- Suspending a DMA transfer does not affect any I/O devices (an I/O device will act as though the transfer is proceeding). The CPU must provide for conditions that may arise if, for example, a device requests a DMA transfer, but the channel does not acknowledge the request because it has been suspended. Similarly, an I/O device may be in a different condition when the operation is resumed.

A suspended operation may be resumed by setting CF to 101. This command causes the channel to reload TP, its tag bit, and the PSW from the first two words of PB. Resuming an operation that has not been suspended will give unpredictable results since the first two words of PB will not contain the required channel state data. A resume command does not affect any channel registers other than TP.

The CPU may abort a channel operation by issuing a "halt" command (CF=111). The channel clears its BUSY flag to 0H and then idles. Again, the CPU must be prepared for the effect aborting a DMA transfer may have on an I/O device.

DRQ (DMA Request)

The synchronizing device in a DMA transfer uses the DRQ line to indicate when it is ready to send or receive the next byte or word. The channel recognizes a signal on this line only during a DMA transfers, i.e., after the instruction following XFER has been executed and before a termination condition has occurred. The channels have separate DMA request lines (DRQ1 and DRQ2).

EXT (External Terminate)

An external device (typically the synchronizing device) can terminate a DMA transfer by signaling on this line. Each channel has its own external terminate line (EXT1 and EXT2). The channel stops the transfer as soon as the current fetch or store cycle is completed. An external terminate in an unsynchronized transfer could result in a loss of data, although this would not be a typical use of EXT. In a synchronized transfer, the synchronizing device will normally issue EXT instead of DRQ following the last transfer cycle. If EXT is activated during a transfer cycle, a fetched byte may not be stored as explained in section 3.4.

A channel does not recognize EXT if it is not performing a DMA transfer. If EXT1 and EXT2 are activated simultaneously, EXT1 is recognized first.

Interrupts

Each channel has a separate system interrupt line (SINTR1 and SINTR2). A channel program may generate a CPU interrupt request by executing a SINTR instruction. Whether this instruction actually activates the SINTR line, however, depends upon the state of the interrupt control bit (bit 3 of the PSW; see figure 3-17). If this bit is set, interrupts from the channel are enabled, and execution of the SINTR instruction activates SINTR. If the interrupt control bit is cleared, the SINTR instruction has no effect; interrupts from the channel are disabled.

The CPU can alter a channel's interrupt control bit by sending any command to the channel with the value of ICF (interrupt control field) in the CCW set to 10 (enable) or 11 (disable). Thus, the CPU can prevent interrupts from either channel.

Once activated, SINTR remains active until the CPU sends a channel command with ICF set to 01 (interrupt acknowledge). When the channel receives this command, it clears the interrupt service bit in the PSW (figure 3-17) and removes the interrupt request. Disabling interrupts also clears the interrupt service bit and lowers SINTR.

Status Lines

The IOP emits signals on the $\overline{S0}$-$\overline{S2}$ status lines to indicate to external devices the type of bus cycle the processor is starting. Table 3-12 shows the signals that are output for each type of cycle. These status lines are connected to an 8288 Bus Controller. The bus controller decodes these lines and outputs the signals that control components attached to the bus. The IOP indicates "instruction fetch" on these lines when it is reading and writing memory operands as well as when it is fet-

ched instructions. In the remote configuration, an 8289 Bus Arbiter monitors the $\overline{S0}$-$\overline{S2}$ status lines to determine when a system bus access is required.

Table 3-12. Status Signals S0-S2

$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Type of Bus Cycle
0	0	0	Instruction fetch from I/O space
0	0	1	Data fetch from I/O space
0	1	0	Data store to I/O space
0	1	1	(not used)
1	0	0	Instruction fetch from system space
1	0	1	Data fetch from system space
1	1	0	Data store to system space
1	1	1	Passive; no bus cycle run

Status lines S3-S6 indicate whether the bus cycle is DMA or non-DMA, and which channel is running the cycle (see table 3-13). Note that when the IOP is not running a bus cycle (e.g., when it is idle or when it is executing an internal cycle that does not use the bus), the status lines reflect the last bus cycle run.

Table 3-13. Status Signals S3-S6

S6	S5	S4	S3	Bus Cycle
1	1	0	0	DMA cycle on channel 1
1	1	0	1	DMA cycle on channel 2
1	1	1	0	Non-DMA cycle on channel 1
1	1	1	1	Non-DMA cycle on channel 2

3.7 Instruction Set

This section divides the IOP's 53 instructions into five functional categories:

1. data transfer,
2. arithmetic,
3. logic and bit manipulation,
4. program transfer,
5. processor control.

The description of each instruction in these categories explains how the instruction operates and how it may be used in channel programs. Instructions that perform essentially the same operation (e.g., ADD and ADDB, which add words and bytes respectively), are described together. A reference table at the end of the section lists every instruction alphabetically and provides execution time, encoded length, and sample ASM-89 coding for each permissable operand combination. For information on how the 8089 machine instructions are encoded in memory, see section 4.3.

In reading this section, it is important to recall that the instruction set does not differentiate between memory addresses and I/O device addresses. Instructions that are described as accepting byte and word memory operands may also be used to read and write I/O devices.

Data Transfer Instructions

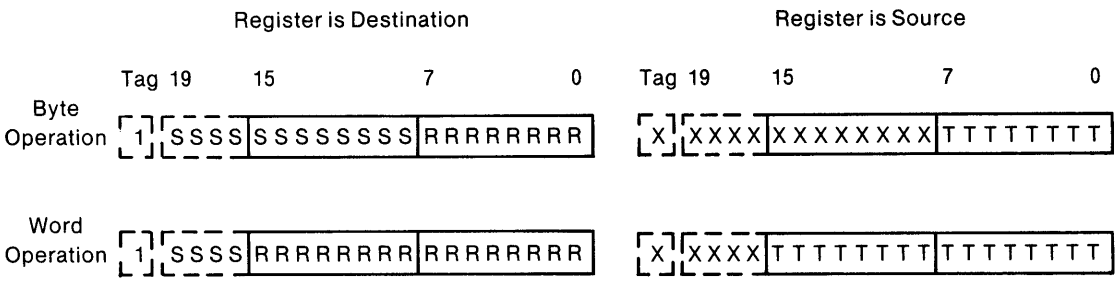
These instructions move data between memory and channel registers. Traditional byte and word moves (including memory-to-memory) are available, as are special instructions that load addresses into pointer registers and update tag bits in the process.

MOV *destination, source*

MOV transfers a byte or word from the source to the destination. Four instructions are provided:

MOV	Move Word Variable,
MOVB	Move Byte Variable,
MOVI	Move Word Immediate,
MOVBI	Move Byte Immediate.

Figure 3-35 shows how these instructions affect register operands. Notice that when a pointer register is specified as the destination of a MOV, its tag bit is unconditionally set to 1. MOV instructions are therefore used to load I/O space addresses into pointer registers.



T = bit is transferred to destination operand
R = bit is replaced by source operand
S = bit is sign extension of high-order bit transferred
X = bit is ignored
1 = bit is unconditionally set

Figure 3-35. Register Operands in MOV Instructions

MOVP *destination, source*

MOVP (move pointer) transfers a physical address variable between a pointer register and memory. If the source is a pointer register, its content and tag bit are converted to a physical address pointer (see figure 3-23). If the source is a memory location, the three bytes are converted to a 20-bit physical address and a tag value, and are loaded into the pointer register and its tag bit. MOVP is typically used to save and restore pointer registers.

