of this format are shown below.

"5 is interpreted as 65_8
"A is interpreted as 101_8
"% is interpreted as 45_8

You can also use this format as part of expressions. The examples below illustrate this. The numbers in the examples are octal values.

"A+4 is interpreted as 101_8+4
"C*5 is interpreted as 103_8*5
"#-"% is interpreted as 43_8-45_8

The Macroassembler packs this format in memory as shown in the following:

"**A** (i.e., 101_8) is packed as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1

"**5** (i.e., 65_8) is packed as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1

The second special integer-generating format converts up to two ASCII characters to a 16-bit integer. The format is:

'string'

or

'string<↓>

where

' is an apostrophe and *string* consists of any number of ASCII characters. Note that the Macroassembler uses only the first two characters in string to form the integer value. Also, two apostrophes without an intervening character will generate a word containing absolute (as opposed to relocatable) zero.

The Macroassembler packs characters in this format left to right in a word as shown below.

'CE' (i.e., 103_8, 105_8) is packed as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	1	1	0	1	0	0	0	1	0	1

'M<↓> (i.e., 115_8) is packed as:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0

Double Precision Integers

A double precision integer is represented in two contiguous words; the first word is the high order word. Using two's complement notation, a double precision integer is represented as folllows

0	1–15
S	

0–15

where Bit 0 of the high order word is the sign bit.

Double precision integers are represented using the format shown below:

{*sign*} *dd...d* {.} **D** *break*

where

{*sign*} is an optional + or -,

d is a digit in the range of the current input radix,

{.} is an optional decimal point,

D indicates a double precision integer, and

break is a digit outside the current input radix, any terminal character, or a period(.).

Note that the first digit must be in the range 0-9. Also, an operator may not terminate a double precision integer, or a format error (F) will result.

The radix of a double precision integer may be in the range 2 - 20. If the radix is greater than or equal to 14, the letter **D** will be interpreted as a digit. To force the Macroassembler to interpret **D** as indicating double precision, use the notation _**D**. For example,

```
            .RDX 16
000455      12D         ;D REPRESENTS DIGIT
                        ;13 (DECIMAL)
000000      12_D        ;D SIGNALS THAT 12 IS
000022                  ;A DOUBLE PRECISION INTEGER
```

Some assembled data statements containing double precision integers follow:

```
            .RDX 8
000000      1D
000001
177777      -1D
177777
000001      200000D
000000
000003      262147.D
000003
000001      100000.D
103240
```

Single Precision Floating Point Constants

A single precision floating point constant is represented in two contiguous words with the format:

S	INT CHAR	MANTISSA
0	1 – 7	8 – 15

MANTISSA
0 – 15

Bit 0 of the high order word is the sign bit which is set to zero for positive numbers and set to 1 for negative numbers.

The integer characteristic (*int char*) is the integer exponent of 16, expressed in excess-64_{10} (100_8) notation. Exponents from -64 to +63 are represented by the binary equivalents of 0 to 127_{10} (0 to 177_8). Zero exponent is represented as 100_8.

The mantissa is represented as a 24-bit binary fraction. It can be viewed as six 4-bit hexadecimal digits. The range of the mantissa's magnitude is:

$16^{-1} <= mantissa <= 1\text{-}16^{-6}$

The negative form of a number is obtained by complementing Bit 0. The characteristic and mantissa remain the same.

The range of magnitude of a floating point constant is:

$16^{-1} * 16^{-64} <=$
floating point number $<=$
$1 - 16^{-6} * 16^{63}$

which is approximately

$5.4 * 10^{-79} <=$
floating point number $<=$
$7.2 * 10^{75}$

Most routines that process floating point numbers assume that all non-zero operands are normalized, and they normalize a non-zero result. A floating point number is considered normalized if the fraction is greater than or equal to 1/16 and less than 1. In other words, it has a 1 in the first four bits of the high order word. All floating point conversions by the Macroassembler are normalized.

Much of the floating point constant format is optional. The minimum format is one digit, followed by either a decimal point or the letter **E** (exponent), followed by another digit, as follows:

d {.} *d break*

or

d {*E*} *d break*

where *d* is a digit in the range 0-9.

A single precision floating point constant is represented in source format as:

{*sign*}*d*{*d...d*}.*d*{*d...d*}{**E**{*sign*}*d*{*d*}}*break*

or

{*sign*}*d*{*d...d*}**E**{*sign*}*d*{*d*}*break*

where

each *d* is a digit in the range 0 - 9;

the mantissa and exponent are always converted in the decimal;

one or more digits may represent an exponent following the letter **E**; and

break is typically one of the terminals ☐ or ; or ↓.

You can format the floating point number with the letter **E**, the decimal point, or both, as shown below.

```
141376    254.33
052172
141376    254.33E0
052172
141376    25433E-02
052172
141376    25433E-2
052172
141376    2543.3E-1
052172
```

If the current radix is 15 or greater, the letter **E** can cause the Macroassembler to interpret the number preceding the **E** as an integer in the current radix rather than as a floating point number. To avoid this ambiguity, use the _**E** notation:

```
000020   .RDX    16
155035   -25E3             ;E is hexadecimal
                           ;14
142141   -25_E3            ;E indicates floating
                           ;point
124000
```

Examples of floating point constants in source statements, with the resulting stored values, are shown below.

```
040420   1.0
000000
040462   3.1415926
041766
140420   -1EO
000000
040200   +5.OE-1
000000
041421   +273.OEO
010000
```

Examples of Numbers

Some additional source program numbers and their assembled values follow.

```
000020   .RDX 16
053175   567D        ;Hexadecimal single precision
                     ;integer
000000   576_D       ;Hexadecimal double precision
002547               ;integer
001067   567.        ;Decimal single precision
                     ;integer
000000   567._D      ;Decimal double precision
001067               ;integer
002547   567         ;Hexadecimal single precision
                     ;integer
05316    567_B14     ;Hexadecimal single precision
                     ;integer, bit shifted one bit
012634   567_B13     ;Hexadecimal single precision
                     ;integer, bit shifted two bits
042026   567_E1      ;Decimal floating point
023000               ;constant
```

Special Atoms

Three atoms are transparent during an assembly line scan. These atoms, @, #, and **, affect a line only after it has been scanned.

At Sign (@)

When an @ sign appears in either a memory reference instruction word or an expression or data word, the Macroassembler will set the indirect bit of the word to 1. In many instructions, the indirect bit is bit 5. In this case the @ sign may appear anywhere in the instruction. In the expression or data word format, the indirect bit is bit 0. In this case the @ sign must appear before the data word, or before or within the expression. Examples of this are shown below.

```
LDA     1,34     assembles to    024034
LDA     1,@34    assembles to    026034
STA   @ 0,20     assembles to    042020
STA     3,@111   assembles to    056111
        25       assembles to    000025
        @25      assembles to    100025
```

Number Sign

The # sign may appear in ALC instructions. After the rest of the instruction has been evaluated, a # sign appearing anywhere in the instruction causes the macroassembler to set the load/no load bit to 1. This bit is bit 12. An example of how to use the # sign is shown below.

```
ADDL    1,2  SZC   assembles as   133102
ADDL #  1,2  SZC   assembles as   133112
```

Asterisks (**)

Two consecutive asterisks anywhere in a source program line (except the comment field) will suppress the listing of that line. The example below illustrates this.

```
        LDA    0,0,2        ;Source program
        LDA    0,0,3**
        LDA    0,0,3
        .END
00000   LDA    0,0,2        ;Listing
00002   LDA    0,0,3
        .END
```

Note that the location counter in the listing (the leftmost column) jumps from 0 to 2 since the Macroassembler assembled all lines of the source program but did not list the second statement.

The Macroassembler would not list any of the statements given as examples below.

```
**.NOMAC 1
.NOMAC   0**
.MACRO   X**
.NOLOC   ZZ**
```

Chapter 3
Syntax

Introduction

Expressions are made up of operands separated by operators, such as A+4, 3<5, C&S, or 4+2-P*3. An expression can represent many things, such as a value, a memory address, or a counter. Associated with every expression is a relocation property.

Symbols can be permanent symbols, semipermanent symbols, or user-defined. Permanent symbols are pseudo ops, which you can include in your programs to provide information to the Macroassembler. Semipermanent symbols are instruction mnemonics, which usually make up the major portion of your programs. User-defined symbols allow you to name elements of your program, and to define your own instructions to perform specialized tasks.

Instructions consist of an instruction mnemonic, which can be followed with argument fields. They allow you to perform some operation, such as adding two numbers or moving data from one place to another.

This chapter describes the elements of syntax the Macroassembler uses to recognize expressions, symbols and instructions. A discussion of relocation properties of expressions is also included.

Expressions

An expression has the general format

[()][sign]operand$_1$*[operator operand*$_2$*]...[)]*

where

operand$_1$ and *operand*$_2$ are integers, or symbols or expressions which evaluate to integers,

sign is an optional + or -, and

Operator is any Macroassembler operator (see Table 2.1).

Each operator must be preceded by an operand, unless you are using the unary operators + and -. In these two cases, you may place a unary operator before an expression.

When more than one operator appears in an expression, the Macroassembler evaluates them in order of their priority. Table 3.1 lists the priorities of all operands.

Note that all relational operators have the same priority, and all arithmetic operators have the same priority.

Operator	Priority level
B	3 (highest priority)
+	
-	
*	
/	
&	
!	2
==	
<>	
<=	1 (lowest priority)
>=	
<	
>	

Table 3.1 Priority levels of operators

If you use two operators of the same priority in an expression, the Macroassembler evaluates them left to right. You can use parentheses to alter the normal sequence of evaluation; the expression within the parentheses will be evaluated first.

An expression containing one of the operators ==,<> , <=,>=,>, or < is a relational expression. It will evaluate to either absolute zero (false) or absolute 1 (true). The term *absolute* means that these values are not relocatable.

Table 3.2 shows some examples of expressions and their values. In the examples, *A* has the value 3, *B* is 5, *C* is 13, and *D* is 10. These are octal values.

Expression	Value
A+B	10
C+B+D/A	10
B+A/D	1
C/(D+A)	1
A!C	13
B&C!D	11
B&(C!D)	1
A<>B	1
C<=(-B)+4	0
A-B+(D!C!B)	15
D==(C-A)	1

Table 3.2 Expressions

Relocation Properties of Expressions

Any expression you write will have a relocation property as well as a value. This property depends upon the relocation properties of the operands (which can be addresses, user symbols, literals, labels, etc.) within the expression. Your expressions will have one of the seven different relocation properties listed below:

Absolute addresses are not to be modified or relocated at bind time. Numeric constants have absolute relocation.

Page zero (ZREL) *word-relocatable* addresses will be relocated or translated at bind time to an absolute address in the range of $50\text{-}377_8$.

Page zero (ZREL) *byte-relocatable* addresses are byte addresses. These addresses will be relocated or translated at bind time to an absolute byte address in the range of $120\text{-}777_8$ (word addresses $50\text{-}377_8$).

Impure code (NREL) *word-relocatable* addresses are relocated or translated at bind time to an absolute address in impure NREL memory.

Impure code (NREL) *byte-relocatable* addresses are byte addresses. These addresses will be relocated or translated at bind time to an absolute byte address in impure NREL memory.

Pure code (NREL) *word-relocatable* addresses are relocated or translated at bind time to an absolute address in pure NREL memory.

Pure code (NREL) *byte-relocatable* addresses are byte addresses. These addresses are relocated or translated at bind time to an absolute byte address in pure NREL memory.

A symbol, literal, etc., does not need to represent an address in order to have a relocation property. The relocation property is necessary because an address may be part of an expression with other operands (which may or may not be addresses). The relocation property of the whole expression depends upon the relocation properties of its operands.

Since an expression can be used as an address or as an operand in another expression, it must have a valid relocation property. Therefore certain restrictions are placed on combinations of operands and operators. In general, operands of different relocation types may not appear in the same expression. There are two exceptions to this rule:

- An operand with absolute relocation may usually be used with an operand of any other relocation property.
- Any two operands, regardless of relocation property, may be compared using a relational operator.

Table 3.3 shows which combinations of operands and operators are legal, and what relocation type the result has. In the table,

a represents an operand with absolute relocation.

r represents an operand with word relocation.

t represents an operand with byte relocation.

k represents an operand with a value of 1 or 2 having absolute relocation.

& represents logical AND.

! represents logical OR.

B represents the bit shift operator.

Expression	Relocation Property
aBa	a
aBr	Illegal
rBa	Illegal
a+a	a
a+r	r
r+r	t
nr+mr	(n+m)r (see Note)
a-a	a
r-a	r
a-r	-1r (see Note)
r-r	a
nr-mr	(n-m)r (see Note)
a*a	a
a*r	ar (see Note)
r*r	Illegal
a/a	a
kr/k	(k/k)r (see Note)
a/r	Illegal
a&a	a
r&r	Illegal
a&r	Illegal
a!a	a
a!r	Illegal
r!r	Illegal

Table 3.3 Relocation properties of expressions

> **NOTE:** *This is an illegal value if there are no other operator-operand groups in the expression. In order for the expression containing this operator-operand group to have a legal relocation value, it must contain other terms which resolve the relocation property to a value of* a, r, *or* t. *(See example below.)*
>
> *((nr+mr)*a - (nr+mr-(a+a+ a))*a)*
> *((nr+mr-nr-mr)+a+a+a) *a*
> *((a+a)+a)*a*
> *(a+a)*a*
> *a*a*
> *a*

Expressions with operands of the same relocation property and a relational operator will have a value of either absolute 0 (false) or absolute 1 (true). Expressions with operands of different relocation properties and a relational operator will always evaluate to absolute 0 (except when you use the operator <> , which will evaluate to absolute 1).

The Macroassembler flags all expressions with an illegal relocation type as relocation errors R.

Symbols

Symbol names are composed of one or more letters, digits, . (period), ?, or _ (underscore). The first character in the name cannot be a digit. Only the first five characters of a symbol name are significant.

If an underscore is to be used in a symbol name, it must be preceded by another underscore to distinguish it from the special meaning of the underscore in macros.

The Macroassembler recognizes three classes of symbols. These are:

- Permanent
- Semipermanent
- User

Permanent Symbols

These symbols are defined within the Macroassembler, so you cannot alter them in any way. The pseudo ops make up the biggest class of permanent symbols and they direct the assembly process. Others, such as **.PASS** and **.MCALL**, represent numerical values of internal assembly variables. Chapters 4 and 5 discuss permanent symbols in detail.

The Macroassembler interprets a symbol as being a pseudo op or a value by the symbol's position in a line. If a symbol is the first atom in a line, then the Macroassembler interprets it as a pseudo op. Otherwise, a symbol represents some value. The example below shows the difference.

```
.TXTM      1          ; TXTM represents a pseudo op.
(.TXTM)               ; TXTM represents a word which
                      ; contains the value of the last
                      ; expression you used to set
                      ; the text mode.
+.TXTM+3              ; This expression represents a
                      ; word containing absolute 4.
```

Semipermanent Symbols

This class of symbols consists of instruction mnemonics. Within their definitions these symbols carry syntax information for the Macroassembler. From this information the Macroassembler can tell how many fields to expect with a mnemonic, what range of values to expect, whether to perform some task associated with the main purpose of the instruction, etc. You can use any of the semipermanent symbols described in the assembly language portion of this manual, or you can define your own symbols with pseudo ops.

You can save and use semipermanent symbols without redefining them for each assembly by using the /S function switch in your **MASM** command line. To list semipermanent symbols in your cross reference listing, you must use the /**P** function switch in your **MASM** command line.

User Symbols

You can define user symbols for many purposes: to name a location, to name some external value, to define some external value, etc. You must make sure that symbols of this class do not conflict with permanent or semipermanent symbols. For example, you cannot define a user symbol with the name **.TXT**, or **LDA**, because these are already defined. During assembly, the Macroassembler maintains your user symbols in a symbol table.

User symbols can be either local or global symbols. Each local symbol has a value which is known only during the single assembly in which it was defined. All global symbol values are known at bind time. This means you can use them in one module to reference data defined in another module.

Instructions

An instruction is made up of a semipermanent symbol (the instruction mnemonic) followed by zero or more fields. It is either 16 or 32 bits long (note that you cannot execute 32-bit instructions on an MP/Computer). There are many types of instructions, each with its own defining pseudo op and set of instruction mnemonics. Table 3.4 lists all the types of instructions and the pseudo op you must use to define an instruction of that type. For a discussion of all but the last three of these instruction types refer to *Assembly Language Reference*. See chapters 5 and 6 of this section for a description of the pseudo ops.

Instr Type	Instr Subtype	Pseudo Op
ALC	Arithmetic and logic class (ALC)	**.DALC**
Accumulator (AC)	2 AC instructions, no skip 2 AC instructions, skip allowed 2 AC instructions, no skip	**.DISD** **.DISS** **.DTAC**
Memory Reference (MRI)	MRI MRI with AC Extended MRI Extended MRI with AC Commercial MRI	**.DMR** **.DMRA** **.DEMR** **.DERA** **.DCMR**
I/O	I/O without AC I/O with AC I/O without device code	**.DIO** **.DIOA** **.DIAC**
Floating Point (FP)	FP with two required arguments	**.DFLM**
	FP with one required argument	**.DFLS**
Immediate	Immediate Extended immediate	**.DICD** **.DIMM**
XOP	Extended operation Extended without arguments	**.DXOP** **.DEUR**
---	Define user symbol as semipermanent without arguments	**.DUSR**

Table 3.4 Instruction types and defining pseudo ops

Chapter 4
Macros, Literals, Labels, and Symbols

Introduction

The macro facility allows you to write a number of source lines, give those lines a name, and subsequently use that name to reference the source lines. The source lines you write define a macro. The name you give the source lines is a symbol which represents the source lines. After you define the macro, you can use the macro name in your program; during assembly, the Macroassembler expands the macro name to the original source lines.

Literal references allow you to define a value without concerning yourself about the value's location in memory. A literal can be used in the displacement field of an instruction.

There are two formats available that allow you to generate numbers and labels in your programs. These formats can be useful when defining tables, or when defining macros that will be used many times in a program.

This chapter describes the macro facility and some of its applications, as well as literal representation and generated numbers and labels.

Macros

Defining a Macro

Each macro definition has the form

.MACRO□*usym*
source line
.
.
.
source line
%

where

usym is the name you will use to identify the macro,

source lines are strings of ASCII characters to be substituted for the macro call,

% ends the macro definition. (It is not part of the definition.)

You define a macro only once, but you may invoke it as often as you need it.

Note that you must follow *usym* with a New-line character to distinguish it from the macro definition. That is, you must clearly separate the macro name from the macro definition.

In most cases, you will write macro definitions in the form of one or more lines. You place the % immediately after the last line of the macro definition as the first character of the next line. You can also define a macro as zero or more lines followed by part of another line. You do this by placing the % immediately after the last character in the partial line. Any characters following the % will not be part of the macro definition.

```
.MACRO    Z         ;This macro definition
STA       0,SV%     ;is made up of a partial
                    ;line.
.MACRO    W         ;This macro definition
```

```
        STA         0,SVO       ;is made up of two full
        STA         1,SV1       ;lines.
%
        .MACRO      R           ;This macro definition
        STA         2,SV2       ;is made up of a full
        STA         3,SV3%      ;line and a partial line.
```

If you define a macro and then decide your definition is not complete, you can append new source lines to the old lines. There are two conditions: you cannot insert new source lines in the middle of the old source lines; also, you must complete one macro definition before going on to another. You cannot define macro A, then define macro B, then append lines to the definition of macro A.

When you do define a macro in stages, the Macroassembler appends the second and subsequent stages in order of their appearance to the first part of the definition. For example:

```
        .MACRO      SMPL
        I=0
%
        .MACRO      SMPL
        J=0
        K=I+J
%
```

is equivalent to

```
        .MACRO      SMPL
        I=0
        J=0
        K=I+J
%
```

Special Symbols

Within the macro definition two symbols have special meanings. When the Macroassembler recognizes an underscore (_) in a macro it stores the next character without interpretation. The Macroassembler otherwise ignores the underscore. The underscore is convenient when you want to use a character the Macroassembler would generally interpret in some other way. You can use the underscore with any ASCII character.

If you want to include the line

```
ABC is 15% of D
```

in a macro, you must place an underscore before the % sign (since the Macroassembler usually interprets % as the end of the macro definition). The correct form of the line would be:

```
ABC is 15_% of D
```

For a macro source line containing a symbol such as

```
X_AB
```

you would have to use the format

```
X__AB
```

Suppose you need to use different values each time you call a particular macro. You can do this by including a number of *formal arguments* in your macro definition. Each time you call the macro, you specify in the call a number of *actual arguments*. The actual arguments will replace the formal arguments as each macro call expands the macro name.

Usually you would specify in your macro call the same number of actual arguments as there are formal arguments. However, if you specify too many actual arguments, the Macroassembler ignores the extra actual arguments. If you specify too few actual arguments, the Macroassembler replaces the leftover formal arguments with null strings.

The other special character, the uparrow (↑), allows you to specify the formal arguments in your macro. An uparrow followed by either an alphanumeric character (↑ *a*) or a question mark and an alphanumeric character (↑?*a*) specifies an argument. To specify arguments, use the following forms:

↑ *n* where *n* is a digit between 1 and 9,

↑ *a* where *a* is a single letter from A to Z (upper or lower case),

↑?*a* where *a* is a single character from the following set: A-Z, 0-9, and ?.

A digit following ↑ represents the position of an actual argument in the argument list of the macro call. The argument in position *n* will replace formal argument *n* wherever ↑ *n* appears within the macro definition. For example, if ↑ **1** appears in a macro definition, then it will be replaced by the first argument specified in the macro call. ↑ **3** will be replaced by the third argument specified in the macro call. If you include ↑ **0** in your macro, it will be unconditionally replaced by the null string. An example of this type of formal argument is:

```
        .MACRO   A
        ADD      ↑1,↑2      ;The macro definition.
        STA      ↑2,0,↑3
    %
        A        3,1,2      ;The macro call.
        ADD      3,1        ;This Macro expands to.
        STA      1,0,2
```

An *a* or ?*a* following an uparrow is a symbol whose value the Macroassembler looks up when it expands the macro. As with the ↑ *n* format, the value of the symbol indicates which argument will replace it. The value of *a* or ?*a* must be in the range of 0-63, since no macro can have more than 63 arguments. An example of this is:

```
        .MACRO   A
        ADD      ↑C,↑D      ;The macro with formal
                            ;arguments.
        STA      ↑D,0,↑?E   ;C will be replaced by the
                            ;first actual argument,
    %                       ;D by the second,
                            ;?E by the third.
                 C=1        ;C will be replaced by the
                            ;first actual argument.
                 D=2        ;D will be replaced by the
                            ;second.
                 ?E=3       ;?E will be replaced by the
                            third.
        A        1,2,3      ;The macro call.
        ADD      1,2        ;Macro expands to this.
        STA      2,0,3
```

Except for ↑, _, and %, the Macroassembler returns all characters from macro expansion as you wrote them.

Macro Calls

You can call a macro any number of times in a program. Generally, you call a macro by including in your program the macro name followed by a list of arguments. There are three specific forms of macro calls:

USYM

USYM $arg_1...arg_n$

USYM [$arg_1...arg_n$]

where

USYM is the name of the macro, and

arg_n is an actual argument that will replace the appropriate formal argument during macro expansion.

During macro expansion arg_1 replaces every occurrence of ↑ **1** (or ↑ *a* where *a* is the equivalent of the first formal argument) within the macro. Arg_2 replaces every occurrence of ↑ **2** (or the equivalent of the second formal argument) within the macro. In general, arg_n replaces every occurrence of ↑ *n* (or the equivalent of the *n*th formal argument) within the macro.

You use the first form of a macro call for macros which have no formal arguments within their definitions, or for those macros which accept null arguments.

If you are calling a macro which requires arguments, you can use any of the macro forms, although if you use the first form, all formal arguments will be replaced with null strings. Most macro calls require the second or third form. In the second form, a New-line character terminates the argument list; in the third form, a right bracket (]) terminates the argument list. If you have more arguments than you can fit on one line, use the third form. In that form, a New-line character serves only as a delimiter between arguments, just like a space character. You must make sure you do not separate the last argument on a line and the new line character with any commas; the Macroassembler will assume the intervening commas represent other arguments. For example:

```
ABC [1,2<↓>
3,4]
ABC [1,2,<↓>
3,4]
```

In the first example, the New-line character separates the second and third arguments. In the second example, the New-line character separates the third and fourth arguments. In this example, the third argument follows the second comma and is a null argument.

Note that, using the third form, if you follow your argument list with some characters, the Macroassembler will list them after the macro expansion.

Using the third form, if you begin your macro reference with the special atom **, then the Macroassembler will suppress the listing of the first line of the macro call (the line containing the **). Any arguments appearing on other lines, plus any characters following the argument list, will still appear in the listing. If you begin a macro reference line with ** and suppress the listing of the macro expansion using the .NOMAC pseudo op, all lines making up the macro reference, and any characters which follow the argument string, will be suppressed.

Macro definition and reference:

```
                .MACRO   NIUN
                LDA      ↑1,↑2
                LDA      ↑3,↑4
                MOV      ↑5,↑6
                JMP      ↑7
%
                **       NIUN [1,MRI,
                         2,SUON
                         1,3<↓>
                         MAR]
                ;This is the comment string; <↓>
```

Expansion with .NOMAC 0:

```
00000 024000    LDA      1,JIR
00001 024000             MAR] LDA 1,MRI
00002 030000    LDA      2,SHON
00003 135000    MOV      1,3
00004 000000    JMP      MAR
                ;This is the comment string
00005 044000    STA      1,KSRTH
```

Expansion with .NOMAC 1:

```
00006 024000    LDA      1,JIR
                ;This is the comment string
00013 044000    STA      1,KSRTH
```

Listing Macro Expansions

Macro definitions replace macro calls in the object file and in listings of macro expansions. The listing shows both macro calls and macro expansions; the object file, however, contains only the object code for the macro expansions with actual arguments.

```
.MACRO   DSP               ;This is the macro.
↑ 1%
LDA      0,DSP[121],3      ;Source listing line with macro.
LDA      0,DSP[121] 121,3
                           ;Expanded line as it appears in
                           ;the listing file.
LDA      0,121,3           ;Expanded line to be translated
                           ;to object file.
```

You can use the pseudo op .NOMAC to suppress the listing of macro expansions. If you suppress macro expansions, the load instruction in the example above will appear in the listing exactly as it does in the source listing line.

.DO Loops and Conditionals

A .DO psuedo op allows you to assemble a portion of your program a number of times. You type the desired portion of code only once; the Macroassembler will assemble that portion the number of times specified by the .DO pseudo op. An .IF pseudo op allows you to assemble a portion of your program if some condition is met. You type the portion of code in your program, but the macroassembler will only assemble it if the condition specified by the .IF pseudo op is true.

When you use a .DO or .IF pseudo op in your program, you indicate the end of the loop or conditional assembly with an .ENDC pseudo op. In any macro the Macroassembler discards all unterminated .DOs and .IFs when the macro definition ends with %. An example of an incorrect .DO loop is:

```
    .MACRO  F1
    .DO     6
%
```

If you nest .DO loops but do not terminate one of the loops properly, the Macroassembler will assemble the unterminated loop only once, but will assemble the terminated loops the correct number of times.

Macro Examples

The first example is a macro which computes the logical OR of two values. To call this macro, use the form:

OR *acs acd*

where

acs is the source accumulator, and

acd is the destination accumulator.

```
        .MACRO     OR
        COM        ↑1,↑1         ;Complement AC↑1.
        AND        ↑1,↑2         ;Clear "on" bits of AC↑1.
        ADC        ↑1,↑2         ;OR result to AC↑2.
   %
```

FACT is a macro that computes the factorial of a number. The factorial of a number, $n!$, is the value:

$n! = n*(n\text{-}1)*(n\text{-}2)*(n\text{-}3)*...*(2)*(1)$

The macro uses the *recursive* formula

$n! = n*(n\text{-}1)!$

to calculate the factorial value, where n is the input integer. *Recursive* means that the macro calls itself repeatedly.

If the input integer is not 1, the macro cannot calculate the value of the factorial because $(n\text{-}1)!$ is unknown. The macro saves the value of the input integer, decrements it, and uses the decremented value as the new input integer when the macro calls itself to calculate $(n\text{-}1)!$.

If n-1 is not 1, the macro repeats the decrement and call procedure. When the macro calls itself with an input integer of 1, the macro can calculate 1!, return to the next higher level, calculate the next factorial using the saved value of the input integer for this level, return to the next level, and so on, until it calculates $(n\text{-}1)!$ and finally $n!$. The format used for calling this macro is:

FACT *n i*

where

n is the number to be factorialized, and

i will be the factorial of *n*.

The **FACT** macro is shown below.

```
          .MACRO     FACT
          .DO        ↑1==1         ;Is input integer 1?
          ↑2=        1             ;If so, set the initial
                                   ;value of the
          .ENDC                    ;factorial to be 1.
          .DO        ↑1<>1         ;If input integer is
                                   ;not 1, then
          FACT       ↑1-1,↑2       ;decrement it and
                                   ;call FACT again.
                                   ;Macroassembler
                                   ;saves the old value
                                   ;of the input integer
                                   ;for use when the
                                   ;macro returns to
                                   ;this level.
          ↑2=        ↑1*↑2         ;When input integer
                                   ;is 1, the value of
                                   ;the factorial
                                   ;becomes the
                                   current
                                   ;value of the
                                   ;factorial times the
                                   ;value of the
                                   ;input integer at
                                   ;this level.
          .ENDC
%

          FACT       6,I
000000    .DO        6==1
          I=         1
          .ENDC
000001    .DO        6<>1
          FACT       6-1,I
000000    .DO        6-1==1
          I=         1
          .ENDC
000001    .DO        6-1<>1
          FACT       6-1-1,I
000000    .DO        6-1-1==1
          I=         1
          .ENDC
000001    .DO        6-1-1<>1
          FACT       6-1-1-1,I
000000    .DO        6-1-1-1==1
          I=         1
          .ENDC
000001    .DO        6-1-1-1<>1
          FACT       6-1-1-1-1,I
```

```
000000   .DO      6-1-1-1-1==1
         I=       1
         .ENDC
000001   .DO      6-1-1-1-1<>1
         FACT     6-1-1-1-1-1,I
000001   .DO      6-1-1-1-1-1==1
000001   I=       1
         .ENDC
000000   .DO      6-1-1-1-1-1<>1
         FACT     6-1-1-1-1-1-1,I
         I=       6-1-1-1-1-1*I
         .ENDC
000002   I=       6-1-1-1-1*I
         .ENDC
000006   I=       6-1-1-1*I
         .ENDC
000030   I=       6-1-1*I
         .ENDC
000170   I=       6-1*I
         .ENDC
001320   I=       6*I
         .ENDC
         .END
```

The next macro we show allows you to structure your programs with **IF-THEN-ELSE** statements. The macro requires 5 arguments. The first two are accumulators. The third is a condition that allows you to test the two accumulators. The table below describes what values the third argument can have and what test each value specifies.

Value	Test Performed
0	Test if first accumulator > second accumulator
1	Test if first accumulator = second accumulator
2	Test if first accumulator < second accumulator
3	Test if first accumulator <> second accumulator

If the test condition is true, then the macro jumps to the address given as argument 4 (the **THEN** address). If the test condition is not true, the macro jumps to the address given as argument 5 (the **ELSE** address). So the format of the actual macro call would be:

IF ac_1, ac_2, *test*,adr_t, adr_f

where

ac_1 and ac_2 are the two accumulators,

test specifies the test you want to make,

adr_t is the address the macro returns to if the test condition is true, and

adr_f is the address the macro returns to if the test condition is false.

The **IF-THEN-ELSE** macro is shown below.

```
   .MACRO    IF
   .DO       ↑3==0         ;This is the GREATER
                           ;THAN routine.
   SUBZ#     ↑1,↑2,SNC     ; If ↑1>↑2 then go to
                           ;THEN routine.
   JMP       ↑5            ; Else go to ELSE routine.
   JMP       ↑4
   .ENDC
   .DO       ↑3==1         ;This is the equal routine.
   SUB#      ↑1,↑2,SZR     ; If ↑1=↑2 then go to
                           ;THEN routine.
   JMP       ↑5            ; Else go to ELSE routine.
   JMP       ↑4
   .ENDC
   .DO       ↑3==2         ;This is the LESS THAN
                           ;routine.
   ADCZ#     ↑1,↑2,SNC     ; If ↑1<↑2 then go to
                           ;THEN routine.
   JMP       ↑5
   JMP       ↑4            ; Else go to ELSE routine.
   .ENDC
   .DO       ↑3==3         ;This is the NOT EQUAL
                           ;routine.
   SUB#      ↑1,↑2,SNR     ; If ↑1<>↑2 then go to
                           ;THEN routine.
   JMP       ↑5            ; Else go to ELSE routine.
   JMP       ↑4
   .ENDC
%
```

Literals

Usually, when you use a memory reference instruction, you specify some particular location in memory. There are times, however, when you want to reference some value and are not concerned with the location containing it. You can create such a value with a *literal* reference.

A literal is a data value, usually a numeric constant or an address. The Macroassembler defines a literal value when it encounters a literal reference contained in the displacement field of a memory reference instruction. You can use literals only in such displacement fields.

```
LDA        3,=400/2    ;Loads the value 200 (which has
                       ;absolute relocation) into AC3
```

When the instruction in the example is assembled, the Macroassembler knows the value of the literal but not its location. You must set aside a place for the Macroassembler to assemble code into your program. You do this by placing **.LPOOL** pseudo ops in your program at places where literal values may be assembled. When the Macroassembler encounters an **.LPOOL** pseudo op, it assembles any literal values which have not been assigned locations on the current pass, forming a block of data words called a *literal pool.* These literal values can then be addressed by the appropriate memory reference instructions (as long as the instructions are within range; this is largely a function of where the **.LPOOL** is placed in relation to the instruction).

If you define the **.LPOOL** pseudo ops in your program to have **.ZREL** relocation, then all parts of your program will be able to access it.

The format of a literal reference is:

MRI □ [*ac*,] = <*exp* | *ext* | *inst*>

where

MRI is the mnemonic for some memory reference instruction,

exp is made up of legal operands and operators,

ext is an external symbol (**.EXTD** or **.EXTN**), and

inst is an instruction mnemonic followed by some number of fields.

You can use any absolute or relocatable expression, external symbol, or one-word instruction as a literal.

If two instructions reference the same literal value and relocation, and if both references are destined for the same literal pool, then the literal will be defined only once within the literal pool.

You can use literals for a variety of things. The statement

```
LDA 1,=3
```

loads the constant 3 into accumulator 1. The statements

```
LDA 2,=T*2
LDA 2,=(6/2)+"D/2
LDA 2,=ADD 1,2
```

load accumulator 2 with the value of a byte pointer, an expression, and an instruction respectively. The statement

```
STA 1,=0
```

allocates a literal pool word initially containing 0 (until the instruction is executed).

You can use literal labels for communication with subroutines. If the subroutine is out of addressing range of the **JSR** instruction, you can address it indirectly through a literal so long as the literal value in the literal pool is within addressing range of the **JSR** instruction. An example of this is:

```
JSR @=subr
```

where

subr is the name of the subroutine.

> **NOTE:** *The* **.END** *pseudo op contains an implicit* **.LPOOL**. *This insures that every literal value will be assigned an address, even if there are no* **.LPOOL** *pseudo ops in the program.*

Generated Numbers and Labels

You may use the format *symbol* anywhere in normal mode assembly code. The Macroassembler replaces *symbol* with a three-digit number representing the current value of *symbol* in the current input radix. If necessary, the Macroassembler truncates the value of *symbol* to fit into the three-digit format.

\symbol may stand alone in the code to form an integer, or it may immediately follow characters that, together with the value of *\symbol*, will form a number or symbol. The number or symbol consists of any number of combined characters.

```
.RDX      8
I         =1234
A\I:                    ;Is equivalent to A234 (the 1 is
                        ;truncated)
BB\I:                   ;Is equivalent to BB234
CCC\I:                  ;Is equivalent to CCC234
J         =45\I         ;Is equivalent to 45234
```

The Macroassembler will print *\symbol* in the assembly listing, even if though it is suppressed in the generated relocatable code:

```
ONES      =111          ;Source code
A\ONES
ONES      =111          ;Listing
A\ONES111
```

You can increment *\symbol* using the **.DO** facility to generate labels for a table:

```
          .RDX     8          ;Source code
**        I        =0
TABLE:    .DO      100
A\I:      0
**        I        =I+1
          .ENDC
          .RDX     8          ;This is the assembly
TABLE:    .DO      100        ;of the above code
A\I000:   0
A\I001:   0
A\I002:   0
A\I003:   0
          ...
A\I077:   0
          .ENDC
```

Note that the **\I** included in the labels is not included in the actual symbol. The labels appear in the symbol table as A000:, A001:, etc.

You can also use the dollar sign ($) to generate unique labels within macros. In non-string mode, the Macroassembler replaces each occurrence of $ with three characters from the set 0-9, A-Z. The Macroassembler determines which three characters to use by converting the count (in radix 36) of the total number of all macro calls to ASCII. In nested macros, the Macroassembler saves the replacement value for $ in the outer macro when the inner macro is expanded.

Generally you should not use $ as the first character in a label, since the first replacement character may be a digit.

The example shows how to use the $ sign to generate labels.

```
        000000          .NREL   0
                        .MACRO  TRK
                        DSZ     N
                 L$:    .               ;This is a label.
           %
00000'000000' N:        .
        000003          .DO     3       ;Generate 3 label
                                        ;entries.
                        TRK
00001'014777            DSZ     N
       000002'  L$001:  .               ;This is a label.
                        .ENDC
                        TRK
00002'014776            DSZ     N
       000003'  L$002:  .               ;This is a label.
                        .ENDC
                        TRK
00003'014775            DSZ     N
       000004'  L$003:  .               ;This is a label.
                        .ENDC
                        .END
```

Chapter 5
Pseudo Ops and Value Symbols

Introduction

This chapter describes the different types of pseudo ops. Under each heading, you will find a general description of what that type does and a list of the member pseudo ops. The list gives a brief summary of what each pseudo op does. For a more detailed discussion of the pseudo ops, refer to Chapter 6.

Symbol Table Pseudo Ops

You can use the pseudo ops in this category to define instructions and symbols, as well as to erase the symbol table. All symbol table pseudo ops except **.XPNG** have the form:

pseudo op usym = <*inst* | *exp*>

where

pseudo op is a symbol table pseudo op,

usym is a symbol you choose,

exp is made up of legal operands and operators, and

inst contains an instruction mnemonic and an appropriate number of fields.

Usym is defined as a semipermanent symbol with the value of *inst* or *exp*. For more information about instructions and expressions, see Chapter 3.

XPNG has the form

.XPNG

Most of the symbol table pseudo ops (except **.XPNG**) define a particular type of instruction. After you define an instruction, you must make sure you use the semipermanent symbol with the correct fields. For example, if you define an instruction which requires you to specify an accumulator, you must code an accumulator with that instruction each time you use it. If a field cannot contain the value you give it, an overflow error (O) will occur and the field will remain unaltered. If you specify the wrong number of fields a format error (F) will occur.

If you have defined a semipermanent symbol with one pseudo op and you decide to change your definition, you may do so by redefining your semipermanent symbol with another pseudo op. The Macroassembler will assign the last definition to a semipermanent symbol. If you try to redefine a semipermanent symbol, the Macroassembler will issue a multiple definition (M) error if you used the **/M** switch in your **XEQ MASM** command line.

Table 5.1 lists all the symbol table pseudo ops and briefly describes each one.

Pseudo Op	Instruction
.DALC	Defines an ALC instruction or expression.
.DCMR	Defines a commercial MRI instruction or expression.
.DEMR	Defines an extended MRI instruction or expression requiring an accumulator.
.DERA	Defines an extended MRI instruction or expression requiring an accumulator.
.DEUR	Defines an extended user instruction or expression.
.DFLM	Defines a floating load or store instruction requiring an accumulator.
.DFLS	Defines a floating load or store instruction requiring no accumulator.
.DIAC	Defines an I/O instruction requiring an accumulator.
.DICD	Defines an instruction requiring an accumulator and a count.
.DIMM	Defines an immediate-reference instruction requiring an accumlator.
.DIO	Defines an I/O instruction requiring no accumulator.
.DIOA	Defines an I/O instruction having two required fields.
.DISD	Defines an instruction with source and destination accumulators which does not allow skips.
.DISS	Defines an instruction with source and destination accumulators which allows skips.
.DMR	Defines an MRI instruction requiring an index and a displacement.
.DMRA	Defines an MRI instruction requiring two fields.
.DTAC	Defines an instruction with source and destination accumulators which does not allow skips.
.DUSR	Defines a user symbol without implied formatting.
.DXOP	Defines an instruction requiring source, destination, and operation fields.
.XPNG	Removes all semipermanent symbol definitions and macros.

Table 5.1 Symbol table pseudo ops

Location Counter Pseudo Ops

Location counter pseudo ops reserve a block of memory or specify a memory location or class of relocatable locations. Table 5.2 lists the location counter pseudo ops and describes each of them briefly.

Pseudo Op	Action
.BLK	Allocates a fixed block of storage locations.
.LOC	Sets the current location counter.
.NREL	Specifies pure or impure code relocation.
.ZREL	Specifies page zero relocation.

Table 5.2 Location counter pseudo ops

The period symbol (.) has the value and relocation property of the current location counter.

Communication Pseudo Ops

Pseudo ops of this type allow modules to reference symbols defined in other modules. You can also use them to declare named and unnamed common areas for communication. Table 5.3 lists these pseudo ops.

Pseudo Op	Action
.COMM	Reserves a named common area.
.CSIZ	Reserves an unnamed common area.
.ENT	Declares an entry symbol.
.ENTO	Declares an overlay identifier.
.EXTD	Declares an external displacement reference.
.EXTN	Declares an external normal reference.
.EXTU	Treats undefined symbols as external displacements.
.GADD	Adds a constant to an external symbol value.
.GLOC	Reserves an absolute data block.
.GREF	Assigns an expression value to a symbol without affecting the sign bit.

Table 5.3 Communication pseudo ops

Note that you must define symbols with **.ENT** if you want to reference them from another program.

Repetition Pseudo Ops

When you use one of these pseudo ops, you cause the Macroassembler to assemble code in a different way from the normal sequential procedure. Table 5.4 describes these pseudo ops and gives a brief description of each.

Pseudo Op	Action
.DO	Assembles the source lines between this statement and the corresponding .ENDC statement a specified number of times.
.ENDC	Defines the end of a series of repetitive assembly or conditional assembly source lines.
.GOTO	Suppresses assembly of source lines between this statement and the specified symbol.

Table 5.4 Repetition pseudo ops

Conditional Pseudo Ops

Conditional psuedo ops allow you to assemble source lines based on some condition. You specify the condition by choosing the appropriate pseudo op; if the condition is true, the Macroassembler will assemble the source lines between the conditional pseudo op and its corresponding **.ENDC**. If the condition is false, the Macroassembler will not assemble the appropriate source code. Table 5.5 lists the pseudo ops of this type.

Pseudo Op	Action
.ENDC	Defines the end of a series of conditional assembly or repetitive assembly source lines.
.IFE	Assembles only if the specified expression equals zero.
.IFG	Assembles only if the specified expression exceeds zero.
.IFL	Assembles only if the specified expression is less than zero.
.IFN	Assembles only if the specified expression is nonzero.

Table 5.5 Conditional pseudo ops

Stack Pseudo Ops and Values

You can save the value and relocation property of any valid expression on an internal stack maintained by the assembler. Table 5.6 lists the pseudo ops you need for manipulating stack data.

Pseudo Op	Action
.POP	Removes the top value and relocation off the stack and returns the value and relocation.
.PUSH	Places a value on the stack.
.TOP	Returns the value and relocation property of the expression most recently pushed on the stack.

Table 5.6 Stack pseudo ops

Macro Definition Pseudo Op and Values

The **.MACRO** psuedo op defines the start of a macro definition. You also specify the name of the macro with this pseudo op. Chapter 4 discusses macros in detail.

When you are writing macros, you may find two particular pseudo ops to be useful. Both of these pseudo ops return values. **.ARGCT** has as its value the number of macro arguments specified by the current macro call. Thus, if your current macro call had two arguments, **.ARGCT** would have the value 2.

The other symbol value, **.MCALL**, is a count which tells you whether the macro containing it has been called once, or more than once. If this is the first call to the macro containing **.MCALL** , then **.MCALL** has the value 0. If the macro containing **.MCALL** has been called before in this asssembly pass, then **.MCALL** has the value 1. Outside a macro, **.MCALL** has the value -1.

Text String Pseudo Ops and Values

You can specify ASCII text strings within source programs in several ways. Depending on which one of this type of pseudo ops you use, you can set parity, change how the Macroassembler packs the text string, or set the highest bit in each byte of your text string. Table 5.7 lists the pseudo ops which affect text strings.

Pseudo Op	Action
.TXT	Sets the leftmost bit to 0.
.TXTE	Sets the leftmost bit for even parity.
.TXTF	Sets the leftmost bit to 1 unconditionally.
.TXTM	Sets byte packing to left/right or right/left.
.TXTN	Terminates a string containing an even number of bytes with no zero bytes or two zero bytes.
.TXTO	Sets the leftmost bit of each character for odd parity.

Table 5.7 Text string pseudo ops

If you enclose the **.TXTN** pseudo op in parentheses and supply no argument, then this returns the value of the **.TXTN** expression. Likewise, enclosing **.TXTM** in parentheses returns the most recent **.TXTM** value.

Listing Pseudo Ops

Using one of these pseudo ops allows you to modify your output listings. By default, the Macroassembler lists all of your source lines, including macros, conditional assembly source lines, and lines lacking a location field. Table 5.8 lists the pseudo ops that will alter your output listing.

Pseudo Op	Action
.EJEC	Begins a new listing page.
.NOCON	Omits or restores listing of conditionally suppressed source lines.
.NOLOC	Omits or restores the listing of lines lacking location fields.
.NOMAC	Omits or restores the listing of macro expansions.

Table 5.8 Listing pseudo ops

You can totally suppress the assembly listing by omitting the /L function switch in the **XEQ MASM** CLI command line.

Other Pseudo Ops

The pseudo ops listed in Table 5.9 do not fall into any specific class of pseudo ops.

Pseudo Ops	Action
.END	Specifies the end of assembler source. If **.END** is missing, the system automatically supplies one. Can also specify a starting address for the program file. One module bound into each program file must specify a starting address by an **.END** statement. Includes an implicit **.LPOOL.**
.EOF, **.EOT**	Specifies an explicit end-of-file for the source module. If **.EOF** or **.EOT** is missing, the system automatically supplies one.
.FORC	Unconditionally binds the object file containing this pseudo op from a library.
.LPOOL	Allocates a variable block of storage locations (a literal pool).
.OB	Names the .OB file.
.PASS	Returns a value corresponding to the current assembly pass. **.PASS** is 0 on pass 1, and 1 on pass 2.
.RDX	Specifies the radix to be used in evaluating all numeric expressions input to the assembler.
.RDXO	Specifies the radix to be used for numeric listing output.
.TITL	Assigns a name to an object module.
.TSK	Specifies the number of TCBs which the binder must reserve for multitask use within the program.

Table 5.9 Other pseudo ops

Chapter 6
Pseudo OP Dictionary

This chapter contains descriptions of the Macroassembler pseudo ops. The pseudo op entries are arranged in alphabetical order according to mnemonic. Each entry lists the name of the pseudo op, the mnemonic, and the format, then describes the action of the pseudo op. Examples which illustrate how to use the pseudo ops complete each entry.

.(Period)

Current location counter

.

The period symbol (.) has the value and the relocation property of the current location counter.

Example

```
        000001 .NREL    1
   000001000003 3
        000005 .LOC     .+2
   000031020010 LDA     0,10
```

.ARGCT

Number of arguments in current macro invocation

.ARGCT

Lists the number of actual arguments given in the most recent macro call. If you use **.ARGCT** outside a macro, its value is -1.

Example

```
                        .NREL    1
                        .MACRO   X
                        ↑1+↑2
                        .ARGCT
              %
                        X        4,5      ;This macro call
                                          ;has two
                                          ;arguments
000001000011            4+5
000011000002            .ARGCT            ;ARGCT has value
                                          ;of 2, showing
                                          ;there were
                                          ;two arguments
                                          ;given in last
                                          ;macro call
```

Example

```
   .
   .
   .
   .MACRO   TABLE                        ;
   .IFE     .ARGCT-1                     ; If there is only
                                         ; one argument,
   ↑1                                    ; it is a pointer
   .ENDC                                 ; to a table.
   .IFE     .ARGCT-4                     ; If there are four
   .+1                                   ; arguments then
                                         ; .+1 is the
   ↑1                                    ; pointer. The
   ↑2                                    ; four arguments
   ↑3                                    ; are the table
   ↑4                                    ; entries.
   .ENDC
   .IFN     (.ARGCT-1)* (.ARGCT-4)       ; If the number of
   JMP      OTHER                        ; arguments is not
                                         ; one or four then
   .ENDC                                 ; jump to
%                                        ; another
   .                                     ; subroutine.
   .
   .
```

.BLK

Allocate a fixed block of storage

.BLK *exp*

Allocates a block of storage. The integer value of *exp* indicates the number of words in the block. Increments the current location counter by the integral value of *exp*.

> **NOTE:** *A* **.BLK** *appearing in absolute code does not allocate storage, although the location counter is incremented.*

Example

```
                         .NREL     0
00000'040405             STA       0,F
00001'044405             STA       1,F+1
00003'050405             STA       2,F+2
00004'054405             STA       3,F+3
00005'000004 F:          .BLK 4
00011'034510             LDA       3,110
                          .
                          .
                          .
```

.COMM

Reserve a named common area

.COMM *usym exp*

Reserves a named common area. The common area will be *exp* words big and will have the name *usym*. This common area will be used for passing data between programs and will be reserved by the first loaded routine which declares *usym*. Note that in your program, you must place the **.COMM** declaration before all other declarations that reserve storage locations.

Usym is an entry in the program and cannot be redefined elsewhere in the program. You may reference *usym* from other programs loaded together using **.EXTN**, **.EXTD**, **.GLOC**, or **GADD** pseudo ops.

Example

```
         .TITL    FIRST
000032   .COMM    A,32        ;Reserves a
                              ;common area
                              ;named A of 32
                              ;words
000064   .COMM    B,64        ;Reserves a
                              ;common
                              ;area named B of 64
                              ;words
         .
         .
         .
         .END
         .TITL    SECOND
000032   .COMM    A,32        ;Refers to common
                              ;area A
00000064 .COMM    B,64        ;Refers to common
                              ;area B
         .
         .
         .
         .END
```

.CSIZ

Specify an unlabeled common area

.CSIZ *exp*

Specifies the word size of an unlabeled common area to be used for passing data between programs.

The Macroassembler evaluates *exp* and passes the value to the Binder. If more than one **.CSIZ** pseudo op appears in a program, the Binder uses the largest value specified.

Example

```
         .TITL FIRST
000032   .CSIZ 32       ;Common area has 32 words
         .
         .
         .
         .END
         .TITL NEXT
000064   .CSIZ 64       ;Common area now has 64
                        ;words because this
         .              ;statement specifies a
                        ;larger size than the
         .              ;last CSIZ statement
         .END
```

.DALC

Define ALC instruction

.DALC *usym*=<*inst* | *exp*>

Defines *usym* as a semipermanent symbol having the value ("inst") or *exp*. You must specify at least two fields (the third is optional). Field 1 and field 2 represent source and destination accumulators, respectively. The optional third field represents a skip field. Specify these fields with the following format:

usym[c][sh][#] acs acd [skip]

The bit pattern below shows how the fields are assembled.

	ACS	ACD		SH	C	#	SKIP
0	1 2	3 4	5 7	8 9	10 11	12	13 15

You can use several options with the ALC instruction you define to make it more powerful. Refer to the section on ALC Instructions in Chapter2, of *Assembly Language Reference* for a detailed description of how these options work. In general, you can add mnemonics to your instruction that manipulate the value of carry, shift the specified data, and perform a skip test. Table 6.1 lists the mnemonics that affect the carry. Table 6.2 lists the mnemonics that perform shift operations. Table 6.3 lists the mnemonics that specify a skip test.

Mnem	Sets bits 8-9 to	Action
-	00	No action
Z	01	Sets carry to zero
O	10	Sets carry to one
C	11	Complements carry

Table 6.1 Carry mnemonics

Mnem	Sets bits 10-11 to	Action
-	00	No action
L	01	Shifts data left one bit
R	10	Shifts data right one bit
S	11	Swaps bytes of data

Table 6.2 Shift mnemonics

Mnem	Sets bits 13-15 to	Action
—	000	No skip
SKP	001	Skip unconditionally
SZC	010	Skip if carry bit is zero
SNC	011	Skip if carry bit is one
SZR	100	Skip if ALC result is zero
SNR	101	Skip if ALC result is nonzero
SEZ	110	Skip if either ALC result or carry bit is zero
SBN	111	Skip if both ALC result and carry bit are nonzero

Table 6.3 Mnemonics for optional skip field

The atom # may be used anywhere as a break character. It assembles as a 1 in bit 12.

Example

```
         103000  .DALC   ADD=103000
   00000 103000  ADD     0,0         ;These three
                                     ;statements
   00001 103002  ADD     0,0,SZC     ;specify fields
                                     ;correctly
   00002 133001  ADD     1,2,SKP
  F00003 123000  ADD     1           ;These two
                                     ;statements
 FF00004 103000  ADD                 ;do not specify
                                     ;fields correctly
```

.DCMR

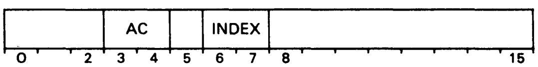
Define commercial memory reference instruction

.DCMR *usym*=<*inst* | *exp*>

Defines *usym* as a semipermanent symbol with the value *inst* or *exp*. *Usym* will be treated as an instruction requiring an accumulator and a displacement. You may specify an optional index field as well. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify at least two fields when you use *usym* (the third is optional). Use the format shown below.

usym ac disp [index]

The three fields assemble as shown below.



Example

```
         102170     .DCMR    ELDB=102170
         000001     .NREL    1
 000000l112570      ELDB     2,PT1       ;AC2 contains
                                         ;the
         001376                          ;character A
 000002l113570      ELDB     3,PT2       ;AC3 contains
         001373                          ;character B
            .
            .
            .
 000600l040502 ALPHA: .TXT   "AB"
         001400&    PT1=     ALPHA*2
         001401&    PT2=     ALPHA*2+1
```

.DEMR

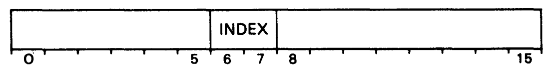
Define extended memory reference instruction

.DEMR *usym*=<*inst* | *exp*>

Defines *usym* as a semipermanent symbol with the value *inst* or *exp*. *Usym* will be treated as an instruction requiring a displacement. You may specify an optional index field as well. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify at least one field when you use *usym* (the second is optional). Use the format shown below.

usym disp [index]

The two fields assemble as shown below.



You can use @ to indicate indirect addressing. This atom sets bit 0 of the second word to 1.

Example

```
        102070 .DEMR    EJMP=102070
               .EXTN    ADDR
        000001 .NREL    1
 000000l102070 EJMP     ADDR
       000001$
 000002l102470 EJMP     .+3
        000002
F000004l102070 EJMP     2,.+3       ;This specifies an
        000002                      ;accumulator,
                                    ;which is
                                    ;incorrect
```

.DERA

Define extended memory reference instruction requiring an accumulator

.DERA *usym=<inst | exp>*

Defines *usym* as a semipermanent symbol with the value *inst* or *exp. Usym* will be treated as an instruction requiring an accumulator and a displacement. You may specify an optional index field as well. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify at least two fields when you use *usym* (the third is optional). Use the format shown below.

usym ac disp [index]

The three fields assemble as shown below.

0–2	3–4	5	6–7	8–15
	AC		INDEX	

0	1–15
@	DISPLACEMENT

You can use the atom @ anywhere in the instruction to indicate indirect addressing. This atom sets bit 0 in the second word to 1.

Example

```
          122070 .DERA     ELDA=122070
                 .NREL     1
    0000001122470 ELDA     0,.+3
          000002
 FF0000002!122070ELDA      .+3        ;This specifies
                                      ;the
                                      ;wrong number
                                      ;of
                                      ;arguments
          000000
```

.DEUR

Define extended user instruction

.DEUR *usym=<inst | exp>*

Defines *usym* as a semipermanent symbol with the value *inst* or *exp*, where *exp* can be an expression, an external normal, or an external displacement. *Usym* will be treated as an instruction requiring an *expression* (where the *expression* may be an expression, an external normal, or an external displacement). After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify one field when you use *usym*. Use the format shown below.

usym expression

The field assembles as shown below.

0–15

0–15
DISPLACEMENT

Example

```
       163710      .DEUR    SAVE=163710
       061777      .DEUR    VCT=061777
       000001      .NREL    1
  0000001163710    SAVE     4
       000004
       000000      SYMB     =0
  0000021061777    VCT      SYMB
       000000
```

.DFLM

Define floating load or store instruction requiring an accumulator

.DFLM *usym*=<*inst* | *exp*>

Defines *usym* as a semipermanent symbol with the value *inst* or *exp*. *Usym* will be treated as a floating point load or store instruction requiring an accumulator and a displacement. You may specify an optional index field as well. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify at least two fields when you use *usym* (the third is optional). Use the format shown below.

usym fpac disp [index]

The three fields assemble as shown below.

	INDEX	FPAC	
0	1 2	3 4	5 … 15

@	DISPLACEMENT
0	1 … 15

You can use the atom @ anywhere in the instruction to indicate indirect addressing. This atom sets bit 0 in the second word to 1.

Example

```
         102050 .DFLM     FLDS=102050
         000000 .NREL     0
  000000'122050 FLDS      0,.+2
         000001
F000002'102050 FLDS       .+3        ;This specifies the
                                     ;wrong number
                                     ;of arguments
```

.DFLS

Define floating load or store instruction

.DFLS *usym*=<*inst* | *exp*>

Defines *usym* as a semipermanent symbol with the value *inst* or *exp*. *Usym* will be treated as a floating point load or store instruction requiring a displacement. You may specify an optional index field as well. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify at least one field when you use *usym* (the second is optional). Use the format shown below.

usym disp[index]

The two fields assemble as shown below.

	INDEX	
0 2	3 4	5 … 15

@	DISPLACEMENT
0	1 … 15

You can use the atom @ anywhere in the instruction to indicate indirect addressing. This atom sets bit 0 of the second word to 1.

Example

```
         123350 .DFLS     FLST=123350
         000000 .NREL     0
                .EXTN     DR
  000000'123350 FLST      DR
        000001$
 00002'123350F  FLST                 ;This specifies no
                                     ;displacemennt
                                     ;field
         000000
```

.DIAC

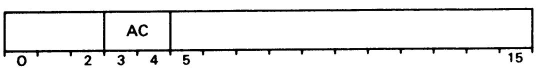
Define an instruction requiring an accumulator

.DIAC *usym*=*<inst | exp>*

Defines *usym* to be a semipermanent symbol with the value *inst* or *exp*. *Usym* will be treated as an instruction requiring an accumulator. After defining *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify an accumulator field with *usym*. Use the following format:

usym ac

The field assembles as shown below.



Example

```
      061000  .DIAC    CM=061000
      062000  .DIAC    MA=062000
00000 061000  CM       0            ;This command is
                                    ;DOA 0,0
00002 066000  MA       1            ;This command is
                                    ;DOB 1,0
```

.DICD

Define an instruction requiring an accumulator and count

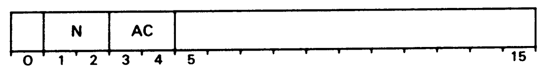
.DICD *usym*=*<inst | exp>*

Defines *usym* as a semipermanent symbol with the value *inst* or *exp*. *Usym* will be treated as an instruction requiring count and destination fields. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify two fields when you use *usym*. Use the format shown below.

usym n ac

where *n* is an integer between 1 and 4.

The two fields assemble as shown below.



where *N* has a value between 0 and 3.

Example

```
        100010  .DICD   ADI=100010
 00000  104010  ADI     1,1
 00001  110010  ADI     1,2
 00002  160010  ADI     4,0
000003  104010  ADI     5,1         ;Specifies illegal
                                    ;count field
F00004  100010  ADI     1           ;Specifies too few
                                    ;arguments
```

.DIMM

Define an instruction requiring an accumulator and an immediate value

.DIMM *usym=<inst | exp>*

Defines *usym* as a semipermanent symbol with the value *inst* or *exp. Usym* will be treated as an instruction requiring an accumulator and an immediate value. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify two fields with *usym*. Use the following format:

usym immed ac

The fields assemble as shown in the diagram below.

0–2	3–4	5–15
	AC	

0–15
IMMEDIATE

Example

```
        163770  .DIMM   ADDI=163770
        000001  .NREL   1
  000001173770  ADDI    1002,2      ;This statement is
                                    ;correct
        001002
 F00002163770   ADDI    0           ;Specifies
                                    ;incorrect
        000000                      ;number of
                                    ;arguments
```

.DIO

Define an I/O instruction without an accumulator

.DIO *usym[f]=<inst | exp>*

Defines *usym* to be a semipermanent symbol with the value *inst* or *exp. Usym* will be treated as an I/O instruction requiring no accumulator field. After you define *usym* use it as you would an instruction mnemonic. Note, however, that you must specify one field with *usym*. Use the following format:

usym[f] device_code

The fields assemble as shown below.

0–7	8–9	10–15
	F	DEVICE CODE

NOTE: *If you define a three-character* usym *with this pseudo op, you can follow it immediately with one of the letters from Table 6.4. Each letter represents an optional function code which sets bits 8-9 of the instruction word. (See example below.)*

Optional Mnem	Sets bits 8-9 to	Action*
S	01	Sets Busy flag, clears Done flag, starting device.
C	10	Clears Done and Busy flags, idling device.
P	11	Sets Done and Busy flags, pulsing I/O bus control line.

Table 6.4 Function Mnemonics

**The actions of these flags are device dependent. For a more detailed discussion on specific I/O devices, refer to the appropriate sections of* Microproducts Hardware Systems Reference.

Example

```
.DIO    NIO=060000
.DIO    SKPBZ=063500
```

.DIOA

Define an I/O instruction with an accumulator

.DIOA *usym[f]=<inst | exp>*

Defines *usym* to be a semipermanent symbol with the value *inst* or *exp. Usym* will be treated as an I/O instruction which requires an accumulator. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify two fields with *usym*. Use the format shown below.

usym[f] ac device_code

The fields assemble as shown in the diagram shown below.

0–2	3–4	5–7	8–9	10–15
	AC		F	DEVICE CODE

> **NOTE:** *If you define a three-character* usym *with this pseudo op, you can follow it immediately with one of the letters from Table 6.4. Each letter represents an optional function code which sets bits 8-9 of the instruction word. (See example below.)*

Example

```
      060400 .DIOA   DIA=060400
      000001 .NREL   1
000011070410 DIA     2,TTI        ;Correct
000021070610 DIAC    2,TTI        ;Correct
```

.DISD

Define an instruction with source and destination accumulators

.DISD *usym=<inst | exp>*

Defines *usym* to be a semipermanent symbol with the value *inst* or *exp. Usym* will be treated as an instruction requiring source and destination accumulators. Note that this pseudo op does not allow you to specify the load/no-load, carry, shift, or skip conditions. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify two fields with *usym*. Use the format shown below.

usym acs acd

The fields assemble as shown in the diagram shown below.

0	1–2	3–4	5–15
	ACS	ACD	

Example

```
      102710         .DISD   LDB=102710
      000001         .NREL   1
000001030410         LDA     2,.PTR
000011146710         LDB     2,1      ;Byte addressed
                                      ;by AC2
                     .                ;is loaded into
                                      ;AC1.
                     .
                     .
000101000022& .PTR:  .+1*2
000111040502         .TXT    "ABCDE"
      041504
      042400
```

.DISS

Define a skip instruction with source and destination accumulators

.DISS *usym*=<*inst* | *exp*>

Defines *usym* to be a semipermanent symbol with the value *inst* or *exp*. *Usym* will be treated as an instruction requiring source and destination accumulators. Note that this pseudo op does not allow you to specify the load/no-load or skip options; however, the instruction you wish to define may cause a skip (such as the ECLIPSE *Skip If Greater Than* instruction). After defining *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify two fields with *usym*. Use the format shown below.

usym acs acd

The fields assemble as shown below.

0	1–2	3–4	5–15
	ACS	ACD	

Example

101010	.DISS	SGT=101010	;This an ECLIPSE ;instruction
	.NREL	1	
000001131010	SGT	1,2	
F000011121010	SGT	1	;Not enough ;arguments

.DMR

Define a memory reference instruction with displacement and index

.DMR *usym*=<*inst* | *exp*>

Defines *usym* to be a semipermanent symbol with the value *inst* or *exp*. *Usym* will be treated as a memory reference instruction requiring either a displacement (or address), or a displacement (or address) and index. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify one or two fields with *usym*. Use one of the formats shown below.

usym displacement
usym displacement index

The fields assemble as shown below.

0–4	5	6–7	8–15
	@	INDEX	DISPLACEMENT

You can use the atom @ anywhere in the instruction to indicate indirect addressing. This atom sets bit 5 to 1.

Example

000001		.NREL	1	
000000		.DMR	JMP=000000	
000001000402	WIZ:	JMP	.+2	
F000011000400		JMP	0,1,2	;Incorrect number ;of arguments
000002I003001		JMP	@1,2	;Correct number ;of arguments ;plus correct use ;of indirect bit
000003I000775		JMP	WIZ	;Correct number of arguments

.DMRA

Define a memory reference instruction with two or three fields

.DMRA *usym=<inst | exp>*

Defines *usym* to be a semipermanent symbol with a value of *inst* or *exp*. *Usym* will be treated as a memory reference instruction requiring two or three fields. The first field specifies an accumulator. When you specify only two fields, the second field is a displacement. When you specify three fields, the second field is a displacement and the third field is an index. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify the two or three fields with *usym*. Use one of the formats shown below.

usym ac displacement
usym ac displacement index

The fields assemble as shown in the diagram below.

0–2	3–4	5	6–7	8–15
	AC	@	INDEX	DISPLACEMENT

You can specify the atom @ anywhere in the instruction as a break character. This atom assembles a 1 in bit 5.

Example

```
          000001 .NREL   1
          020000 .DMRA   LDA=20000
    00000'030204 LDA     2,.+4
    00001'025400 LDA     1,0,3
    00002'031401 LDA     2,1,3
    00003'033401 LDA     2,@1,3
```

.DO

Assemble source lines repetitively

.DO *exp*

Assembles the lines of source program between the **.DO** and the corresponding **.ENDC** *exp* number of times.

.DO statements may be nested to any depth. The innermost **.DO** corresponds the the innermost **.ENDC**, etc. See Chapter 4 for information about using **.DO**s with macros.

Example

This is the source program.

```
           .NREL    0
FIRST:     1
SECOND:    3
SUM:       0
           .DO      3
           LDA      1,FIRST
           LDA      2,SECOND
           ADD      1,2
           STA      1,SUM
           .ENDC
           .END
```

This is the listing of the source program.

```
      000000            .NREL     0
00000'000001 FIRST:     1
00001'000003 SECOND:    3
00002'000000 SUM:       0
      000003            .DO       3
00003'024775            LDA       1,FIRST
00004'030775            LDA       2,SECOND
00005'133000            ADD       1,2
00006'044774            STA       2,SUM
                        .ENDC
00007'024771            LDA       1,FIRST
00010'030771            LDA       2,SECOND
00011'133000            ADD       1,2
00012'044770            STA       2,SUM
                        .ENDC
00013'024765            LDA       1,FIRST
00014'030765            LDA       2,SECOND
00015'133000            ADD       1,2
00016'044764            STA       2,SUM
                        .ENDC
                        .END
```

.DTAC

Define an instruction with two accumulator fields

.DTAC *usym=<inst | exp>*

Defines *usym* to be a semipermanent symbol with the value of *inst* or *exp*. *Usym* will be treated as an instruction requiring source and destination accumulators. After *usym* has been defined, it is used as an instruction mnemonic. Note, however, that you must specify two fields with *usym*. Use the format shown below.

usym acs acd

The fields assemble as shown in the diagram below.

0–2	3–4	5–7	8–9	10–15
	ACD		ACS	

Example

```
      060401          .DTAC   LDB=060401
00000 030410          LDA     2,.PTR
00001 064601          LDB .   2,1      ;Byte addressed
                      .                ;by AC2 is loaded
                      .                ;into AC1
                      .
00010 000022 .PTR:    (.+1)*2
00011 040502          .TXT    "ABCDE"
      041504
      042400
```

.DUSR

Define user symbol without format

.DUSR *user symbol=<inst | exp>*

Defines *usym* to be a semipermanent symbol with the value *inst* or *exp*. *Usym* is given a value with no implied format. You can use the *usym* defined by **.DUSR** anywhere you would use a single precision operand.

Example

```
          000000 .NREL   0
          000025 .DUSR   A=25
          000124 .DUSR   B=A*4
    00000'000077 B-A
    00001'003345 A*B+1
```

Example

You can use **.DUSR** to assign mnemonics to displacements in user data bases.

```
          000001 .DUSR   TAC0=1
          000002 .DUSR   TAC1=2
    00000 021001 LDA     0,TAC0,2
    00001 025002 LDA     1,TAC1,2
                  .
                  .
                  .
```

.DXOP

Define an instruction with source, destination, and operation fields

.DXOP *usym=<inst | exp>*

Defines *usym* as a semipermanent symbol with the value *inst* or *exp*. *Usym* will act as an instruction requiring source and destination accumulators and an operation number. After you define *usym*, use it as you would an instruction mnemonic. Note, however, that you must specify three fields with *usym*. Use the format shown below.

usym acs acd op_no.

The fields assemble as shown below.

	ACS	ACD	OP CODE	
0	1 2	3 4	5 7	8 15

Example

```
          100030 .DXOP   XOP=100030
          000001 .NREL   1
    000001130130 XOP     1,2,1
   F000011100030 XOP     1,2          ; Incorrect
                                      ;number
                                      ;of arguments
```

.EJEC

Begin a new listing page

.EJEC

Begins a new page in the listing output.

Example

This is the source code.

```
        MOV     1,2
        .EJEC
        LDA     1,0,1
```

This is the listing of the above code.

```
      page 1
00000 131000 MOV     1,2
             .EJEC
      page 2
00001 024401 LDA     1,0,1
```

.END

End-of-assembly indicator

.END *[exp]*

Terminates a source file by providing an end-of-module indicator for the Macroassembler. Also builds a pool of literal values if necessary (see **.LPOOL** entry). The *exp* is an optional argument specifying a starting address for execution. The Binder initializes the program address in the initial TCB to the last address, if any, specified by an object binary at load time. Execution of the bound program file begins at this address. (If the Binder finds no starting address among the modules loaded, it issues an error message.)

Example

```
                        .TITL    SUBR
              VAR:      0
00000 024100  SUBR:     LDA      1,VAR
00001 132400            SUB      1,2
00002 050100            STA      2,VAR
                        .END     SUBR
```

.ENDC

Specify the end of conditional assembly

.ENDC *[usym]*

If the syntax is **.ENDC**, this pseudo op terminates lines for repetitive assembly (lines following **.DO**), or lines whose assembly is conditional (lines following **.IFE**, **.IFG**, **.IFL**, or **.IFN**).

If the syntax is **.ENDC** *usym*, this pseudo op declares a *usym*, which marks the end of the next block of conditionally assembled code. If the first block (i.e., the statements between the **.DO** and the **.ENDC**) is assembled, then the second block (i.e., the statements between the **.ENDC** and the *usym*) will not be assembled. If the first block is not assembled, then the second block will be assembled. If the second block follows a **.DO** block which is not assembled, then the second block will only be assembled once.

Example

```
        .IFN     ALPHA     ;Assemble only if
                           ;ALPHA is not zero
        SUB      0,0
        .ENDC    FAIL      ;End of conditional
                           ;assembly. If ALPHA
                           ;does not equal 0,
                           ;then do not
                           ;assemble
                           ;the code
                           ;between
                           ;this .ENDC
                           ;statement
                           ;and [FAIL]
        ADD      1,0
        MUL
        SUB      0,0
[FAIL]
        .DO      ALPHA     ;If ALPHA=0, then
                           ;do not assemble the
                           ;code between
        ADD      0,0       ;.DO and the next
                           ;.ENDC
        .ENDC              ;End the .DO loop
```

.ENT

Declare a symbol entry

.ENT $usym_1$[... $usym_n$]

Declares each *usym* to be defined within the current file that may be referenced by separately-assembled files.

A *usym* appearing in a **.ENT** pseudo op must be defined as a user symbol within the file. This symbol must be unique from entries defined in other object modules bound together to form a program file. If not, the binder will issue a message indicating multiply-defined entries.

You can reference entries in separately assembled files using either the **.EXTD** or the **.EXTN** pseudo ops.

Example

```
                              .TITL    A
                              .ENT     ONE,TWO
                              .EXTN    THREE
                              .ZREL
00000-000000$ ONE:            THREE
     000001                   .NREL    1
 000001024100 TWO:            LDA      1,100
 00001 1 1012400-             JSR      @ONE
                              .
                              .
                              .
                              .END
```

.ENTO

Define an overlay entry

.ENTO *usym*

Associates *usym* with the node number and overlay number of the overlay in which this module resides. If the module is loaded outside an overlay, the value of *usym* is set to -1. You may reference the overlay from another file by using *usym*. Note that you must declare *usym* as an **.EXTN** in the referencing file.

> **NOTE:** *The value of* usym *is assigned at bind time.*

You can place both pure and impure code in the same assembly module. If you place such a module in an overlay, code in this module will be split. The pure code will become an overlay in the pure area; the impure code will be loaded as resident code in the impure area.

The format of the resolved value of *usym* is:

1	NODE	OVERLAY
0	1 – 7	8 – 15

Example

```
.TITL    C       ;The whole module is named C
.NREL    0       ;This is the impure code overlay
.ENTO    EO1     ;EO1 is the entry into this overlay
.
.
.
.NREL    1       ;This is the pure code overlay
.ENTO    EO2     ;EO2 is the entry into this overlay
.
.
.
.END
```

.EOF

Explicit end-of-file

.EOF

Provides an explicit end-of-file for any source module except the last in a series for assembly. If you leave **.EOF** pseudo ops out of your source modules, the system supplies them implicitly.

Example

```
.TITL    FUTZ
.
.
.
.EOF
```

.EOT

Explicit end-of-tape

.EOT

Provides an explicit end-of-file for any source module except the last in a series for assembly. If you leave **.EOT** pseudo ops out of your source modules, the system supplies them implicitly.

Example

```
.TITL      FUTZ
.
.
.
.EOT
```

.EXTD

Declare an external displacement reference

.EXTD *usym₁[... usymₙ]*

Allows a file to reference a *usym* defined in some other file. *Usym* must be declared by an **.ENT** pseudo op in the file defining it.

Any **.EXTD** *usym* may specify the contents of a 16-bit storage word. *Usym* may also be an address or displacement of a memory reference instruction. If you use *usym* to specify a page zero address or a displacement for a memory reference instruction, then *usym* must meet specific requirements:

0<=page zero address>=377

-200<=displacement>=+200

Example

```
         .TITL    FIZL
         .ENT     WUMP
         .ZREL
WUMP:    .+1
         VAR1
         VAR2
         VAR3
         .
         .
         .
         .END
         .TITL    BUMP
         .EXTD    WUMP
         .NREL    1
         .
         .
         .
         LDA      3,WUMP     ;Load AC3 with
                             ;address of VAR1
         LDA      0,0,3      ;Load ACs 0,1,2
                             ;with the three
         LDA      1,1,3      ;variables VAR1,
                             ;VAR2, and VAR3
         LDA      2,2,3
         .
         .
         .
         .END
```

.EXTN

Declare an external normal reference

.EXTN *usym*$_1$[... *usym*$_n$]

Allows the currrent file to reference some *usym* defined in another file. You must use an **.ENT** pseudo op to declare *usym* in the file defining it.

An **.EXTN** *usym* specifies the contents of a 16-bit storage word. The value at bind time of this word can be an unsigned number in the range:

$0 <= \text{value of storage word} >= 2^{16}\text{-}1$

Example

```
         .TITL    CRANK
         .EXTN    GEAR
         .NREL    1            ;Start of pure code
.GEAR:   GEAR
         .
         .
         .
         JSR      @.GEAR
         .END
         .TITL    COG
         .ENT     GEAR
         .NREL    1            ;Start of pure code
GEAR:    LDA      2,1
         .
         .
         .
         .END
```

.EXTU

Treat undefined symbols as external displacements

.EXTU

Causes the assembler to treat all symbols that are undefined after pass 1 as if they had appeared in an **.EXTD** statement.

> **NOTE:** *This pseudo op is not recommended for use in untested user programs. It is used by compiler-level languages where undefined symbols are resolved at bind time by the inclusion of run-time libraries.*

Example

```
                  .TITL    GEEZ
                  .EXTU
00000 024000$     LDA      1,IBUS
                  .
                  .
                  .
                  .END
```

.FORC

Force load this module from a library

.FORC

Forces the Binder to unconditionally load the module into the current program file.

Normally, the Binder binds a library object file into a program file only if it satisfies a currently unsatisfied external reference appearing in another module. If the **.FORC** pseudo op appears in a module, the Binder will unconditionally load that module into the program file.

If a **.FORC**ed module multiply-defines a symbol, the Binder uses the first definition it encounters during the bind.

You can use **.FORC** for a library module if you want it to be force-bound whenever the library name occurs in the Binder command line.

Example

```
        .TITL   SQUARE   ;SQUARE is part of
                         ;a library. Whenever
        .NREL   1        ;the library name
                         ;occurs in the bind
        .ENT    CUBE     ;command line, the
                         ;program will have
        .FORC            ;access to the CUBE
                         ;routine, even if
        .                ;the current version
                         ;of the program does
        .                ;not reference
                         ;CUBE.
        .
CUBE:   ...
        .END
```

.GADD

Add a scalar value to an external symbol value

.GADD *usym exp*

Generates a storage word whose contents are resolved at bind time. The Binder searches for the value of *usym* and, if found, adds it to *exp* to form the contents of the storage word. If the Binder does not find the value of *usym*, it generates a Binder error. In this case the storage word will contain just the value of *exp*.

Usym must be a symbol defined in some separately assembled file and must appear in that program in a **.ENT**, **.ENTO**, or **.COMM** statement.

Example

```
                           .TITL   ZEBRA
                           .ENT    ZEB
         000200            .LOC    200
   00200 000201 ZEB:       .+1              ;ZEB has the
                                            ;value 200
                           .END
                           .TITL   STRIPE
                           .EXTN   ZEB
   00100 000006 ZEBL:      .GADD   ZEB,5+1  ;Value at ZEB
                                            ;will be 206
                                            ;at bind time
                           .END
```

.GLOC

Reserve an absolute data block

.GLOC *usym*

Begins a block of absolute data which starts at the value of the external user symbol, *usym*, at bind time. The next .LOC, **.NREL**, **.ZREL**, or **.END** pseudo op will terminate the block.

You must declare *usym* as an **.EXTN** in the file in which the .GLOC appears. Omitting this will cause an undefined symbol (U) error.

You can make no external references, label definitions, or label references within the block.

.GOTO

Suppress assembly

.GOTO *usym*

Suppresses the assembly of lines until the scan encounters another *usym* enclosed in square brackets.

Example

```
                    .GOTO   START   ;Start assembly
                                    ;at START.
                    LDA     0,0,2   ;Do not assemble
                    MOVL    0,0     ;these
                                    ;instructions.
                    SUB     1,1
00000 021001 [START]
00001 131100        LDA     0,1,2   ;Start assembly
                                    ;here.
                    MOV     1,2
                    .
                    .
                    .
```

.GREF

Assign expression value to symbol

.GREF *usym exp*

Functions exactly like the **.GADD** pseudo op except that bit zero of the storage word will always retain its original value.