LPD *destination, source*

LPD (load pointer with doubleword) converts a doubleword pointer (see figure 3-22) to a 20-bit physical address and loads it into the destination, which must be a pointer register. The pointer register's tag bit is unconditionally cleared to 0, indicating a system address. Two instructions are provided:

LPD	Load Pointer With Doubleword Variable
LPDI	Load Pointer With Doubleword Immediate

An 8086 or 8088 can pass any address in its megabyte memory space to a channel program in the form of a doubleword pointer. The channel program can access the location by using LPD to load the location address into a pointer register.

Arithmetic Instructions

The arithmetic instructions interpret all operands as unsigned binary numbers of 8, 16 or 20 bits. Signed values may be represented in standard two's complement notation with the high-order bit representing the sign (0=positive, 1=negative). The processor, however, has no way of detecting an overflow into a sign bit so this possibility must be provided for in the user's software.

The 8089 performs arithmetic operations to 20 significant bits as follows. Byte and word operands are sign-extended to 20 bits (e.g., bit 7 of a byte operand is propagated through bits 8-19 of an internal register). Sign extension does not affect the magnitude of the operand. The operation is then performed, and the 20-bit result is

returned to the destination operand. High-order bits are truncated as necessary to fit the result in the available space. A carry out of, or borrow into, the high-order bit of the result is not detected. However, if the destination is a register that is larger than the source operand, carries will be reflected in the upper register bits, up to the size of the register.

Figure 3-36 shows how the arithmetic instructions treat registers when they are specified as source and destination operands.

ADD *destination, source*

The sum of the two operands replaces the destination operand. Four addition instructions are provided:

ADD	Add Word Variable
ADDB	Add Byte Variable
ADDI	Add Word Immediate
ADDBI	Add Byte Immediate

INC *destination*

The destination is incremented by 1. Two instructions are available:

INC	Increment Word
INCB	Increment Byte

DEC *destination*

The destination is decremented by 1. Word and byte instructions are provided:

DEC	Decrement Word
DECB	Decrement Byte

Logical and Bit Manipulation Instructions

The logical instructions include the boolean operators AND, OR and NOT. Two bit manipulation instructions are provided for setting or

X = bit is ignored in operation
R = bit is replaced by operation result
P = bit participates in operation

Figure 3-36. Register Operands in Arithmetic Instructions

clearing a single bit in memory or in an I/O device register. As shown in figure 3-37, the logical operations always leave the upper four bits of 20-bit destination registers undefined. These bits should not be assumed to contain reliable values or the same values from one operation to the next. Notice also that when a register is specified as the destination of a byte operation, bits 8-15 are overwritten by bit 7 of the result. Bits 8-15 can be preserved in AND and OR instructions by using word operations in which the upper byte of the source operand is FFH or 00H, respectively.

AND *destination, source*

The two operands are logically ANDed and the result replaces the destination operand. A bit in the result is set if the bits in the corresponding positions of the operands are both set, otherwise the result bit is cleared. The following AND instructions are available:

AND	Logical AND Word Variable
ANDB	Logical AND Byte Variable
ANDI	Logical AND Word Immediate
ANDBI	Logical AND Byte Immediate

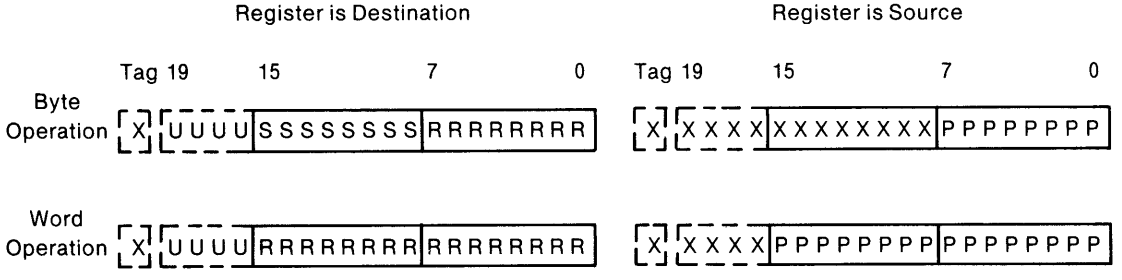
AND is useful when more than one bit of a device register must be cleared while leaving the remaining bits intact. For example, ANDing an 8-bit register with EEH only clears bits 0 and 4.

OR *destination, source*

The two operands are logically ORed, and the result replaces the destination operand. A bit in the result is set if either or both of the corresponding bits of the operands are set; if both operand bits are cleared, the result bit is cleared. Four types of OR instructions are provided:

OR	Logical OR Word Variable
ORB	Logical OR Byte Variable
ORI	Logical OR Word Immediate
ORBI	Logical OR Byte Immediate

OR can be used to selectively set multiple bits in a device register. For example, ORing an 8-bit register with 30H sets bits 4 and 5, but does not affect the other bits.



X = bit is ignored in operation
U = bit is undefined following operation
R = bit participates in operation and is replaced by result
S = bit is sign-extension of high-order result bit
P = bit participates in operation, but is unchanged

Figure 3-37. Register Operands in Logical Instructions

NOT *destination/destination, source*

NOT inverts the bits of an operand. If a single operand is coded, the inverted result replaces the original value. If two operands are coded, the inverted bits of the source replace the destination value (which must be a register), but the source retains its original value. In addition to these two operand forms, separate mnemonics are provided for word and byte values:

NOT	Logical NOT Word
NOTB	Logical NOT Byte

NOT followed by INC will negate (create the two's complement of) a positive number.

SETB *destination, bit-select*

The bit-select operand specifies one bit in the destination, which must be a memory byte, that is unconditionally set to 1. A bit-select value of 0 specifies the low-order bit of the destination while the high-order bit is set if bit-select is 7. SETB is handy for setting a single bit in an 8-bit device register.

CLR *destination, bit-select*

CLR operates exactly like SETB except that the selected bit is unconditionally cleared to 0.

Program Transfer Instructions

Register TP controls the sequence in which channel program instructions are executed. As each instruction is executed, the length of the instruction is added to TP so that it points to the next sequential instruction. The program transfer instructions can alter this sequential execution by adding a signed displacement value to TP. The displacement is contained in the program transfer instruction and may be either 8 or 16 bits long. The displacement is encoded in two's complement notation, and the high-order bit indicates the sign (0=positive displacement, 1=negative displacement). An 8-bit displacement may cause a transfer to a location in the range −128 through +127 bytes from the end of the transfer instruction, while a 16-bit displacement can transfer to any location within −32,768 through +32,767 bytes. An instruction containing an 8-bit displacement is called a short transfer and an instruction containing a 16-bit displacement is called a long transfer.

The program transfer instructions have alternate mnemonics. If the mnemonic begins with the letter "L," the transfer is long, and the distance to the transfer target is expressed as a 16-bit displacement regardless of how far away the target is located. If the mnemonic does not begin with "L," the ASM-89 assembler may build a short or long displacement according to rules discussed in section 3.9.

The "self-relative" addressing technique used by program transfer instructions has two important consequences. First, it promotes position-independent code, i.e., code that can be moved in memory and still execute correctly. The only restriction here is that the entire program must be moved as a unit so that the distance between the transfer instruction and its target does not change. Second, the limited addressing range of these instructions must be kept in mind when designing large (over 32k bytes of code) channel programs.

CALL/LCALL *TPsave, target*

CALL invokes an out-of-line routine, saving the value of TP so that the subroutine can transfer back to the instruction following the CALL. The instruction stores TP and its tag bit in the TPsave operand, which must be a physical address variable, and then transfers to the target address formed by adding the target operand's displacement to TP. The subroutine can return to the instruction following the CALL by using a MOVP instruction to load TPsave back into TP.