.IFE, .IFG, .IFL, .IFN

Perform conditional assembly

.IFE *n*

.IFG *n*

.IFL *n*

.IFN *n*

The lines between these pseudo ops and the corresponding **.ENDC**s will be assembled if the condition defined in the pseudo op is true. The pseudo ops define the following conditions:

.IFE *n* - assemble if $n=0$
.IFG *n* - assemble if $n>0$
.IFL *n* - assemble if $n<0$
.IFN *n* - assemble if $n<>0$

The value field of the listing is 1 if the condition is true and 0 if the condition is false.

NOTE: **.IF**s *may be nested to any depth, with the innermost* **.IF** *corresponding to the innermost* **.ENDC**, *etc. Note that all* **.IF** *conditions are degenerate forms of* **.DO**s. *For example,* **.IFG A** *is equivalent to* **.DO A>0**.

Chapter 4 describes handling of **.IF**s *and* **.DO**s *within macros.*

Example

```
000004 A=4
000005 B=5
000000 C=0
000001 .NREL     1
000000 .IFE      B-A       ;This expression
                           ;does not meet
       LDA       0,A       ;the condition, so
                           ;the LDA
       .ENDC               ;instruction is not
                           ;assembled
000000 .IFG      C+C-A     ;This expression
                           ;does not meet
       LDA       0,A       ;the condition, so
                           ;the LDA
       .ENDC               ;instruction is not
                           ;assembled
000001 .IFL      A-B+C     ;This expression
                           ;meets the
000001020004 LDA 0,A       ;condition, so the
       .ENDC               ;LDA instruction
                           ;is assembled
000001 .IFN      A+B       ;This expression
                           ;meets the
000011020004 LDA 0,A       ;condition, so the
       .ENDC               ;LDA instruction
                           ;is assembled
```

.LOC

Set the current location counter

.LOC *exp*

Sets the current location counter to the value and relocation property given by *exp*. The initial value of **.LOC** is zero with absolute relocation.

Note that if you PUSH **.LOC** onto the stack, do not try to use it to restore the value of the location counter. That is, when you do this:

```
.PUSH  .LOC
.LOC   .POP
```

the Macroassembler will ignore the value of **.POP**, but will use its relocation property. This is to allow you to save the current relocation mode within a macro and restore it correctly, without affecting the relative location counter value (which may have been altered since the **.PUSH**).

Example

```
       000000         .NREL   0
00000'000010 A:       10
                      .ZREL
00000-000021 Z:       21
       000100         .LOC    100
00100 000000'         A
00101 000000-         Z
       000000'        .LOC    A
00000'000000'         A
00001'000000-         Z
       000000-        .LOC    Z
00000-000000'         A
00001-000000-         Z
                      .END
```

.LPOOL

Permit construction of a literal pool

.LPOOL

Marks locations where the Macroassembler may deposit the values of literal expressions which have not yet been assigned addresses during the current assembler pass. The number of values in a literal pool determines the pool's size. Each value occupies one word. The Macroassembler ignores an **.LPOOL** pseudo op if there are no currently unassigned values.

Since the **.LPOOL** pseudo op marks a potential data area, you should make sure it does not appear in the path of executable code. Usually you place **.LPOOL** pseudo ops after branching instructions, returns from subroutines, and other areas typically used for storing data. If you do not not include an .LPOOL pseudo op in a program that uses literals, then the literal pool will be constructed by the .END pseudo op (see .END).

You can set the relocation property of literal pool addresses using the **.LOC**, **.ZREL**, and **.NREL** pseudo ops.

Example

```
000001  .NREL    1
        .
        .
        .
1020000 LDA      0,="A       ;NREL instruction
                             ;references literal
        .
        .
        .
        .ZREL                ;ZREL definition
                             ;means that all
                             ;code can
-000101 .LPOOL               ;reference a literal
                             ;value ("A)
000001  .NREL    1           ;Continue
                             ;program in NREL
        .
        .
        .
        .EXTN                TRIPE
        .NREL                1
        LDA                  2,@=TRIPE
        LDA                  1,=-400/2
        SUB                  0,0
  GUTZ: STA                  0,0,2
        INC                  1,1,SNR
        RET
        INC                  2,2
        JMP                  @=GUTZ
000000$ .LPOOL
177600
000031
```

.MACRO

Name a macro definition

.MACRO *usym*

Defines *usym* as the name of the macro definition that follows. Any character following the **.MACRO** line is part of the macro definition up to the first % character encountered. The % may appear within a line.

After definition, *usym* calls the macro. Chapter 4 discusses macros in detail.

Example

```
                    .MACRO   TEST        ;SAMPLE is the
                                         ;name of the
                                         ;macro
                    ↑1                   ; These three lines
                    ↑2                   ; are the macro
                    ↑3                   ; definition
                 %                       ; This symbol
                                         ; ends the macro
                    TEST     4,5,6       ;Macro call with
                                         ;args 4,5,6
   00000 000004 4
   00001 000005 5
   00002 000006 6
                    .RDX     16          ;Change the radix
                    TEST     OA,OB,OC    ;Macro call with
                                         ;args OA,OB,OC
   00003 000012 OA
   00004 000013 OB
   00005 000014 OC
```

.MCALL

Indicate current macro usage

.MCALL has value 1 if the macro containing it has been previously called on this assembly pass. It has value 0 if the macro containing it has not been previously called on this assembly pass. If used outside a macro, its value is -1.

Example

```
      .MACRO    Z
      .DO       .MCALL<>O ;If this is not the first
                          ;macro call assemble
      JSR       @.X       ;this code
      .ENDC
      .DO       .MCALL==O ;If this is the first macro
                          ;call then generate
                          ;subroutine X
      .PUSH     .         ;Save the location counter
      .ZREL
 .X:  X
      .LOC      .POP      ;Restore location counter
      JSR       X         ;Call subroutine X
      JMP       XEND      ;Jump to end
 X:   .                   ;This would be the code
                          ;for X
      .
      .
      JMP       O,3       ;Return to main routine
XEND:
      .ENDC               ;End the .DO loop
 %                        ;End the macro
```

.NOCON

Inhibit or re-enable listing conditional lines

.NOCON *exp*

Either inhibits or permits listing of those conditional portions of the source program that do not meet the conditions given for assembly. If the value of *exp* is not zero, listing is inhibited; if the value of *exp* equals zero, listing occurs. The macroassembler defaults to listing the conditional portions of the program.

.NOCON does not affect conditional portions of the source proram that would be assembled.

The value of **.NOCON** is the value of the last *exp*.

Example

This is the source program.

```
A=3
.NOCON    0
.DO       4==A
5
3
.ENDC
.DO       4==(A+1)
5
3
.ENDC
.NOCON    1
.DO       4==A
5
3
.ENDC
.DO       4==(A+1)
5
3
.ENDC
```

This is the program listing.

```
      000003 A=3
      000000 .NOCON    0
      000000 .DO       4==A
             5
             3
             .ENDC
      000001 .DO       4==(A+1)
00000 000005 5
00001 000003 3
             .ENDC
      000001 .NOCON    1
      000001 .DO       4==(A+1)
00002 000005 5
00003 000003 3
             .ENDC
```

.NOLOC

Inhibit or re-enable listing source lines which have no location fields

.NOLOC *exp*

Either inhibits or permits listing of lines lacking a location field. If the value of *exp* is not zero, listing is inhibited; if the value of *exp* is 0, listing occurs. In the default condition listing occurs.

The value of .NOLOC is the value of the last *exp*.

Example

This is the source program.

```
.NOLOC   0
.NREL    1
.TXT     "ABCDEF"    ;This prints.
.NOLOC   1           ;This does not
                     ;print.
.TXT     "GHIJ"      ;This line prints,
                     ;but the second
                     ; line does not.
LDA      0,TEMP      ;This prints
.LOC     .+12        ;This will not
                     ;print.
STA      0,TEMP      ;This prints.
.END                 ;This will not
                     ;print.
```

This is the program listing.

```
      000000 .NOLOC   0
000001000001 .NREL    1
000011040502 .TXT     "ABCDEF"   ;This prints.
      041504
      042506
      000000
000051043510 .TXT     "GHIJ"     ;This line prints,
                                 ;but the second
                                 ;line
                                 ;does not.
000101020411 LDA      0,TEMP     ;This prints.
000231040411 STA      0,TEMP     ;This prints.
```

.NOMAC

Inhibit or re-enable listing macro expansions

.NOMAC *exp*

Either inhibits or permits the listing of macro expansions. If the value of *exp* is not zero, then macro expansions are inhibited; if the value of *exp* is zero, macro expansions will be listed. Listing occurs in the default condition. .NOMAC can also be used within a macro to selectively inhibit listing.

The value of .NOMAC is the value of the last *exp*.

Example

```
                    .MACRO  TEST
                    5
                    6
                    .NOMAC  1       ; You can inhibit
                                    ;or permit macro
                    7               ;expansion at any
                                    ;time within the
                                    ;macro
                    3
             %
                    TEST
00000 000005        5
00001 000006        6
                    .END
```

Example

```
             .MACRO   OR         ;OR is the name
                                 ;of the macro.
             COM      ↑1,↑1
             AND      ↑1,↑2
             ADC      ↑1,↑2
             %
      000001 NOMAC    1          ;Inhibit the macro
                                 ;expansion.
             OR       1,2
      000000 .NOMAC   0          ;Permit the macro
                                 ;expansion.
             OR       3,0
00003 174000 COM      3,3        ;Here the macro is
                                 ;expanded with
00004 163400 AND      3,0        ;arguments
                                 ;inserted.
00005 162000 ADC      3,0
```

Example

You can use **.NOMAC** as a value as shown in this example.

```
.MACRO      BUSH
**.PUSH     .NOMAC
**.NOMAC    1
.
.
.
**.NOMAC    .POP
%
```

Suppose that in the program which calls the macro **BUSH** you must preserve the original value of **.NOMAC**. The macro, however, suppresses its own listing by setting **.NOMAC** and using the ** atom. It is convenient to save the original value of **.NOMAC** on the stack. At the end of the macro definition the original value of **.NOMAC** is restored by popping it off the stack.

.NREL

Specify pure or impure code relocation

.NREL *[exp]*

Causes subsequent source statements to be assembled using either pure code relocation or impure code reloaction. If *exp* is not present, then the Macroassembler uses the location counter for impure code relocation.

If present, the optional *exp* is evaluated. If the result is zero, the Macroassembler uses the location counter for impure code relocation; if non-zero, the Macroassembler uses the location counter for pure code relocation.

Example

```
            000000          .NREL       0
   00000'000123 EX1:        123
            000001          .NREL       1
000001000000' EX2:          EX1
```

.OB

Name an object file

.OB *filename*

Names the object file, *filename*, that is the output of assembly. If more than one **.OB** pseudo op occurs in the source, the extra **.OB**s will be flagged with an **M** (multiply defined symbol) and the object file will have the name given in the first pseudo op encountered.

If an /**N** function switch appears in the **MASM** command line, denoting that no object file is to be created, the **.OB** pseudo op is ignored. If a /**B** function switch appears in the **MASM** command line, the **.OB** pseudo op will be overridden and the object file will have the name given by the /**B** switch. The procedure for naming an object file is thus:

Precedence	Object File Name
First	/**B** name
Second	**.OB** name
Third	Default name (first source file name in the **MASM** command line that is not followed by the /**S** switch)

One of the primary uses of **.OB** is in conditional assembly code when alternative file names are to be used, as for example, slightly different versions of a single **.OB** file name. The default object file extension **.OB** will be appended to the specified name if it does not already have a **.OB** extension.

Example

```
        .
        .
        .
.IFE    MSW
.OB     SYSTEM      ;Creates SYSTEM.OB
.ENDC
.IFN    MSW
.OB     MSYST       ;Creates MSYST.OB
.ENDC
```

.PASS

Number of assembly pass

.PASS has a value of zero on pass 1 and a value of one on pass 2 of assembly.

Example

This example defines several parameters for later use in the assembly. Since the value remains constant and no code is generated, there is no need to assemble these lines during pass 2. (This is similar to using the /S switches in the **XEQ MASM** command line.

```
.IFE    PASS        ;Define stack frame offsets
.DUSR   ?OAC0=-4    ;Old AC0
.DUSR   ?OAC1=-3    ;Old AC1
.DUSR   ?OAC2=-2    ;Old AC2
.DUSR   ?OFP=-1     ;Old frame pointer
.DUSR   ?ORTN=0     ;Carry and return
.ENDC
```

.POP

Pop the value and relocation of last item pushed onto stack

The value of **.POP** is the value and relocation property of the last expression pushed onto the stack (see the **.PUSH** pseudo op). In addition, **.POP** removes the value and relocation property from the top of the stack.

If there are no values on the stack, **.POP** has a value of zero with absolute relocation and an **O** (overflow flag) will appear on the line in which the **.POP** occurred.

Example

```
      000025  A=25          ;Set A to 25
00000 000025  A             ;Show A's present
                            ;value
      000025  .PUSH   A     ;Push value of A
                            ;onto stack
      000015  A=15          ;Assign A a new
                            ;value
00001 000015  A             ;Show A's present
                            ;value
      000025  A=.POP        ;Assign A the
                            ;value of a
                            ;stack word
00002 000025  A             ;Show A's present
                            ;value
              .END
```

.PUSH

Push a value and its relocation onto a stack

.PUSH *exp*

Allows you to save the value and relocation properties of any valid assembler expression on the assembler stack. Additional expressions may be pushed until the stack space is exhausted (at which point the Macroassembler will issue an overflow flag (**O**)). You reference the stack with the permanent symbols **.POP** and **.TOP**. As with any push-down stack, the last expression pushed is the first expression to be popped.

Example

```
000010  .RDX    8
000010  .PUSH   .RDX
000012  .RDX    10
000010  .RDX    .POP
```

.RDX

Radix for numeric input conversion

.RDX *exp*

Defines the radix to be used for numeric input conversion by the assembler. *Exp* is evaluated in decimal and must be in the range between 2 and 20. The numeric value of **.RDX** is the current input radix. The default input radix is 8.

Example

```
                                    ;Assume the output
                                    ;radix is 8.
      000010 .RDX    8
00000 000123 123
      000012 .RDX    10
00001 000173 123
      000020 .RDX    16
00002 000443 123
00003 000020 (.RDX)                 ;List the current value
                                    ;of the input radix.
```

.RDXO

Radix for numeric output conversion

.RDXO *exp*

Defines the radix to be used for numeric conversion by the assembler. *Exp* is evaluated in decimal and must be in the range of 8 to 20.

The numeric value of .RDXO is always expressed in the listing as 10. (**.RDXO**) represents the value of .RDXO. The default output radix is 8.

Example

```
000012 .RDX    10        ;Input radix is 10
 00010 .RDXO   10        ;Output radix is 10
 00077 77
 00022 22
 00045 45
 00008 .RDX    8         ;Input radix is 8
000010 .RDXO   8         ;Output radix is 8
000077 77
000022 22
000045 45
000020 .RDX    16        ;Input radix is 16
  0010 .RDXO   16        ;Output radix is 16
  0077 77
  0022 22
  0045 45
000010 .RDXO   8         ;Input radix is 16, output
                         ;radix is 8
000167 77                ;Hex input, octal listing
000042 22
000105 45
 00010 .RDXO   10        ;Input radix is 16, output
                         ;radix is 10
 00119 77                ;Hex input, decimal listing
 00034 22
 00069 45
000010 (.RDXO)           ;Current value of output
                         ;radix always prints as 10
```

.REV

Set the revision level

.REV *maj_rev min_rev*

Identifies the revision level of a program. You enter the major and minor revision levels as numbers in the current radix or as legal expressions. Revision levels are carried into the object file and then into the program file. Both the major and minor revision levels have a numeric range of 0-255_{10}. This pseudo op may appear anywhere within the source module.

If two or more object modules containing revision numbers are to be bound into a program, the Binder chooses the revision level for the program file as follows:

- If a program file has the same name as an OB file, the OB revision level will be carried over.
- Otherwise, revision level information will be selected from the first OB bound that contains such information.
- If none of the modules being loaded contains revision information, the program file will be assigned major and minor revision numbers of 255 each.

For example, assume that SCHED.OB, IODRIV.OB, and DISP.OB are to be bound into SCHED.PR. If SCHED.OB contains revision information, that revision information will be passed to the program file. If SCHED.OB does not contain revision information, the revision information contained in either IODRIV.OB or DISP.OBL will be passed, depending on which is bound first.

Use the CLI **REV** command to obtain revision information of a program file.

```
.TITL    QUIZ
.ENT     S1,S2
.EXTN    S3
.REV     12,1        ;Revision data is in octal (default
                     ;input radix)
```

.TITL

Entitle an object program

.TITL *usym*

Supplies a title for the assembly and error listings. Also supplies a name (internal to the OB file) for library reference, and which the Binder will require.

Example

```
.TITL    CHEEZ
```

.TOP

Value and relocation of top stack word

.TOP

.TOP has the value and relocation property of the last expression pushed onto the variable stack. **.TOP** differs from **.POP** in that the symbol does not pop the last pushed expression from the stack. If you have not pushed any expressions before you issue a **.TOP** pseudo op, the Macroassembler returns zero (absolute relocation) and flags the **.TOP** statement with the overflow flag (**O**).

Example

```
.PUSH     .RDX
.RDXO     .TOP          ;Set output radix equal
.                       ; to input radix.
.
.
.RDX      .POP          ; Restore the stack.
```

.TSK

Declare a number of tasks

.TSK *number of tasks*

Specifies how many TCBs are to be available for initiating a multitask environment within a given program. You can override the number specified by the pseudo op by specifying an alternate number with the **/TASKS**=*x* switch in the Binder command line. If several modules making up a program contain **.TSK** declarations, the Binder uses the largest value.

Example

```
.TSK      10            ;Reserve 8 TCBs.
```

.TXT, .TXTE, .TXTF, .TXTO

Specify a text string

.TXT **string**

.TXTE **string**

.TXTF **string**

.TXTO **string**

These pseudo ops cause the assembler to scan the input following the character * up to the next occurrence of the character * in string mode. The character * may be any character not used within the string except null, any New-line character, any space character, or Rubout; * delimits but is not part of the string. You may use a New-line character to continue the string from line to line or page to page, but this character is not stored as part of the text string.

Every two bytes generate a single storage word containing ASCII codes for the bytes. Storage of a character of a string requires seven bits of an eight-bit byte. The leftmost (parity) bit may be set to 0, 1, even parity, or odd parity as follows:

.TXT	- sets leftmost bit to 0
.TXTF	- sets leftmost bit to 1 unconditionally
.TXTE	- sets leftmost bit for even parity on byte
.TXTO	- sets leftmost bit for odd parity on byte

The packing mode can be altered using the **.TXTM** pseudo op. If an even number of bytes are assembled, the null word following these packed bytes can be suppressed by the **.TXTN** pseudo op.

Within the string, you can use angle brackets (<>) to delimit an arithmetic expression. The Macroassembler will evaluate the expression and mask it to eight bits. (This means you can set the parity bit for a **.TXT** string here. The Macroassembler masks out the parity bit for **.TXTE**, **.TXTO**, and **.TXTF** strings later.) Angle brackets are the only means, for example, to store a New-line character as part of the text string (see example). Note that you cannot use logical operators within the expression.

In the default condition, bytes are packed left/right, and a null byte is generated as the terminating byte.

Example

```
041101   .TXT    #AB<12>CDE#
                             ;All the examples
                             ;will assemble.
043012
042504
041101   .TXTE   $AB CD$
141640
000104
141301   .TXTF   @AB CD@
141640
000304
141301   .TXTO   EAB CDE
041440
000304
```

See the **TXTM** entry for other examples.

.TXTM

Change text byte packing

.TXTM *exp*

Changes the packing of bytes generated using the text pseudo ops, **.TXT**, **.TXTE**, **.TXTF**, or **.TXTO**. If *exp* evaluates to zero, bytes are packed right/left; if *exp* evaluates to non-zero, bytes are packed left/right (default condition).

The value of **.TXTM**, expressed as **(.TXTM)**, is the value of the last *exp*.

Example

```
      000000  .TXTM     0
00000 041101  .TXT      #AB CD#     ;Pack bytes right
                                    ;to left
      041440
      000104
00003 000000 (.TXTM)                ;The current value
                                    ;of .TXTM is 0
      000001  .TXTM     1           ;Pack bytes left to
                                    ;right
00004 040502  .TXT      #AB CD#
      020103
      042000
00007 000001 (.TXTM)                ;The current value
                                    ;of.TXTM is 1
```

.TXTN

Determine text string termination

.TXTN *exp*

Determines whether or not a string that contains an even number of characters will terminate with a word consisting of two zero bytes. (When the number of characters in the string is odd, the last word contains a zero byte in all cases.)

If *exp* evaluates to zero, all text strings containing an even number of bytes will terminate with a full word zero (the default condition). If *exp* evaluates to non-zero, any text string containing an even number of bytes terminates with a word containing the last two characters of the string.

The value of **.TXTN**, expressed as **(.TXTN)**, is the value of the last *exp*.

Example

```
      000000  .TXTN     0
00000 030462  .TXT      /1234/      ;End string with a
                                    ;word of zeroes
      031464
      000000
00003 000000 (.TXTN)
      000001  .TXTN     1
00004 030462  .TXT      /1234/      ;Do not add
                                    ;zeroes to this
                                    ;string
      031464
00006 000001 (.TXTN)
```

.XPNG

Remove all non-permanent macro and symbol definitions

.XPNG

Deletes the symbol table file and recreates an empty symbol table file with the same name as the deleted one. **XPNG** is used primarily as follows:

- You write a source file containing **.XPNG** followed by definitions of any semipermanent symbols.
- You assemble the source file using the **/S** function switch with the **MASM** command line. This causes the assembler to stop the assembly after pass 1 and save the symbols in MASM.PS.
- You can then use the new symbol table containing the semipermanent symbols defined in step 2.

Example

```
        .TITL     XP
        .XPNG
020000  .DMRA     LDA=20000
040000  .DMRA     STA=40000
        .END
```

Example

The CLI command form is: X MASM/S XP<↓>

The Macroassembler symbol table now contains LDA and STA.

.ZREL

Specify lower page zero relocation

.ZREL

Assembles subsequent source lines using addresses extending from 50_8 to 377_8. If you exit **.ZREL** mode during assembly, the assembler will maintain the current **.ZREL** value and use that value if you enter **.ZREL** mode again.

Example

```
00000 000000 AL:   0
                   .ZREL
00000-000000 Z:    0
00001-000000 ZL:   0
     000100        .LOC     100
00100 000000       AL
                   .ZREL
00002-000000       AL
```

Part 3
MP/OS System Reference

Chapter 1
Introduction

System Overview

The MP/OS system is a general purpose operating system for the MP/computers. It provides features such as multitasking, memory management, and device independent I/O, usually associated with larger computer systems.

The MP/OS system can be used either for general purpose systems oriented toward program development, or for smaller stand-alone applications such as real-time process control. You can generate a MP/OS system containing a desired subset of the full system's power, and tailor it to any configuration of memory boards and peripherals. You can put all software in read-only memory (ROM) to eliminate the need for mass storage, or you can take advantage of the powerful MP/OS file management system for storing large amounts of data on disks.

Programs communicate with the system through calls that you place in the program code. This section of the manual (Part 3) describes the system's facilities and the system calls that apply them. The remainder of this chapter describes the major MP/OS facilities and basic programming concepts.

Program Management

The MP/OS system controls user programs with a stack structure, which enables programs to call other programs in a nested fashion. Programs can pass messages to their parent and descendant programs. A program may also be interrupted, either by system call or keyboard command, and its state saved for resumption at a later time. Programs can examine and change memory allocation at run time, and you can cut down on a program's memory needs by using overlaying, a technique in which little-used parts of a program are kept in a disk file until they are needed. Timing control is provided by a time-of-day clock and a delaying function that allows a program to suspend its execution for a specified period of time.

File Management

The system supports a hierarchical file structure for disk devices. Files may be referenced by symbolic names, and all input/output devices in the system also have names, so that a program may easily be written to communicate with any device or file. You have control over file element size (clustering of disk blocks), file protection, and other attributes of a file, including some you may define and use in any manner you find useful.

Input/Output

A MP/OS program may communicate with up to 16 files or devices at one time, using software-defined data paths called I/O channels. System calls permit reading or writing of a specified number of bytes, or a line of bytes terminated by a delimiter. You can use file positioning system calls for random access to data in disk files. I/O channels may be passed unaltered from one program to another, so that a program may open channels which can then be used for I/O by a descendant program.

I/O Device Management

There are system calls that enable you to introduce a disk unit to the system, and to release it from system control in order to mount a new disk on the drive. MP/OS also supports a variety of options for console devices, and recognizes certain control characters for input editing and interrupt functions.

You can use custom or special purpose I/O devices under the MP/OS system by defining interrupt handling routines for them. This gives you a high degree of control over interrupts and fast response to them.There is no need to modify the MP/OS software to handle your own devices.

Multitasking

Multitasking is a powerful technique that enables you to divide a program into a number of subprograms called tasks. The system provides a scheduling routine, which switches control among the tasks to create the appearance of parallel processing. Multitasking can greatly simplify the code for any program which must "do several things at the same time".

System calls are available to create or delete tasks at run time. As many as 255 tasks can be active at once. You can assign variable priorities to tasks, giving you control over which one runs in response to a given event. You can also turn multitasking off with a system call, to ensure that a critical operation is performed without interruption from other tasks. Tasks may use system calls to communicate with each other and to suspend or resume their operation.

Cross-Development on AOS Systems

AOS is a powerful multiprogramming system that runs on Data General's ECLIPSE-line computers. Since MP/OS system calls are a functionally compatible subset of those on AOS systems, you can run a program on either system if it uses only MP/OS calls. But the two systems are different internally, so you need translating software to run MP/OS programs on AOS systems.

The translating software consists of object files (provided by Data General in the form of a library) which are bound along with your program. The result is an AOS-format program file. This process enables you to move programs from one type of system to the other with little or no rewriting. The specific procedures are described in Appendix J.

Chapter 2
Programming with the MP/OS System

Introduction

System Calls

System calls are the means by which a program communicates with the system. A system call is like a subroutine call, except that the called routine is part of the MP/OS supervisor instead of your program. You code a system call in your program in the same manner you would use an instruction. All the system calls have mnemonics recognized by the assembler. There are also mnemonics for numbers and flags used in programming the system calls.

Normal and Error Returns

A system call is like a conditional instruction in that the word following the call in your program may be skipped under certain conditions. The system uses the convention that, if the system call executes properly, the next word is skipped. If the system encounters some error in trying to execute the call, the next word is executed. The word following a system call in your program is called the *error return;* the word after the error return is the *normal return.* When your program takes an error return from a system call, the system places a number in AC0 identifying the cause of the error.

The system provides a file containing error messages that correspond to all the error codes. You can use the **?ERMSG** library routine to read messages from this file for use by your program. See Appendix **** for a complete list of the error codes and their messages.

Accumulator Usage

System calls generally require arguments, called *inputs,* which your program must place in the proper accumulators before executing the call. Some system calls also return *outputs* in accumulators. Only AC0, AC1, and AC2 are used for inputs and outputs; the system always sets AC3 to the value of the frame pointer on return from a call. Any accumulators not used for outputs are returned to your program unchanged.

MP/OS system calls and library routines observe the following conventions for accumulator usage:

- Input/output calls use AC0 for the I/O channel number.
- Multitasking calls use AC2 for the task identifier.
- Calls that reference files use AC0 for the byte pointer to the pathname.
- Calls that require packets (see below) use AC2 for the packet address.

Packets

Some calls require more information than the accumulators alone will hold. In this case you pass additional arguments to the call in a *packet.* A packet is simply a block of consecutive words in memory. The number of words depends on the particular call; there is a mnemonic for each packet size. The first word of every packet contains a number indicating what type it is; there is a mnemonic for each type. The system checks this number for validity when handling the call.

Stacks

A MP/OS program must have a stack area in memory for use by system calls. The stack size must be equal to or greater than the value of the mnemonic **?STKMIN**. You can initialize the stack control words by using the assembler's **.LOC** directive. When a program starts, the contents of locations 40_8 and 41_8 are placed in the stack pointer and frame pointer, respectively. You must also initialize location 42_8 with the stack limit, and you may initialize location 43_8 with the address of a *stack overflow handling routine.*

The stack overflow handling routine will be called by the system if your program attempts to exceed the specified stack limit. The routine may perform functions such as allocating more memory, or simply shutting down the program. Before calling the routine, the system pushes five words onto the stack, whose contents (in the order pushed) are:

- The accumulators (AC0 through AC2).
- The frame poiner.
- A word containing the carry in bit 0, and the contents of the program counter (where the overflow occured) in bits 1-15.

  NOTE: *Since the system's handling of a stack overflow involves pushing more words onto the stack, you should make sure that your stack is actually five words larger than the size that you specify in the stack limit word. Otherwise, part of your program code may be destroyed during the handling of the overflow. You should also allow for any stack space that may be needed by the overflow handling routine itself.*

If your program uses multitasking, then each task must have its own stack area. In this case, the system maintains the stack control words so that each task always has the proper values for them.

The following sequence of instructions can be used by a stack overflow handling routine to return control to the main program. (Presumably you would only do this if you had already remedied the overflow condition.)

```
; RETURN TO MAIN PROGRAM
        POPA   0          ; get PC and carry
        MOVL   0,0        ; rotate carry out
        MOVR   1,1        ; and save it in AC1
        MOVZR  0,0        ; clear bit 0 of PC word
        STA    0,TEMP     ; and save in memory
        MOVL   1,1        ; restore carry
        POPA   3          ; now restore all ACs
        POPA   2
        POPA   1
        POPA   0
        JMP    @TEMP      ; and jump back
        .
        .
        .
TEMP:  0
; NOTE: if your program is multitasked,
; you should be sure that no other task
; modifies TEMP while this task
; is using it.
```

Table 2.1

Naming Conventions

All symbols containing a ? are reserved for use by the system. All symbols starting with **ER** are reserved for error codes.

All mnemonics for system calls, library routines, error codes, and other symbols used in this manual are defined in the file MASM.PS, the assembler's permanent symbol table. Many of them are defined in the user parameter file, MPARU.SR, which is described in Appendix I of this manual. You can refer to the parameter file if you need to know the value of one of these symbols; usually, though, you will simply use the mnemonics in your program without needing to know their values.

Mnemonics which represent status flags have values such that the named bit is set to 1 and all other bits are 0; thus you can use the mnemonic's value in a logical AND to determine the setting of the flag. You can also code an assembler expression containing the sum of several mnemonics as a way to set several flags at once.

System Call Options

Some system calls have *options* you may specify to modify the action of the call. Options are specified by two letter abbreviations, which you code after the call's mnemonic in your program. For instance, if you want to create a disc file with the **?CREATE** call, and you wish to delete any existing file with the same name, you can use the delete (**DE**) option by coding **?CREATE DE**. You can specify more than one option by separating their abbreviations with commas.

Non-Pended Calls

Some system calls, notably those that perform I/O, can take a relatively long time to execute. Normally your program (or the calling task in a multitasked program) is suspended from running during this interval, resulting in a loss of potentially useful processor time. In order to eliminate this waste, you can use *non-pended* system calls.

You specify a non-pended call by coding the **NP** option on any system call allowing it. When you execute the call, instead of suspending your program, the system creates a new task and assigns it the job of executing your call. Your program is free to continue operation. Obviously, you cannot assume that the results of the system call are valid; for instance, if you read data with a **?READ NP** system call, you must still wait for the data to arrive before you can operate on it. However, you can perform other types of computation while waiting for the new data.

To find out when the non-pended call is complete, you must execute an **?AWAIT** system call. This enables you to either check the call's progress, or to suspend your program until the call is complete. You must issue an **?AWAIT** to obtain the results of every non-pended system call that you execute; otherwise system memory space will be wasted.

Library Routines

The system provides a number of convenient functions that some users will be willing to do without, for the sake of speed and compactness. These functions are implemented as *library routines* instead of system calls. Library routines are called in exactly the same manner as system calls; the only difference is that the code implementing the function will be part of your program, instead of being in system memory. The library routines are described in detail in Chapter 4.

Program Management

You can perform the following functions with system calls:

- Transfer control from one program to another.
- Pass a message between programs.
- Create a restartable program file *break file* containing the complete state of an interrupted program.
- Examine or modify a program's memory allocation.
- Manipulate overlays to reduce a program's size.
- Determine the current time, and control program timing.
- Restart the system.

Program Level

Program management in the MP/OS system is largely based on the concept of *program level*. The system maintains information about programs in a stack that your program can manipulate, using system calls. The following examples illustrate this concept.

> **NOTE:** *The MP/OS program stack is eight levels high. For the sake of brevity, only the first four levels are shown in the examples.*

Figure 2.1 represents the state of the program stack when the system is started up. As soon as the system is initialized, it invokes the MP/OS Command Line Interpreter (CLI). The CLI is the first program to enter the stack: it is at *program level 1*.

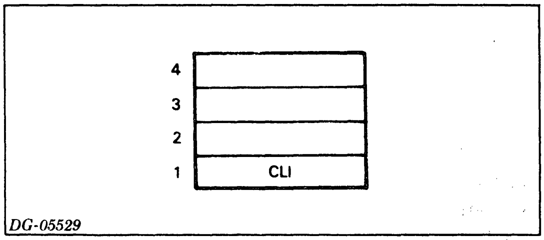


Figure 2.1

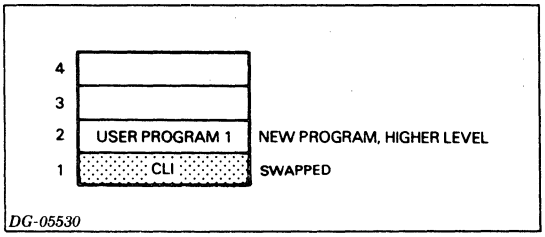


Figure 2.2

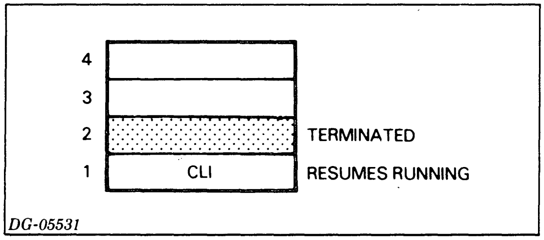


Figure 2.3

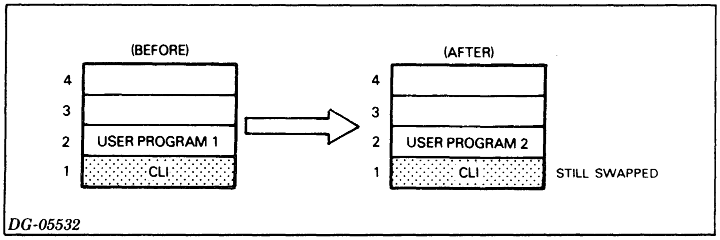


Figure 2.4

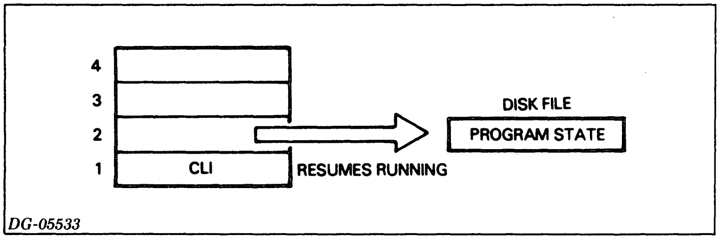


Figure 2.5

NOTE: *If you wish to have some other program run at level 1, you can place it on the system master disk (see "File Management") with the name CLI.PR. You should be aware that the system passes an inter-program message (explained shortly) to the level 1 program, which consists of the string "CLI.PR, LOGON". This causes the MP/OS CLI to look for and execute a macro called LOGON.CLI. (For more information on the CLI, see MP/OS Utilities Reference, Section 1, "Command Line Interpreter".)*

When you want to execute a program, you type a CLI **XEQ** (execute) command. The CLI performs an **?EXEC** system call, which causes the CLI to be swapped. All information on its operating state is saved in the stack and on disk, and your program begins executing at *level 2.*

This operation is called a program *swap.* The CLI is the *parent* program, and your program is its *descendant.* The state of the parent program is saved in a disc file, which is normally transparent to you. However you should be aware that **?EXEC** will give an error return if there is not enough disk space left for the file.

When a program finishes running, it executes a **?RETURN** system call. The system removes it from the stack and returns control to the parent program (the CLI).

Alternatively, your program can transfer control to a companion program, causing it to execute at the same level.

This operation is called a program *chain.* The difference between a swap and a chain is that, in a chain, the state of the calling program is lost. Swaps and chains are both performed by the **?EXEC** system call.

Upon termination, your program may request that the system preserve its operating state, so that it can resume execution at a later time. You can use the *break* function to save its state in a disk file before returning to the parent program. To perform this function, you specify the **BK** option on the **?RETURN** system call.

You can resume execution of the *break file* at some later time with the CLI **XEQ** command or **?EXEC** system call. In addition to using **?RETURN BK**, you can cause a program break at any time by typing a CTRL C CTRL E sequence on the console keyboard (see "I/O Device Management").

You must observe certain cautions when restarting a break file. Any files in use by the program must be in the same state that they were when the break file was created. Also, break files cannot be transported to other MP/OS systems the way regular program files can.

Parallel Call Errors

A conflict may arise in a multitasked program if one task executes an **?EXEC** or **?RETURN** while some other task has a system call in progress. A similar situation may occur, even in a single-task program, if you interrupt the program from the console. In these cases, any outstanding calls will be aborted, and they will give an error return with code **ERPCA**. Note that if a break file is produced, the error return will not occur until execution is resumed.

Debugger Starting Address

The MP/OS Debugger is an interactive routine which permits you to examine and modify your program and its data during execution. To use the Debugger you request the Binder program to include it in your program. It will be bound into your program file, where it resides without affecting the program until you call it. To call the Debugger, you start your program at the Debugger starting address, either by an **?EXEC** call or by the CLI **DEBUG** command. Complete information on the Debugger is contained in Section 5 of the *MP/OS Utilities Reference* (DGC No. 093-400002)

Inter-Program Communication

MP/OS programs can send messages to each other with an option on the **?EXEC** and **?RETURN** system calls. They can receive messages with the **?GTMSG** call.

The system maintains a buffer which holds one message at a time. An **?EXEC** or **?RETURN** which does not pass a message causes the buffer to be cleared, so you must read the message before executing either of these calls.

A message may be up to 2047 bytes long. Its format is entirely up to the user. However, a standard format is used for messages from the CLI, and there is a **?TMSG** library routine that translates CLI-format messages. To ensure compatibility between programs, you will find it convenient to use this message format, which is described in detail in Appendix H.

I/O Channel Status Passing

An important feature of the MP/OS system is its ability to pass I/O channels between programs. An I/O channel (see "Input/Output", below) is a system-defined data path between your program and an external device or disk file. When a program performs an **?EXEC**, the states of any active I/O channels are passed to the new program. The new program may perform input and output on these channels without having to reopen them.

I/O channel status is passed to a descendant program, and to another program at the same level (by a chain), but not to a parent. When a program executes a **?RETURN**, the parent resumes execution with the same I/O status it had when it performed its **?EXEC**.

Since the existence of open I/O channels can be troublesome to programs that do not need them, the system provides a **?RESET** system call, which closes any or all channels. It is good practice to start all MP/OS programs with a **?RESET** to close any channels the program does not expect to receive from its parent. (For more details on **?RESET** see "Input/Output" below.)

Memory Management

MP/OS programs are divided into two main data areas: *pure* and *impure*. The pure area consists of code and data that are never modified during the program's execution. Separating this part of the program from the rest enables the system to increase its efficiency, since the pure area does not need to be saved when the program is swapped. It can be recovered from the program file when the program resumes execution. You control the partitioning of your program with the assembler **.NREL** directive.

Since the MP/OS system and your program both occupy the same address space, there is a potential danger that your program could accidentally overwrite part of the system. This problem is most prevalent in programs that acquire and release segments of impure memory during execution. To minimize this danger, the system provides the **?MEMI** call, which allows programs to acquire and release memory in an orderly manner, and the **?INFO** call, which provides the program with information about its memory usage.

You will use either of two basic memory organizations, depending on whether you are working with a program that will be loaded into read-write memory (RAM), or one that will be permanently stored in read-only memory (ROM).

Figure 2.6 shows the organization of *lower page zero* (locations 0-377_8), which is the same for both types of systems. Locations 0-15_8 and 17_8 are reserved for the system (some of these have special functions for the CPU). Location 16_8, **?USP**, is a general-purpose word whose contents are unique to each task in a multitasked program; i.e., if there is more than one task in your program, the system maintains a separate copy of this word for each task. Locations 20-37_8 are the auto-increment and

-decrement locations, all of which are available to your program. Locations 40_8 and 41_8 hold values which the system loads into the stack pointer and frame pointer, respectively, when the program starts. Locations 42_8 and 43_8 hold values for the stack limit and the stack overflow handling address. You may initialize these words, but you may not alter them during execution. Locations $44\text{-}47_8$ are reserved for the system. The rest of lower page zero (locations $50\text{-}377_8$) is available to your program for general use. You can specify these locations with the assembler's **.ZREL** directive.

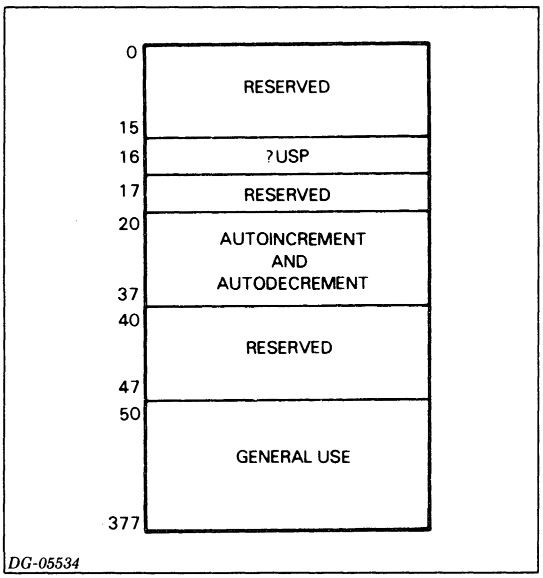


Figure 2.6

Figure 2-7 shows the contents of the rest of memory for a typical program development system, which consists entirely of read-write memory (RAM). Your program's pure area is placed in memory starting at location 400_8. The impure area is placed above the pure area. The MP/OS code and data are placed at the top of memory.

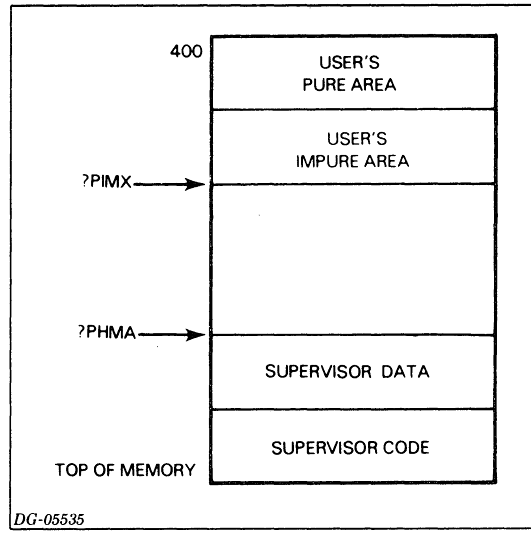


Figure 2.7

For a stand-alone system (one which is dedicated to a single application), you will probably want to put all pure code in read-only memory (ROM). In this case, memory is allocated in separate RAM and ROM segments, as shown in Figure 2.8. Your pure code and the MP/OS code are placed together so that they may occupy a single ROM board. The system's data areas, as well as your program's impure area, are placed on a RAM board that also contains lower page zero.

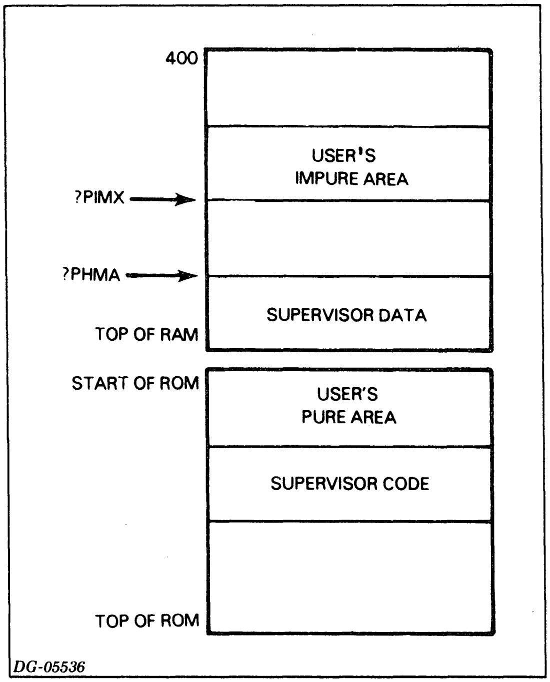


Figure 2.8

For either type of system, there is usually an area of free memory between the MP/OS system's data and your program's data. The **?PIMX** word (which you can read with the **?INFO** call) contains the address of the highest word in use by your program. The **?PHMA** word (also accessible by **?INFO**) contains the address of the first free word below the system's data.

When your program uses **?MEMI** to request more memory, the system increases **?PIMX**. When the MP/OS system needs more memory, it decreases **?PHMA**. An error is indicated if either your program or the supervisor attempts to acquire memory which is already in use. To prevent this (and to minimize the time for a program swap), you should always use **?MEMI** to release memory when you no longer need it.

For information on how to generate MP/OS systems for different applications, see Appendix L.

Overlays

Overlaying is a technique that allows you to reduce the memory requirements of a program by taking advantage of the fact that, in a typical program, much of the code is used only infrequently. Overlaying allows you to divide up your program so that routines can reside on disk until they are actually needed.

To use overlaying, you form one or more *overlay nodes* in your program. An overlay node is a block of memory which may contain one of several different *overlays*. An overlay is a routine, or a group of routines. Each node has its own set of overlays, and can hold one overlay at a time.

The MP/OS overlay facility is designed to be flexible. The exact distribution of nodes and overlays is not specified until bind time, so no program modification is needed to try out several different strategies. This makes it easy to reorganize the overlays for greatest efficiency.

The system manages overlays by a combination of assembler directives, Binder commands, and library routines. The assembler generates object modules containing overlay information, and creates special symbols called *overlay descriptors*. The Binder links together the various inter-module references and allocates space for the overlay nodes. It also builds an *overlay file* containing all the overlays, and places overlay control information in the program file. The overlay file has the same filename as the program, but with a suffix of **.OL** instead of **.PR**.

At run time, your program uses the **?OVLOD** and **?OVREL** routines to load and release overlays.

The system keeps track of which overlays are currently loaded. If your program is multitasked, then several tasks may share a node if they all request the same overlay. If a task requests a new overlay in a node that is already in use by another task, then the requesting task is blocked from execution until the node becomes available.

This handling of overlays under multitasking ensures that the nodes are correctly managed in a way that is transparent to all tasks, provided that your program observes the following conventions:

- All tasks use the **?OVLOD** and **?OVREL** routines to call and release overlays.
- Tasks that use overlays can tolerate some delays when calling them. (You can minimize these delays by optimizing your overlay structure at bind time.)

Program Revision Number

The system maintains a revision number in every program file, to help you keep track of different versions of a program. This number consists of a major and minor revision number, each of which may range from 0 to 255. You may set the number with the **.REV** assembler directive or the Binder **/REV** switch. You may read it with the **?INFO** call. You may also use the CLI **REVISION** command to read or set the number.

Real-Time Support

The MP/OS system provides several facilities for real-time and other programs which must accurately time their operations. The system keeps track of the current date and time, and the **?GTIME** call enables you to read these values, encoded in two 16-bit words. You can use the **?CDAY** and **?CTOD** library routines to convert the encoded value into an easy-to-use form. There is also an **?STIME** system call that enables you to change the system time without shutting down and restarting the system.

You can suspend operation of your program for a specified period of time with the **?DELAY** routine. If your program uses multitasking, you can suspend some tasks while others continue to run. Highly accurate timing is possible, since delay times are expressed in milliseconds. You can use the **?MSEC** library routine to calculate the time. The system measures time with as much accuracy as the CPU's real-time clock permits.

System Restart

The **?BOOT** system call enables you to shut the system down in an orderly manner, ensuring that no data is lost. You can also use **?BOOT** to restart the system by reading a bootstrap loader from a disk.

File Management

The MP/OS file system provides you with simple, efficient ways to communicate with input/output devices, and to store and retrieve data in files. All devices and files are handled by the same system calls; this makes it easy to write device independent programs.

Basic Organization

A MP/OS file is either an I/O device, such as a printer, or a collection of data stored in a disk file. Since both kinds are handled identically, we will use the term *file* to mean either one.

A file is referenced by its *filename*, which is a string of 1 to 15 characters. Legal characters in filenames are:

- The letters a to z and A to Z. You may use upper and lower case interchangeably; the system considers them to be equivalent, and uses only upper case internally.
- The digits 0 to 9.
- Punctuation marks ?, $, _, and .(period).

The symbols :, @, =, and ↑ may also be used to specify a file. Their meanings are discussed below.

In general, the contents of a file are entirely up to the user; however, the system recognizes several types of files as having special significance. In particular, there is a type of file called the *directory*, whose contents are other files.

Any file in a directory may itself be a directory containing more files. A directory contained in another directory is called a *subdirectory*. This *nesting* of directories may continue indefinitely.

Files within directories are referenced by using the : character. For example, X:Y references a file named Y in a directory named X. An expression of this form is called a *pathname*. Pathnames are explained in detail below.

The Device Directory

There is a special directory in every MP/OS system called the *device directory*. Its filename is the special symbol @. The device directory is the "highest" directory in the system: the one which contains all the others.

The filenames in the device directory correspond to all the input/output devices in the system. When you wish to reference a device, you use its name prefixed by @. The name may be followed by a one- or two-digit unit number. Typical device names are @LPT for a line printer or @DPD0 for a disk drive.

Since the device directory contains all I/O devices including disks, it obviously cannot be contained on any device. The device directory is unique in the system in that it has no physical representation on any disk. It is actually a table in MP/OS memory space.

Root Directories

Every disk device has a *root* directory which is the highest directory on the device. All other files on a disk are contained in its root. The root directory is referenced by appending a : to the device name, e.g., @DPD0:.

System Master Device

One disk unit in every MP/OS system is designated the *system master device*. This is the unit on which the MP/OS system program files reside. It often holds many other commonly used files. To make it easy for you to reference this device, MP/OS accepts the : character as a prefix which refers to the root directory of the system master device. For example, if your system's master device is @DPD0, then the pathname :CLI.PR is equivalent to @DPD0:CLI.PR.

Pathnames

The system allows one filename to be used simultaneously for several files in different directories. Therefore there is a need for a way to reference any file uniquely. This capacity is provided by *pathnames*. As its name suggests, a pathname represents a path through the directory structure to a particular file.

A pathname consists of a series of filenames separated by colons (:). Pathnames may be up to 127 characters long. All the filenames except the last one must be directories, and each directory named must be a subdirectory of the preceding one. For example, the pathname A:B:C references a file called C in subdirectory B of directory A.

A pathname which begins at the device directory is called a *fully qualified* pathname, since it is guaranteed to identify no more than one file in the entire system. An example of a fully qualified pathname is @DPD0:A:B:C.

When you supply a pathname as an argument to a system call or library routine, it must be terminated by a null (zero) byte. The system always uses this format when passing pathnames to your program.

Figure 2.10 shows a typical fragment of a MP/OS file system. There are several I/O devices including one disk drive. The fully qualified pathnames of the devices are shown in bold type.

The disk's root directory contains three files. Their filenames are FILE1, FILE2 and DIR1. DIR1 is a directory that is a subdirectory of the root. It contains two files called X and Y. The fully qualified pathnames of these files are also shown in bold face type.

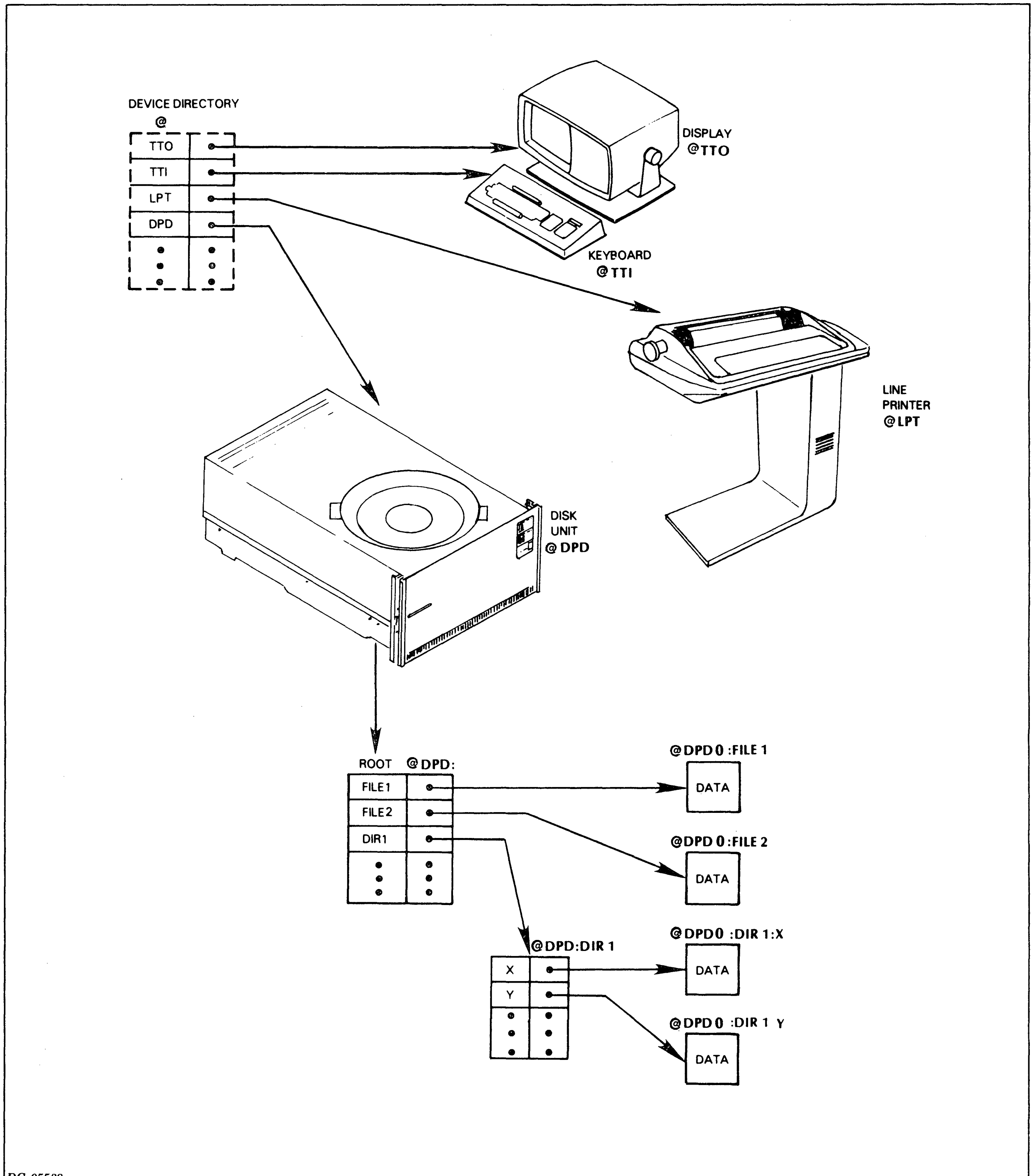

DEVICE DIRECTORY
@
TTO
TTI
LPT
DPD
DISPLAY
@TTO
KEYBOARD
@TTI
LINE
PRINTER
@LPT
DISK
UNIT
@ DPD
ROOT @DPD:
FILE1
FILE2
DIR1
@DPD 0 :FILE 1
DATA
@DPD 0 :FILE 2
DATA
@DPD:DIR 1
X
Y
@DPD0 :DIR 1:X
DATA
@DPD 0 :DIR 1 Y
DATA
DG-05538

Figure 2.10

The Working Directory

A MP/OS system typically contains many more files than are shown in Figure 2.10. As directory structures become more complex, pathnames get longer and more cumbersome. In order to reduce the necessity of using long pathnames, the system assigns a *working directory* to every program. You may think of the working directory as your current location in the file structure.

Whenever you reference a filename or pathname that is not fully qualified, the system will look for the file in your working directory. This enables you to use simple filenames instead of pathnames, and all file activity will take place in the working directory. A name such as A:B refers to a file called B in subdirectory A of your working directory.

Since you will typically assign a directory to each project you are working on, or to each user of the system, this concept assures that related files are kept together. You can change your current working directory at any time with the **?DIR** system call, and you can find out what your current working directory is with the **?GNAME** call. You can also perform these functions with the CLI **DIR** command.

The Searchlist

Sometimes it is inconvenient to try to confine all your work to one directory. For this reason the system provides you with a *searchlist* as a concise method of referencing multiple directories. The searchlist is simply a list of pathnames of directories. If you reference a filename that is not fully qualified, and if no such file exists in your working directory, then the system will search all the directories in your searchlist before determining that the file does not exist. All pathnames are searched for in this manner, except for those specified in a **?CREATE**, **?DELETE**, or **?RENAME** call.

You can read your searchlist with the **?GLIST** system call, and you can clear or extend your searchlist with the **?ALIST** call. There is also an **?SLIST** library routine which is a convenient way to set up your searchlist with one call, and a **SEARCHLIST** CLI command that reads or sets the searchlist. When the system is started up, it sets your searchlist to contain only the system master directory, :.

Pathname Prefixes

We have already explained the use of the @ and : characters in pathnames. There are two other characters may be used as *prefixes,* i.e., they may appear only at the beginning of a pathname.

The = character is equivalent to the pathname of the current working directory. You use this character when you wish to explicitly reference the working directory; the searchlist will not be used if the file is not found. The = by itself may be used as the name of the current working directory.

The ↑ character refers to the *parent* directory, i.e. the one which contains the current working directory. For instance, if your current working directory is @DPD0:A:B and you want to reference the file @DPD0:A:XYZ, you can use the pathname ↑XYZ. You can also use several ↑'s in sequence: for instance, to reference @DPD0:X you could use ↑↑ X.

Note that a pathname beginning with = or ↑ is not, strictly speaking, a fully qualified pathname, since the exact meaning of the pathname depends on your current working directory. However, such a pathname is like a fully qualified one because it specifies a directory, thus the searchlist will not be scanned.

File Element Size

The system allows you to optimize disk file organization by controlling the size of *file elements.* A file element is either a single disk block (512 bytes), or a group of disk blocks physically contiguous on the disk surface. The system allocates and deallocates file space in elements rather than blocks.

You specify a file's element size when you create it. A large element size means that data in a file will be organized as a number of large segments. This means that reading or writing the file can be done more efficiently, since the disk heads do not need to be continuously moved around the disk to find the proper data. Small element sizes give the system greater freedom in allocating disk blocks. You should choose the element size that gives you the best compromise between speed and efficient use of space.

File Types and Attributes

We have already mentioned one specific type of file, the directory. Every file in the system has a 16-bit number that defines its type. You may specify the file type when you create a file with the **?CREATE** call, and you may read it with the **?FSTAT** call.

A range of file type numbers is reserved for the user. You may assign any meanings to these file types that you find useful. Table 2.1 summarizes the available file types.

Mnem	Meaning
?DCHR	Character (non-disk) device.
?DLPT	Line printer.
?DDVC	Directory (disk) device.
?DDIR	Directory.
?DMSG	Inter-program message.
?DPSH	Program break file
?DSMN to ?DSMX	Range of values for files used by the system:
	?DPRG Program file.
	?DOBF Object file.
	?DLIB Library.
	?DSTF Symbol table file.
	?DOVL Overlay file.
	?DBPG Bootable (stand-alone) program.
	?DPST Permanent symbol table (used by assembler).
	?DUDF General purpose data file.
	?DTXT Text file.
?DUMN to ?DUMX	Range of values reserved for users.

Table 2.1 File types

The system also maintains an attribute word for each file. The right half (bits 8-15) of this word is used or reserved by the system. The left half (bits 0-7) is reserved for the user. As with file types, you may assign any meanings to these bits that you wish.

You can read a file's attributes with the **?GTATR** call, and change them with the **?STATR** call. Table 2.2 summarizes file attributes.

Mnem	Meaning
?ATPM	Permanent: the file may not be deleted or renamed while this bit is set to 1.
?ATRD	Read protect: this file may not be read.
?ATWR	Write protect: this file may not be written.
?ATAT	Attribute protect: the attributes of this file may not be changed (this bit is only used for devices and root directories of disks).

Table 2.2 File attributes

Basic Operations on Files

The system provides a number of system calls for manipulating files and managing directories. Table 2.3 summarizes these operations.

FILE OPERATIONS

Mnem	Function
?CREATE	Creates a file: creates an entry in the directory structure with a specified name, type and element size.
?DELETE	Deletes a file: removes the named file from the directory structure, and returns its disc space to the system.
?RENAME	Renames a file: changes the specified file's pathname. (Can be used to move a file to a new directory.)
?FSTAT	Gets file status: retrieves information about a file, including type, attributes, size, etc.
?GNAME	Gets pathname: Given a simple filename or pathname with or without prefixes, returns the fully qualified pathname. Scans the searchlist if necessary.
?DELDIR	Deletes directory (library routine): deletes the named directory after first deleting any files contained in it.
?GNFN	Gets next filename (library routine): retrieves names of files contained in a directory.

Table 2.3 File operations

Input/Output

I/O Channels

All data transfers between your program and a device or file take place via an I/O *channel.* A channel is a system-defined data path. Sixteen channels, numbered 0-15, are available to a program.

In order to use a channel, you must *open* it, i.e., you must connect it to some device or file. Your program does this with the **?OPEN** system call. When you finish using a channel, you can release (*close*) it with the **?CLOSE** call. You can use the **?INFO** call to learn the status of all channels. **INFO** returns a word in which a bit is set to 1 if the corresponding channel is currently open.

When your program begins to run, some channels may already be open, due to activity by a previously running program (see "Program Management", earlier in this chapter). This is a useful form of communication, since one program can open channels that are processed by another.

This is also a potentially hazardous situation if the second program is not "expecting" any channels to be open. Therefore, it is a useful precaution to have all programs start by closing any unneeded channels. The **?RESET** call is convenient for this purpose; it can close any or all 16 channels at once.

Standard Input and Output

The MP/OS CLI always opens two channels for console I/O. The CLI always passes these two channels to other programs, since almost all programs use them. The CLI always closes all other channels before calling any program.

The standard input channel has the mnemonic **?INCH**; it is opened to device @TTI. The standard output channel has the mnemonic **?OUCH**; it is opened to device @TTO.

File Positioning

When you open a channel to a disk file, the system keeps track of your position in the file with a 32-bit *file pointer*. This pointer is the number of the next byte in the file to be read or written. Normally this pointer is simply incremented for each byte transferred, so that the entire file is processed sequentially. When the pointer is zero, the channel is positioned at the beginning of the file.

You can use the **?GPOS** system call to find out the current value of the file pointer for any channel. You can use the **?SPOS** call to change the value of this pointer, thus permitting random access to any byte in the file.

Types of I/O

The MP/OS system provides two different techniques for I/O transfers. Generally, when you write data to a file, it is buffered in system memory, so that the system can combine several short transfers into one long one. You should always **?CLOSE** a file when you are finished using it to make sure that the system buffers are written (*flushed*) to the file.

Dynamic I/O

Dynamic I/O is performed by the **?READ** and **?WRITE** system calls. In this technique, you can read or write any number of bytes with a single system call. You place the number of bytes to be transferred in an accumulator; the data itself is transferred directly between the file and main memory (subject to buffering by the system).

You can use a special case of dynamic I/O to improve efficiency of transfers to and from disk devices. Data on a disk is divided by hardware into *blocks* of 512 bytes. When you request a data transfer to some part of a disk block, the system stores the entire block in a system buffer. Then your **?READ** or **?WRITE** call moves the data from/to this buffer.

If you use the **?READ** and **?WRITE** calls to transfer entire disk blocks, you eliminate the need for the system to buffer the transfer. To do this, you must program the data transfer to be a multiple of 512 bytes in length, and you must make sure that the transfers start on a block boundary, i.e., the file pointer must be set to a multiple of 512. Also your buffer area must begin on a word boundary, i.e., it must begin with the high order byte of a word. The result is a significant improvement in speed. For maximum efficency, you should not mix operations of this type with conventional dynamic I/O.

Data Sensitive I/O

Data sensitive I/O is performed by the **?READ** and **?WRITE** system calls with the **DS** option. In this case, the actual number of bytes to be transferred is not specified. The system transfers bytes until it reaches a limit specified by you, or it encounters a *delimiter:* a byte containing either a New-Line (12_8), Carriage Return (15_8), Form Feed (14_8), or null (0_8). As with dynamic I/O, the data is transferred between the channel and main memory. After the transfer, the number of bytes moved is placed in an accumulator.

I/O Device Management

The MP/OS system divides I/O devices into two categories: those with directory structures *(disks)*, and those without (*character devices*). Character devices include consoles, paper tape readers, and line printers.

Table 2.4 lists the standard MP/OS I/O devices with their mnemonics. Note that in a system with several devices of the same type, the mnemonic may be followed by a number, e.g., **@DPD0**, **@LPT1**.

Mnem	Type	I/O Name
DPD	In/out	Disk cartridge unit (10MByte).
DPH	In/out	Fixed-media disc unit (12.5MByte).
DPX	In/out	Diskette unit (315KByte).
DPY	In/out	Diskette unit (1.26 MByte).
TTI	In	Console keyboard.
TTO	Out	Console display.
PTR	In	Paper tape reader.
LPT	Out	Line printer.

Table 2.4 Input/Output devices

Disk Devices

The system provides two calls that initialize disk devices, and release them so that you can remove disk units from the system to mount new media on the drives. The **?MOUNT** system call introduces a disk to the system. The **?DISMOUNT** call shuts down a disk device in a consistent manner, ensuring that any I/O data that is still in system memory space will be written out. When the system is started up, only the system master device is mounted.

The system performs consistency checks at **?MOUNT** and **?DISMOUNT** time. A flag on every MP/OS disk indicates whether it was dismounted properly,(i.e., the system did not crash or some other circumstance did not impede dismounting). This flag is set by **?DISMOUNT**, and tested by **?MOUNT**. If the flag is not set at **?MOUNT** time, you will have to run the disk FIXUP program to restore the disk to a proper state. If this occurs for the system master device when you start up the system, the bootstrap loader automatically runs Fixup. You may also use **?MOUNT** to check the label on a disk, to make sure you have the one you want.

Note that you can only **?MOUNT** a MP/OS-format disk. If a disk is not in the proper format, you can use the DINIT utility to prepare it. DINIT is described in the *MP/OS Utilities Reference*, Section 6, *"Disk Initialization"*.

If you wish to access a disk without using the MP/OS file structure, you can **?OPEN** it without **?MOUNT**ing it first. In this case, the disk is treated as a single file with an element size equal to the number of blocks on the disk.

There is a **?DSTAT** call that you may use to retrieve status information that pertains to a disk. This call provides such data as the number of blocks in use, and the number of I/O errors that have occurred.

Console Devices

Console devices have a number of unique attributes, since they communicate directly with users. A console is actually two separate devices: the keyboard for input, and the printer or CRT for output.

The system performs a number of preprocessing functions on data from consoles. Normally, all characters received by the system from the keyboard are *echoed*, or retransmitted to the display, so that you can see what you are typing. Most control characters are echoed in the standard way, e.g., ↑ A for CTRL A. However, some control characters, such as newline, are echoed explicitly since they have special meanings to the console. Some others are assigned special meanings by the system (see below).

To ensure compatibility with the standard ASCII, the system sets to 0 the high-order bit of any byte that is sent to or from a console. Thus character values range from 0 to 177_8. All the above preprocessing functions, and the others listed in the tables below, can be enabled or disabled using system calls.

Control Characters

The system assigns special meanings to certain control characters. These meanings are summarized in Table 2.5.

Char	Octal	Function
null	0	Standard delimiter: signals the end of a data sensitive ?READ or ?WRITE.
CTRL C	3	Starts a control sequence (described below).
CTRL D	4	Indicates end of file (not passed to program).
New Line	12	Standard delimiter (like null).
Form Feed	14	Standard delimiter (like null).
Carriage Return	15	Standard delimiter (like null).
CTRL O	17	Reserved for future use.*
CTRL P	20	Reserved for future use.*
CTRL Q	21	Restarts output after CTRL S
CTRL R	22	Reserved for future use.*
CTRL S	23	Suspends output (so you can read material on a CRT screen).
CTRL T	24	Retype the current line (so you can check what you have typed).
CTRL U	25	Delete the current input line.
CTRL V	26	Reserved for future use.*
Rubout	177	Deletes the last character you typed from the current input line.

Table 2.5 Control characters

**Reserved characters are ignored (except in binary mode; see Table 2.7).*

Control Sequences

A *control sequence* is a CTRL C followed by one of the characters described in Table 2.6.

Char	Octal	Function
CTRL A	1	Signals a console interrupt, which may be passed to your program (see "Multitasking").
CTRL B	2	Causes termination of the currently running program (unless the current program is the level 1 CLI, in which case it is restarted).
CTRL C	3	Reserved for future use.*
CTRL D	4	Reserved for future use.*
CTRL E	5	Terminates the current program, and saves its state in a break file (see "Program Management"). This sequence is ignored if the current program is the level 1 CLI.
(Others)	---	No function: character is passed to your program.

Table 2.6 Control sequence characters

**Reserved characters are echoed, but not passed to your program (except in binary mode, see Table 2.7).*

Console Characteristics

Table 2.7 summarizes the console characteristics. Each is controlled by a bit in the device's characteristics word. You can read this word with the **?GCHAR** system call, and set it with the **?SCHAR** call.

Mnem	Affects	Meaning when 1
?CECH	input	Echo mode: echoes all typed characters (some receive special handling as described in text).
?CLST	output	List mode: echoes Form Feeds (014_8) as "]L" to prevent them from erasing CRT screen.
?CESC	input	Escape mode: handles Escape (33_8), the same as CTRL C CTRL A.
?CBIN	both	Binary mode: disables all special control characters; passes all characters exactly as received (8 bits).
?CST	output	Simulates tabs: converts all tab characters (011_8) to the appropriate number of spaces. Cursor moves to the beginning of the next 8-character tab column.
?CNAS	both	Non-ANSII-standard console: supports terminals using older standard for control characters by converting Carriage Returns (015_8) into New-Lines (012_8), and vice versa, on input. On output, converts New-Line to Carriage Return followed by New-Line followed by null.
?C605	both	DGC 6052, 6053, or similar device: uses cursor movement characters to echo Rubout and CTRL U by erasing characters from the screen.

Table 2.7 Console characteristics

The system keeps track of the number of characters that will fit on a line, and automatically inserts a newline whenever a program attempts to write more than this number. Similarly, the system keeps track of the number of lines on a page, and supplies a form feed when necessary. You may specify the line and page size when you create the system with SYSGEN (see Appendix L). You may read or change these sizes at any time with the **?GCHAR** or **?SCHAR** system calls, respectively. Note that form feeds are never inserted on output to a CRT terminal, and all auomatic insertion is disabled when you select binary mode for I/O.

Other Character Devices

The system's handling of paper tape readers and line printers is similar to that for console input and output, respectively. Both devices have a characteristics word which is a subset of the one for consoles. For line printers, the system keeps track of the line and page size in the same way that it does for consoles (as described previously).

The applicability of the various characteristics to these devices is summarized by the Table 2.8.

Characteristic	LPT	PTR
?CECH	unused	unused
?CLST	unused	unused
?CESC	unused	unused
?CBIN	used	always on
?CST	used	unused
?CNAS	used	used
?C605	unused	unused

Table 2.8 Character device characteristics

Special Device Support

The system allows you to take control of input/output devices at interrupt level by means of system calls. This enables you to achieve fast response to changing external conditions, while permitting the system to continue handling all other devices.

With the **?IDEF** call, you specify an *interrupt handler*: a routine in your program that is to receive control whenever a specified device causes an interrupt. You also specify the address of a *device control table* (DCT). This is a block of memory containing control data as summarized in table 2.9.

Word	Mnem	Contents
0	?IHND	Address of interrupt handler.
1	?IMSK	Mask word to be logically OR'ed with CPU interrupt mask.

Table 2.9 Device control table

Interrupt handlers should be written to run as fast as possible, so that they will not degrade the system's response to other devices. Interrupts will be enabled when an interrupt handler is called. The system passes data to it in the accumulators as shown in Table 2.10.

Accumulator	Contents
ACO	Current CPU interrupt mask.
AC1	Device code of interrupting device.
AC2	Address of DCT for this device.
AC3	Frame pointer to the system stack.

Table 2.10 Interrupt handler accumulators

> **NOTE:** *Interrupt handlers use the MP/OS stack for the sake of speed. If your handler modifies the contents of the frame pointer, it must restore it before exiting, or the state of the system will be destroyed.*

The only system calls which an interrupt handler may execute are **?IUNPEND** and **?IXIT**. **?IUNPEND** enables the routine to communicate with other tasks (see "Multitasking", below). **?IXIT** returns control to the system, and must be executed as the exit from the routine.

Your program must deactivate all interrupt handlers before the system will permit it to call another program with the **?EXEC** call. Interrupt handlers are deactivated with the **?IRMV** call.

Multitasking

Multitasking greatly simplifies certain types of programs, notably those where a number of operations must be performed in parallel. The system allows you to divide a program into a number of subprograms called *tasks*. The system contains a routine called the task *scheduler*, which switches control among various tasks. Since the switching from one task to another can be done very rapidly, it may seem as if several routines are running simultaneously.

Multitasking is similar to multiprogramming (e.g., timesharing) in that each of the separate tasks can be largely ignorant of what the others are doing. However, all tasks are part of a single program, so they must share memory, I/O channels, and other system resources.

An example of a multitasking program is a multi-user editing system that supports several people working at consoles. Without multitasking, the program would have to contain some sort of scanning routine that checked all users to see who needed service. Under the MP/OS system, you simply assign a separate task to each user. The system takes charge of deciding which user to service, freeing you from the need to write and debug a long, complex scanning routine.

Managing Tasks

The system uses some memory to hold task control information. You must specify the maximum number of tasks your program will require, so the system will know how much memory to allocate. You use the assembler's **.TSK** directive or the Binder's **/TASK** switch for this. The limit on the number of tasks that a program can use is specified when you generate a MP/OS system. In systems supplied by Data General, the maximum is 255.

At run time, you *create* tasks with the **?CTASK** system call. Creating a task is similar to calling a subroutine, except that the calling routine continues to run, rather than waiting for the called routine to exit. The contents of AC0, AC1, and AC2 may be passed to the created task. AC3 is set to the address of a MP/OS routine to which the task should jump when it finishes running. Because of this accumulator handling, a task may be written to use the **SAV** and **RET** instructions in a manner identical to a subroutine.

When you create a task, the system assigns it a *task identifier:* a 16-bit number which you use with system calls to reference the task. A task can retrieve its own identifier with the **?MYID** call.

Tasks are deleted (*killed*) when they jump to the address in AC3. You can also kill a task at any time with the **?KTASK** call. If you have specified a *kill post-processing routine* for the task, it will be executed at this time. This routine can perform such functions as deallocating memory used by the task. When it is entered, AC2 will contain the identifier of the task that is being killed, and AC3 will contain a return address to which the routine should jump when it finishes executing.

> **NOTE:** *A task kill post-processing routine may not execute any system calls.*

Each task must have its own stack space. The length of the stack must be equal to or greater than the value of the mnemonic **?STKMIN**. This ensures that there is enough space in the stack to store the task's state during execution of a system call. If you need to use the stack for local storage, you must allocate extra space.

When you create a task, you may specify a routine to be called in case the task causes a stack overflow. When this routine is called, AC2 will contain the identifier of the task that caused the overflow, and AC3 will contain a return address to which the routine should jump when it finishes executing. If you do not specify an overflow handling routine, any stack error will kill the task.

Parallel Call Errors

A conflict may arise in a multitasked program if one task executes an **?EXEC** or **?RETURN** while some other task has a system call in progress. A similar situation may occur, even in a single-task program, if you interrupt the program from the console. In these cases, any outstanding calls will be aborted, and they will give an error return with code **ERPCA**. Note that if a break file is produced, the error return will not occur until execution is resumed.

Task Priority

You may modify the scheduler's operation by changing the *priority* of some tasks. A task priority is a number between 0 and 255; lower value numbers represent higher priority. The system always runs higher priority tasks first; lower priority tasks are run only when all higher priority tasks are blocked from running.

You specify a task's priority when you create it with the **?CTASK** system call. You can also change the priority of a task at any time with the **?PRI** call. When you start a program, the system assigns a priority of 127 to its initial task.

Scheduling

At times you will need to suspend multitasking activity; for instance, you may need to read and modify a critical memory location without having some other task modify the same location at the same time. There are two system calls that support these activities. The **?DRSCH** call disables the task scheduler, ensuring that no task can run except the one that executed the **?DRSCH**. When it completes the critical activity, the task uses the **?ERSCH** call to reenable the scheduler. You can also use **?DRSCH** to determine whether or not multitasking is currently enabled, as explained in Chapter III.

Intertask Communication

Tasks are able to control each other's actions. The system permits tasks to synchronize their activities with the **?PEND** and **?UNPEND** calls. When a task executes a **?PEND**, it is blocked from running until a particular event occurs. The event is specified by a 16-bit number. This number must be used by some other task in an **?UNPEND** call to unblock the pended task. **?UNPEND** can also unblock a particular task by specifying its task identifier.

Event number values must be between zero and the value of the mnemonic **?EVMAX**. Certain values - those between zero and mnemonic **?EVMIN** - are reserved for system-defined events, which you may specify on a **?PEND** but not on an **?UNPEND**. You may use values between **?EVMIN** and **?EVMAX**, inclusive, for any purpose you choose. **?UNPEND** also allows you to pass a one-word message to AC0 of the unpended task.

When you unpend a task, you may cause it to take either the normal or error return from its **?PEND** call. The unpended task should then examine the contents of AC0 to determine the cause of the error return. You should be sure that the value of the message word is not the same as one of the **?PEND** error codes; otherwise, a task that takes an error return will not be able to determine the cause.

Console Interrupt Tasks

The system provides a method for programs to be interrupted by a user typing a CTRL-C CTRL-A sequence on his console keyboard. To receive this interrupt, your program must create a task that pends on an event number equal to (**?EVCH** + the channel number of the console keyboard). The task will be unpended if the user types CTRL-C CTRL-A, and it can then perform actions such as accepting a command from the user, or shutting down the program.

Chapter 3
System Calls

This chapter describes the MP/OS system calls. This chapter describes the MP/OS library routines. For each entry in this chapter, we give the following information:

- The mnemonic that you place in your program code.
- A description of the function performed, along with a diagram showing the format of the required packet (if any).
- A list of tables as described below.

Inputs
This table lists information which your program must place in accumulators before executing the call.

Outputs
This table lists information which will be in the accumulators when control returns to your program. Any accumulators that are not used for outputs will be unchanged, except for AC3 which is always set to the value of the frame pointer.

Errors
This table lists the error codes which are likely to be returned if you use a call improperly. Note that this list is not complete: some calls may return codes other than those listed under certain conditions. A complete list of the MP/OS error codes is contained in Appendix I of this manual.

For more general information on MP/OS programming, refer to Chapter 2 of this manual.

?ALIST

Add a Name to the Searchlist

ACTION - Appends the specified directory name to your searchlist. AC0 must contain a byte pointer to the fully qualified pathname, which must be terminated by a null byte. If AC0 contains 0, the searchlist is cleared; i.e., all entries are removed. The maximum length of a searchlist is 5 pathnames.

Inputs

AC	Contents
AC0	Byte pointer to pathname of directory (or 0).

Outputs

None.

Options

None.