Notice that the 8089's facilities for implementing subroutines, or procedures, is less sophisticated than its counterparts in the 8086/8088. The principal difference is that the 8089 does not have a built in stack mechanism. 8089 programs can implement a stack using a base register as a stack pointer. On the other hand, since channel programs are not subject to interrupts, a stack will not be required for most channel programs.

JMP/LJMP *target*

JMP causes an unconditional transfer (jump) to the target location. Since the task pointer is not saved, no return to the instruction following the JMP is implied.

JZ/LJZ *source, target*

JZ (jump if zero) effects a transfer to the target location if the source operand is zero; otherwise the instruction following JZ is executed. Word and byte values may be tested by alternate instructions:

JZ/LJZ	Jump/Long Jump if Word Zero
JZB/LJZB	Jump/Long Jump if Byte Zero

If the source operand is a register, only the low-order 16 bits are tested; any additional high-order bits in the register are ignored. To test the low-order byte of a register, clear bits 8-15 and then use the word form of the instruction.

JNZ/LJNZ *source, target*

JNZ operates exactly like JZ except that control is transferred to the target if the source operand does not contain all 0-bits. Word and byte sources may be tested using these mnemonics:

JNZ/LJNZ	Jump/Long Jump if Word Not Zero
JNZB/LJNZB	Jump/Long Jump if Byte Not Zero.

JMCE/LJMCE *source, target*

This instruction (jump if masked compare equal) effects a transfer to the target location if the source (a memory byte) is equal to the lower byte in register MC as masked by the upper byte in MC. Figure 3-15 illustrates how 0-bits in the upper half of MC cause the corresponding bits in the lower half of MC and the source operand to compare equal, regardless of their actual values. For example, if bits 8-15 of MC contain the value 01H, then the transfer will occur if bit 0 of the source and register MC are equal. This instruction is useful for testing multiple bits in 8-bit device registers.

JMCNE/LJMCNE *source, target*

This instruction causes a jump to the target location if the source is not equal to the mask/compare value in MC. It otherwise operates identically to JMCE.

JBT/LJBT *source, bit-select, target*

JBT (jump if bit true) tests a single bit in the source operand and jumps to the target if the bit is a 1. The source must be a byte in memory or in an I/O device register. The bit-select value may range from 0 through 7, with 0 specifying the low-order bit. This instruction may be used to test a bit in an 8-bit device register. If the target is the JBT instruction itself, the operation effectively becomes "wait until bit is 0."

JNBT/LJNBT *source, bit-select, target*

This instruction operates exactly like JBT, except that the transfer is made if the bit is not true, i.e., if the bit is 0.

Processor Control Instructions

These instructions enable channel programs to control IOP hardware facilities such as the $\overline{\text{LOCK}}$ and SINTR1-2 pins, logical bus width selection, and the initiation of a DMA transfer.

TSL *destination, set-value, target*

Figure 3-38 illustrates the operation of the TSL (test and set while locked) instruction. TSL can be used to implement a semaphore variable that controls access to a shared resource in a multiprocessor system (see section 2.5). If the target operand specifies the address of the TSL instruction, the instruction is repetively executed until the semaphore (destination) is found to contain zero. Thus the channel program does not proceed until the resource is free.

WID *source-width, dest-width*

WID (set logical bus widths) alters bits 0 and 1 of the PSW, thus specifying logical bus widths for a DMA transfer. The operands may be specified as

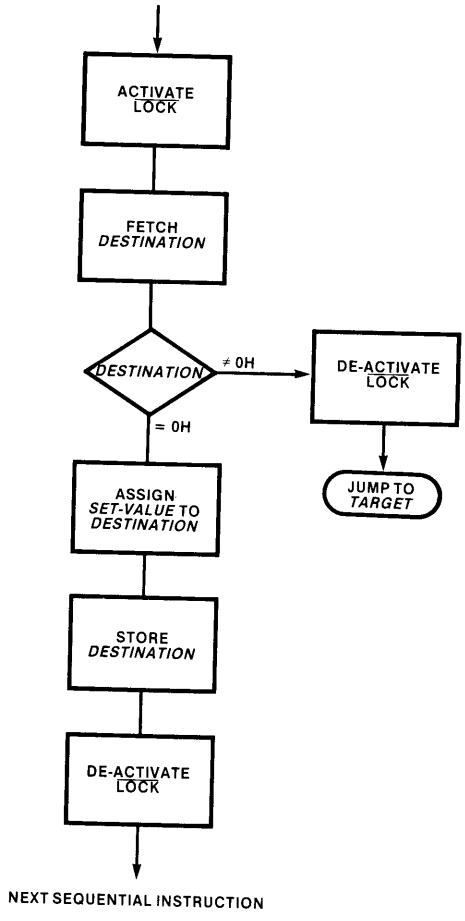


Figure 3-38. Operation of TSL Instruction

8 or 16 (bits), with the restriction that the logical width of a bus cannot exceed its physical width. The logical bus widths are undefined following a processor RESET; therefore the WID instruction must be executed before the first transfer. Thereafter the logical widths retain their values until the next WID instruction or processor RESET.

XFER *(no operands)*

XFER (enter DMA transfer mode after following instruction) prepares the channel for a DMA transfer operation. In a synchronized transfer, the instruction following XFER may ready the synchronizing device (e.g., send a "start" command or the last of a series of parameters). Any instruction, including NOP and WID, may follow XFER, except an instruction that alters GA, GB or GC.

SINTR *(no operands)*

This instruction sets the interrupt service bit in the PSW and activates the channel's SINTR line if the interrupt control bit in the PSW is set. If the

interrupt control bit is cleared (interrupts from this channel are disabled), the interrupt service bit is set, but SINTR1-2 is not activated. A channel program may use this instruction to interrupt a CPU.

NOP *(no operands)*

This instruction consumes clock cycles but performs no operation. As such, it is useful in timing loops.

HLT *(no operands)*

This instruction concludes a channel program. The channel clears its BUSY flag and then idles.

Instruction Set Reference Information

Table 3-16 lists every 8089 instruction alphabetically by its ASM-89 mnemonic. The ASM-89 coding format is shown (see table 3-14 for an explanation of operand identifiers) along with the instruction name. For every combination of operand types (see table 3-15 for key), the instruction's execution time and its length in bytes, and a coding example are provided.

The instruction timing figures are the number of clock periods required to execute the instruction with the given combination of operands. At 5 MHz, one clock period is 200 ns; at 8 MHz a clock period is 125 ns. Two timings are provided when an instruction operates on a memory word. The first (lower) figure indicates execution time when the word is aligned on an even address and is accessed over a 16-bit bus. The second figure is for odd-addressed words on 16-bit buses and any word accessed via an 8-bit bus.

Instruction fetch time is shown in table 3-17 and should be added to the execution times shown in table 3-16 to determine how long a sequence of instructions will take to run. (Section 3.2 explains the effect of the instruction queue on 16-bit instruction fetches.) External delays such as bus arbitration, wait states and activity on the other channel will increase the elapsed time over the figures shown in tables 3-16 and 3-17. These delays are application dependent.