Errors

Mnemonic	Meaning
ERFDE	File does not exist.
ERSTL	Searchlist too long.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERIFT	Incorrect file type (not a directory).
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device is off line.

?AWAIT

Await Completion of a Non-Pended System Call

ACTION - This call is used in conjunction with any non-pended system call (**NP** option) to find out if the call's action is finished. For example, if you executed a non-pended **?READ**, you would use **?AWAIT** to determine that the input data was available before you began operating on it. You specify the particular system call to be **AWAIT**ed by a task identifier, which must be the one that was returned to you by the system when you executed the non-pended call.

If the non-pended call is not yet finished, the task that executed the **?AWAIT** will be suspended until it completes execution, unless you use the **CK** option described below.

> **NOTE:** *You must issue a successful* **?AWAIT** *for every non-pended system call; otherwise a task control block (TCB) will be wasted.*

Inputs

AC	Contents
AC2	Task identifer for non-pended call.

Outputs

AC	Contents
AC0-2	All accumulators are set to the outputs of the non-pended call. Those not used for outputs are set to their values at the time of the non-pended call.

Options

Mnemonic	Meaning
CK	Check: if the non-pended call is not yet complete, does not suspend this task; instead, returns the **ERTIP** error code.

Errors

Mnemonic	Meaning
ERTIP	Task in progress: the non-pended call is still executing (**CK** option only).
ERTID	Invalid task identifier.

NOTE: *This call may also return any error codes that were produced by the non-pended call.*

?BOOT

Restart the System

ACTION - Causes the current MP/OS system to be shut down, and a new bootstrap loader to be read from the specified disc device and executed . All I/O channels are closed, and all disc devices are dismounted.

This call is also used to start a stand-alone user program. In this case, you specify the pathname of the program file instead of a device name.

The name must be terminated by a null byte. If no device is specified, the system shuts down but does not restart.

Inputs

AC	Contents
ACO	Byte pointer to file or device name (zero to shut down system).

Outputs
None.

Options
None.

Errors

Mnemonic	Meaning
ERFDE	File does not exist.
ERRAD	Read access denied.
ERWAD	Write access denied.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERIFF	Invalid file format.

?CLOSE

Close an I/O Channel

ACTION - Removes the specified I/O channel's connection to a device or file. If there is any data from previous **?WRITE** calls in a system buffer, it is written to the file. No more I/O may be performed on the channel until it is opened again.

Inputs

AC	Contents
ACO	Channel number.

Outputs

None.

Options

Mnemonic	Meaning
DE	Delete the file.

Errors

Mnemonic	Meaning
ERICN	Invalid channel number.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device is off line.
ERPRM	Permanent file: cannot be deleted.

?CREATE

Create a File

ACTION - Creates an entry for the specified pathname in the directory structure. The pathname must be terminated by a null byte. You can specify the type of the file and its element size. Its attribute word is set to 0.

You may not create new files in the device directory. However, in order to simplify device independent programming, the system gives a normal return if a program attempts to **?CREATE** a device that already exists.

> **NOTES:** *If the specified pathname is not fully qualified, the file is created in the working directory. The searchlist is* not *scanned.*
>
> *If the pathname contains any directories, they must already exist.*

Inputs

AC	Contents
ACO	Byte pointer to pathname.
AC1	Type of file to create.
AC2	File element size in disc blocks.

Outputs

None.

Options

Mnemonic	Meaning
DE	If the file already exists, deletes the old one.

Errors

Mnemonic	Meaning
ERNAI	File already exists (**DE** option not used).
ERIFT	Invalid file type.
ERPRM	Permanent file: cannot be deleted (**DE** option only).
ERSPC	Insufficient file space.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device is off line.
ERWAD	Write access denied.

?CTASK

Create a Task

ACTION - Introduces a new task to the scheduler. AC2 must contain the address of a task definition packet, in which you specify the new task's parameters as defined in the table below.

TASK DEFINITION PACKET
Type: ?TDP
Length: ?TLN

Mnem.	0	1–15
?TYPE	PACKET TYPE (?TDP)	
?TPRI	RESERVED | PRIORITY	
?TSTA		STARTING ADDRESS
?TSTB		STACK BASE (start address)
?TSTL		STACK LIMIT (end address)
?TSTE		STACK ERROR HANDLER ADDRESS
?TAC2	NEW TASK'S AC2	
?TUSP	NEW TASK'S ?USP WORD	
?TKPP		KILL POST-PROCESSING ADDRESS

If you specify zero in **?TSTE**, the system will provide a stack error handling routine. In this case, the task will be killed if it overflows its stack.

If you specify zero in **?TKPP**, the system assumes that you do not wish to perform any kill post-processing for the task.

?USP is a general purpose word in lower page zero whose contents are unique for each task in the program.

An error return will be taken if no task control block (TCB) is available to support the new task, unless the **AW** option is specified as described below.

Inputs

AC	Contents
ACO, AC1	Passed to new task.
AC2	Address of task definition packet.

Outputs

AC	Contents
AC2	Task identifier of the new task.

Options

Mnemonic	Meaning
AW	If no TCB is available, waits for one.

Errors

Mnemonic	Meaning
ERNOT	No free TCBs.
ERSTS	Invalid stack definition.
ERADR	Invalid start address.
ERPRP	Invalid priority.

?DELETE

Delete a File

ACTION - Removes the specified file from the directory structure and returns its disk space to the system. The pathname must be terminated by a null byte. If the file is open, the filename is removed from the directory, but the disk blocks are not released until all channels open to the file are closed.

Directories cannot be deleted if they contain any files. You can use the **?DELDIR** library routine for this function, since it automatically deletes all subordinate files.

Devices cannot be deleted; however, for the sake of compatibility, **?DELETE** does not take an error return if you attempt to delete one.

> **NOTE:** *If the specified pathname is not fully qualified, and the file is not found in the working directory, the* **ERFDE** *error return is taken. The searchlist is* not *scanned.*

Inputs

AC	Contents
ACO	Byte pointer to pathname.

Outputs
None.

Options
None.

Errors

Mnemonic	Meaning
ERFDE	File does not exist.
ERPRM	Permanent file: cannot be deleted.
ERDID	Directory is not empty.
ERNAD	Non-directory name in pathname
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device is off line.

?DIR

Select a Working Directory

ACTION - Sets the specified directory to be your current working directory. The pathname must be terminated by a null byte. If an error occurs, the current working directory is unchanged.

Inputs

AC	Contents
ACO	Byte pointer to pathname.

Outputs

None.

Options

None.

Errors

Mnemonic	Meaning
ERIFT	Invalid file type (not a directory).
ERFDE	File does not exist.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in pathname.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device is off line.

?DISMOUNT

Remove a Disk from the System

ACTION - Causes the specified disk device to be disabled from further I/O activity, and prepares the disk to be removed from the drive. The device name must be terminated by a null byte. Any data that is left in memory from previous I/O is flushed to the disk, and all pointers and directories on the disk are left in an orderly state. A flag is set on the disk to indicate that it was successfully ?DISMOUNTed.

Inputs

AC	Contents
ACO	Byte pointer to device name.

Outputs
None.

Options
None.

Errors

Mnemonic	Meaning
ERFDE	File does not exist.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.
ERDAI	Device in use (some I/O channels are open).
ERDNM	Device is not mounted.
ERIOD	Specified name is not a device.

?DRSCH

Disable Task Rescheduling

ACTION - Disables the system task scheduler, suspending the execution of all other tasks. Multitasking will resume only when an **?ERSCH** call is executed, or this task executes a **?PEND**. If multitasking is already disabled, this call has no efffect.

You can use this call to find out whether multitasking is enabled by using the **CK** option as described below. Since this is a "destructive test," you may then need to execute a **?ERSCH** to restore the scheduler's state.

Inputs
None.

Outputs
None.

Options

Mnemonic	Meaning
CK	Check: if multitasking is already disabled, cause the program to take an error return with code **ERSAD**.

Errors

Mnemonic	Meaning
ERSAD	Scheduling already disabled (**CK** option only).

?DSTAT

Get a Disk's Status Information

ACTION - Retrieves status information about the specified disk. You specify the disk by its pathname, which must be terminated by a null byte. The status information is placed in a packet, which has the format shown in the table below:

Disk Status Packet
TYPE: ?DSP
LENGTH: ?DLN

Mnem.	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
?TYPE	PACKET TYPE (?DSP)
?DFB	NUMBER OF FREE
	DISK BLOCKS (2 words)
?DAB	NUMBER OF ALLOCATED
	DISK BLOCKS (2 words)
?DTMX	MAXIMUM NO. OF FILES
?DTAL	(internal status information)
?DTSW						(status flags)
?DRER	NO. OF RECOVERABLE ERRORS
?DUER	NO. OF UNRECOVERABLE ERRORS

The status flags in the **?DSTW** word are described below:

Mnem.	Meaning when 1
?DLE1	Bad primary label block.
?DLE2	Bad secondary label block
?DME1	Bad primary MDV (internal information).
?DME2	Bad secondary MDV.

Inputs

AC	Contents
ACO	Byte pointer to device name of disk.
AC2	address of packet.

Outputs
None.

Options
None.

Errors

Mnemonic	Meaning
ERIOD	Specified device is not a disk.
ERMPR	Invalid packet address.
ERBTL	Buffer too long.
ERFDE	File does not exist.
ERFTL	Filename is too long.
ERIFC	Invalid character in filename.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device is off line.

?ERSCH

Enable Task Rescheduling

ACTION - Enables the system scheduler, allowing other tasks to be executed under control of the system scheduler. This call has no effect if multitasking is already enabled.

Inputs
None.

Outputs
None.

Options
None.

Errors
None.

?EXEC

Execute a Program

ACTION - Starts execution of the specifed program or break file. The pathname must be terminated by a null byte. All open I/O channels are made available to the new program. You may specify either a program swap, where the new program runs as a descendant, or a chain, where the old program's state is not saved. You may pass a message to the new program, which may be up to 2047 bytes long.

> **NOTE:** **?EXEC** *will give an error return if any user device interrupt handlers are active.*

Inputs

AC	Contents	
ACO	Byte pointer to pathname.	
AC1	Byte pointer to message (if message length is nonzero).	
AC2	Bit O:	O = swap 1 = chain
	Bit 1:	If one start the new program at the debugger starting address.
	Bits 5-15:	Message length (O if no message).

Outputs

AC	Contents
AC1	(only in case of error return)
?ECCP	The error code in ACO was returned by the called program.
?ECEX	The error code was returned by the system; the called program did not run.
?ECRT	The error code was caused by **?RETURN**, which was unable to resume the parent program; in this case, control is passed to the "grandparent" program, i.e., program level is decreased by 2.
?ECBK	The error code was returned by the system while trying to write a break file.
?ECAB	The error code was returned by the system to indicate an abnormal program termination such as a console interrupt.
AC2	Length of returned message.

Options

None.

Errors

Mnemonic	Meaning
ERABT	Called program terminated by CTRL-C CTRL-A sequence from console.
ERABK	Called program terminated by CTRL-C CTRL-A (break) sequence from console.
ERUIH	User device interrupt handlers are active.
ERSPC	Insufficient file space.
ERPCA	Some other task has already issued an **?EXEC** or **?RETURN**.
ERFDE	File does not exist.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device is off line.
ERIRB	Message buffer is too short.
ERBTL	Message buffer extends into system space.
ERNAD	Non-directory name in pathname.
ERFTL	Filename is too long.
ERIFC	Invalid character in pathname.
ERIFT	Invalid file type (not a program or break file).
ERNDP	No Debugger present (when you specify the Debugger starting address).
EREXS	Attempt to swap beyond program level 8.
ERVNS	Program file is for a different revision of the MP/OS system.

?FSTAT

Get a File's Status Information

ACTION - Retrieves a packet of information about the specified file. The file may be specified by channel number, if you have a channel open to it. Otherwise, you can specify the file by its pathname, which must be terminated by a null byte. The status information is placed in a block, which has the format shown in the table below:

File Status Packet
TYPE: ?FSP
LENGTH: ?FLN

Mnem.	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
?TYPE	PACKET TYPE (?FSP)
?FTYP	FILE TYPE
?FATR	ATTRIBUTE WORD
?FESZ	FILE ELEMENT SIZE (in blocks)
?FTLA	DATE AND TIME OF LAST ACCESS II
?FTLM	DATE AND TIME OF LAST MODIFICATION (2 words)
?FLEN	LENGTH OF FILE (in bytes) (2 words)

> **NOTE:** *If the specified file is a device, the contents of the* **?FESZ**, **?FTLA**, *and* **?FTLM** *words are meaningless. If the file is a character device, the* **?FLEN** *word is also unused.*

Inputs

AC	Contents
AC0	Byte pointer to pathname, or channel number (if **CH** option is used).
AC2	Address of packet.

Outputs

None.

Options

Mnemonic	Meaning
CH	AC0 contains a channel number instead of a byte pointer to a pathname.

Errors

Mnemonic	Meaning
ERICN	Invalid channel number.
ERMPR	Invalid packet address.
ERBTL	Buffer too long.
ERFDE	File does not exist.
ERNAD	Non-directory name in pathname.
ERFTL	Filename is too long.
ERIFC	Invalid character in filename.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device is off line.

?GCHAR

Get Device Characteristics

ACTION - Places the characteristics word of the specified device into an accumulator. The device name must be terminated by a null byte. The device must be a character device. See **?SCHAR** for a list of the characteristics.

Inputs

AC	Contents
AC0	Byte pointer to device name.

Outputs

AC	Contents
AC1	Device characteristics word, number of characters per line (**LL** option only), or number of lines per page (**PG** option only).

Options

Mnemonic	Meaning
LL	Return the number of characters per line in AC1.
PG	Return the number of lines per page in AC1.

Errors

Mnemonic	Meaning
ERFDE	File does not exist.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERIFT	Not a character device.

?GLIST

Get the Searchlist

ACTION - Retrieves the contents of your current searchlist into a buffer. The searchlist is represented by a series of pathnames, separated by commas and terminated by a null byte. All pathnames are fully qualified, i.e., they start at the device directory.

Inputs

AC	Contents
AC0	Byte pointer to buffer.
AC1	Length of buffer in bytes.

Outputs

AC	Contents
AC1	Length of searchlist (not counting final null byte).

Options

None.

Errors

Mnemonic	Meaning
ERIRB	Buffer too short.
ERBTL	Buffer extends into system space.
ERFIL	Device read error.
ERDOL	Device off line.

?GNAME

Get the Fully Qualified Pathname

ACTION - Accepts a filename or pathname and returns a fully qualified pathname (i.e., one that starts at the device directory) corresponding to it. If no such file is found in the current working directory, and no prefixes (@, ↑, or =) are present, then **?GNAME** scans the entire searchlist looking for the filename. The output pathname is placed in a buffer, and terminated with a null byte.

The input filename may contain prefixes; this enables you to find the name of your current working directory by calling **?GNAME** with the filename = You can also use **?GNAME CH** to find the name of the file that is open on a specified I/O channel, or **?GNAME PR** to find out the name of the currently running program.

Inputs

AC	Contents
ACO	Byte pointer to filename, channel number (**CH** option), or ignored (**PR** option).
AC1	Byte pointer to buffer for pathname.
AC2	Length of pathname buffer in bytes.

Outputs

AC	Contents
AC2	Length of returned pathname in bytes (not counting final null byte).

Options

Mnemonic	Meaning
CH	ACO contains a channel number.
PR	Get the pathname of the calling program (ACO is ignored).

Errors

Mnemonic	Meaning
ERFDE	File does not exist.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERIRB	Buffer too short.
ERBTL	Buffer extends into system space.
ERFIL	Device read error.
ERDOL	Device off line.

?GPOS

Get the File Position

ACTION - Retrieves the 32-bit file pointer for the specified I/O channel, and places it into two accumulators.

Inputs

AC	Contents
ACO	Channel number.

Outputs

AC	Contents
AC1	High order 16 bits of file pointer.
AC2	Low order 16 bits of file pointer.

Options

None.

Errors

Mnemonic	Meaning
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.
ERICN	Invalid channel number.
ERIOD	Invalid operation for device.

?GTATR

Get File Attributes

ACTION - Places the attribute word and type number of the specified file into two accumulators. The pathname must be terminated by a null byte. See **?STATR** for a list of the attributes.

Inputs

AC	Contents
ACO	Byte pointer to pathname.

Outputs

AC	Contents
AC1	Attribute word.
AC2	File type.

Options

None.

Errors

Mnemonic	Meaning
ERFDE	File does not exist.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.

?GTIME

Get the Current System Time and Date

ACTION - Gets the current time and date, in MP/OS internal format. Internal format is a 32-bit number representing the number of seconds since midnight, January 1, 1900. You may also use the **?GDAY** and **?GTOD** library calls to retrieve this number in decoded form.

Inputs
None.

Outputs

AC	Contents
AC0	High order 16 bits of system time.
AC1	Low order 16 bits of system time.

Options
None.

Errors
None.

?GTMSG

Get an Interprogram Message

ACTION - Reads the current interprogram message into a buffer. This message may have been transmitted by an **?EXEC** or a **?RETURN**, whichever occurred most recently. The system maintains only one message at a time.

The message may be any string of up to 2047 bytes. If you a specify a buffer that is too short, your program will take the error return but AC1 will contain the actual message length; thus you can try again after allocating more memory.

Inputs

AC	Contents
AC0	Byte pointer to message buffer.
AC1	Length of buffer in bytes.

Outputs

AC	Contents
AC1	Actual length of message.

Options
None.

Errors

Mnemonic	Meaning
ERIRB	Buffer too short.
ERBTL	Buffer extends into system space.

?IDEF

Define an Interrupt Handling Routine

ACTION - Informs the system that your program will handle interrupts from the specified device. You may specify device code 77_8 to define a power-up restart routine.

You must specify a device control table (DCT) for the device. The format of a DCT is as follows:

Word	Mnem	Contents
0	?IHND	Address of interrupt handler.
1	?IMSK	Mask word to be logically OR'ed with CPU interrupt mask.

Device control table

> **NOTE:** *Your service routine must not contain any* **MSKO** *instructions.*

Inputs

AC	Contents
ACO	Device code.
AC2	Address of DCT.

Outputs

None.

Options

None.

Errors

Mnemonic	Meaning
ERADR	Invalid routine address.
ERUIH	Service routine already defined for this device.
ERDVC	Invalid device code.

?INFO

Get Program Information

ACTION - Retrieves a packet containing information about the program's current memory allocation and running state. The format of the packet is given in the table below:

PROGRAM INFORMATION PACKET TYPE: ?PIP LENGTH: ?PLN

Mnem.	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
?TYPE	PACKET TYPE (?PIP)
?PPMN	LOWEST PURE ADDRESS
?PPMX	HIGHEST PURE ADDRESS
?PIMN	LOWEST IMPURE ADDRESS
?PIMX	HIGHEST IMPURE ADDRESS
?PREV	PROGRAM REVISION NUMBER
?PLEV	CURRENT PROGRAM LEVEL
?PHMA	HIGHEST ADDRESS AVAILABLE TO USER
?POCH	I/O CHANNEL STATUS MASK

In the **?POCH** word, each bit (0 - 15) is set to 1 if the corresponding I/O channel is open.

Inputs

AC	Contents
AC2	Address of packet.

Outputs

None.

Options

None.

Errors

Mnemonic	Meaning
ERIPT	Invalid packet type.
ERMPR	Invalid packet address.

> **NOTE:** *In runtime systems (see Appendix M), this call only places data into the* **?PIMN**, **?PIMX**, *and* **?PHMA** *words of the packet; the other words are not modified.*

?IRMV

Remove an Interrupt Handling Routine

ACTION - Informs the system that your program will no longer handle interrupts from the specified device.

Inputs

AC	Contents
ACO	Device code.

Outputs

None.

Options

None.

Errors

Mnemonic	Meaning
ERNUI	No handling routine currently defined, or you attempted to remove the system's control of a standard I/O device.

?IUNPEND

Unpend a Task from Interrupt Handling Routine

ACTION - This call functions in a manner identical to the **?UNPEND** call. The difference is that only **?IUNPEND** may be used by an interrupt handler. **?IUNPEND** may not be used at any other time.

It is advisable to use the **ID** option whenever possible, since it is faster than the other forms of the call, and speed is generally important to interrupt handlers.

Inputs

AC	Contents
ACO	Message word to unpended task.
AC2	Event number or task identifier.

Outputs

AC	Contents
ACO	Number of tasks unpended.

Options

Mnemonic	Meaning
BD	Unpend all tasks waiting for this event.
ER	Causes the unpended task(s) to take the error return from the **?PEND** call.
ID	AC2 contains a task identifier, not an event number.

NOTE: *Do not specify the* **BD** *and* **ID** *options together.*

Errors

Mnemonic	Meaning
ERTID	Invalid task identifier.
EREVT	Invalid event number.

?IXIT

Exit from an Interrupt Handling Routine

ACTION - Returns the system to normal operation after completion of a user interrupt handler. All interrupt service routines *must* exit by this call.

> **NOTES:** *You must not execute* **?IXIT** *at any time other than during an interrupt handler.*
>
> *When you execute* **?IXIT**, *the frame pointer must have the same value that it did when the interrupt handler was started.*

Inputs
None.

Outputs
None.

Options
None.

Errors
None.

?KTASK

Kill a Task

ACTION - Terminates execution of the specified task. If a kill post-processing routine is defined for the task, it will be executed. If the killed task has an outstanding system call, that call will be aborted; the degree of completion that the outstanding call reaches is undefined.

Inputs

AC	Contents
AC2	Task identifier, or zero to kill this task.

Outputs
None.

Options
None.

Errors

Mnemonic	Meaning
ERTID	Invalid task identifier.

?MEMI

Change Impure Memory Allocation

ACTION - Allocates or releases a section of the program's impure memory area. Memory is always added or removed at the top of the impure area.

Inputs

AC	Contents
AC0	Number of words to allocate (if positive) or release (if negative).

Outputs

AC	Contents
AC1	New highest impure address.

Options

None.

Errors

Mnemonic	Meaning
ERMEM	Invalid request: attempt to acquire or release too much memory.

?MOUNT

Introduce a Disk to the System

ACTION - Prepares the specified disk device for I/O. This call must be executed before any directories on the disk can be accessed.

The system checks a flag on the disk to see if it was properly **?DISMOUNT**ed. If it was not, your program will take the error return with code **ERFIX**, and you must run the Disk Fixup program. You can also check the label of the disk to make sure you have the correct one. The disk name and label must be terminated by null bytes. If AC2 is nonzero, then the system returns the disk label (terminated by a null byte) in a buffer at the specified address.

Inputs

AC	Contents
AC0	Byte pointer to device name.
AC1	Byte pointer to disk label, or zero to suppress label check.
AC2	If nonzero, address of a buffer to receive the disk label.

Outputs

None.

Options

None.

Errors

Mnemonic	Meaning
ERFDE	File does not exist.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERFIX	Disk requires FIXUP.
ERLAB	Disk label does not match specified one.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.

?MYID

Get Task Identity

ACTION - Places the calling task's identifier and priority into two accumulators.

Inputs

None.

Outputs

AC	Contents
AC0	Task priority.
AC2	Task identifier.

Options

None.

Errors

None.

?OPEN

Open an I/O Channel

ACTION - Connects an I/O channel to a specified device or disk file. The pathname must be terminated by a null byte. The system selects a channel and returns the channel number to you in an accumulator. When a file is opened, the file pointer is set to zero (the first byte of the file), unless the **AP** option is used as detailed below.

Normally, when you allocate new blocks to a disk file, the system writes zeroes to all bytes in these blocks. You can use the **NZ** option to suppress this action, and save some processing time. You should be sure that your program will not be disturbed by random data in the file; this is usually the case, since most files are written before they are read.

If you open a channel to a character output device, the system sends a Form Feed character (14_8) to it, unless the **AP** option is used.

Inputs

AC	Contents
AC0	Byte pointer to pathname.
AC1	File type (required only for **CR,** **DE**, or **UC** option).
AC2	File element size (required only for **CR**, **DE**, or **UC** option.

Outputs

AC	Contents
AC0	Channel number.

Options

Mnemonic	Meaning
EX	Exclusive access: no other program may use the file while this channel is open (gives an error return with code **EREOP** if the file is already in use).
CR	If the file does not exist, creates it.
DE	Deletes any existing copy of the file, then creates it.
UC	Unconditionally creates the file (gives an error return if the file already exists).
AP	For files, opens for appending; sets the file pointer to the end of the file. For character output devices, suppresses the sending of a form feed character.
NZ	Do not set newly allocated blocks to 0.

NOTE: *If you specify either the* **DE** *or* **UC** *option, the searchlist is* not *scanned in attempt to find the file. If it does not exist in the working directory, it is created there.*

Errors

Mnemonic	Meaning
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.
EREOP	File in use (**EX** option only), or file already **?OPEN**ed with **EX** option.
ERFDE	File does not exist.
ERNAE	File already exists (**UC** option only).
ERNMC	No more channels.
ERIOO	Illogical option combination.
ERPRM	Permanent file: cannot be deleted.

?PEND

Suspend a Task

ACTION - Causes the calling task to become blocked from execution until a specified event occurs. The event is defined by a 16-bit number which may be used by another task in an **?UNPEND** call. Event numbers must be greater than or equal to 0 and less than or equal to **?EVMAX**. When execution resumes, the system passes a message word from the task which executed the **?UNPEND**.

The calling task may also resume execution in response to an **?UNPEND ID** call, or after a timeout interval elapses. The length of the interval in milliseconds is specified as a 32-bit number in two accumulators. You may also request the system default timeout interval (about 1 minute) by setting both accumulators to 0.

If **?PEND** is issued when scheduling is disabled, **?PEND** reenables scheduling after blocking the calling task.

?PEND may also be issued while interrupts are disabled; in this case, it blocks the calling task and then reenables interrupts.

Inputs

AC	Contents
AC0	Event number.
AC1	High order word of timeout duration.
AC2	Low order word of timeout duration.

Outputs

AC	Contents
AC0	Message word from **?UNPEND**.

Options

None.

Errors

Mnemonic	Meaning
ERTMO	Timeout interval has elapsed.
EREVT	Invalid event number.

?PRI

Change Task Priority

ACTION - Sets the value of the specified task's priority. Priorities may range from 0 to 255 (lower values have higher priorities).

Inputs

AC	Contents
AC0	New task priority.
AC2	Task identifier: zero means this task.

Outputs

None.

Options

None.

Errors

Mnemonic	Meaning
ERTID	Invalid task identifier.
ERPRP	Invalid priority.

?READ

Read Data from a Device or File

ACTION - Reads one or more bytes from the specifed I/O channel. **?READ** may operate in either dynamic or data sensitive mode.

For dynamic input, you specify the number of bytes to be read, as well as the address at which to store the data. If you are reading from a disk, you can improve the efficiency of your programs by transferring entire disk blocks. To do this you must:

- Set the file pointer to a multiple of 512 before the transfer.
- Specify a multiple of 512 bytes to read.
- Specify a buffer beginning with the high order byte of a word.

For data sensitive I/O, you specify a maximum number of bytes, and reading proceeds until either a New-Line (12_8), Carriage Return (15_8), Form Feed (14_8), or null (00_8) is read. Data sensitive I/O is specified by the **DS** option.

Note that when you execute a data sensitive **?READ**, you may encounter the end of file before finding a delimiter. Similarly, on a dynamic **?READ** there may not be as many bytes left as you requested. In either of these cases your program will take the error return, but the data will be read and AC2 will contain the number of bytes read.

Inputs

AC	Contents
AC0	Channel number.
AC1	Byte pointer to buffer to receive data.
AC2	Byte count (dynamic), or maximum byte count (data sensitive).

Outputs

AC	Contents
AC2	Actual number of bytes read.

Options

Mnemonic	Meaning
NP	Non-pended call.
DS	Data sensitive read.

Errors

Mnemonic	Meaning
ERNOT	No free task control blocks (**NP** option only).
EREOF	End of file encountered.
ERIRB	Buffer too short.
ERBTL	Buffer extends into system space.
ERRAD	Read access denied.
ERLTL	Line too long: too many bytes without a delimiter (**DS** option only).
ERICN	Invalid channel number.

?RENAME

Rename a File

ACTION - Give a new pathname to a file. The new and old pathnames must both be on the same disk device. If a file with the new pathname already exists, the call gives an error return, unless the **DE** option is specified. The file must not be open. Note that if the new pathname points to a different directory, you can effectively "move" the file into the new directory.

> **NOTE:** *If the specified new pathname is not fully qualified, and the file is not found in the working directory, it is created there. The searchlist is* not *scanned.*

Inputs

AC	Contents
AC0	Byte pointer to current pathname.
AC1	Byte pointer to new pathname.

Outputs

None.

Options

Mnemonic	Meaning
DE	If the new filename exists, delete that file before renaming.

Errors

Mnemonic	Meaning
ERREN	Attempt to rename across devices, or to rename a root directory or an open file.
ERFDE	File does not exist.
ERPRM	Permanent file: cannot be renamed.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.
ERIFT	Illegal file type (attempt to rename a device).

?RESET

Close Multiple I/O Channels

ACTION - Closes any or all I/O channels, as specified by a bit mask that you place in ACO. No error is produced if you attempt to close a channel that is already closed.

Note that channels **?INCH** and **?OUCH** are set up by the CLI for standard input and output; therefore, it is generally convenient for you to keep them open.

Inputs

AC	Contents
ACO	Any bit (O - 15) set to 1 causes the corresponding channel to be closed.

Outputs

None.

Options

None.

Errors

Mnemonic	Meaning
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.

?RETURN

Return to the Next Lower Program Level

ACTION - Terminates execution of this program, and resumes the program which called it (the *parent* program). You may cause the parent to take the error return from its **?EXEC** call. All I/O channels are closed; however, the parent program will have the same I/O status that it did when it performed its **?EXEC**. You may pass an error code and/or a message of up to 2047 bytes to the parent.

If you specify the **BK** option, **?RETURN** creates a restartable program (*break file*) from the current program. The break file is named *?program_name*.**BRK** (If *program_name* ends in **.PR** or **.BRK**, that suffix is deleted first.) The break file is written into the current working directory; any existing file of the same name is overwritten. When the break file is restarted, execution will begin at the normal return of the **?RETURN** call.

The states of all open I/O channels will be preserved in the break file. When you restart the program, any open channels from the calling program will be passed to it; if a currently open channel has the same number as a channel that was saved at break time, the currently open one will take precedence. Note that break files cannot be transported to another MP/OS system the way **.PR** files can, unless the other system is running an identical version of the MP/OS system program.

If a **?RETURN** is executed at program level 1, the system sets the working directory and searchlist to **:**, and executes **:CLI.PR**.

?RETURN never takes the error return.If the parent program cannot be resumed, an error code is passed to the "grandparent" program, i.e., two program levels down instead of one.

Inputs

AC	Contents
ACO	Error code to return to parent program; if zero, the parent will take the normal return from its **?EXEC** call.
AC1	Byte pointer to message (if AC2 is nonzero).
AC2	Message length in bytes.

Outputs

None.

Options

Mnemonic	Meaning
BK	Save program state in a break file.

Errors

None (see above).

?SCHAR

Set Device Characteristics

ACTION - Sets the characteristics of the specified device. The device name must be terminated by a null byte. The specified device must be a character device. The characteristics are summarized by the table below. Note that not all characteristics have meaning for all devices (see Chapter 2, "I/O Device Management").

Mnem	Affects	Meaning when 1
?CECH	input	Echo mode: echoes all typed characters (some receive special handling as described in text).
?CLST	output	List mode: echoes Form Feeds (014_8) as "↑L" to prevent them from erasing CRT screen.
?CESC	input	Escape mode: handles Escape (33_8), the same as CTRL C CTRL A.
?CBIN	both	Binary mode: disables all special control characters; passes all characters exactly as received (8 bits).
?CST	output	Simulates tabs: converts all tab characters (011_8) to the appropriate number of spaces. Cursor moves to the beginning of the next 8-character tab column.
?CNAS	both	Non-ANSII-standard console: supports terminals using older standard for control characters by converting Carriage Returns (015_8) into New-Lines (012_8), and vice versa, on input. On output, converts New-Line to Carriage Return followed by New-Line followed by null.
?C605	both	DGC 6052, 6053, or similar devices uses cursor movement characters to echo Rubout and CTRL U by erasing characters from the screen.

Table 3.12 Console characteristics

Inputs

AC	Contents
AC0	Byte pointer to device name.
AC1	Characteristics word, number of characters per line (**LL** option only), or number of lines per page (**PG** option only).

Outputs

None.

Options

Mnemonic	Meaning
LL	Set the number of characters per line to the value in AC1.
PG	Set the number of lines per page to the value in AC1.

Errors

Mnemonic	Meaning
ERFDE	File does not exist.
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.
ERIFT	Not a character device.
ERICH	Invalid characteristics.

?SPOS

Set the Current File Position

ACTION - Sets the file pointer for the specified I/O channel to a specific byte. Normally, if you try to position past the current end of the file, the system will extend the file as needed. However, an attempt to exceed the file size will produce an error if:

- You have specified the **EF** option (described below).
- You have opened a disk device as a file, since the end-of-file for a disk is a physical limitation (no space left).

Inputs

AC	Contents
AC0	Channel number.
AC1	High order 16 bits of file position.
AC2	Low order 16 bits of file position.

Outputs

None.

Options

Mnemonic	Meaning
EF	If the program attempts to position past the end of file, give an error return with code **EREOF**.

Errors

Mnemonic	Meaning
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.
ERIOD	Invalid operation for device.
ERICN	Invalid channel number.
EREOF	End of file encountered.

?STATR

Set File Attributes

ACTION - Sets the attributes of the specified file. The filename must be terminated by a null byte. The meanings of the attributes are as follows:

Mnem	Meaning
?ATPM	Permanent: the file may not be deleted or renamed while this bit is set to 1.
?ATRD	Read protect: this file may not be read.
?ATWR	Write protect: this file may not be written.
?ATAT	Attribute protect: the attributes of this file may not be changed (this bit is only used for devices and root directories of disks).

File attributes

Inputs

AC	Contents
AC0	Byte pointer to pathname.
AC1	Attribute word.

Outputs

None.

Options

None.

Errors

Mnemonic	Meaning
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERIRB	Buffer too short.
ERBTL	Buffer extends into system space.
ERFDE	File does not exist.
ERATP	File is attribute protected.
ERIAT	Invalid attribute word.

?STIME

Set the Current System Time and Date

ACTION - Sets the system time and date to the specified value. You must use the MP/OS internal format, which is a 32-bit number representing the number of seconds since midnight, January 1 1900.

Inputs

AC	Contents
AC0	High order 16 bits of system time.
AC1	Low order 16 bits of system time.

Outputs

None.

Options

None.

Errors

None.

?UNPEND

Resume Execution of a Task

ACTION - Resumes execution of the specified task. You may specify the task either by its identifier, or by a 16-bit event number. Event numbers must be greater than or equal to **?EVMIN**, and less than or equal to **?EVMAX**. You may also specify that the unpended task should take the error return from its **?PEND** call. A message word is passed to all tasks that are unpended.

If you specify an event number that several tasks are waiting for, only one will be unpended (unless you use the **BD** option described below). The system unpends the task with the least remaining time in its timeout interval. If there are several tasks with the same time remaining, the one that **?PEND**ed first will be unpended.

Inputs

AC	Contents
AC0	Message word to unpended task(s).
AC2	Event code or task identifier (**ID** option).

Outputs

AC	Contents
AC0	Number of tasks unpended.

Options

Mnemonic	Meaning
BD	Unpends all tasks waiting for this event.
ER	Unpends task(s) at the error return.
ID	AC2 contains task identifier, not event code.

NOTE: *Do not specify the* **BD** *and* **ID** *options together.*

Errors

Mnemonic	Meaning
ERTID	Invalid task identifier.
EREVT	Invalid event number.

?WRITE

Write Data to a Device or File

ACTION - Writes data to the device or file on the specified I/O channel. You can write data using either dynamic or data sensitive mode.

To use dynamic writing, you must specify the number of bytes to be written. If you are writing to a disk, you can improve the efficiency of your program by transferring entire disk blocks. To do this you must:

- Set the file pointer to a multiple of 512 before the transfer.
- Specify a multiple of 512 bytes to write.
- Specify a buffer starting with the high order byte of a word.

Data sensitive writing is selected by the **DS** option. In this case, you specify the maximum number of bytes to transfer, and the system writes until it has written a New-Line (12_8), Carriage Return (15_8), Form Feed (14_8), or null (00_8) byte.

If you attempt to write past the end of file, your program will take the error return if you specified the **EF** option. If you did not specify the option, the system will extend the file as needed.

After the transfer, AC2 will contain the number of bytes written, whether or not the error return was taken.

Inputs

AC	Contents
ACO	Channel number.
AC1	Byte pointer to data to write.
AC2	Byte count (dynamic), or maximum byte count (data sensitive).

Outputs

AC	Contents
AC2	Number of bytes written.

Options

Mnemonic	Meaning
EF	Cause an error return if the program attempts to write past the end of file.
NP	Non-pended call.
DS	Data sensitive write.

Errors

Mnemonic	Meaning
ERNOT	No free task control blocks (**NP** option only).
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.
ERIRB	Buffer too short.
ERBTL	Buffer extends into system space.
ERICN	Invalid channel number.
ERLTL	Too many bytes without a delimiter (**DS** option only).
EREOF	End of file encountered.
ERWAD	Write access denied.

Chapter 4
Library Routines

This chapter describes the MP/OS library routines. This chapter describes the MP/OS library routines. For each entry in this chapter, we give the following information:

- The mnemonic that you place in your program code.
- A description of the function performed, along with a diagram showing the format of the required packet (if any).
- A list of tables as described below.

Inputs

This table lists information which your program must place in accumulators before executing the call.

Outputs

This table lists information which will be in the accumulators when control returns to your program. Any accumulators that are not used for outputs will be unchanged, except for AC3 which is always set to the value of the frame pointer.

Errors

This table lists the error codes which are likely to be returned if you use a call improperly. Note that this list is not complete: some calls may return codes other than those listed under certain conditions. A complete list of the MP/OS error codes is contained in Appendix I of this manual.

For more general information on MP/OS programming, refer to Chapter 2 of this manual.

?CDAY

Convert System Time/Date to Date

ACTION - Accepts a time and date in 32-bit MP/OS format, and returns the day, month, and year.

Inputs

AC	Contents
AC0	High order 16 bits of time.
AC1	Low order 16 bits of time.

Outputs

AC	Contents
AC0	Day.
AC1	Month.
AC2	Year (minus 1900).

Errors

None.

?CTOD

Convert System Time/Date to Time of Day

ACTION - Accepts a time and date in 32-bit MP/OS format, and returns the hour, minute, and second.

Inputs

AC	Contents
AC0	High order 16 bits of time.
AC1	Low order 16 bits of time.

Outputs

AC	Contents
AC0	Second.
AC1	Minute.
AC2	Hour.

Errors

None.

?DELAY

Delay Execution of a Task

ACTION - Causes the calling task to be suspended for the specified length of time. The time, specified in milliseconds, is a 32-bit quantity which you place in two accumulators. You may use the **?MSEC** library routine to convert hours/minutes/seconds/ to milliseconds. If you set both accumulators to 0, your task will be delayed for the system default timeout interval (about 1 minute).

The maximum delay time is about 6 days.

This routine uses the **?PEND** system call, so if multitasking has been disabled, it will be resumed.

Inputs

AC	Contents
AC0	High order 16 bits of the delay time.
AC1	Low order 16 bits of the delay time.

Outputs

None.

Errors

None.

?ERMSG

Retrieve a System Error Message

ACTION - Reads a message from the MP/OS error message file, **:ERMES**. If the specified error code has no corresponding message, then the text *UNKNOWN ERROR CODE n* is returned, where *n* is the error code in octal. If the error file cannot be found, the message *ERROR CODE n* is returned.

Inputs

AC	Contents
AC0	Error code.
AC1	Byte pointer to message buffer.
AC2	Buffer size in bytes.

Outputs

AC	Contents
AC2	Actual length of message.

Errors

Mnemonic	Meaning
ERIRB	Buffer too short.
ERBTL	Buffer extends into system space.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.
ERFDE	File **:ERMES** does not exist.
ERNMC	No more I/O channels.

?GDAY

Get the Current Date

ACTION - Gets the system time, decodes it into year, month, and day, and returns these values in accumulators. The year is measured since 1900.

Inputs
None.

Outputs

AC	Contents
AC0	Day.
AC1	Month.
AC2	Year.

Errors
None.

?GNFN

Get the Next Filename in the Working Directory

ACTION - Retrieves filenames from the specified directory. Each filename is placed in a buffer in your memory space, and terminated by a null byte.

To use this call, you **?OPEN** the directory and place the I/O channel number in AC0. Then each call to **?GNFN** will return one filename. An **EREOF** error return is taken after the last filename has been read.

Inputs

AC	Contents
AC0	Channel number.
AC1	Byte pointer to filename buffer.
AC2	Length of buffer.

Outputs

AC	Contents
AC2	Length of returned filename (not counting the terminating null byte).

Errors

Mnemonic	Meaning
ERIRB	Buffer too short.
ERBTL	Buffer extends into system space.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.
EREOF	End of file encountered.

?GTOD

Get the Current Time of Day

ACTION - Gets the system time, decodes it into hours, minutes, and seconds, and returns these values in accumulators. The hour ranges from 0 to 23.

Inputs

None.

Outputs

AC	Contents
AC0	Seconds.
AC1	Minutes.
AC2	Hours.

Errors

None.

?MSEC

Convert a Time to Milliseconds.

ACTION - Accepts a time in hours/minutes/seconds form, and returns a single 32-bit number representing the equivalent number of milliseconds. All inputs will be range checked, i.e., the hours must be in the range of 0 to 23, and the minutes and seconds must both be in the range of 0 to 59.

Inputs

AC	Contents
AC0	Seconds.
AC1	Minutes.
AC2	Hours.

Outputs

AC	Contents
AC0	High order 16 bits of the time in milliseconds.
AC1	Low order 16 bits of the time in milliseconds.

Errors

Mnemonic	Meaning
ERANG	Input out of range.

?OVLOD

Load an Overlay

ACTION - Checks to see if the specified overlay is currently in its node. If it is, then the node's use count is incremented. If it is not, and the use count is zero, then the overlay is loaded into the node and the use count is set to 1.

If the node is occupied by another overlay and the use count is nonzero, the calling task is pended until the node becomes available.

Inputs

AC	Contents
ACO	Overlay descriptor.

NOTE: *For information about the format of overlay descriptors, see Appendix G, "Using Overlays".*

Outputs

None.

Errors

Mnemonic	Meaning
EROVN	Invalid overlay descriptor.

?OVREL

Release an Overlay

ACTION - Releases control of the specified overlay by decrementing its use count. If the new use count is zero, any pended task awaiting the node will be unpended. If several tasks are pended awaiting different overlays for the node, the system selects one on the basis of remaining time and order of request (the same selection method used by the **?UNPEND** call).

Inputs

AC	Contents
ACO	Overlay descriptor.

NOTE: *For information about the format of overlay descriptors, see Appendix G, "Using Overlays".*

Outputs

None.

Errors

Mnemonic	Meaning
EROVN	Invalid overlay descriptor.
EROVC	Specified overlay is not currently loaded.

?SLIST

Set the Searchlist

ACTION - Sets the searchlist to the specified list of fully qualified pathnames. The pathnames must be separated by commas, with no intervening blanks, and the list must be terminated by a null byte. (This is the format returned by the **?GLIST** system call.)

Inputs

AC	Contents
ACO	Byte pointer to list of pathnames.

Outputs

None.

Errors

Mnemonic	Meaning
ERNAD	Non-directory name in pathname.
ERFTL	Filename too long.
ERIFC	Invalid character in filename.
ERSTL	Searchlist is too long.
ERIFT	Filename is not a directory.
ERFIL	Device read error.
ERPWL	Device write error.
ERDOL	Device off line.

?TMSG

Translate a CLI-format Message

ACTION - Retrieves selected portions of an interprogram message in CLI format. The specified message must be terminated by a null byte. This call uses a packet; its format is given in the table below.

?TMSG PACKET
TYPE: depends on request (see below).
LENGTH: ?GTLN

Mnem.	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
?GREQ	REQUEST TYPE (see below)
?GAR	ARGUMENT NUMBER
?GSW	SWITCH SPECIFIER (see below)
?GRES	BYTE POINTER TO RESULT BUFFER

The types of requests are outlined in the table below.

MNEM	MEANING	OUTPUTS ACO	AC1
?GCMD	Get the message: issue a ?GTMSG call and place the message in the buffer pointed to by ?GRES.	Length of message.	Unused
?GCNT	Get the argument count.	Number of arguments.	Unused
?GARG	Copy the argument specified by ?GAR to the result buffer.	Argument length.	Unused
?GTSW	Test if the switch specified by the byte pointer in ?GSW is attached to argument specified by ?GAR. If so copy its value (if any) to the result buffer.	Test result.*	Unused
?GSWS	Get the switch set: check for single-letter switches, and set the corresponding bits in ACO and AC1.	Flags for /A through /P (bit O = /A, bit 1 = /B, bit 15 = /P).	Flags for /Q through /Z (bit O = /P, bit 9 = /Z, bits 10-15 unused).
?GSWI	Test if the switch specified by the switch number in ?GSW is attached to the argument specified by ?GAR. If found copy the switch value (if any) to the result buffer.	Test result.*	Length of returned string.

**Test results are:*
-1 if the switch was not found.
0 if the switch has no value.
> 0 for the length of the switch value.

NOTES: *The command or program name is referenced as the 0th argument.*

?TMSG *regards upper and lower case letters as equivalent on input; on output it converts all letters to upper case.*

Inputs

AC	Contents
ACO	Byte pointer to message buffer.
AC2	Address of packet.

Outputs

AC	Contents
ACO	Depends on request type.
AC1	Depends on request type.

Errors

Mnemonic	Meaning
ERNAR	No argument for specified **?GAR**.
ERNSS	No such switch.
ERBTL	Buffer extends into system space.
ERIRB	Buffer too short.
ERMRP	Invalid packet address.

Appendices

Appendix A
The ASCII Character Set

DECIMAL	OCTAL	HEX	KEY SYMBOL	MNEMONIC
0	000	00	↑@	NUL
1	001	01	↑A	SOH
2	002	02	↑B	STX
3	003	03	↑C	ETX
4	004	04	↑D	EOT
5	005	05	↑E	ENQ
6	006	06	↑F	ACK
7	007	07	↑G	BEL
8	010	08	↑H	BS (BACKSPACE)
9	011	09	↑I	TAB
10	012	0A	↑J	NEW LINE
11	013	0B	↑K	VT (VERT. TAB)
12	014	0C	↑L	FORM FEED
13	015	0D	↑M	CARRIAGE RETURN
14	016	0E	↑N	SO
15	017	0F	↑O	SI
16	020	10	↑P	DLE
17	021	11	↑Q	DC1
18	022	12	↑R	DC2
19	023	13	↑S	DC3
20	024	14	↑T	DC4
21	025	15	↑U	NAK
22	026	16	↑V	SYN
23	027	17	↑W	ETB
24	030	18	↑X	CAN
25	031	19	↑Y	EM
26	032	1A	↑Z	SUB
27	033	1B	ESC	ESCAPE
28	034	1C	↑\	FS
29	035	1D	↑]	GS
30	036	1E	↑↑	RS
31	037	1F	↑←	US

DECIMAL	OCTAL	HEX	KEY SYMBOL
32	040	20	SPACE
33	041	21	!
34	042	22	" (QUOTE)
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	' (APOS)
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	, (COMMA)
45	055	2D	-
46	056	2E	. (PERIOD)
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	–
62	076	3E	<
63	077	3F	?
64	100	40	@

DECIMAL	OCTAL	HEX	KEY SYMBOL
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	↑ OR ^
95	137	5F	← OR _
96	140	60	` (GRAVE)

DECIMAL	OCTAL	HEX	KEY SYMBOL
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~ (TILDE)
127	177	7F	DEL (RUBOUT)

Appendix B
Soft Control Panel (SCP) Programs

SCP for the MP/100 System

The SCP is a program which aids you in working with an MP/100 system by allowing you to interact with the computer through your terminal. You enter simple commands on a terminal keyboard to examine and/or modify any processor register or memory location. A breakpoint feature allows you to stop the execution of a program at a selected place in order to find program bugs.

The SCP is supplied by Data General as a set of read-only memory (ROM) chips. These chips reside in sockets on the MP/100 SPU board. They are not in the program's address space, so they are transparent to program operation.

To use the SCP, your SPU must have its jumper word register set to 77_8 so that the SCP will be entered upon power-up. You can also enter the SCP at any time by depressing the Break key on the console.

When the SCP takes control of the SPU, it types out the contents of the program counter when the SCP was entered. (Upon power-up, this number is meaningless.) The SCP then types a ! on the terminal. This is the SCP *prompt*; it tells you that the SCP is in control and ready to accept a command.

Command Format

An SCP command consists of a single character. Some commands must be preceded by an *argument*, which is either an octal number or an *expression* consisting of several octal numbers separated by + or - signs. The *dot* symbol (.) is also allowed in arguments; its value is the address of the location that SCP is currently working with.

Special Commands

The SCP has several commands which help you to correct typing errors. You can use the Rubout key to delete the last character you typed, in which case the SCP prints it on the terminal again to notify you that it has been deleted. Typing more Rubouts will continue to delete characters from right to left.

If you wish to cancel the entire line that you have just entered, type a **K**. The SCP prints a ? followed by a New-line, and also closes the current cell if it is open (described in detail below). The ? followed by a New-line is also printed if you type a character SCP does not recognize or recognizes as a user error.

Cells

The SCP operates on *cells*. A cell is either a memory location or an internal processor register (*internal cell*), such as an accumulator. In order to examine or modify any cell, you must *open* it. Opening a cell causes its contents to be printed, in octal, on your terminal.

To open an internal cell, use the command *n***A**, where *n* is one of the numbers listed in Table B.1.

To open a memory location, use the slash (/) command. This command takes a variety of arguments, as summarized below. The symbol *expr* in the table means you may type any octal number or expression. The term *current cell* means the last cell that you opened.

Cell	Contents
0-3	The accumulators AC0 through AC3.
4	Return address: the contents of the program counter when the SCP was called.
5	Stack pointer.
6	Frame pointer.
7	Status bits: Bit 15: Done flag of the asynchronous output interface. Bit 14: Interrupt enable flag: 0 = interrupts off. 1 = interrupts on. Bit 13: Carry bit.
12	Previous contents of breakpoint location.
11	Current breakpoint address.
12-15	Temporary locations used by the SCP.

Table B.1 MP/100 SCP internal cells

Command	Function
expr/	Opens the memory location whose address is equal to *expr*.
./	Opens the current memory location.
.+*expr/*	Opens the location whose address is equal to the current location's address plus *expr*.
.-*expr/*	Opens the location whose address is equal to the current location's address minus *expr*.
Carriage Return	Closes the current cell, and opens the next consecutive cell.
New-Line	Closes the current cell, but does not open another.
/	Opens the location whose address is equal to the contents of the current cell.
+*expr/*	Opens the location whose address is equal to the current cell's contents plus *expr*.
-*expr/*	Opens the location whose address is equal to the current cell's contents minus *expr*.

Table B.2 MP/100 SCP memory location commands

Modifying a Cell

Once you have opened a cell, you may change its contents by simply typing the number or expression whose value is to be placed in the cell. Terminate the expression with a Carriage Return or New-line to close the cell. Note that if you type Carriage Return, the next cell will also be opened. This is convenient when you need to enter data into several consecutive locations.

If you type an expression starting with + or -, the value of the expression will be added to or subtracted from the current contents of the cell. The dot symbol (.) may also be used in these expressions.

If you type any Rubouts immediately after opening a cell, the SCP will delete the rightmost digits of the cell's contents as though you had just typed them yourself. You may then type in new values for these digits.

> **NOTE:** *You may not alter any location that is contained in read-only memory (ROM).*

Breakpoints and Program Control

The SCP breakpoint facility allows you to place a breakpoint at any location in your program. When the program encounters the breakpoint during execution, it will enter SCP so that you can examine or modify any cells. This can be a great aid in debugging a program, since you can stop your program at a point where you think there is a problem, and then resume execution with no loss of data.

To set the breakpoint, type *expr* **B**. The breakpoint will be set at the address specified by *expr*. To delete the breakpoint, type **D**.

Typing **B** causes the address of an existing breakpoint to be printed on the terminal.

When you set a breakpoint, the SCP uses *instruction emulation* on all subsequent returns to your program, so that it can trap the entry to the breakpoint address. This means that, instead of jumping back into your program, the SCP examines the instructions and emulates their action without releasing control of the SPU.

When the breakpoint is encountered, the SCP places the address of that instruction into the **4A** internal cell. Typing **P** causes the SCP to return to the location specifed by **4A**. Thus you can use **P** to resume program execution after a breakpoint. When the SCP is called by a means other than a breakpoint (i.e., by a halt or power-up), typing **P** will cause the SCP to resume program execution at the location specified by the program counter +1. Note that the instruction at the breakpoint address is not executed until you return from the breakpoint.

You can also return to a program by typing *expr***R**. In this case, the SPU resumes program execution at the location specifed by *expr*. You can restart the program at location 0 by typing **R**.

You can execute your program one instruction at a time by using the **O** or **Q** commands. These commands manipulate the breakpoint to provide a single-step facility. The difference between the two commands is that **O** may only be used for single-stepping through RAM, whereas **Q** may be used for RAM or ROM.

The **O** or **Q** command causes the SCP to set a breakpoint at the instruction which will be executed after the one addressed by the **4A** internal cell, and to return to the location addressed by **4A**. This results in execution of one instruction, followed by a return to the SCP.

> **NOTES:** *The* **O**, **P**, *and* **Q** *commands all assume that the* **4A** *internal cell has been initialized by a return from a breakpoint. If this is not the case, you must set the* **4A** *cell and the breakpoint address to the value of the instruction at which single-stepping is to begin.*
>
> *Interrupts are not enabled during instruction emulation, i.e., whenever a breakpoint is set in ROM.*
>
> *You cannot use instruction emulation to execute a stack instruction. If you attempt to do so, the SCP will stop executing your program and type a* **?** *on the terminal.*

Additional Command

You can program load from an I/O device by typing *dvc***L**, where *dvc* is the device code of the I/O device to be used. (For data channel devices, remember that bit 0 of *dvc* should be set to 1.)

SCP for the MP/200 System

The Soft Control Panel (SCP) program can aid you in working with an MP/200 system by allowing you to interact with the computer through your terminal. Simple commands which you enter on a terminal keyboard allow you to examine and/or modify any processor register or memory location.

The SCP resides in read-only-memory (ROM) chips on the MP/200 multi-function controller board. SCP ROMs occupy memory locations 77000_8-77777_8.

To access the SCP, insert device code 77_8 in location 77776_8, the jumper word register. Upon power-up the SCP firmware examines this location, and if it contains device code 77_8, the SCP retains control. If it contains any other device code, the SPU performs an automatic program load from that device.

The SCP program is also called when the user's program encounters a **HALT**, if the CPU is jumpered for auto start.

Once called, the SCP prints the contents of the program counter at the time SCP was invoked on the terminal. If the SCP was called during a power-up, this data is meaningless. If the SCP was called by a **HALT**, this data is the address of the instruction. If the SCP was called by a non-maskable interrupt request, this data is the program-counter minus 1 at the time when the interrupt occured.

The SCP then types a **!** on the terminal. This is the SCP *prompt*; it tells you that SCP is ready to accept a command.

Command Format

An SCP command consists of a single character. Some commands must be preceded by an *argument* which is an octal number. The *dot* (.) symbol (see Table B.4) is also allowed as an argument; its value is the address of the location that the SCP is currently working with.

Special Commands

The SCP has several commands which help you to correct typing errors. You can use the Rubout key to delete the last character you typed, in which case the SCP prints it on the terminal again to notify you that it has been deleted. Typing more Rubouts will continue to delete characters from right to left.

If you wish to cancel the entire line that you just entered, type a **K**. The SCP prints a **?** followed by a New-line, and also closes the current cell if it is open (described in detail below). If you type a character which the SCP does not recognize, the **?** followed by a New-line is also printed, however the current cell is not closed.

Cells

The SCP operates on *cells*. A cell is either a memory location (*memory cell*), or an internal register (*internal cell*) such as an accumulator. In order to examine or modify any cell, you must *open* it. Opening a cell causes its contents to be printed, in octal, on your terminal.

To open an internal cell, use the *n***A** command where *n* is one of the numbers listed in Table B.3.

Number	Cell
0-3	The accumulators AC0 through AC3, respectively.
4	The contents of the program counter when SCP was entered by a HALT instruction.
5	Stack pointer.
6	Frame pointer.
7	Status bits: Bit 15: Done flag of the terminal output interface. Bit 14: Interrupt enable flag (ION): 0 = interrupts off. 1 = interrupts on. Bit 13: Carry bit.

Table B.3 MP/200 SCP internal cells

To open a memory location, use one of the commands listed in Table B.4. In the table, the term *current cell* means the last cell that you opened.

Command	Function
n/	Open the memory location whose address is equal to *n*.
./	Open the current memory location.
Carriage Return	Close the current cell, and open the next consecutive cell.
New-line	Close the current cell, but do not open another.
/	Open the cell whose address is equal to the contents of current cell.

Table B.4 MP/200 SCP memory location commands

Modifying a Cell

Once you have opened a cell, you may change its contents by simply typing the number or expression whose value is to be placed in the cell. Terminate the expression with a Carriage Return or New-Line to close the cell. Note that if you type a Carriage Return the next cell will also be opened. This is convenient when you need to enter data into several consecutive locations.

If you type any Rubouts immediately after opening a cell, SCP will delete the rightmost digits of the cell's contents as though you had just typed them yourself. You may then type in new values for these digits.

Additional Commands

The SCP has two commands which allow you to resume program execution. Typing **P** restarts program execution at the location specified by **4A**, the return address. (See Table B.3, Internal Cells.) You can also return to a program by typing *n***R**. In this case, the SPU issues a system reset command **IORST**, and resumes program execution at the location specified by the octal number *n*.

Typing *n***L** causes the SPU to perform a program load from an I/O device whose device code is equal to the octal number *n*. Typing *n***H** causes the CPU to perform a program load from a data channel device whose device code is equal to *n*.

Typing **S** causes the contents of accumulators 0-3, the stack pointer, and the frame pointer to be printed on the terminal.

Appendix C
Standard I/O Device Codes

Octal Device code	Mnem	Priority mask bit	Device name
00	-	-	Reserved
01	-	-	Reserved
02	-	-	Reserved
03	-	-	Reserved
04			
05			
06			
07			
10	TTI	14	Async. controller rec.
11	TTO	15	Async. controller trans.
12	PTR	11	Paper tape reader
13			
14	RTC	13	Model 4220 prog. R.T.C.
15			
16			
17	LPT	12	Line printer
20			
21	ADCV	8	A-D interface
22			
23	DACV	8	D-A interface
24			
25			
26			
27	DHP	7	12.5 Mbyte disc
	DPD	7	10 Mbyte cartridge disc
30			
31			
32			
33	DPX	10	Diskette subsystem
34	MUX	8	Sync/async controller
35	CRC	8	Cyclic redundancy checker
36			
37			

Table C.1

Octal Device code	Mnem	Priority mask bit	Device name
40			
41			
42	DIO	5	Digital I/O interface
43			
44			
45			
46			
47			
50	TTI1	14	Remote restart receiver
51	TTO1	15	Remote restart transmitter
52	PTR1	11	Second paper tape reader
53			
54			
55			
56			
57	LPT1	12	Second line printer
60			
61	ADCV1	8	Second A-D interface
62			
63	DACV1	8	Second D-A interface
64			
65			
66			
67	DHP	7	Second 12.5 Mbyte disc
	DPD1	7	Second 10 Mbyte cartridge disc
70			
71			
72			
73	DPX	10	Second diskette subsystem
74			
75			
76			
77	CPU	-	Reserved

Appendix D
DGC Standard Floating Point Format

Word for word, floating point format provides a much larger range than integer format, at the expense of some precision. It also provides the ability to operate on fractions. The maximum range of floating point format is equivalent to a 16-word multiple precision integer. In addition, floating point operations are executed faster than most multiple precision integer operations.

We represent a floating point value using a 4-byte number (for single precision) or an 8-byte number (for double precision). The 4- or 8-byte aggregate contains 3 fields:

- A fractional part called the *mantissa* , which, at the end of all floating point operations, is always adjusted to be greater than or equal to 1/16 and less than 1 (i.e., *normalized*);
- An *exponent*, which is adjusted to maintain the correct value of the number;
- A sign.

Operations on numbers in memory employing the floating point arithmetic instructions require that the number be *word aligned*, that is, bit 0 of the first byte of the number is bit 0 of the first word of a 2-word or 4-word area in memory.

The magnitude of a floating point number is defined to be:

$\text{MANTISSA X } 16^{(\text{TRUE VALUE OF THE EXPONENT})}$

The magnitude of a single or double precision number is thus on the range, approximately:

$5.398 \text{ X } 10^{-79}$ to $7.237 \text{ x } 10^{-79}$.

We represent zero in floating point by a number with all bits zero, known as *true zero*. When a calculation results in a zero mantissa, the number is automatically converted to a true zero.

Sign

Bit 0 of the first byte is the sign bit. If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative.

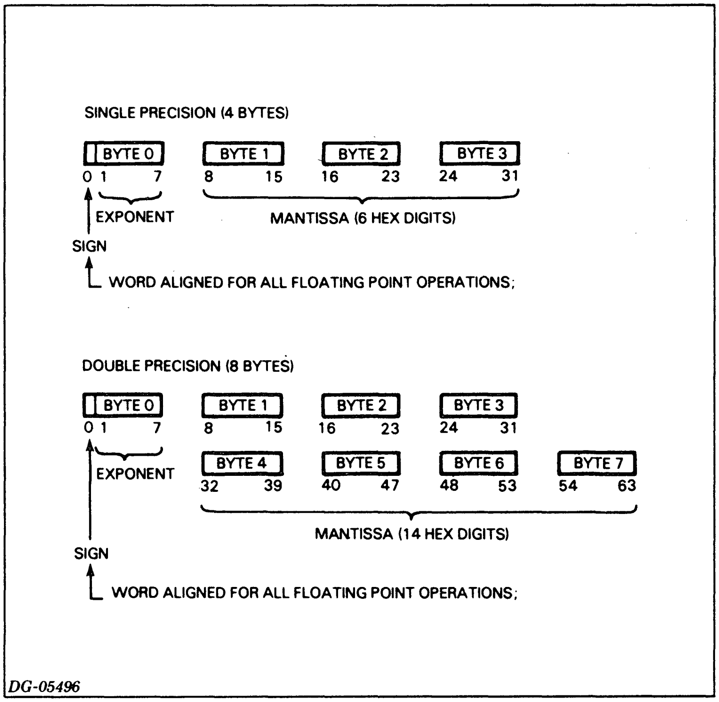


Figure D.1 Figure D.1

Exponent

The right-most 7 bits of the first byte contain the exponent. We use *excess 64* representation. For both positive and negative exponents, the value is 64 greater than the true value of the exponent. Table D.1 illustrates this:

Exponent Field	True Value of Exponent
0	-64
64	0
127	63

Table D.1 Excess 64 representation of exponents

Mantissa

Bytes 1-3 (single precision) or bytes 1-7 (double precision) contain the mantissa. By definition, the binary point lies *between* byte 0 and byte 1 of a floating point number. In order to keep the mantissa in the range of 1/16 to 1, the results of each floating point calculation are *normalized.* A mantissa is normalized by shifting it left one hex digit (4 bits) at a time, until the high-order four bits (the left-most four bits of byte 1) represent a nonzero quantity. For every hex digit shifted, the exponent is decreased by one.

Appendix E
Macroassembler Operating Instructions

The CLI command line used to invoke the Macroassembler is

XEQ MASM*[func_sw] pathnames[arg_sw]*↓

where:

[func_sw] are optional function switches, and

[arg_sw] are optional argument switches.

When you issue this command, the Macroassembler assembles one or more source files *(pathnames)* and produces an object file, an assembly listing, and/or an error listing. By default, the object file bears the name of the first source filename not followed by /S which you supplied in the pathname argument list. Also by default, the macroassembler produces no assembly listing and sends error codes to the console.

Tables E.1 and E.2 describe the function and argument switches you can use in the MASM command line.

Symbol	Action
/**B**=*file name*	Gives the name *filename* to the object file. Ordinarily the object file has the name of the first source file in the assembly command line, less the extension **.SR** (if any) and with a new extension **.OB**.
/**E**	Produces a summary listing at the console of any assembly errors. If no assembler listing (/**L**) or error listing (/**E**) is requested, error messages will appear at the console as though /**E** had been specified.
/**E**= *error_file*	Same as /**E** except that the error messages will be sent to *error_file*. If the *error_file* already exists, then new error messages will be appended to the file.
/**F**	Generates or suppresses Form Feeds as required to produce an even number of assembly pages. This keeps the first page of successive listings on the outsides of paper folds, making refolding unnecessary. By default, the Macroassembler generates a Form Feed at the end of a listing, whether the number of pages is odd or even.
/**K**	Keeps the Macroassembler's temporary symbol file at the end of assembly. Since virtually no programs require the use of this file, the Macroassembler deletes it by default.
/**L**	Produces a listing file on the line printer. Listings always include a cross reference of symbols in the program showing the page and line number where each symbol is used. If you use the /**L** switch, program MASMXR.PR must be present in the same directory as the Macroassembler itself. MASMXR.PR is nearly always present where MASM.PR itself is found, but, if it is missing, then an error message will be displayed at the console.
/**L**=*list file*	Produces a listing file, but instead of sending it to the line printer, sends it to the file designated by *listfile*. If there is already a listing in this file, then the new listing follows it. *Listfile* can be any filename or pathname permitted by the operating system.

Table E.1 Macroassembler command line function switches

Symbol	Action
/**M**	Flags any redefinition of semipermanent symbols as multiple definition errors (M).
/**N**	Produces no object file.
/**O**	Overrides all listing control pseudo ops: **.NOCON**, **.NOLOC**, and **.NOMAC**. Also overrides ** listing suppression.
/**P**	Adds semipermanent symbols to the cross-reference listing. By default they are not included.
/**PS**= *pathname*	Uses *pathname* instead of MASM.PS to build symbol table file.
/**R**	Produces an object file even if there is an assembly error. By default, if there is an assembly error, the Macroassembler does not produce an object file.
/**S**	Skips the second assembly pass (produces no .OB file) and saves the Macroassembler's symbol table, renaming it MASM.PS. (See below, "Macroassembler Symbol Table Files.")
/**S**= *ps_file*	Same as /**S** except that *ps_file* is used as the name of the new permanent symbol file instead of MASM.PS. The original MASM.PS remains unchanged and the old copy, if any, of *ps_file* will be deleted.
/**U**	Includes user symbols in the object file. When the /**U** switch is also applied to the Binder command line, then the Debugger is able to find user symbols. This makes debugging easier. The user symbol facility is only available under AOS.

Table E.1 Macroassembler command line function switches cont

Symbol	Action
Source file/**S**	Skips the file named *sourcefile* on the *file*/**S** second pass of assembly. *Sourcefile* must not define any storage words. Typical files that might be skipped include parameter definition files and macro definition files. Skipping such a file on the second assembly pass does not hinder the assembly of other files in the command line. It merely decreases the size of the output listing and reduces assembly time.

Table E.2 Macroassembler command line argument switches

File Names

Even if a source file called *filename* does not end with **.SR**, the Macroassembler will always search first for *filename*.**SR**. If the Macroassembler does not find this file, then it will search for *filename*. The object module produced by the assembly will have the name of the first source file in the command line (unless you specify the /**B** switch or use the **.OB** pseudo op). Therefore, both of the following commands produce the file named A.OB.

) XEQ MASM A B C<↓>

) XEQ MASM A.SR B.SR C<↓>

Both of these commands cause any error messages to go to the console; neither produces a list file.

The command

) XEQ MASM/L=@LPT A B C<↓>

produces the file A.OB, does not produce an error file, and sends the assembly listing to the line printer.

The Macroassembler adds the extension .OB to a specified object file name only if the extension is not already present. Both of the following commands produce an object file named Z.OB (if there are no assembly errors).

) XEQ MASM/B=Z A B C<↓>

) XEQ MASM/B=Z.OB A B C<↓>

Macroassembler Symbol Table Files

The Macroassembler stores semi-permanent and user symbols and macro definitions in disk files. You can define, update, redefine, or delete these symbols and macro definitions in the disk files. For convenience, we refer to disk files as "symbol files", except for the file ?MASM.ST.TMP, which we refer to as the "symbol table file".

Symbol files contain symbol and macro definitions used to construct the Macroassembler's symbol table file, ?MASM.ST.TMP. This symbol table file is constructed each time you invoke the Macroassembler. Symbol files usually contain the definitions of instruction mnemonics, device code mnemonics, MP/OS system call macros and parameters, etc., which are used from assembly to assembly. When you invoke the Macroassembler, you may specify which symbol file will be used. The default file name is MASM.PS; you can use the global **/PS**=*filename* switch to specify a different file. If the Macroassembler cannot find the file you specify, it creates an empty ?MASM.ST.TMP. Otherwise the Macroassembler copies the symbol file into ?MASM.ST.TMP.

Symbol files can be created and updated. Suppose you want to add the definitions from the assembly language source files A.SR, B.SR, and C.SR to MASM.PS. An appropriate CLI command line to invoke the Macroassembler is:

) X MASM/S A B C<↓>

The global **/S** switch indicates that the Macroassembler is producing a new symbol file, MASM.PS, which contains the original MASM.PS, together with A.SR, B.SR and C.SR.

If you want to create, rather than update, the symbol file, then you follow the same procedure with one change. You must modify the first source file, A.SR, so that it starts with an **.XPNG** pseudo op. When the Macroassembler encounters this during the assembly, it deletes ?MASM.ST.TMP, then creates a new, empty ?MASM.ST.TMP. At the end of pass 1 the symbol table file contains the definitions from the source files only. The Macroassembler deletes MASM.PS as before and renames ?MASM.ST.TMP. The result is a new MASM.PR containing only the definitions from the assembly language source lines.

Note that you should write the assembly language files used to make symbol files in the form of parameter files. This kind of file can contain symbol and macro definitions but does not produce any object code. Make sure that all symbols and macros will be defined during pass 1, since the assembly terminates before pass 2. A parameter file must not define a symbol with a value during pass 1 that would be different if it were assembled in pass 2.

You can also use parameter files in full two-pass assembly which will produce object code. Suppose PROG.SR is the source file for a user-program and that it uses definitions from a parameter file PAR.SR. An appropriate command to assemble the sources would be:

) X MASM PAR/S PROG<↓>

The local **/S** switch on the parameter file name indicates that the file is a parameter file and need not be scanned during pass 2. This is because all of the symbol and macro definitions involved will have been recorded in the symbol table file by the end of pass 1 and will not change during pass 2. You do not have to use the /S switch, but its use will make the assembly take less time.

You can save additional time if you are going to use the parameter file PAR.SR in several assemblies. You can do this by building a symbol file, perhaps called SYMB.PS, containing the definitions from PAR.SR as well as all the definitions from the original symbol file, MASM.PS.

```
X MASM/S=SYMB.PS PAR<↓>
```

Now besides the original MASM.PS, there is a new symbol file called SYMB.PS which can be used to assemble PROG.SR without using the parameter file PAR.SR.

```
X MASM/PS=SYMB.PS PROG<↓>
```

Symbol files and symbol table files have the same internal format. Each file consists of a symbol definition section and a macro definition section. The symbol section of a file can hold approximately 8,000 symbols. The Macroassembler truncates long symbol names, retaining only the first five characters. Using names shorter than five characters will not increase the number of symbols a symbol file can hold. The macro definition section of the file can hold approximately half a million characters of macro definition strings.

Appendix F
Macroassembler Error Codes

Macroassembler error messages appear as single letter flags in the first three character positions of a listing line. If a line of code contains one error, the error flag appears in character position three of that line. If there is a second error in a line, the second error flag appears in character position two. A third error in a line causes an error flag to appear in character position one. A fourth or subsequent error does not cause a flag to appear in the code, but it is included in the total error count.

The Macroassembler will write lines containing errors to the specified error file and as part of your assembly listing. You can usually suppress output of errors to the error file. If you suppress the program listing, the error listing is written to the error file. If you suppress both the assembly and the error listings, then the error listing is sent to the console. Table F.1 lists the error codes and their symbols.

The following pages provide explanations and examples of each of the error codes. However, the examples do not show all possible causes of assembly errors.

Symbol	Error Code
A	Address error
B	Bad character, bad line
C	Macro error
D	Radix error
E	Equivalence error
F	Format error
G	Global reference error
K	Repetitive assembly or conditional error
L	Location counter error
M	Multiply-defined symbol error
N	Number error
O	Overflow field error or stack error
P	Phase error
Q	Questionable line error
R	Relocation error
U	Undefined symbol error
V	Variable label error
X	Text error

Table F.1 Error codes and their symbols

A

Addressing Error

Indicates an addressing error in a memory reference instruction.

Example

In this example a page zero relocatable instruction tries to reference a normal relocatable (.NREL) address.

```
                      .NREL
  00003'000010 G:     10
                      .ZREL
 A00000-044000        STA 0,G
```

Example

In this example an .NREL location tries to reference an address outside the program location counter's address range.

```
                                ;(.-200<=disp
                                ;<=.+177).
              .NREL    0
A00004'020000' LDA     0,Y      ;Y is outside the
       004423' .LOC    .+416    ;instruction's
  00423'000002 Y:2              ;address range.
```

B

Bad Character

Indicates an illegal character in some symbol. This type of error often leads to other errors.

Example

In this example the label contains an illegal character: %.

```
                        .NREL
B00000'024023 .A%:      LDA     1,23
```

C

Macro Error

Occurs under the following circumstances:

- You specify more than 63_{10} arguments.
- The macro has exhausted assembler working space. This should only occur if your macro definition causes endless recursion.
- You attempted to continue the definition of a macro which was not the last one you defined.

The example illustrates the last circumstance.

Example

```
     .MACRO A     ;This defines macro A.
     .
     .
     .
  %
     .            ; other code
     .
     .
     .MACRO A     ;This is a legal continuation of macro
     .            ;A's definition.
     .
  %
     .MACRO B     ;This is a new macro.
     .
     .
     .
  %
     .            ;Other code
     .
     .
  C  .MACRO A     ;Since you have begun a new macro, B,
                  ;you cannot continue to define macro
                  ;A.
```

D

Radix Error

Occurs in three instances:

- The argument in a **.RDX** command is not within the range of 2-20.
- The argument in a **.RDXO** command is not within the range of 8-20.
- You use a digit not within the current input radix.

Example

```
      D 000030          .RDX      4*6
```

Example

```
        000002          .RDX      2
  D00000 000013 B:      35
```

E

Equivalence Error

Occurs when an equivalence line contains an undefined symbol on the righthand side of the equal sign. This error may occur on pass 1 before the symbol on the righthand side has been defined or on pass 2 if the symbol was never defined.

Example

```
EUU             A=B     ;Pass 1 - B is
                        ;unidentified.

EUU    000000   A=B     ;Pass 2 - B is
                        ;unidentified.
```

F

Format Error

Occurs when you try to use a format that is illegal for the current line. This error often occurs in conjunction with other errors.

When you make a format error in an instruction, the instruction-generated code includes only those fields assembled before the error was detected.

Example

```
F00000 143000 ADD    2            ;Not enough
                                  ;arguments.
```

Example

```
F00000 041410 STA    0,10,3,SNC  ;Too many
                                 ;arguments and
                                 ;wrong operand
                                 ;for instr. type.
```

Example

```
       060512 .DUSR  C=DIAS
F00000 060512 C      1            ;This symbol ;does not need
                                  ;arguments.
```

G

Global Symbol Error

Occurs when there is an error in the declaration of an external or entry symbol.

Example
In this example HH is never defined.

```
GU  .ENT        HH
    .END
```

Example
In this example, AA is an entry in a program which declares AA external.

```
G  AA:
            .EXTN       AA
            .END
```

K

Conditional Assembly Error

Occurs when an **.ENDC** pseudo op does not have a preceding **.DO** or **.IF***x* psuedo op.

Example

```
000002  .DO      2
        .
        .
        .
        .ENDC
     K  .ENDC
```

L

Location Error

Occurs when errors are detected in lines affecting the location counter.

If the expression in a **.LOC** or a **.BLK** statement evaluates to less than zero, then the Macroassembler will flag the line with an **L**. An **L** will also flag such a line if the expression in a **.LOC** or a **.BLK** statement cannot be evaluated on the first pass of the assembler. In either case, the Macroassembler ignores the **.LOC** or the **.BLK** and leaves the location counter unchanged.

Example

```
L 177777          .LOC     -1
```

Example

```
77711'000000 A:   0
     L 100012'    .BLK     .+100
```

Example

```
LU 000000         .BLK B            ;B undefined.
```

M

Multiple Definition Error

Occurs each time you attempt to redefine a symbol in a program. Within an assembly program, labels may be defined only once. Also, when **IM** is set, semi-permanent symbols may not be redefined. The Macroassembler will flag any such multiply-defined symbol with an **M** each time the symbol appears.

Multiple occurrences of the **.OB** pseudo-op will also result in **M** errors.

Example

```
 M00000'000000 ALPH:    0
PM00001'000001 ALPH:    1
```

Note that the second definition of **ALPH** is also flagged as a phase error (**P**) on the second pass. See the Phase Error entry.

N

Number Error

Occurs if a single precision integer is greater than or equal to 2^{16}, or if a double precision number is greater than or equal to 2^{32}. The Macroassembler truncates the former number to 16 bits, and the latter to 32 bits.

Example

```
000012     .RDX        10
N000140  65539
000003
```

O

Field Overflow Error

Occurs in the following cases:

- You exceed the size of the stack.
- You issue a **.POP** or a **.TOP** without having previously issued a **.PUSH**.
- You code an instruction operand which is not within the required limits.
- You supply a value for a field which already contains a value.

When overflow occurs in an instruction field the field remains unchanged.

Example

```
000000 020775 LDA    5,.-3     ;AC field is too
                               ;large.
       070400 .DIAC  R=DIA
                     2,0
000001 070400 R      1         ;AC field already
                               ;has value.
000002 000000 .POP             ;Stack is empty.
000003 000000 .TOP             ;Stack is empty.
```

P

Phase Error

Occurs during pass 2 when the Macroassembler detects some unexpected difference from the source program scan on pass 1. For example, a symbol defined on the first pass which has a different value on the second pass will cause a phase error. If you multiply-define a symbol, the **M** error will flag each statement containing the symbol; the **P** error will flag the second and later statements containing the symbol.

Example

```
 M00001 000000 B:      0
PM00002 000001 B:      1
```

Example

```
  00000 000001         .BLK        .PASS
 P00001 000000 C:      0
```

Q

Questionable Line

Occurs under the following conditions:

- You used a # or @ atom improperly.
- You used a **.ZREL** value where the Macroassembler expected an absolute value.
- You used a conditional skip instruction immediately before a two-word instruction.
- You wrote an illogical ALC instruction.

Example

```
Q00002 113000            ADD    0,@2      ;Incorrect use of
                         .ZREL            ;@.
  00000-000010 FLD:      .BLK   10
        000001           .NREL  1
Q000000!000000           LDA    0,FLD,2   ;Macroassembler
                                          ;expects an
                                          ;absolute field
  000011!125015          MOV#   1,1,SNR   ;The MOV
                                          ;instruction
Q000002!000000           ELDA   0,SYMB    ;precedes a
                                          ;two-word
                                          ;instruction.
Q000003!105010           MOV#   0,1       ;No-load bit is
                                          ;set here, but no
                                          ;skip specified.
```

R

Relocation Error

Indicates one of the following:

- The Macroassembler cannot evaluate an expression to a legal relocation type (absolute, word-relocatable, or byte-relocatable).
- The expression mixes .ZREL and pure .NREL symbols.
- The expression mixes impure .NREL and .ZREL symbols.
- The expression mixes pure and impure .NREL symbols.