Table 3-14. Key to ASM-89 Operand Identifiers

IDENTIFIER	USED IN	EXPLANATION
destination	data transfer, arithmetic, bit manipulation	A register or memory location that may contain data operated on by the instruction, and which receives (is replaced by) the result of the operation.
source	data transfer, arithmetic, bit manipulation	A register, memory location, or immediate value that is used in the operation, but is not altered by the instruction.
target	program transfer	Location to which control is to be transferred.
TPsave	program transfer	A 24-bit memory location where the address of the next sequential instruction is to be saved.
bit-select	bit manipulation	Specification of a bit location within a byte; 0=least-significant (rightmost) bit, 7=most-significant (leftmost) bit.
set-value	TSL	Value to which destination is set if it is found 0.
source-width	WID	Logical width of source bus.
dest-width	WID	Logical width of destination bus.

Table 3-15. Key to Operand Types

IDENTIFIER	EXPLANATION
(no operands)	No operands are written
register	Any general register
ptr-reg	A pointer register
immed8	A constant in the range 0-FFH
immed16	A constant in the range 0-FFFFH
mem8	An 8-bit memory location (byte)
mem16	A 16-bit memory location (word)
mem24	A 24-bit memory location (physical address pointer)
mem32	A 32-bit memory location (doubleword pointer)
label	A label within −32,768 to +32,767 bytes of the end of the instruction
short-label	A label within −128 to +127 bytes of the end of the instruction
0-7	A constant in the range: 0-7
8/16	The constant 8 or the constant 16

Table 3-16. Instruction Set Reference Data

ADD destination, source		Add Word Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem16	11/15	2-3	ADD BC, [GA].LENGTH
mem16, register	16/26	2-3	ADD [GB], GC

ADDB destination, source		Add Byte Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem8	11	2-3	ADDB GC, [GA].N_CHARS
mem8, register	16	2-3	ADDB [PP].ERRORS, MC

ADDBI destination, source		Add Byte Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed8	3	3	ADDBI MC,10
mem8, immed8	16	3-4	ADDBI [PP+IX+].RECORDS, 2CH

ADDI destination, source		Add Word Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed16	3	4	ADDI GB, 0C25BH
mem16, immed16	16/26	4-5	ADDI [GB].POINTER, 5899

Table 3-16. Instruction Set Reference Data (Cont'd.)

AND destination, source		Logical AND Word Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem16	11/15	2-3	AND MC, [GA].FLAG_WORD
mem16, register	16/26	2-3	AND [GC].STATUS, BC

ANDB destination, source		Logical AND Byte Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem8	11	2-3	AND BC, [GC]
mem8, register	16	2-3	AND [GA+IX].RESULT, GA

ANDBI destination, source		Logical AND Byte Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed8	3	3	GA, 01100000B
mem8, immed8	16	3-4	[GC+IX], 2CH

ANDI destination, source		Logical AND Word Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed16	3	4	IX, 0H
mem16, immed16	16/26	4-5	[GB+IX].TAB, 40H

CALL TPsave, target		Call	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem24, label	17/23	3-5	CALL [GC+IX].SAVE, GET_NEXT

CLR destination, bit select		Clear Bit To Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7	16	2-3	CLR [GA], 3

DEC destination		Decrement Word By 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
register	3	2	
mem16	16/26	2-3	DEC [PP].RETRY

Table 3-16. Instruction Set Reference Data (Cont'd.)

DECB destination		Decrement Byte By 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8	16	2-3	DECB [GA+IX+].TAB

HLT (no operands)		Halt Channel Program	
Operands	**Clocks**	**Bytes**	**Coding Example**
(no operands)	11	2	HLT

INC destination		Increment Word by 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
register mem16	3 16/26	2 2-3	INC GA INC [GA].COUNT

INCB destination		Increment Byte by 1	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8	16	2-3	INCB [GB].POINTER

JBT source, bit-select, target		Jump if Bit True (1)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	3-5	JBT [GA].RESULT_REG, 3, DATA_VALID

JMCE source, target		Jump if Masked Compare Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	3-5	JMCE [GB].FLAG, STOP_SEARCH

JMCNE source, target		Jump if Masked Compare Not Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	3-5	JMCNE [GB+IX], NEXT_ITEM

JMP target		Jump Unconditionally	
Operands	**Clocks**	**Bytes**	**Coding Example**
label	3	3-4	JMP READ_SECTOR

Table 3-16. Instruction Set Reference Data (Cont'd.)

JNBT source, bit-select, target		Jump if Bit Not True (0)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	3-5	JNBT [GC], 3, RE_READ

JNZ source, target		Jump if Word Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	3-4	JNZ BC, WRITE_LINE
mem16, label	12/16	3-5	JNZ [PP].NUM_CHARS, PUT_BYTE

JNZB source, target		Jump if Byte Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	3-5	JNZB [GA], MORE_DATA

JZ source, target		Jump if Word is Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	3-4	JZ BC, NEXT_LINE
mem16, label	12/16	3-5	JZ [GC+IX].INDEX, BUF_EMPTY

JZB source, target		Jump if Byte Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	3-5	JZB [PP].LINES_LEFT, RETURN

LCALL TPsave, target		Long Call	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem24, label	17/23	4-5	LCALL [GC].RETURN_SAVE, INIT_8279

LJBT source, bit-select, target		Long Jump if Bit True (1)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	4-5	LJBT [GA].RESULT, 1, DATA_OK

LJMCE source, target		Long jump if Masked Compare Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	4-5	LJMCE [GB], BYTE_FOUND

Table 3-16. Instruction Set Reference Data (Cont'd.)

LJMCNE source, target		Long jump if Masked Compare Not Equal	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	14	4-5	LJMCNE [GC+IX+], SCAN_NEXT

LJMP target		Long Jump Unconditional	
Operands	**Clocks**	**Bytes**	**Coding Example**
label	3	4	LJMP GET_CURSOR

LJNBT source, bit-select, target		Long Jump if Bit Not True (0)	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, 0-7, label	14	4-5	LJNBT [GC], 6, CRCC_ERROR

LJNZ source, target		Long Jump if Word Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	4	LJNZ BC, PARTIAL_XMIT
mem16, label	12/16	4-5	LJNZ [GA+IX].N_LEFT, PUT_DATA

LJNZB source, target		Long Jump if Byte Not Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	4-5	LJNZB [GB+IX+].ITEM, BUMP_COUNT

LJZ source, target		Long Jump if Word Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, label	5	4	LJZ IX, FIRST_ELEMENT
mem16, label	12/16	4-5	LJZ [GB].XMIT_COUNT, NO_DATA

LJZB source, target		Long Jump if Byte Zero	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, label	12	4-5	LJZB [GA], RETURN_LINE

LPD destination, source		Load Pointer With Doubleword Variable	
Operands	**Clocks**	**Bytes**	**Coding Example**
ptr-reg, mem32	20/28*	2-3	LPD GA, [PP].BUF_START

*20 clocks if operand is on even address; 28 if on odd address

Table 3-16. Instruction Set Reference Data (Cont'd.)

LPDI destination, source		Load Pointer With Doubleword Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
ptr-reg, immed32	12/16*	6	LPDI GB, DISK_ADDRESS

*12 clocks if instruction is on even address; 16 if on odd address

MOV destination, source		Move Word	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem16	8/12	2-3	MOV IX, [GC]
mem16, register	10/16	2-3	MOV [GA].COUNT, BC
mem16, mem16	18/28	4-6	MOV [GA].READING, [GB]

MOVB destination, source		Move Byte	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, mem8	8	2-3	MOVB BC, [PP].TRAN_COUNT
mem8, register	10	2-3	MOVB [PP].RETURN_CODE, GC
mem8, mem8	18	4-6	MOVB [GB+IX+], [GA+IX+]

MOVBI destination, source		Move Byte Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed8	3	3	MOVBI MC, 'A'
mem8, immed8	12	3-4	MOVBI [PP].RESULT, 0

MOVI destination, source		Move Word Immediate	
Operands	**Clocks**	**Bytes**	**Coding Example**
register, immed16	3	4	MOVI BC, 0
mem16, immed16	12/18	4-5	MOVI [GB], 0FFFFH

MOVP destination, source		Move Pointer	
Operands	**Clocks**	**Bytes**	**Coding Example**
ptr-reg, mem24	19/27*	2-3	MOVP TP, [GC+IX]
mem24, ptr-reg	16/22*	2-3	MOVP [GB].SAVE_ADDR, GC

*First figure is for operand on even address; second is for odd-addressed operand.

NOP (no operands)		No Operation	
Operands	**Clocks**	**Bytes**	**Coding Example**
(no operands)	4	2	NOP

Table 3-16. Instruction Set Reference Data (Cont'd.)

NOT destination/destination, source — Logical NOT Word

Operands	Clocks	Bytes	Coding Example
register	3	2	NOT MC
mem16	16/26	2-3	NOT [GA].PARM
register, mem16	11/15	2-3	NOT BC, [GA+IX].LINES_LEFT

NOTB destination/destination, source — Logical NOT Byte

Operands	Clocks	Bytes	Coding Example
mem8	16	2-3	NOTB [GA].PARM_REG
register, mem8	11	2-3	NOTB IX, [GB].STATUS

OR destination, source — Logical OR Word

Operands	Clocks	Bytes	Coding Example
register, mem16	11/15	2-3	OR MC, [GC].MASK
mem16, register	16/26	2-3	OR [GC], BC

ORB destination, source — Logical OR Byte

Operands	Clocks	Bytes	Coding Example
register, mem8	11	2-3	ORB IX, [PP].POINTER
mem8, register	16	2-3	ORB [GA+IX+], GB

ORBI destination, source — Logical OR Byte Immediate

Operands	Clocks	Bytes	Coding Example
register, immed8	3	3	ORBI IX, 00010001B
mem8, immed8	16	3-4	ORBI [GB].COMMAND, 0CH

ORI destination, source — Logical OR Word Immediate

Operands	Clocks	Bytes	Coding Example
register, immed16	3	4	ORI MC, 0FF0DH
mem16,immed16	16/26	4-5	ORI [GA], 1000H

SETB destination, bit-select — Set Bit to 1

Operands	Clocks	Bytes	Coding Example
mem8, 0-7	16	2-3	SETB [GA].PARM_REG, 2

SINTR (no operands) — Set Interrupt Service Bit

Operands	Clocks	Bytes	Coding Example
(no operands)	4	2	SINTR

Table 3-16. Instruction Set Reference Data (Cont'd.)

TSL destination, set-value, target		Test and Set While Locked	
Operands	**Clocks**	**Bytes**	**Coding Example**
mem8, immed8, short-label	14/16*	4-5	TSL [GA].FLAG, 0FFH, NOT_READY

*14 clocks if destination ≠ 0; 16 clocks if destination = 0

WID source-width, dest-width		Set Logical Bus Widths	
Operands	**Clocks**	**Bytes**	**Coding Example**
8/16, 8/16	4	2	WID 8, 8

XFER (no operands)		Enter DMA Transfer Mode After Next Instruction	
Operands	**Clocks**	**Bytes**	**Coding Example**
(no operands)	4	2	XFER

Table 3-17. Instruction Fetch Timings (Clock Periods)

INSTRUCTION LENGTH (BYTES)	BUS WIDTH		
	8	16	
		(1)	(2)
2	14	7	11
3	18	14	11
4	22	14	15
5	26	18	15

(1) First byte of instruction is on an even address.

(2) First byte of instruction is on an odd address. Add 3 clocks if first byte is not in queue (e.g., first instruction following program transfer).

3.8 Addressing Modes

8089 instruction operands may reside in registers, in the instruction itself or in the system or I/O address spaces. Operands in the system and I/O spaces may be either memory locations or I/O device registers and may be addressed in four different ways. This section describes how the channel processes different types of operands and how it calculates addresses using its addressing modes. Section 3.9 describes the ASM-89 conventions that programmers use to specify these operands and addressing modes.

Register and Immediate Operands

Registers may be specified as source or destination operands in many instructions. Instructions that operate on registers are generally both shorter and faster than instructions that specify immediate or memory operands.

Immediate operands are data contained in instructions rather than in registers or in memory. The data may be either 8 or 16 bits in length. The limitations of immediate operands are that they may only serve as source operands and that they are constant values.

Memory Addressing Modes

Whereas the channel has direct access to register and immediate operands, operands in the system and I/O space must be transferred to or from the IOP over the bus. To do this, the IOP must calculate the address of the operand, called its

effective address (EA). The programmer may specify that an operand's address be calculated in any of four different ways; these are the 8089's memory addressing modes.

The Effective Address

An operand in the system space has a 20-bit effective address, and an operand in the I/O space has a 16-bit effective address. These addresses are unsigned numbers that represent the distance (in bytes) of the low-order byte of the operand from the beginning of the address space. Since the 8089 does not "see" the segmented structure of the system space that it may share with an 8086 or 8088, 8089 effective addresses are equivalent to 8086/8088 physical addresses.

All memory addressing modes use the content of one of the pointer registers, and the state of that register's tag bit determines whether the operand lies in the system or the I/O space. If the operand is in the I/O space (tag = 1), bits 16-19 of the pointer register are ignored in the effective address calculation. Section 4.3 describes the two fields (AA and MM) in the encoded machine instruction that specify addressing mode and base (pointer) register.

Based Addressing

In based addressing (figure 3-39), the effective address is taken directly from the content of GA, GB, GC or PP. Using this addressing mode, one instruction may access different locations if the register is updated before the instruction executes. LPD, MOV, MOVP or arithmetic instructions might be used to change the value of the base register.

Offset Addressing

In this mode (figure 3-40) an 8-bit unsigned value contained in the instruction is added to the content of a base register to form the effective address. The offset mode provides a convenient way to address elements in structures (a parameter block is a typical example of a structure). As shown in figure 3-41, a base register can be pointed at the base (first element) in the structure, and then different offsets can be used to access the elements within the structure. By changing the base address, the same structure can be relocated elsewhere in memory.

Indexed Addressing

An indexed address is formed by adding the content of register IX (interpreted as an unsigned quantity) to a base register as shown in figure 3-42. Indexed addressing is often used to access