Example

```
      000000           .NREL 0
  00000'000010 E:      10
  00001'000000"        E+E       ;Contents are .NREL
                                 ;byte-reloctable.
 R00002'000000'        E+E+E     ;Contents not absolute,
                                 ;word-relocatable, or
                                 ;byte-relocatable.
```

Example

```
                       .ZREL
  00000-000000 A:      0
      000001           .NREL 1
  000001000000         0
 R00001!000000! B:     A+B       ;A and B are of different
                                 ;relocation types.
```

U

Undefined Symbol Error

Occurs on pass 2 when the Macroassembler encounters a symbol whose value was not defined after pass 1. Occurs on pass 1 when a symbol definition depends on another symbol whose value is unknown at that time.

> **NOTE:** *A symbol does not have to be defined on pass 1 if it is already defined in MASM.PS.*

Example

```
U00002 030000 LDA     2,B        ;B is as yet unknown.
```

See also the example given in the entry for Equivalence Error (E).

V

Variable Label Error

Occurs if anything other than a symbol follows the **.GOTO** or **.ENDC** pseudo ops.

Example

```
            .GOTO  AUG      ;Legal if AUG is defined.
FV 000000   .GOTO  14       ;14 is an illegal symbol.
                   .
                   .
                   .
            [AUG]  ...
```

Example

```
            .ENDC  HQF      ;Legal if HQF is defined.
            .ENDC  14       ;14 is an illegal symbol.
            .
            .
            .
            [HQF]  ...
```

X

Text Error

Occurs if the two expression delimiters < and > within a text string do not enclose a recognizable arithmetic or logical expression. You cannot use relational operators within text strings.

Example

```
                     .NREL  0
   00000'00001 X:    1
   00001'00002 Y:    2
X00002'054476        .TXT   #<X+ Y>#     ;Spaces not
                                         ;allowed in
       000000                            ;expressions.
X00004'000000        .TXT   #<+>#        ;Lacks operands.
X00005'000076        .TXT   #<X=>Y>#     ;Illegal relational
                                         ;operator
       054476                            ;The
                                         ;Macroassembler
                                         ;sees
       000000                            ;<X=> as the
                                         ;expression,
                                         ;which is not a
                                         ;legal arithmetic
                                         ;or logical
                                         ;expression.
                     .END
```

Appendix G
Using Overlays

Overlay Programming Considerations

Only pure code (specified by the assembler **.NREL 1** directive) may be placed in an overlay. If any overlay object files contain impure code, that code is placed in the main program's impure area by the Binder.

If an **.ENTO** directive is found in a routine that you do not bind into an overlay, then the Binder sets the overlay descriptor to -1. The **?OVLOD** and **?OVREL** routines perfrom no action if they are called with a -1 descriptor. This means that you can wait until bind time to decide whether or not to put a routine in an overlay.

The Binder pre-loads each node with the code of its first overlay, so these overlays will already be in main memory when your program begins running.

When you use **?OVLOD** to request an overlay that is already loaded, the system does *not* load a new copy. Therefore overlay code should not be self-modifying.

An overlay routine may not call another overlay into the same node.

To protect multitasked programs, the system maintains a *use count* for each overlay node. This count is incremented whenever a task executes an **?OVLOD** for the node, and decremented whenever a task executes an **?OVREL**.

When a task requests an overlay, some other task may be using a different overlay in the same node. In this case, the requesting task is blocked from execution until the node's use count becomes zero. Then the system loads the new overlay and unblocks any tasks that are waiting for it. Multitasking is explained fully later in this chapter.

If your program is not multitasked, and you neglect to release an overlay, then the next **?OVLOD** that requests a different overlay for the node will "hang" your program (block it from execution indefinitely).

A program may contain up to 128 nodes, and each node may have up to 256 overlays.

Before doing an **?EXEC** system call in an overlayed program, you must load each node with its number 0 overlay. To illustrate the use of overlays, we will discuss a typical program called MPRG. The programmer has decided that there are six subroutines which are little used and should therefore be placed in overlays. Some of these routines call each other, so two overlay nodes are needed. One node is to hold subroutines A, B, and C, each in its own overlay. In the other node are two overlays: one containing subroutine D, and another containing two subroutines, E and F.

Assembling Overlay Programs

To use overlays with a program, you must declare the names of the entry points of the overlay routines. You must also declare the names of the overlays themselves. You use the assembler's **.EXTN** directive to declare all of these names as external symbols.

The names of the overlays must be placed in your program's data area so that they can be referenced at run time. The Binder will replace the names with *overlay descriptors* (described shortly) which are used by the library routines.

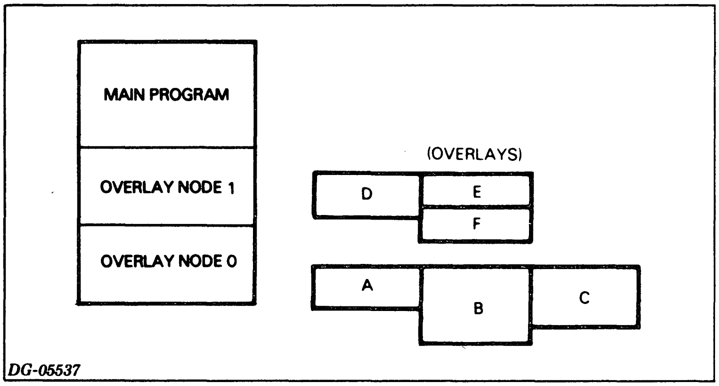


Figure G.2

The code for our sample program, MPRG, contains declarations of the following form:

```
          ;MAIN PROGRAM DECLARATIONS
          .EXTN    OVL1,OVL2,OVL3,   ;Overlay descriptors
          .EXTN    OVL4,OVL5
          .EXTN    A,B,C,D,E,F       ;Subroutine entries
DESC1:    OVL1
DESC2:    OVL2
DESC3:    OVL3
DESC4:    OVL4
DESC5:    OVL5
.A:       A
.B:       B
.C:       C
.D:       D
.E:       E
.F:       F
```

Within the overlay source files, you must declare all the entry points with the **.ENT** directive. You must also declare the overlay names with the **.ENTO** directive.

Each of the six subroutines contains declarations of the following form:

```
;SUBROUTINE DECLARATIONS (subroutine A)
      .ENTO OVL1              ;Overlay descriptor
      .ENT A                  ;Name of entry
      .
      .
      .
A:    ;(subroutine entry point)
      .
      .
      .
```

It is not necessary for you to explicitly allocate space for the overlay nodes. This is taken care of by the Binder.

The format of an overlay descriptor is shown in the following diagram:

*	NODE NUMBER	OVERLAY NUMBER
0	1–7	8–15

Reserved for future use.

For normal programming, there is no need for you to know this format; it is included for the sake of completeness.

Binding Overlay Programs

After assembling the main program and the overlays, you use the Binder to determine the actual distribution of nodes and overlays. The Binder then creates the program and overlay files.

Our sample program has seven object modules: MPRG.OB for the main program, and A.OB, B.OB, etc. for the subroutines. The programmer binds the program using a command of the form:

X BIND MPRG !* A ! B ! C *! !* D ! E F *!

This command contains special symbols that define the overlay structure to the Binder. The symbols **!*** and ***!** indicate the start and end of an overlay node. The symbol **!** defines separate overlays within a node. For instance, the string **!* D ! E F *!** identifies an overlay node with two overlays: one for D, and another for E and F.

You must use delimiters (such as spaces) to separate the symbols **!**, **!***, and ***!** from each other, and from the object program names.

The Binder analyzes the command, and allocates space for the overlay nodes. Each node will be allocated enough memory to hold its largest overlay. The Binder then assigns values to the overlay descriptors, and places these values in locations DESC1 through DESC5 of the main program. The Binder also resolves the references to the subroutine entries. The result of the binding process is a program file, MPRG.PR, and an overlay file, MPRG.OL.

For more information on binding overlays, refer to the MP/OS Binder documentation, Section 4 of *MP/OS Utilities Reference.*

Overlay Library Routines

There are two library routines that support overlays. You use the **?OVLOD** routine to load the overlay into its node. You then jump to the desired entry address. After exiting from the routine, you use the **?OVREL** routine to release the overlay.

The main program in our example must contain library calls to manipulate the overlays, as shown by the following example:

```
;MAIN PROGRAM OVERLAY CODE
MPRG:   (start of program)      ;Initialization.
        LDA 0,OVL1              ;Get descriptor for routine.
        ?OVLOD                  ;Load the overlay.
         JMP ERROR              ;(Error return)
         JSR @.A                ;Call the subroutine.
        LDA 0,OVL1              ;Then set up for ?OVREL.
        ?OVREL                  ;Release the overlay.
         JMP ERROR              ;(Error return)
        .
        .
        .
```

Appendix H
CLI Message Format

This appendix describes the format of messages passed to programs by the MP/OS operating system. The syntax that the CLI provides to the user is more complex than that described here: there are a number of features, such as command repetition and filename templates, which are interpreted by the CLI and not passed to programs. This appendix only describes the format of CLI messages as the program sees them. You can use the **?TMSG** library routine to translate such a message.

For more information on the CLI command language, see *MP/OS Utilities Reference*.

Arguments

CLI messages consist of a program name alone or of a program name followed by one or more arguments. An argument may consist of a filename, function name, or any other string of characters. If the message contains one or more arguments, they are separated by commas. The last argument (or the program name itself if there are no arguments) is always followed by a null byte.

Switches

Switches are modifiers that may follow the program name or any argument. All switches are preceded by a slash (/) and may consist of one or more characters. Switches have two forms: simple and keyword. A simple switch has the format:

/name

A keyword switch has the format:

/name=value

Value may be any number, filename, etc.

Consider the following example of a command to MASM, the MP/OS Macroassembler program. You might type a CLI command on your console:

```
X MASM/U/L=@LPT DEFS/S PROG
```

The **X** is the minimum unique abbreviation of **XEQ**. MASM is the name of the program you want to run. **/U** is a simple switch which instructs MASM to include user symbols in the object file it generates. **/L=@LPT** is a keyword switch: it instructs MASM to send the listing file to the line printer (device name @LPT). The arguments DEFS and PROG are the names of two files to be assembled. The **/S** on the first argument is a simple switch that tells MASM to skip that file on the second assembly pass.

From this command the CLI creates an inter-program message which is of the form:

MASM/U/L=@LPT,DEFS/S,PROG (plus a terminating null byte)

As you can see, the **X** has been removed. Also all strings of spaces have been converted to single commas, and a trailing null byte has been appended to the message as a terminator.

Appendix I
User Parameter Files

All MP/OS systems include a set of *parameter files*. These are assembler source files that contain symbol definitions without any executable code. You use these files to prepare permanent symbol tables for the Macroassembler, using the /S function switch as explained in Appendix E. The table below describes all the parameter files.

File	Contents
NBID.SR	The basic NOVA instruction set.
MP100ID.SR	Instructions used only on MP/100 systems.
MP200ID.SR	Instructions used only on MP/200 systems.
MPID.SR	MP/200 instructions that are convertible from microNOVA to ECLIPSE. Only used with the System Call Translator.
NSKID.SR	Conditional skip mnemonics that simplify programming and improve program readability.
SYSID.SR	Definitions of system calls.
MPARU.SR	Error codes and other mnemonics used with system calls.

Table I.1

This Appendix contains a listing of one parameter file, MPARU.SR, since you will refer to it when you need to know the numeric value of a MP/OS symbol. The list of error codes in MPARU.SR is particularly useful when debugging programs.

> **NOTE:** *This listing of MPARU.SR reflects the state of the system at the time when this manual was printed. The values of some mnemonics may change from time to time. If there is any doubt about the value of a symbol, you should check the copy of MPARU.SR that was released with your system.*

```
0001 MPARU AOS ASSEMBLER REV 02.05                07/15/79 21:46:29

              ; ==========================
              ; MPARU - USER PARAMETER FILE
              ; ==========================


                     .TITLE  MPARU            ; MICRON User Parameter File


     000000          .NOMAC  0

     000040 group = 40
            ;start of error codes
     040001 .dusr    ERNAR =    group*1000+1     ;*Argument does not exist
     040002 .dusr    ERBTL =    group*1000+2     ;*Buffer extends into
                                                 ;system space
     040003 .dusr    ERIRB =    group*1000+3     ; Buffer too short
     040004 .dusr    ERPRM =    group*1000+4     ; Cannot delete
                                                 ;permanent file
     040005  dusr    ERREN =    group*1000+5     ;*Renaming error (file is
                                                 ;open  cross device)
     040006  dusr    ERDVC =    group*1000+6     ;*Invalid device code
     040007 .dusr    ERDAI =    group*1000+7     ; Device is in use
     040010  dusr    ERDFT =  group*1000+10      ;*Fatal device error
     040011 .dusr    ERDOL =  group*1000+11      ;*Device is off line
     040012  dusr    ERFIL =  group*1000+12      ; Device read error
     040013 .dusr    ERPWL =  group*1000+13      ; Device write error
     040014 .dusr    ERDID =  group*1000+14      ; Directory delete error
     040015 .dusr    ERLAB =  group*1000+15      ;*Disk label does not
                                                 ; match disk name
     040016 .dusr    ERFIX =  group*1000+16      ; Disk requires fixup
     040017 .dusr    EREOF =  group*1000+17      ; End of file
     040020 .dusr    ERUIH =  group*1000+20      ;*Extant user interrupt
                                                 ; handler
     040021 .dusr    ERNAE =  group*1000+21      ; File already exists
     040022 .dusr    ERFDE =  group*1000+22      ; File does not exist
     040023 .dusr    EREOP =  group*1000+23      ;*File is in use
     040024 .dusr    ERATP =  group*1000+24      ;*File is attribute
                                                 ; protected
     040025 .dusr    ERFTL =  group*1000+25      ; File name is too long
     040026 .dusr    ERIFT =  group*1000+26      ; Illegal file type
     040027 .dusr    ERIOO =  group*1000+27      ; Illegal option
                                                 ; combination
     040030 .dusr    ERSTS =  group*1000+30      ; Invalid stack
                                                 ; definition (too small.
                                                 ; system space)
     040031 .dusr    ERSPC =  group*1000+31      ; Insufficient file space
     040032 .dusr    ERMPR =  group*1000+32      ; Invalid address
     040033  dusr    ERMWT =  group*1000+33      ;*Multiple waiters
                                                 ; for single NPSC
     040034  dusr    ERIAT =  group*1000+34      ;*Invalid attributes
     040035 .dusr    ERICN =  group*1000+35      ; Invalid channel number
     040036  dusr    ERIFC =  group*1000+36      ; Invalid character
                                                 ; in pathname
     040037  dusr    ERICH =  group*1000+37      ;*Invalid
                                                 ; characteristics
     040040 .dusr    EREVT =  group*1000+40      ;*Invalid event number
                                                 ;( > ?EVMAX or < ?EVMIN )
     040041 .dusr    ERMEM =  group*1000+41      ; Invalid memory request
     040042 .dusr    ERIOD =  group*1000+42      ;*Invalid operation
                                                 ; for device
     040043 .dusr    ERPRP =  group*1000+43      ; Invalid priority
     040044 .dusr    ERADR =  group*1000+44      ; Invalid starting
                                                 ; address
     040045 .dusr    ERTID =  group*1000+45      ; Invalid task
                                                 ; identifier
     040046  dusr    ERLTL =  group*1000+46      ; Line is too long
     040047 .dusr    ERNDP =  group*1000+47      ;*No debugger present
     040050  dusr    ERNMC =  group*1000+50      ; No free channels
     040051 .dusr    ERNOT =  group*1000+51      ; No free TCB
                                                 ; available
     040052 .dusr    ERNUI =  group*1000+52      ;*No such user interrupt
                                                 ; service routine exists
     040053 .dusr    ERNAD =  group*1000+53      ; Non-directory entry
                                                 ; in pathname
     040054 .dusr    ERNSY =  group*1000+54      ;*Non-system name
                                                 ; specified
     040055 .dusr    ERTMO =  group*1000+55      ;*Pend timeout
     040056 .dusr    ERANG =  group*1000+56      ;*Range error
     040057 .dusr    ERRAD =  group*1000+57      ; Read access denied
```

```
0002 MPARU
      040060 .dusr    ERSTL =  group*1000+60      ;*Searchlist overflow
      040061 .dusr    ERNSS =  group*1000+61      ;*Switch not found
      040062 .dusr    ERTIP =  group*1000+62      ;*Task in progress
      040063 .dusr    ERWAD =  group*1000+63      ; Write access denied
      040064 .dusr    ERYSL =  group*1000+64      ;*You should live so long
      040065 .dusr    ERISC =  group*1000+65      ;*Illegal system call
      040066 .dusr    ERINT =  group*1000+66      ;*Internal error
      040067 .dusr    ERRNA =  group*1000+67      ;*No available resource
      040070 .dusr    ERCIN =  group*1000+70      ;*Console interrupt
                                                  ; ( C^A)
      040071  dusr    ERABT =  group*1000+71      ;*Son terminated via ^C^A
      040072  dusr    ERIPT =  group*1000+72      ;*Illegal packet type
      040073  dusr    ERPCA =  group*1000+73      ;*Call aborted due to
                                                  ; program management call
      040074  dusr    ERVNS =  group*1000+74      ; Program file format
                                                  ; revision not supported
      040075  dusr    ERDNM =  group*1000+75      ;*Device not mounted
      040077  dusr    EROVN =  group*1000+77      ; Invalid overlay
                                                  ; descriptor
      040101  dusr    EREXS = group*1000+101      ; Attempt to exceed
                                                  ; maximum swap level
      040102 .dusr    ERNOV = group*1000+102      ;*No overlays defined
                                                  ; for this program
      040103 .dusr    EROVC = group*1000+103      ;*Specified overlay is
                                                  ; not currently in use
      040104 .dusr    ERATD = group*1000+104      ;*All tasks have died
      040105 .dusr    ERUSD = group*1000+105      ;*User and system de-
                                                  ; buggers can not coexist
      040106 .dusr    ERNEM = group*1000+106      ; Not enough memory
      040107 .dusr    ERABK = group*1000+107      ;*Son terminated via ^C^E
      040110 .dusr    ERESZ = group*1000+110      ;*Invalid element size
      040111 .dusr    ERIFF = group*1000+111      ;*Invalid file format
                                                  ; (bad SA file)
      040112  dusr    ERJMO = group*1000+112      ;*User PC is equal to
                                                  ; zero
      040113  dusr    ERSAD = group*1000+113      ;*Scheduling already
                                                  disabled (from ?ERSCH.CK)
             ;end of error codes



             ;     Note:   The * is a MICRON specific error code which

             ;     has no AOS counterpart.
```

```
0003 MPARU
             ;
             ; MICRON error classes returned on ?EXEC
             ;
       000000  DUSR    ?ECCP=  0               ; code returned by
                                               ; called program
       000001 .DUSR    ?ECEX=  ?ECCP+1         ; the error occured
                                               ; while attempting the
                                               ; ?EXEC. the called
                                               program did not run
       000002 .DUSR    ?ECRT=  ?ECEX+1         ; the error occured on a
                                               ; ?RETURN which
                                               ; did not complete. this
                                               ; error is seen
                                               ; by the grandparent of
                                               ; the program
                                               ; attempted the ?RETURN
       000003 .DUSR    ?ECBK=  ?ECRT+1         ; the error occured
                                               ; while trying to
                                               ; write a breakfile
       000004 .DUSR    ?ECAB=  ?ECBK+1         ; the error returned
                                               ; indicates an
                                               ; abnormal termination
                                               ; (e.g   C^B) as
                                               ; opposed to the usual
                                               ; ?RETURN

                        .EJECT
```

```
0004 MPARU

        000020 .DUSR    ?NMCH    = 16.              ; Number of MICRON channels

        000000 .DUSR    ?INCH    = 0                ; Default console input
                                                    ; channel
        000001 .DUSR    ?OUCH    = 1                ; Default console output
                                                    ; channel

        000001  DUSR    ?EVCH    = 1                ; Channel 0  C A event code
        000021 .DUSR    ?EVMIN   = ?EVCH+?NMCH      ; Minimum user pend event
        077777 .DUSR    ?EVMAX   = 77777            ; Maximum user pend event

        000017 .DUSR    ?MXFL    = 15.              ; Maximum MICRON filename
                                                    ; length
        000177 .DUSR    ?MXPL    = 127              ; Maximum MICRON
                                                    ; pathname length


               ; MICRON FILE TYPES

        177772 .DUSR    ?DLPT    = -6               ; Lineprinter
        177773 .DUSR    ?DCHR    = -5               ; Character device
        177774 .DUSR    ?DDVC    = -4               ; Disk (directory device)
        177775 .DUSR    ?DDIR    = -3               ; Directory
        177776 .DUSR    ?DMSG    = -2               ; message file
        177777 .DUSR    ?DPSH    = -1               ; Push (Break) file

        000000 .DUSR    ?DSMN    = 0                ; System min
        000100 .DUSR    ?DSMX    = 100              ; System max
        000101 .DUSR    ?DUMN    = ?DSMX+1          ; User minimum
        000200 .DUSR    ?DUMX    = 200              ; User maximum

                        ; system types

        000000  DUSR    ?DOBF    = ?DSMN            ; OB file
        000001 .DUSR    ?DSTF    = ?DOBF+1          ; Symbol table file
        000002  DUSR    ?DPRG    = ?DSTF+1          ; Program file
        000003  DUSR    ?DOLF    = ?DPRG+1          ; Overlay file
        000004 .DUSR    ?DBPG    = ?DOLF+1          ; Bootable program file
        000005 .DUSR    ?DPST    = ?DBPG+1          ; Permanent symbol file
                                                    ; (x PS)
        000006  DUSR    ?DLIB    = ?DPST+1          ; Library file (x.LB)
        000007  DUSR    ?DUDF    = ?DLIB+1          ; User data file
        000010  DUSR    ?DTXT    = ?DUDF+1          ; Text File

0005 MPARU

               ; MICRON FILE ATTRIBUTES

        000001 .DUSR    ?ATPM    = 1                ; Permanent file. can't
                                                    ; be deleted
        000002 .DUSR    ?ATRD    = 2                ; Can't be  read
        000004 .DUSR    ?ATWR    = 4                ; Can't be written
        000010 .DUSR    ?ATAT    = 8.               ; Attributes can't be
                                                    ; changed
        000020 .DUSR    ?ATDC    = 16.              ; Delete on last close
                                                    ; (user can't set this)
        000040 .DUSR    ?ATMP    = 32.              ; Sector remap (user
                                                    ; can't alter)
        000100 .DUSR    ?ATZR    = 64               ; Don t zero blocks on
                                                    ; allocation


               ; MICRON DEVICE CHARACTERISTICS

        100000  DUSR    ?CST     = 1B0              ; Simulate tabs if asserted
        040000 .DUSR    ?CNAS    = 1B1              ; If asserted  device is
                                                    ; not ANSI standard
        020000 .DUSR    ?CESC    = 1B2              ; Interpret escape as
                                                    ; ^C^A sequence
        010000 .DUSR    ?CECH    = 1B3              ; If asserted, echo input
                                                    ; to output
        004000 .DUSR    ?CLST    = 1B4              ; If asserted, echo form
                                                    ; feed as  L
        002000 .DUSR    ?CBIN    = 1B5              ; If asserted. input is
                                                    ; in binary form (8 bit)
        001000 .DUSR    ?C605    = 1B6              ; If asserted. device is
                                                    ; 605x series
        000400 .DUSR    ?CUCO    = 1B7              ; Convert lowercase as
                                                    ; uppercase
```

```
0006 MPARU

          ;
          ===========================================================
          ; MICRON packets are all typed   The zero th word of each
          ; and  every packet must contain the type code for that
          ; packet  The actual packet begins at offset 1   The packet
          ; length  includes the type word
          ;
          ===========================================================
          ;        MICRON packet types

     000000  DUSR    ?PIP     = 0                ; Rev 0 program information
                                                 ; packet
     000001 .DUSR    ?TDP     = 1                ; Rev 0 task definition
                                                 ; packet
     000002 .DUSR    ?FSP     = 2                ; Rev 0 file status packet
     000003 .DUSR    ?DSP     = 3                ; Rev 0 disk status packet

     000000 .DUSR    ?TYPE    = 0                ; Offset of type word in
                                                 ; the packet



          ;
          ===========================================================
          ;
          ;    THE PROGRAM INFORMATION PACKET USED BY THE ?INFO CALL


     000001 .DUSR    ?PPMN    = ?TYPE+1          ; Lowest pure (code) address
     000002  DUSR    ?PPMX    = ?PPMN+1          ; Highest pure address
     000003 .DUSR    ?PIMN    = ?PPMX+1          ; Lowest impure (data)
                                                 ; address
     000004 .DUSR    ?PIMX    = ?PIMN+1          ; Highest impure address
     000005 .DUSR    ?PREV    = ?PIMX+1          ; Program revision number
     000006  DUSR    ?PLEV    = ?PREV+1          ; Program level
     000007 .DUSR    ?PHMA    = ?PLEV+1          ; Highest memory available
     000010  DUSR    ?POCH    = ?PHMA+1          ; Open channel mask

     000011 .DUSR    ?PLN     = ?POCH+1          ; Length of the P.I.P.



          ;
          ===========================================================
          ;
          ;    THE TASK DEFINITION PACKET USED BY THE ?CTASK CALL


     000001 .DUSR    ?TPRI    = ?TYPE+1          ; Task priority (0 =<x=<255)
     000002 .DUSR    ?TSTA    = ?TPRI+1          ; Task starting address
     000003 .DUSR    ?TSTB    = ?TSTA+1          ; Stack base
     000004 .DUSR    ?TSTL    = ?TSTB+1          ; Stack limit
     000005 .DUSR    ?TSTE    = ?TSTL+1          ; Stack error handler
                                                 ; (0=>system default)
     000006 .DUSR    ?TAC2    = ?TSTE+1          ; New task's initial ac2
     000007 .DUSR    ?TUSP    = ?TAC2+1          ; New task s initial ?usp
     000010 .DUSR    ?TKPP    = ?TUSP+1          ; Task kill post-procesing
                                                 ; Routine address
                                                 ;  (0 => none)

     000011 .DUSR    ?TLN     = ?TKPP+1          ; Length of a TDP

```

```
0007 MPARU

        ;
=========================================================================
        ;
        ;       THE FILE STATUS PACKET USED BY THE ?FSTAT CALL

000001 .DUSR   ?FTYP   = ?TYPE+1        ; File type
000002 .DUSR   ?FATR   = ?FTYP+1        ; Attributes
000003 .DUSR   ?FESZ   = ?FATR+1        ; Element size
000004 .DUSR   ?FTLA   = ?FESZ+1        ; Time last accessed
                                        ; (two words)
000006 .DUSR   ?FTLM   = ?FTLA+2        ; Time last modified
                                        ; (two words)
000010 .DUSR   ?FLEN   = ?FTLM+2        ; File length in bytes
                                        ; (two words)

000012 .DUSR   ?FLN    = ?FLEN+2        ; Length of the file
                                        ; status packet

        ;
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
        ;           *  *  *    N O T E    *  *  *
        ;
        ;  The DIT structure conforms to offsets ?FTYP thru ?FTLM
        ;  Do not alter these offsets and expect ?FSTAT to work
        ;
        ;
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-



        ;
=========================================================================
        ;
        ;       THE MESSAGE PACKET USED BY THE ?TMSG CALL

000000 .DUSR   ?GREQ   = 0              ; Packet/request type
                                        ; (see below)
000001 .DUSR   ?GNUM   = ?GREQ+1        ; Argument number
000002 .DUSR   ?GSW    = ?GNUM+1        ; Switch specifier
000003 .DUSR   ?GRES   = ?GSW+1         ; B.P. to buffer
                                        ; recieving switch

000004 .DUSR   ?GTLN   = ?GRES+1        ; Length of ?TMSG pagket

        ;       PACKET / REQUEST TYPES   (?GREQ)

000000  DUSR   ?GCMD   = 0              ; Get entire message
000001 .DUSR   ?GCNT   = ?GCMD+1        ; Get argument count
000002 .DUSR   ?GARG   = ?GCNT+1        ; Get argument
000003  DUSR   ?GTSW   = ?GARG+1        ; Test a switch
000004 .DUSR   ?GSWS   = ?GTSW+1        ; Get (alphapetic) switches
000005 .DUSR   ?GSWI   = ?GSWS+1        ; Test for switch # ?GSW


        ;=======================================================================
        ;
        ;       The disk status packet used by the ?DSTAT call
        ;

000001 .DUSR   ?DFB    = ?TYPE+1        ; Two word # of free blocks
000003 .DUSR   ?DAB    = ?DFB+2         ; Two word # of allocated
                                        ; blocks
000005 .DUSR   ?DTMX   = ?DAB+2         ; Maximum possible # of
                                        ; files
000006 .DUSR   ?DTAL   = ?DTMX+1        ; Current # of allocated
                                        ; DITs
000007 .DUSR   ?DSTW   = ?DTAL+1        ; Disk status word (see
                                        ; below)
000010 .DUSR   ?DRER   = ?DSTW+1        ; Number of recoverable
                                        ; disk errors
000011 .DUSR   ?DUER   = ?DRER+1        ; Number of unrecoverable
                                        ; disk errors

000012 .DUSR   ?DLN    = ?DUER+1        ; Length of the packet


        ; Disk status bits (offset ?DSTW)


0008 MPARU

100000 .DUSR   ?DWRP   = 1B0            ; Disk is write protected
040000 .DUSR   ?DLE1   = 1B1            ; Primary label block is bad
020000 .DUSR   ?DLE2   = 1B2            ; Secondary label block is
                                        ; is bad
010000 .DUSR   ?DME1   = 1B3            ; Primary MDV block is bad
004000 .DUSR   ?DME2   = 1B4            ; Secondary MDV block is bad
```

```
0009 MPARU

          ;
          -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
          ;
          ;     All of the following parameters/macros are assembler
          ;     parameters and do not need to be included for Pascal
          ;
          ;
          -=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=


          ;        Structure of a MICRON DCT (system imposed structure)


   000000 .DUSR   ?IHNDL  = 0              ; Address of ISR
   000001 .DUSR   ?IMSK   = 1              ; Mask to be or'ed with
                                           ; current system mask


   000062 .DUSR   ?STKMIN= 50.             ; Minimum stack size
   000016 .DUSR   ?USP    = 16             ; User stack pointer
   000042 .DUSR   CSL     = 42             ; Stack limit word


          ; Define the default frame pointer relative offsets into
          ; the save block

   177774 .DUSR   ?OAC0   = -4
   177775  DUSR   ?OAC1   = -3
   177776 .DUSR   ?OAC2   = -2
   177777 .DUSR   ?OFP    = -1
   000000  DUSR   ?ORTN   = 0

   000001 .DUSR   ?TMP    = 1              ; First free stack loc
                                           ; rel to FP



          ; --------  the real-time program description block
            --------
          ;           this is a packet produced for programs
          ;           bound with the /sa or /sp switches. It
          ;           is based on the symbol ?ZSPA



   000000 .dusr   ?rusp = 0                ; ?usp word
   000001 .dusr   ?rusl = ?rusp+1          ; user stack limit
   000002 .dusr   ?rusb = ?rusl+1          ; user stack base
   000003 .dusr   ?rust = ?rusb+1          ; user starting address
   000004 .dusr   ?rsii = ?rust+1          ; start of impure
                                           ; initialization area
   000005  dusr   ?reii = ?rsii+1          ; end of impure
                                           ; initialization area
   000006  dusr   ?rtmt = ?reii+1          ; highest available
                                           ; memory address
   000007  dusr   ?rtsi = ?rtmt+1          ; start of user impure area
   000010 .dusr   ?rtei = ?rtsi+1          ; end of user impure area
   000011 .dusr   ?rtsp = ?rtei+1          ; start of user pure area
   000012 .dusr   ?rtep = ?rtsp+1          ; end of user pure area

   000013 .dusr   ?rtln = ?rtep+1          ; length of the packet



                  .EOF                     ; PARUM.SR


0010 MPARU

**00000 TOTAL ERRORS  00000 PASS 1 ERRORS
```

```
0011 MPARU

CSL    000042    9/19#
ERABK  040107    2/22#
ERABT  040071    2/10#
ERADR  040044    1/49#
ERANG  040056    1/59#
ERATD  040104    2/19#
ERATP  040024    1/33#
ERBTL  040002    1/15#
ERCIN  040070    2/09#
ERDAI  040007    1/20#
ERDFT  040010    1/21#
ERDID  040014    1/25#
ERDNM  040075    2/14#
ERDOL  040011    1/22#
ERDVC  040006    1/19#
EREOF  040017    1/28#
EREOP  040023    1/32#
ERESZ  040110    2/23#
EREVT  040040    1/45#
EREXS  040101    2/16#
ERFDE  040022    1/31#
ERFIL  040012    1/23#
ERFIX  040016    1/27#
ERFTL  040025    1/34#
ERIAT  040034    1/41#
ERICH  040037    1/44#
ERICN  040035    1/42#
ERIFC  040036    1/43#
ERIFF  040111    2/24#
ERIFT  040026    1/35#
ERINT  040066    2/07#
ERIOD  040042    1/47#
ERIOO  040027    1/36#
ERIPT  040072    2/11#
ERIRB  040003    1/16#
ERISC  040065    2/06#
ERJMO  040112    2/25#
ERLAB  040015    1/26#
ERLTL  040046    1/51#
ERMEM  040041    1/46#
ERMPR  040032    1/39#
ERMWT  040033    1/40#
ERNAD  040053    1/56#
ERNAE  040021    1/30#
ERNAR  040001    1/14#
ERNDP  040047    1/52#
ERNEM  040106    2/21#
ERNMC  040050    1/53#
ERNOT  040051    1/54#
ERNOV  040102    2/17#
ERNSS  040061    2/02#
ERNSY  040054    1/57#
ERNUI  040052    1/55#
EROVC  040103    2/18#
EROVN  040077    2/15#
ERPCA  040073    2/12#
ERPRM  040004    1/17#
ERPRP  040043    1/48#
ERPWL  040013    1/24#

0012 MPARU

ERRAD  040057    1/60#
ERREN  040005    1/18#
ERRNA  040067    2/08#
ERSAD  040113    2/26#
ERSPC  040031    1/38#
ERSTL  040060    2/01#
ERSTS  040030    1/37#
ERTID  040045    1/50#
ERTIP  040062    2/03#
ERTMO  040055    1/58#
ERUIH  040020    1/29#
ERUSD  040105    2/20#
ERVNS  040074    2/13#
ERWAD  040063    2/04#
ERYSL  040064    2/05#
GROUP  000040    1/12#  1/14   1/15   1/16   1/17   1/18   1/19
                 1/20   1/21   1/22   1/23   1/24   1/25   1/26
                 1/27   1/28   1/29   1/30   1/31   1/32   1/33
                 1/34   1/35   1/36   1/37   1/38   1/39   1/40
                 1/41   1/42   1/43   1/44   1/45   1/46   1/47
                 1/48   1/49   1/50   1/51   1/52   1/53   1/54
                 1/55   1/56   1/57   1/58   1/59   1/60   2/01
                 2/02   2/03   2/04   2/05   2/06   2/07   2/08
                 2/09   2/10   2/11   2/12   2/13   2/14   2/15
                 2/16   2/17   2/18   2/19   2/20   2/21   2/22
                 2/23   2/24   2/25   2/26
?ATAT  000010    5/07#
?ATDC  000020    5/08#
?ATMP  000040    5/09#
?ATPM  000001    5/04#
?ATRD  000002    5/05#
?ATWR  000004    5/06#
?ATZR  000100    5/10#
?C605  001000    5/21#
?CBIN  002000    5/20#
?CECH  010000    5/18#
?CESC  020000    5/17#
?CLST  004000    5/19#
?CNAS  040000    5/16#
?CST   100000    5/15#
?CUCO  000400    5/22#
?DAB   000003    7/50#  7/51
?DBPG  000004    4/35#  4/36
?DCHR  177773    4/18#
?DDIR  177775    4/20#
?DDVC  177774    4/19#
?DFB   000001    7/49#  7/50
?DLE1  040000    8/03#
?DLE2  020000    8/04#
?DLIB  000006    4/37#  4/38
?DLN   000012    7/57#
?DLPT  177772    4/17#
?DME1  010000    8/05#
?DME2  004000    8/06#
?DMSG  177776    4/21#
?DOBF  000000    4/31#  4/32
?DOLF  000003    4/34#  4/35
?DPRG  000002    4/33#  4/34
?DPSH  177777    4/22#
```

0013 MPARU

?DPST	000005	4/36#	4/37
?DRER	000010	7/54#	7/55
?DSMN	000000	4/24#	4/31
?DSMX	000100	4/25#	4/26
?DSP	000003	6/13#	
?DSTF	000001	4/32#	4/33
?DSTW	000007	7/53#	7/54
?DTAL	000006	7/52#	7/53
?DTMX	000005	7/51#	7/52
?DTXT	000010	4/39#	
?DUDF	000007	4/38#	4/39
?DUER	000011	7/55#	7/57
?DUMN	000101	4/26#	
?DUMX	000200	4/27#	
?DWRP	100000	8/02#	
?ECAB	000004	3/13#	
?ECBK	000003	3/11#	3/13
?ECCP	000000	3/04#	3/05
?ECEX	000001	3/05#	3/07
?ECRT	000002	3/07#	3/11
?EVCH	000001	4/07#	4/08
?EVMA	077777	4/09#	
?EVMI	000021	4/08#	
?FATR	000002	7/07#	7/08
?FESZ	000003	7/08#	7/09
?FLEN	000010	7/11#	7/13
?FLN	000012	7/13#	
?FSP	000002	6/12#	
?FTLA	000004	7/09#	7/10
?FTLM	000006	7/10#	7/11
?FTYP	000001	7/06#	7/07
?GARG	000002	7/39#	7/40
?GCMD	000000	7/37#	7/38
?GCNT	000001	7/38#	7/39
?GNUM	000001	7/29#	7/30
?GREQ	000000	7/28#	7/29
?GRES	000003	7/31#	7/33
?GSW	000002	7/30#	7/31
?GSWI	000005	7/42#	
?GSWS	000004	7/41#	7/42
?GTLN	000004	7/33#	
?GTSW	000003	7/40#	7/41
?IHND	000000	9/13#	
?IMSK	000001	9/14#	
?INCH	000000	4/04#	
?MXFL	000017	4/11#	
?MXPL	000177	4/12#	
?NMCH	000020	4/02#	4/08
?OACO	177774	9/24#	
?OAC1	177775	9/25#	
?OAC2	177776	9/26#	
?OFP	177777	9/27#	
?ORTN	000000	9/28#	
?OUCH	000001	4/05#	
?PHMA	000007	6/29#	6/30
?PIMN	000003	6/25#	6/26
?PIMX	000004	6/26#	6/27
?PIP	000000	6/10#	
?PLEV	000006	6/28#	6/29

0014 MPARU

?PLN	000011	6/32#				
?POCH	000010	6/30#	6/32			
?PPMN	000001	6/23#	6/24			
?PPMX	000002	6/24#	6/25			
?PREV	000005	6/27#	6/28			
?REII	000005	9/46#	9/47			
?RSII	000004	9/45#	9/46			
?RTEI	000010	9/49#	9/50			
?RTEP	000012	9/51#	9/53			
?RTLN	000013	9/53#				
?RTMT	000006	9/47#	9/48			
?RTSI	000007	9/48#	9/49			
?RTSP	000011	9/50#	9/51			
?RUSB	000002	9/43#	9/44			
?RUSL	000001	9/42#	9/43			
?RUSP	000000	9/41#	9/42			
?RUST	000003	9/44#	9/45			
?STKM	000062	9/17#				
?TAC2	000006	6/46#	6/47			
?TDP	000001	6/11#				
?TKPP	000010	6/48#	6/52			
?TLN	000011	6/52#				
?TMP	000001	9/30#				
?TPRI	000001	6/40#	6/41			
?TSTA	000002	6/41#	6/42			
?TSTB	000003	6/42#	6/43			
?TSTE	000005	6/44#	6/46			
?TSTL	000004	6/43#	6/44			
?TUSP	000007	6/47#	6/48			
?TYPE	000000	6/15#	6/23	6/40	7/06	7/49
?USP	000016	9/18#				

Appendix J Running MP/OS Programs Under AOS

The MP/OS System Call Translator is a powerful software package that enables you to develop and run MP/OS programs on an ECLIPSE line computer under AOS, the Advanced Operating System. The System Call Translator translates MP/OS system calls into their AOS counterparts.