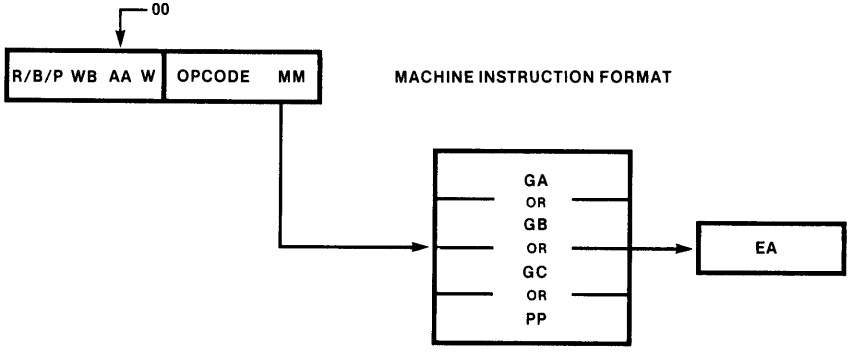


Figure 3-39. Based Addressing

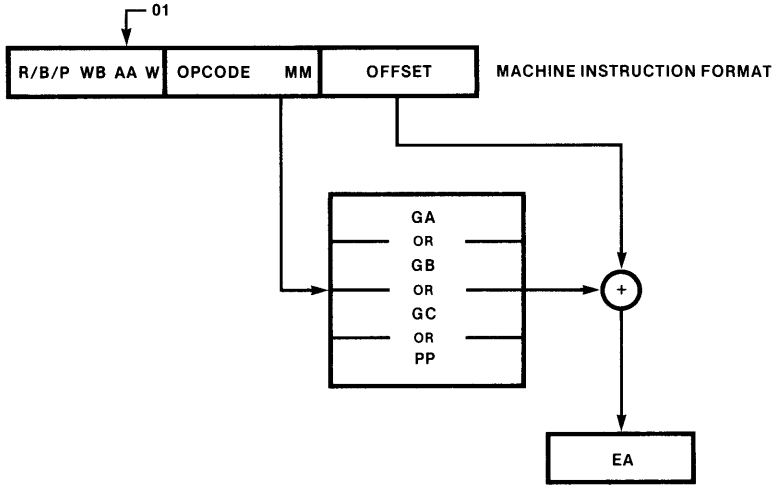


Figure 3-40. Offset Addressing

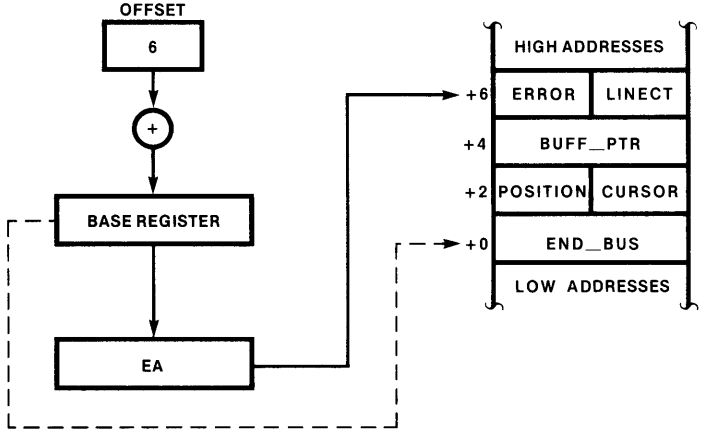


Figure 3-41. Accessing a Structure with Offset Addressing

array elements (see figure 3-43). A base register locates the beginning of the array and the value in IX selects one element, i.e., it acts as the array subscript. The *i*th element of a byte array is selected when IX contains $(i - 1)$. To access the *i*th element of a word array, IX should contain $((i - 1) * 2)$.

Indexed Auto-Increment Addressing

In this variation of indexed addressing, the effective address is formed by summing IX and a base register, and then IX is incremented automatically. (See figure 3-44.) The addition takes place

after the EA is calculated. IX is incremented by 1 for a byte operation, by 2 for a word operation and by 3 for a MOVP instruction. This addressing mode is very useful for "stepping through" successive elements of an array (e.g., a program loop that sums an array).

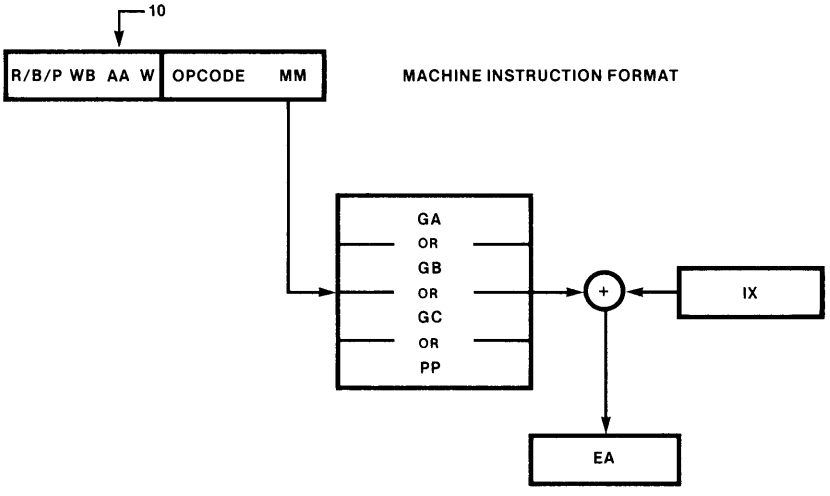


Figure 3-42. Indexed Addressing

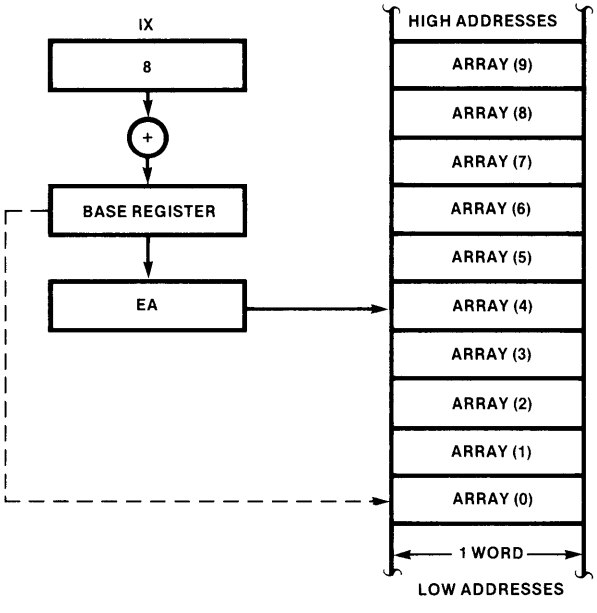


Figure 3-43. Accessing a Word Array with Indexed Addressing

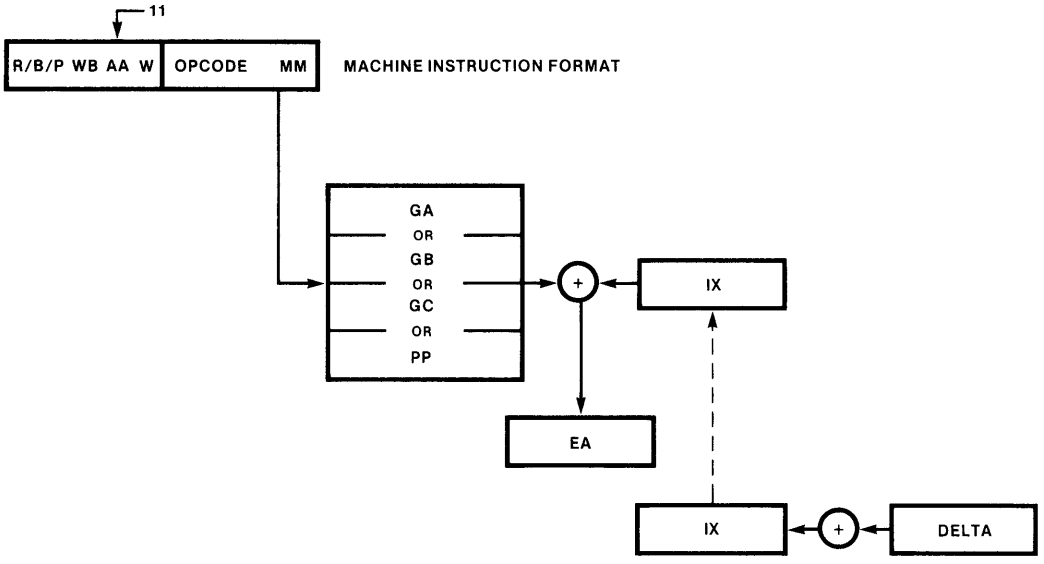


Figure 3-44. Indexed Auto-Increment Addressing

3.9 Programming Facilities

The compatibility of the 8089 with the 8086 and 8088 extends beyond the hardware interface. Comparing figure 3-45, with figure 2-45, one can see that, except for the translate step, the software development process is identical for both 8086/8088 and 8089 programs. The ASM-89 assembler produces a relocatable object module that is compatible with the 8086 family software development utilities LIB-86, LINK-86, LOC-86 and OH-86, described in section 2.9. All of these development tools run on an Intellec® 800 or Series II microcomputer development system.

This section surveys the facilities of the ASM-89 assembler and discusses how LINK-86 and LOC-86 can be used in 8089 software development. For a complete description of the 8089 assembly language, consult *8089 Assembly Language User's Guide*, Order No. 9800938, available from Intel's Literature Department.

ASM-89

The ASM-89 assembler reads a disk file containing 8089 assembly language statements, translates these statements into 8089 machine instructions, and writes the result into a second disk file. The assembly input is called a source module, and the principal output is a relocatable object module. The assembler also produces a file that lists the module and flags any errors detected during the assembly.

Statements

Statements are the building blocks of ASM-89 programs. Figure 3-46 shows several examples of ASM-89 statements. The ASM-89 assembler gives programmers considerable flexibility in formatting program statements. Variable names and labels (identifiers) may be up to 31 characters long, the underscore (_) character may be used to improve the readability of longer names (e.g.,

WAIT_UNTIL_READY). The component parts of statements (fields) need not be located at particular "columns" of the statement. Any number of blank characters may separate fields and multiple identifiers within the operand field. Long statements may be continued onto the next link by coding an ampersand (&) as the first character of the continued line.

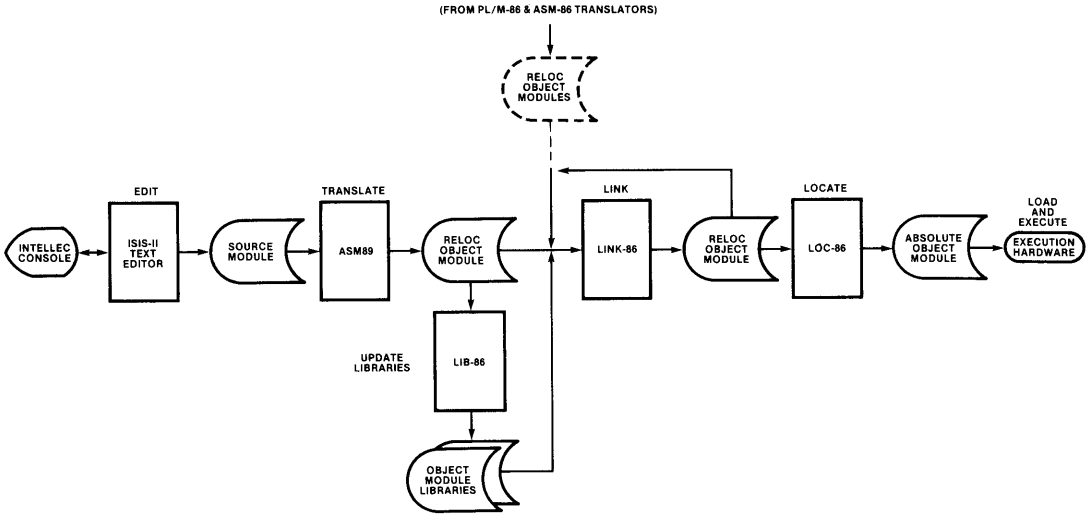


Figure 3-45. 8089 Software Development Process

```
; THIS STATEMENT CONTAINS A COMMENT FIELD ONLY
ADDI   BC,5                    ; TYPICAL ASM89 INSTRUCTION
  ADDI     BC,       5         ; NO "COLUMN" REQUIREMENTS
MOV      [GA].STATUS,
&        6                     ; A CONTINUED STATEMENT
SOURCE     EQU GA              ; A SIMPLE ASM89 DIRECTIVE
LINE_BUFFER_ADDRESS DD         ; A LONG IDENTIFIER
```

Figure 3-46. ASM-89 Statements

A statement whose first non-blank character is a semicolon is a comment statement. Comments have no affect on program execution and, in fact, are ignored by the ASM-89 assembler. Nevertheless, carefully selected comments are included in all well written ASM-89 programs. They summarize, annotate and clarify the logic of the program where the instructions are too "microscopic" to make the operation of the program self-evident.

An ASM-89 instruction statement (figure 3-47) directs the assembler to build an 8089 machine instruction. The optional label field assigns a symbolic identifier to the address where the instruction will be stored in memory. A labelled instruction can be the target of a program transfer; the transferring instruction specifies the label for its target operand. In figure 3-47 the labelled instruction conditionally transfers to itself; the program will loop on this one instruction as long as bit 3 of the byte addressed by [GA].STATUS is not true. The mnemonic field of an instruction statement specifies the type of 8089 machine instruction that the assembler is to build.

The operand field may contain no operands or one or more operands as required by the instruction. Multiple operands are separated by commas and, optionally, by blanks. Any instruction statement may contain a comment field (comment fields are initiated by a semicolon).

An ASM-89 directive statement (figure 3-48) does not produce an 8089 machine instruction. Rather, a directive gives the assembler information to use during the assembly. For example, the DS (define storage) directive in figure 3-48 tells the assembler to reserve 80 bytes of storage and to assign a symbolic identifier (INPUT_BUFFER) to the first (lowest-addressed) byte of this area. The ASM-89 assembler accepts 14 directives; the more commonly used directives are discussed in this section.

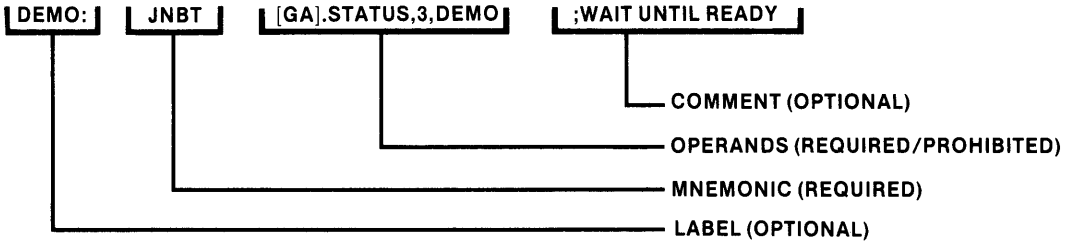


Figure 3-47. ASM-89 Instruction Format

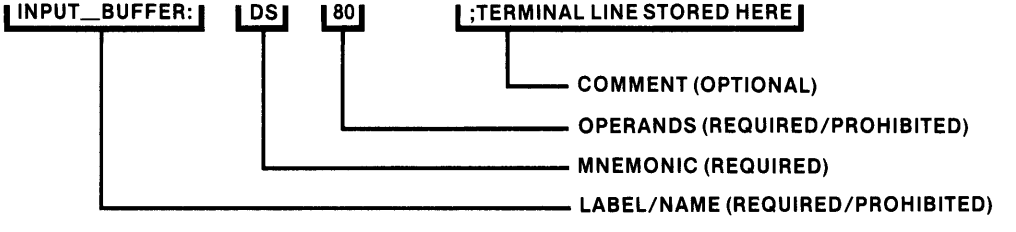


Figure 3-48. ASM-89 Directive Format

The first field in a directive may be a label or a name; individual directives may require or prohibit names, while labels are optional for directives that accept them. A label ends in a colon like an instruction statement label. However, a directive label cannot be specified as the target of a program transfer. A name does not have a colon. The second field is the directive mnemonic, and the assembler distinguishes between instructions and directives by this field. Any operands required by the directive are written next; multiple operands are separated by commas and, optionally, by blanks. A comment may be included in any directive by beginning the text with a semicolon.