Programs used on the MP/OS system AOS can be moved with no modification except for rebinding. This means that you can write programs in, for example, MP/Pascal and run them under AOS by simply rebinding.

The System Call Translator consists of three parts: a set of *parameter files*, a *translator object module*, and a *subroutine library file*. The parameter files contain the definitions of the MP/OS system calls and parameters. The Translator package also includes a copy of MASM.PS, the assembler's permanent symbol table, which has been prepared using this symbol table. You assemble your program with this file, instead of the usual MASM.PS. The object module and the library contain pre-assembled code which interfaces your program to the AOS environment.

Operating Procedures

Assembling

To prepare a MP/OS program to run under AOS, you must assemble it using MMASM, the Macroassembler. You type the following CLI command:

X MMASM *pathname [...pathname]*

Each *pathname* represents the pathname of one or more files to be assembled.

> **NOTE:** *Be sure that the MASM.PS that MMASM sees is the Translator's MASM.PS.*

Binding

After assembly, you use the MP/OS Binder to prepare an executable program file. You type the command:

X MBIND/AOS *pathname[...pathname]* <↓>

For this command to work properly, MICREM.OB and MMSL.LB must be on your searchlist. MICREM and MMSL.LB are the translator object module and the subroutine library, respectively. *Pathname* represents the pathname of one or more object files to be bound. Note that MICREM must come first, and MSL.LB must come last in the command.

The result of this process is the program file *progname*.**PR**, which can be run on an AOS system.

Once the program has been assembled with the MICREM parameter file, a rebind is all that is required to transfer it to a MP/OS system.

For more information refer to the *MP/OS Utilities Reference*, Section 4, "The Binder".

Compatibility of System Calls

Since the AOS environment is somewhat different than that of the MP/OS system, there are some minor differences in the actions of some of the system calls. These are detailed in the following paragraphs.

When an AOS system call causes your program to take an error return, the Translator will pass the corresponding MP/OS error code to your program. If the error code has no MP/OS counterpart, your program will receive the AOS code.

Program Management

The **?RETURN** call with the **BK** option uses the AOS convention for the break file name: *?pid.time*.**BRK**, where *pid* is your process I. D. and *time* is the current time of day. Also, a program which terminates with a **?RETURN BK** may not pass a message to the parent program.

The **?EXEC'd** call, with AC0 pointing to CLI.PR will invoke the AOS CLI with the appropriate message format. The user's message must be in the MP/OS CLI format with the CLI as the zero argument.

An **?EXEC** program will fail if it attempts to use a channel passed from the parent if that channel was opened exclusivly.

The **?BOOT** call does not perform a bootstrap. It attempts to return to the user's CLI (no matter how many levels down it is). The message gives the reason for returning, and the name of the specified bootstrap device or file.

File Management

AOS supports file type numbers which are somewhat different from MP/OS file types. The System Call Translator converts MP/OS file types to their AOS counterparts when you create files. It also converts AOS file types to their MP/OS counterparts when you open files that were created by AOS programs. The correspondences between file types are summarized by Tables J.1 and J.2.

MP/OS	AOS	Meaning
?DDIR	?FDIR	Directory.
?DPRG	?FPRG	Program file.
?DUDF	?FUDF	User data file.
?DTXT	?FTXT	Text file.

Table J. 1 Conversions of MP/OS file types when creating files under AOS

NOTE: *All file types not mentioned in the table above are converted to* **?FUDF**.

AOS	MP/OS	Meaning
?FLPU	?DLPT	Line printer.
?FGFN	?DCHR	AOS generic file.
?FPRG	?DPRG	Program file.
?FDIR	?DDIR	Directory.
?FCPD	?DDIR	AOS control point directory.
?FTXT	?DTXT	Text file.
?FUDF	?DUDF	User data file.

Table J.2 Conversions of AOS file types when opening files with MP/OS programs

NOTE: *All file types not mentioned in the table above are converted to* **?DUDF**.

File attributes are also handled differently on the two systems. The MP/OS System Call Translator intercepts the references in your program to all file attributes except permanence, and translates them into elements on the *access control list* (ACL) of the file. The ACL is a file protection feature provided by AOS, which is described fully in the AOS manuals listed in the Bibliography. The correspondences between attributes and access types are summarized by Table J3:

Note that there is a reversal in polarity between the two systems: setting the MP/OS read protect attribute for a file means that it may *not* be read, while setting the AOS **R** access priviledge for a file means that it *may* be read. (This conversion is handled by the Translator).

MP/OS Attributes	AOS Access Privileges
Read protection	Read access (R)
Write protection	Write access (W)
Attribute protection	Owner access (O)

Table J.3 Reversal in polarity between MP/OS attributes and AOS access privileges

The permanence attribute is handled identically under the AOS and MP/OS systems, so there is no difference in its use under the Translator.

I/O Device Management

The AOS and MP/OS systems have different formats for device characteristics. The **?GCHAR** and **?SCHAR** calls perform the conversion between characteristics, so that the difference is transparent to your program. The correspondences between the two systems are summarized by Table J.4.

MICRON Name	AOS Name
?CST	?CST
?CNAS	?CNAS
?CESC	?CESC
?CECH	?CEOC
?CUCO	?CUCO
?CLST	Not supported.
?CBIN	Supported for @TTI, @TTO, @TTI1, @TTO1, @LPT
?C605	?C605

Table J.4 Correspondences between device characteristics

?MOUNT, ?DISMOUNT, ?IDEF, ?IRMV, ?IUNPEND and **?IXIT** are unimplemented. They produce an error return with code **ERISC** (Illegal System Call).

There is a mapping between the MP/OS system and AOS for the various devices, shown in Table J.5. The System Call Translator recognizes the MP/OS device name and converts it to its AOS counterpart.

MP/OS Device Name	AOS Device Name
@TTIO	@Input (or @null if the program is batched).
@TTOO	@Output
@TTI1	@Data
@TTO1	@List

Table J.5 Device name mapping

Appendix K
MP/OS Fatal Errors

There are some conditions under which the MP/OS operating system may detect an error condition from which it cannot recover. Such errors are called *fatal errors*, and are extremely rare. The most common cause of fatal errors is erroneous behavior by a user program, such as overwriting part of system memory.

When the system detects a fatal error, it shuts itself down at once to prevent further loss of data. At this time it types a message on the console:

FATAL ERROR CODE:

followed by six octal numbers.

The *code* is a number which identifies the cause of the error, as listed in the table. The six numbers are the contents of the accumulators (AC0 - AC3), the stack pointer, and the frame pointer. You should write down these numbers as well as the error code, since they may be of use to you or Data General personnel in finding the cause of the error.

Code	Meaning
0	Internal system call error. AC0 contains the system call error code.
1	System checksum error: the system has detected erroneous internal data.
2	System infinite loop: part of the MP/OS program code has been destroyed. You should write down the contents of memory locations 5, 6, and 7, as well as the numbers mentioned in the above text.
3	I/O or other error occurred during a shutdown or bootstrap operation. AC0 contains the system call error code.
4	An interrupt was received from an unknown device, and the system was unable to clear it. AC1 contains the device code.
5	An interrupt was received with a device code greater than 76_8. AC1 contains the device code.
6	The system was unable to execute :CLI.PR.
7	The system was unable to load or release one of its overlays.
10	Internal inconsistency. You should write down the contents of memory locations 5, 6, and 7, as well as the numbers mentioned in the above text.

Table K.1

Appendix L
Generating MP/OS Systems

This Appendix describes the SYSGEN program, which you use to generate new MP/OS systems that are tailored to your specific needs. SYSGEN asks you a number of questions about the features you need; then it calls the MP/OS Binder to build a suitable system file. It also stores the results of your dialogue in a *script* file; this file can be used as input to a later SYSGEN session.

You can create two basic types of systems:

- Program development systems. These are general purpose, disk-based systems which provide all MP/OS features.
- Runtime systems. These are very small systems which are intended to support stand-alone programs that are either bootstrapped on a development system *(bootable)*, or permanently stored in read-only memory *(PROMable)*.

The procedures for generating runtime systems are detailed in Appendix M.

Using SYSGEN

To use SYSGEN, you type a CLI command of the form:

XEQ SYSGEN[/I=*file***][/DEFAULT][/50HZ]**
    sysname

On AOS systems, the command has the same form, except that the program name is **MSYSGEN** instead of **SYSGEN**.

SYSGEN will then begin its dialogue with you, and will build a system called *sysname*. The name of the program file will have a suffix of **.SY** if you build a program development system, and a suffix of **.RS** if you build a runtime system. For a program development system, SYSGEN will create an overlay file, whose filename has the suffix **.OL**. SYSGEN also creates a script file of your dialogue, which it places in a file having the name *sysname* with no suffix.

Default Responses

For each question that SYSGEN asks, it supplies a *default* answer which it will use if you type a null reply, i.e., a New-line. This saves you time when building a system with mostly standard features. The default answer appears in square brackets; for example:

How many terminals will this system support? (0..6) [1]:

If you type New-line in response to this question, SYSGEN will build your system to support a single terminal.

If you specify an existing script file with the **/I=***file* switch, then SYSGEN will use the contents of that script for the default answers to all questions. This makes it easy for you to build a system that is similar to a previously created one.

If you wish to build a system that is identical to a previously created one, you use the **/DEFAULT** switch along with the **/I=***file* switch. In this case, there is no dialogue; SYSGEN simply builds a system using all the answers in the specified script file.

Real time Clocks

Any Microproducts system may be equipped with the Model 4220 real time clock. If you do not have this device, you have several options depending on the type of CPU, as detailed below.

mN601 systems may use the CPU's internal real time clock, which has a frequency of about 108Hz.

MP/100 systems may use either the CPU's internal clock (also about 108Hz), or the line frequency clock (50 or 60Hz).

MP/200 systems with the basic controller board may use a clock frequency of 10, 100, or 1000Hz, or the line frequency.

SYSGEN normally builds systems to run with an AC line frequency of 60Hz. If your line frequency is 50Hz, be sure to use the **/50HZ** command switch.

Optimizing Memory Usage

SYSGEN asks several questions that permit you to adjust the system's use of memory in order to improve performance. The system allocates memory for four types of data structures:

- **Stacks**. A system stack is required for any system call; in a multitasked program, tasks will be blocked if they execute system calls when all system stacks are in use. Each stack requires about 130 words of memory.
- **Buffers**. These are blocks of memory that are used for disk I/O. If the system runs out of buffers, it begins swapping them to and from disk, resulting in a decrease of execution speed. Each buffer requires 256 words.
- **Task control blocks** (TCBs). The system needs a TCB for each non-pended system call which is active at any given time. Each TCB requires about 30 words. (If your program uses multitasking, you will need additional TCB's; however these are allocated in your program's address space, and are not of any concern at SYSGEN time.)
- **File information blocks** (FIBs). One FIB is needed for each open file, although the system can swap them if necessary. Each FIB requires 64 words.

Examples

The remainder of this Appendix consists of several SYSGEN dialogs that show how to create typical program development systems. In the dialogs, text which was typed by the user is shown in **bold face**; text which was typed by SYSGEN is shown in *italics*. All commands are terminated with a New-line. The symbol <↓> is used to indicate that the user typed a New-line; we use this for the sake of clarity in places where the user did not type any other characters.

For examples of how to create runtime systems, see Appendix M.

Example 1

The first example is not really "typical," since it generates a system with at least one of every possible type of device.

Which CPU is this system for? (mn601,MP/100,MP/200) [mn601]: <↓>

The items in parentheses are the possible answers to the question. The item in brackets is the default answer, which will be used if the user simply types New-line, as she did in this case.

Your tailored MICRON will have one of the following forms:
pd -- a program development system
sp -- a promable run-time system
sa -- a bootable stand-alone system

Which form of system should be generated? (pd,sp,sa) [pd]: <↓>

Do you have an external real-time clock? (4220-BS,4220-S) (y,n) [n]: **Y**
Clock's device code (3..76) [14]: <↓>
Clock frequency (10,60,100,1000) [10]: **60**

The clock questions are somewhat different for MP/100 and MP/200 systems, as shown in Examples 5 and 6.

\- - - - - - - - - - - -

How many disk controllers are there? (1..8) [1]: **6**

The following disk controllers are supported:
a.) 330 kb diskette (6038/6039)
b.) 10 mb cartridge (6095-N)
c.) 1.2 mb diskette (6096a/6096b)
d.) 12.5 mb fixed disk + optional 1.2 mb diskette(s) (6101/6102)

Choose the appropriate letter for controller # 1 (a,b,c,d) [a]: <↓>
Enter the device code for this controller (in octal) (3..76) [33]: <↓>
Enter the number of diskette units on this controller (1..2) [2]: <↓>

Unit 0 is named @DPX0
Unit 1 is named @DPX1

Choose the appropriate letter for controller # 2 (a,b,c,d) [a]: **B**
Enter the device code for this controller (in octal) (3..76) [27]: <↓>

The removable pack is named @DPD0
The fixed platter is named @DPD4

Choose the appropriate letter for controller # 3 (a,b,c,d) [a]: **C**
Enter the device code for this controller (in octal) (3..76) [67]: <↓>
Enter the number of diskette units on this controller (1..4) [2]: **3**

Unit 0 is named @DPY0
Unit 1 is named @DPY1
Unit 2 is named @DPY2

Choose the appropriate letter for controller # 4 (a,b,c,d) [a]: **D**
Enter the device code for this controller (in octal) (3..76) [73]: <↓>
Enter the number of diskette units on this controller (0..3) [1]: **0**

The fixed disk is named @DPH0

Choose the appropriate letter for controller # 5 (a,b,c,d) [a]: **D**
Enter the device code for this controller (in octal) (3..76) [63]: <↓>
Enter the number of diskette units on this controller (0..3) [1]: <↓>

The fixed disk is named @DPH10
Unit 1 is named @DPY11

Choose the appropriate letter for controller # 6 (a,b,c,d) [a]: **C**
Enter the device code for this controller (in octal) (3..76) [57]: <↓>
Enter the number of diskette units on this controller (1..4) [2]: **1**

Unit 0 is named @DPY20

- - - - - - - - - - - -

How many terminals will this system support? (0..6) [1]: **5**

Is terminal # 1 a Dasher (TM) display? (y,n) [y]: <↓>
Keyboard device code (3..76) [10]: <↓>
The keyboard is named @TTI
The display is named @TTO

The device code of the display is always one greater than that of the keyboard.

Is terminal # 2 a Dasher (TM) display? (y,n) [y]: **N**
Characters per line (40..132) [72]: **80**
Keyboard characteristics [?cnas+?cech]: **?CNAS+?CECH**
Display characteristics [?cst+?cnas+?cuco]: **?CST+?CNAS+?CUCO**
Keyboard device code (3..76) [50]: <↓>

The keyboard is named @TTI1
The display is named @TTO1

Is terminal # 3 a Dasher (TM) display? (y,n) [y]: **N**
Characters per line (40..132) [72]: **40**
Keyboard characteristics [?cnas+?cech]: **?CECH**
Display characteristics [?cst+?cnas+?cuco]: **?CUCO+?CST**
Keyboard device code (3..76) [40]: **40**

The keyboard is named @TTI2
The display is named @TTO2

Is terminal # 4 a Dasher (TM) display? (y,n) [y]: **N**
Characters per line (40..132) [72]: **72**
Keyboard characteristics [?cnas+?cech]: **?CNAS+?CBIN**
Display characteristics [?cst+?cnas+?cuco]: **?CBIN**
Keyboard device code (3..76) [30]: <↓>

The keyboard is named @TTI3
The display is named @TTO3

Is terminal # 5 a Dasher (TM) display? (y,n) [y]: <↓>
Keyboard device code (3..76) [20]: <↓>

The keyboard is named @TTI4
The display is named @TTO4

\- - - - - - - - - - -

How many lineprinters do you have? (0..8) [0]: **2**

Enter the device code for printer #1 (3..76) [17]: <↓>
Lines per page (20..120) [63]: <↓>
Characters per line (40..136) [72]: <↓>
Characteristics [?cst+?cuco]: **?CST+?CUCO**

This lineprinter is named @LPT

Enter the device code for printer #2 (3..76) [47]: <↓>
Lines per page (20..120) [63]: <↓>
Characters per line (40..136) [72]: <↓>
Characteristics [?cst+?cuco]: **?CUCO**

This lineprinter is named @LPT1

\- - - - - - - - - - -

Should the default system configuration parameters be used? (y,n) [y]: **N** *Number of system buffers (3..20) [6]:* **8**
Number of file information block buffers (2..10) [4]: **6**
Number of free task control blocks (0..100) [1]: **3**
Number of system stacks (1..10) [2]: **4**

\- - - - - - - - - - -

Example 2

This dialogue generates a more typical program development system: one with a single disk and console.

Which CPU is this system for? (mn601,MP/100,MP/200) [mn601]: <↓>

Your tailored MICRON will have one of the following forms:
pd -- a program development system
sp -- a promable run-time system
sa -- a bootable stand-alone system

Which form of system should be generated? (pd,sp,sa) [pd]: <↓>

Do you have an external real-time clock? (4220-BS,4220-S) (y,n) [n]: <↓>

- - - - - - - - - - - -

How many disk controllers are there? (1..8) [1]: <↓>

The following disk controllers are supported:

a.) 330 kb diskette (6038/6039)
b.) 10 mb cartridge (6095-N)
c.) 1.2 mb diskette (6096a/6096b)
d.) 12.5 mb fixed disk + optional 1.2 mb diskette(s) (6101/6102)

Choose the appropriate letter for controller # 1 (a,b,c,d) [a]: <↓>
Enter the device code for this controller (in octal) (3..76) [33]: <↓>
Enter the number of diskette units on this controller (1..2) [2]: <↓>

Unit 0 is named @DPX0
Unit 1 is named @DPX1

- - - - - - - - - - - -

How many terminals will this system support? (0..6) [1]: <↓>

Is terminal # 1 a Dasher (TM) display? (y,n) [y]: <↓>
Keyboard device code (3..76) [10]: <↓>

The keyboard is named @TTI
The display is named @TTO

- - - - - - - - - - - -

How many lineprinters do you have? (0..8) [0]: <↓>

- - - - - - - - - - - -

Should the default system configuration parameters be used? (y,n) [y]: <↓>

- - - - - - - - - - - -

Example 3

This is a *partial* example which shows the difference in the real-time clock questions for an MP/200 system.

Which CPU is this system for? (mn601,MP/100,MP/200) [mn601]: **mp/200**

Your tailored MICRON will have one of the following forms:
pd -- a program development system
sp -- a promable run-time system
sa -- a bootable stand-alone system

Which form of system should be generated? (pd,sp,sa) [pd]: **SP**

Do you have the basic controller board? (y,n) [y]: <↓>
Clock frequency (10,60,100,1000) [10]: <↓>

Example 4

This is a *partial* example which shows the difference in the real-time clock questions for an MP/100 system.

Which CPU is this system for? (mn601,MP/100,MP/200) [mn601]: **mp/100**

Your tailored MICRON will have one of the following forms:
pd -- a program development system
sp -- a promable run-time system
sa -- a bootable stand-alone system

Which form of system should be generated? (pd,sp,sa) [pd]: **SP**

Do you have an external real-time clock? (4220-BS,4220-S) (y,n) [n]: <↓>
Is the CPU jumpered to use line frequency as the time base? (y,n) [y]: <↓>

If you answer N to this question, the system will use the CPU's internal clock.

Appendix M
Generating Stand-alone Programs

This Appendix describes how you can build MP/OS programs to run in specialized environments. There are two basic types of stand-alone programs:

- Those which are loaded from a disk *(bootable)*.
- Those which are permanently stored in programmable read-only memory (PROM) chips *(PROMable)*.

For either type, you have the option of building a *runtime system* into the program. This is a small subset of the MP/OS system which you create with the SYSGEN program (described in Appendix L). The runtime system provides multitasking, I/O, and several other functions. More specifically, all runtime systems support the following system calls:

?INFO*
?MEMI
?PEND
?UNPEND
?IUNPEND
?IXIT
?DRSCH
?ERSCH

**The action of this call is restricted in runtime systems (see Part 3, Chapter 3).*

You can add any of the following system calls during the SYSGEN dialog:

?CTASK
?KTASK
?PRI
?MYID
?IDEF, ?IRMV
?GTIME, ?STIME

SYSGEN adds the **?READ** or **?WRITE** system calls if you specify any input or output devices in your dialog. Also, if you specify any disk devices, SYSGEN will ask if you want to add the **?GPOS** and **?SPOS** system calls.

Note that the **?OPEN** and **?CLOSE** system calls are not supported; I/O channel numbers are assigned by SYSGEN. This means that you will not know what the channel numbers are until you run SYSGEN. Therefore you must either run SYSGEN before assembling your program, or declare the I/O channel numbers as external symbols, and define them in an additional program module which you assemble after running SYSGEN.

If you have no need for the functions of the runtime system, you can omit it from your program, which will decrease the amount of memory required.

Procedure

To create a stand-alone program file, you execute BIND, the MP/OS Binder, with special function switches. The /**SA** switch instructs BIND to build a bootable program. The /**SP** switch instructs BIND to build a PROMable program. If you are using a runtime system, you specify it at this time with the /**RS**=*name* switch.

BIND produces a special type of program file whose filename has a suffix of .**SA** or .**SP**; this file must then be processed by either the MAKEBOOT or PROMLOAD program.

With an .**SA** file, you run the MAKEBOOT program by typing a CLI command of the form:

XEQ MAKEBOOT *name*.**SA**

MAKEBOOT produces a file called *name*.**BPG**, which can be run by the CLI **BOOT** command or the **?BOOT** system call.

An **.SP** file can be loaded directly into PROM chips. To do this, you run the PROMLOAD program, which is described fully in Appendix N.

Controlling Stand-alone Programs

The MP/OS Binder places a *stand_alone program control block* (SPCB) in an **.SA** or **.SP** file. The format of this block is given in the following table:

Mnem.	Contents
?RUSP	**?USP** (general purpose page zero word).
?RUSL	Stack limit.
?RUST	Stack base (starting address).
?RSII	Start of impure initialization area, or 0 (described below).
?REII	End of impure initialization area, or 0.
?RTMT	Highest available memory address.
?RTSI	Starting address of impure area.
?RTEI	Ending address of impure area.
?RTSP	Starting address of pure area.
?RTEP	Ending address of pure area.

Table M.1 Stand-alone program information block

The Binder places the SPCB in the pure area of the program, and creates the symbol **?ZSPA** with the value of the block's starting address.

The Binder initializes location 77777_8 of the program to have the value 177764_8. Therefore you should place the starting address of your program into location 77764_8, for compatibility with the hardware power-up logic. You should place the address of your power-fail handling routine in location 77765_8.

If you do not specify a top of memory with the Binder's /**MTOP**=*number* switch, the Binder assumes a value of 77763_8.

Initializing the Impure Area

If you create an **.SP** program that declares any data in the impure area of memory, then the Binder builds an *impure initialization block* (IIB). This block is placed in the *pure* area of the program, so that it will be programmed into PROM.

If you are using a MP/OS runtime system, the system will initialize your impure memory upon power-up. First it sets all words in impure memory (RAM) to 0. Then it loads the data from the IIB into the specified locations. If you are not using a runtime system, then your program will need to do these functions. The Binder puts the IIB's starting and ending addresses into the SPCB, so that your program can find the IIB when it starts. (Note that in an **.SA** program, there is no need for an IIB, so the Binder sets the pointers in the SPCB to zero.)

The IIB consists of a series of entries, each of which has one of the two formats shown in Figure M.1. One format is for single words, and the other is for groups of words.

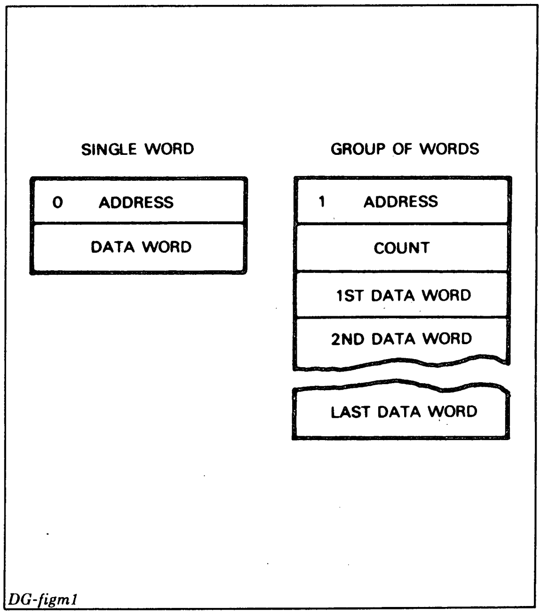


Figure M.1 Impure initialization block entries

Examples

The following sample SYSGEN dialogs show how the program is used to create runtime systems. In the dialogs, text which was type by the user is shown in **bold face**; text which was typed by SYSGEN is shown in *italics*. All commands are terminated by a New-line. The symbol <↓> is used to indicate that the user typed a New-line; we use this for the sake of clarity in places where the user typed no other characters.

For more information on SYSGEN, see Appendix L.

Example 1

This dialogue creates a PROMable runtime system with no standard I/O devices.

Which CPU is this system for? (mn601,MP/100,MP/200) [mn601]: <↓>

Your tailored MICRON will have one of the following forms:
pd -- a program development system
sp -- a promable run-time system
sa -- a bootable stand-alone system

Which form of system should be generated? (pd,sp,sa) [pd]: **SP**

Do you have an external real-time clock? (4220-BS,4220-S) (y,n) [n]: <↓>

\- - - - - - - - - - - -

How many disk controllers are there? (0..8) [0]: <↓>

\- - - - - - - - - - - -

How many terminals will this system support? (0..6) [1]: **0**

\- - - - - - - - - - - -

How many lineprinters do you have? (0..8) [0]: <↓>

\- - - - - - - - - - - -

Should the default system configuration parameters be used? (y,n) [y]: <↓>

\- - - - - - - - - - - -

Allow ?CTASK (y,n) [n]: **Y**
Allow ?KTASK (y,n) [n]: **Y**
Allow ?MYID (y,n) [n]: **Y**
Allow ?PRI (y,n) [n]: **Y**
Allow ?IDEF and ?IRMV (y,n) [n]: **Y**
Allow ?GTIME and ?STIME (y,n) [n]: **Y**

Example 2

This dialogue creates a PROMable runtime system with a console.

Which CPU is this system for? (mn601,MP/100,MP/200) [mn601]: <↓>

Your tailored MICRON will have one of the following forms:
pd -- a program development system
sp -- a promable run-time system
sa -- a bootable stand-alone system

Which form of system should be generated? (pd,sp,sa) [pd]: **SP**

Do you have an external real-time clock? (4220-BS,4220-S) (y,n) [n]: <↓>

- - - - - - - - - - - -

How many disk controllers are there? (0..8) [0]: <↓>

- - - - - - - - - - - -

How many terminals will this system support? (0..6) [1]: **1**

Is terminal # 1 a Dasher (TM) display? (y,n) [y]: <↓>
Keyboard device code (3..76) [10]: <↓>

The keyboard is named @TTI
The display is named @TTO

- - - - - - - - - - - -

How many lineprinters do you have? (0..8) [0]: <↓>

- - - - - - - - - - - -

Should the default system configuration parameters be used? (y,n) [y]: <↓>

- - - - - - - - - - - -

Allow ?CTASK (y,n) [n]: **Y**
Allow ?KTASK (y,n) [n]: **Y**
Allow ?MYID (y,n) [n]: **Y**
Allow ?PRI (y,n) [n]: **Y**
Allow ?IDEF and ?IRMV (y,n) [n]: **Y**
Log fatal errors (y,n) [n]: **Y**

If you answer N to this question, the system will simply halt in the event of a fatal error. If you answer Y, a message will be typed on the console identifying the error (note that some extra memory is needed to support this function.)

Allow ?GTIME and ?STIME (y,n) [n]: **Y**

Appendix N
PROM Generation

This Appendix describes the PROMTAPE program, which is used to store programs and data on programmable read-only memory chips (PROMs). PROMTAPE conducts a dialog with you to obtain the name of the input file, and the type of output that you desire. It can produce three different types of output:

- A binary output file that is compatible with many PROM programmers when punched onto paper tape.
- A series of ASCII characters that can be sent directly to a PROM programmer via an asynchronous interface.
- A listing of the output data in octal and hexadecimal, for use with manually operated PROM programmers.

PROMTAPE may be used with PROMs that are 8 bits wide and 512, 1024, 2048, or 4096 bytes long. It assumes that the PROMs will be arranged so that the starting memory address for each one will be a multiple of the PROM's size: for instance, PROMs with a length of 1024 (2000_8) will start at memory addresses 2000_8, 4000_8, 6000_8, etc.

Operating Instructions

To use PROMTAPE, you type the CLI command:

X PROMTAPE

PROMTAPE then asks the following questions:

Input file?

Type the name of the file that you wish to store in PROMs. This file must be a PROMable program (see Appendix M).

Output file format? (asynch, binary, or <nl> for none)

Type **A** to send the output to an asynchronous interface, or **B** to produce a binary file. If you type New-line without a letter, no output file will be produced.

What is the console number of the PROM programmer?

PROMTAPE only asks this if you typed **A** in response to the previous question. Type the number of the console interface to which the PROM programmer is attached; for example, type **2** to indicate the console whose pathname is **TTI2** (input) and **TTO2** (output).

Output file?

PROMTAPE only asks this if you typed **B** as the output file format. Type the name that you want the output file to have. You may type the pathname of your system's paper tape punch to send the output directly to it.

Listing file? (<nl> for none)

Type the name you want for the listing file; or type New-line without a filename if you do not want a listing.

PROM length? (512, 1024, 2048, or 4096)

Type the number which is the length in bytes of each PROM chip.

Starting address? (octal)

Type the memory address that the first word of your program is to occupy.

NOTES: *This number should not be lower than* 1000_8*, as this would result in PROMs occupying the part of the address space that is reserved for lower page zero.*

If this address falls in the middle of a PROM and you have specified **B** *(binary) as the output type, PROMTAPE rounds the address down to the start of a PROM. If you have not specified binary output, PROMTAPE will start programming in the middle of a PROM.*

Ending address? (octal)

Type the last memory address that is to be loaded.

Output Data Formats

Listing

The following figure shows a section of a typical PROMTAPE listing file.

PROM SET 1
memory address = 01000 to 01200

prom addr	hex high	data low	oct high	data low	memory addr	data
000	20	8D	040	215	01000	020215
001	D4	00	324	000	01001	152000
002	38	8E	070	216	01002	034216
003	43	00	103	000	01003	041400
004	4B	03	113	003	01004	045403
005	53	05	123	005	01005	051405
006	D5	00	325	000	01006	152400
007	53	07	123	007	01007	051407
008	53	08	123	010	01010	051410
009	30	8F	060	217	01011	030217
00A	53	01	123	001	01012	051401
00B	F2	00	362	000	01013	171000
00C	0C	0F	014	017	01014	006017
00D	80	03	200	003	01015	100003
00E	0C	60	014	140	01016	006140
00F	AF	C8	257	310	01017	127710
010	E7	C8	347	310	01020	163710
011	00	00	000	000	01021	000000
012	30	36	060	066	01022	030066
013	85	00	205	000	01023	102400
014	28	38	050	070	01024	024070
015	A8	00	250	000	01025	124000
016	42	00	102	000	01026	041000
017	D3	00	323	000	01027	151400
018	AB	04	253	004	01030	125404
019	01	FD	001	375	01031	000775
01A	AF	C8	257	310	01032	127710
01B	E7	C8	347	310	01033	163710
01C	00	00	000	000	01034	000000
01D	20	2F	040	057	01035	020057
01E	30	34	060	064	01036	030064
01F	97	00	227	000	01037	113400
020	50	31	120	061	01040	050061
021	D2	50	322	120	01041	151120
022	AD	00	255	000	01042	126400
023	20	2A	040	052	01043	020052
024	38	8E	070	216	01044	034216
025	43	00	103	000	01045	041400
026	4B	07	113	007	01046	045407
027	53	08	123	010	01047	051410
028	F2	00	362	000	01050	171000
029	FD	00	375	000	01051	176400
02A	5A	05	132	005	01052	055005
02B	38	91	070	221	01053	034221
02C	5A	01	132	001	01054	055001
02D	0C	0F	014	017	01055	006017
02E	80	16	200	026	01056	100026
02F	0C	62	014	142	01057	006142

Table N.1

Binary File

A PROMTAPE binary file is divided into segments, each of which contains the data for one PROM. Each segment is preceded by a single byte with all bits set to 1 (377_8). Each segment is followed by a series of null bytes (0_8), so that when the file is punched, there is about 3 inches of blank paper tape between segments.

The segments are ordered in pairs, since it takes two 8-bit wide PROMs to occupy a section of 16-bit wide memory. The first segment in the file contains the high order bytes for the first pair of PROMs. It is followed by the low order bytes for the same pair, then the high order bytes for the next pair, etc.

The following diagram shows the format of a PROMTAPE binary file when punched on paper tape.

Asynchronous Output

PROMTAPE can communicate with a suitable PROM programmer over an asynchronous line. The protocol which it uses is detailed below:

1. PROMTAPE sends a sequence of characters of the form *ssseeeP* to the PROM programmer, where *sss* is the starting address, and *eee* is the ending address. The two addresses are represented by ASCII characters that form a hexadecimal number. For example, PROMTAPE sends *0001FFP* to completely program a 512-byte PROM.
2. The PROM programmer replies with a byte containing 6_8 if a blank PROM is loaded in it; otherwise, it sends a byte containing 25_8. Then it sends bytes containing 3_8 and 15_8 to tell PROMTAPE that it is ready to start.
4. PROMTAPE sends the first byte of data.
5. The PROM programmer replies with 6_8 if the it succeeds in programming this byte; otherwise it replies with 25_8.
6. Steps 4 and 5 are repeated for each byte of data.
7. The PROM programmer sends a byte containing 4_8 when it finishes programming.

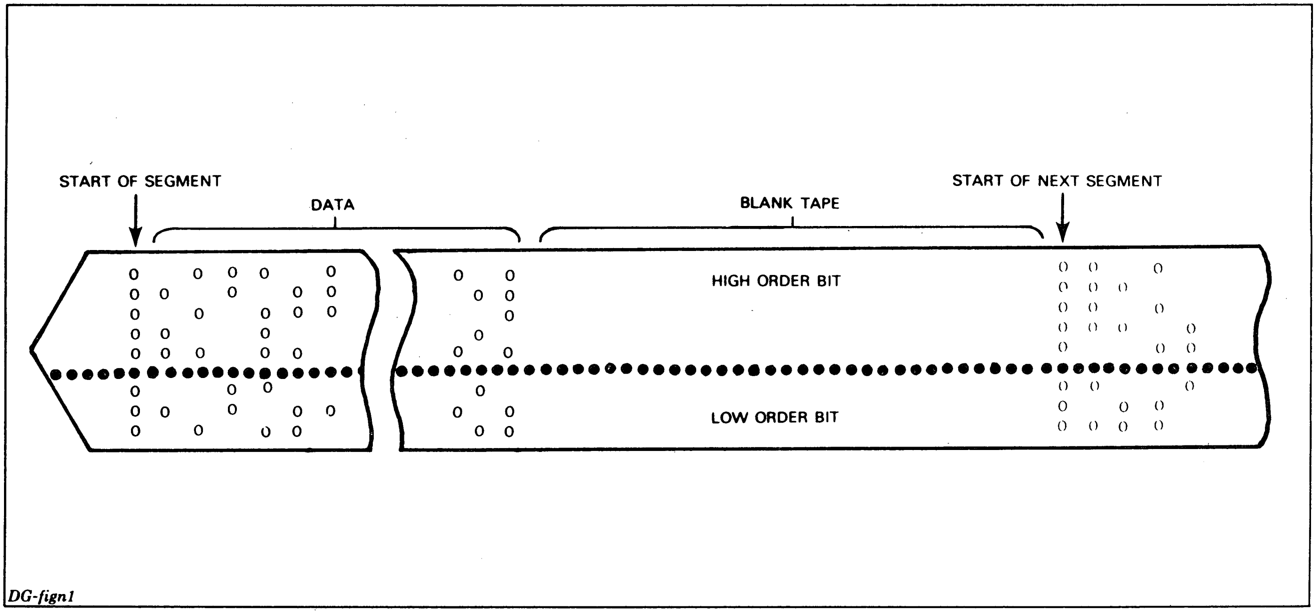


Figure N.4 Figure N.1 Paper tape format

SALES AND SERVICE OFFICES

Alabama: Birmingham
Arizona: Phoenix, Tucson
Arkansas: Little Rock
California: El Segundo, Fresno, Palo Alto, Sacramento, San Diego, San Francisco, Santa Ana, Santa Barbara, Van Nuys
Colorado: Englewood
Connecticut: North Branford
Florida: Ft. Lauderdale, Orlando, Tampa
Georgia: Norcross
Idaho: Boise
Illinois: Peoria, Schaumburg
Indiana: Indianapolis
Kentucky: Louisville
Louisiana: Baton Rouge
Maryland: Baltimore
Massachusetts: Springfield, Wellesley, Worcester
Michigan: Southfield
Minnesota: Richfield
Missouri: Kansas City, St. Louis
Nevada: Las Vegas
New Hampshire: Nashua
New Jersey: Cherry Hill, Wayne
New Mexico: Albuquerque
New York: Buffalo, Latham, Melville, Newfield, New York, Rochester, Syracuse, White Plains
North Carolina: Charlotte, Greensboro
Ohio: Columbus, Dayton, Brooklyn Heights
Oklahoma: Oklahoma City, Tulsa
Oregon: Portland
Pennsylvania: Blue Bell, Carnegie
Rhode Island: Rumford
South Carolina: Columbia
Tennessee: Knoxville, Memphis
Texas: Austin, Dallas, El Paso, Ft. Worth, Houston
Utah: Salt Lake City
Virginia: McLean, Norfolk, Richmond, Salem
Washington: Kirkland
West Virginia: Charleston
Wisconsin: West Allis

Australia: Melbourne, Victoria
France: Le Plessis Robinson
Italy: Milan, Padua, Rome
The Netherlands: Rijswijk
New Zealand: Auckland, Wellington
Sweden: Gothenburg, Malmoe, Stockholm
Switzerland: Lausanne, Zurich
United Kingdom: Birmingham, Dublin, Glasgow, London, Manchester
West Germany: Filderstadt, Frankfurt, Hamburg, Munich, Ratingen, Rodelheim

MANUFACTURER'S REPRESENTATIVES & DISTRIBUTORS

Argentina: Buenos Aires
Costa Rica: San Jose
Ecuador: Quito
Egypt: Cairo
Finland: Helsinki
Greece: Athens
Hong Kong: Hong Kong
India: Bombay
Indonesia: Jakarta
Iran: Tehran
Israel: Givatayim
Japan: Tokyo
Jordan: Amman
Korea: Seoul
Kuwait: Kuwait
Lebanon: Beirut
Malaysia: Kuala Lumpur
Mexico: Mexico City
Nicaragua: Managua
Nigeria: Lagos, Ibadan
Norway: Oslo
Peru: Lima
Philippine Islands: Manila
Puerto Rico: Hato Rey
Saudi Arabia: Riyadh
Singapore: Singapore
South Africa: Johannesburg, Pretoria
Spain: Barcelona, Bilbao, Madrid, San Sebastian, Valencia
Taiwan: Taipei
Thailand: Bangkok
Uruguay: Montevideo
Venezuela: Maracaibo

ADMINISTRATION, MANUFACTURING RESEARCH AND DEVELOPMENT

Massachusetts: Cambridge, Framingham, Westboro, Southboro
Maine: Westbrook
New Hampshire: Portsmouth
California: Anaheim, Sunnyvale
North Carolina: Research Triangle Park, Johnston County

Hong Kong: Kowloon, Tai Po
Thailand: Bangkok

Data General

Data General Corporation, Westboro, Massachusetts 01581