Constants

Binary, decimal, octal and hexadecimal numeric constants (figure 3-49) may be written in ASM-89 instructions and directives. The assembler can add and subtract constants at assembly time. Numeric constants, including the results of arithmetic operations, must be representable in 16 bits. Positive numbers cannot exceed 65,535 (decimal); negative numbers, which the assembler represents in two's complement notation, cannot be "more negative" than −32,768 (decimal).

Character constants are enclosed in single quote marks as shown in figure 3-49. Strings of characters up to 255 bytes long may be written when initializing storage. Instruction operands, however, can only be one or two characters long (for byte and word instructions respectively).

As an aid to program clarity, The EQU (equate) directive may be used to give names to constants (e.g., DISK_STATUS EQU 0FF20H).

Defining Data

Four ASM-89 directives reserve space for memory variables in the ASM-89 program (see figure 3-50). The DB, DW and DD directives allocate units of bytes, words and doublewords, respectively, initialize the locations, and optionally label them so that they may be referred to by name in instruction statements. The label of a storage directive always refers to the first (lowest-addressed) byte of the area reserved by the directive.

The DB and DW directives may be used to define byte- and word-constant scalars (individual data items) and arrays (sequences of the same type of item). For example, a character string constant could be defined as a byte array:

```
SIGN_ON_MSG: DB 'PLEASE ENTER PASSWORD'
```

The DD directive is typically used to define the address of a location in the system space, i.e., a doubleword pointer variable. The address may be loaded into a pointer register with the LPD instruction.

The DS directive reserves, and optionally names, storage in units of bytes, but does not initialize any of the reserved bytes. DS is typically used for RAM-based variables such as buffers. As there is no special directive for defining a physical address pointer, DS is typically used to reserve the three bytes used by the MOVP instruction.

```
MOVBI   GA, 'A'          ; CHARACTER
MOVBI   GA, 41H          ; HEXADECIMAL
MOVBI   GA, 65           ; DECIMAL
MOVBI   GA, 65D          ; DECIMAL ALTERNATIVE
MOVBI   GA, 101Q         ; OCTAL
MOVBI   GA, 101O         ; OCTAL ALTERNATIVE
MOVBI   GA, 01000001B    ; BINARY
; NEXT TWO STATEMENTS ARE EQUIVALENT AND
;    ILLUSTRATE TWO'S COMPLEMENT REPRESENTATION
;    OF NEGATIVE NUMBERS
MOVBI   GA, -5
MOVBI   GA, 11111011B
```

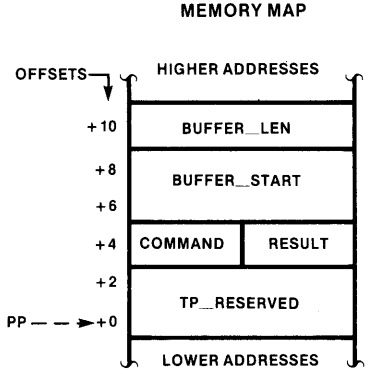
Figure 3-49. ASM89 Constants

```
; ASM89 DIRECTIVE                ; MEMORY CONTENT (HEX)
ALPHA:  DB      1               ; 01
        DB      -2              ; FE (TWO'S COMPLEMENT)
        DB      'A', 'B'        ; 4142
BETA:   DW      1               ; 0100
        DW      -5              ; FAFF
        DW      'AB'            ; 4241
        DW      400, 500        ; 2410F401
        DW      400H, 500H      ; 0004 0005
gamma:  DW      BETA            ; OFFSET OF BETA ABOVE,
                                ; FROM BEGINNING OF PROGRAM
DELTA   DD      GAMMA           ; ADDRESS (SEGMENT & OFFSET)
                                ; OF GAMMA
ZETA:   DS      80              ; 80 BYTES, UNINITIALIZED
```

Figure 3-50. ASM-89 Storage Directives

Structures

An ASM-89 structure is a map or template that gives names and relative locations to a collection of related variables that are called structure elements or members. Defining a structure, however, does not allocate storage. The structure is, in effect, overlaid on a particular area of memory when one of its elements is used as an instruction operand. Figure 3-51 shows how a structure representing a parameter block could be defined and then used in a channel program. The assembler uses the structure element name to produce an offset value (structures are used with the offset addressing mode). Compared to "hard-coded" offsets, structures improve program clarity and simplify maintenance. If the layout of a memory block changes, only the structure definition must be modified. When the program is reassembled, all symbolic references to the structure are automatically adjusted. When multiple areas of memory are laid out identically, a single structure can be used to address any area by changing the content of the pointer (base) register that specifies the structure's "starting address."




```
STRUCTURE DEFINITION

PARM_BLOCK      STRUC
  TP_RESERVED:  DS   4
  COMMAND:      DS   1
  RESULT:       DS   1
  BUFFER_START: DS   4
  BUFFER_LEN:   DS   2
PARM_BLOCK      ENDS
```

USING "HARD-CODED" OFFSETS

```
LPD GA, [PP].6
MOVBI   [PP].5,0
```

USING STRUCTURE ELEMENT NAMES

```
LPD GA,  [PP].BUFFER_START
MOVBI    [PP].RESULT,0
```

Figure 3-51. ASM-89 Structure Definition and Use

Addressing Modes

Table 3-18 summarizes the notation a programmer uses to specify how the effective address of a memory operand is to be computed. Examples of typical ASM-89 coding for each addressing mode, as well as register and immediate operands, are provided in figure 3-52. Notice that a bracketed reference to a register indicates that the content of the register is to be used to form the effective address of a memory operand, while an unbracketed register reference specifies that the register itself is the operand.

The following examples summarize how the memory addressing modes can be used to access simple variables, structures and arrays.

- If GA contains the address of a memory operand, then [GA] refers to that operand.
- If GA contains the base address of a structure, then [GA].DATA refers to the DATA element (field) in that structure. If DATA is six bytes from the beginning of the structure, then [GA].6 refers to the same location.
- If GA contains the starting address of an array, then [GA+IX] addresses the array element indexed by IX. For example, if IX contains the value 4H, the effective address refers to the fifth element of a byte array, or the third element of a word array. [GA+IX+] selects the same element and additionally auto-increments IX by 1 (byte operation), 2 (word operation) or 3 (MOVP instruction) in anticipation of accessing the next array element.

Note that any pointer register could have been substituted for GA in the previous examples.

Table 3-18. ASM-89 Memory Addressing Mode Notation

Notation	Addressing Mode
[*ptr-reg*]	Based
[*ptr-reg*].*offset*	Offset
[*ptr-reg* + IX]	Indexed
[*ptr-reg* + IX +]	Indexed Post Auto-increment

ptr-reg = GA, GB, GC or PP
offset = 8-bit signed value; may be structure element

Program Transfer Targets

As discussed in section 3.7, program transfer instructions operate by adding a signed byte or word displacement to the task pointer. Table 3-19 shows how the ASM-89 assembler determines the sign and size of the displacement value it places in a program transfer machine instruction. In the table, the terms "backward" and "forward" refer to the location of a label specified as a transfer target relative to the transfer instruction. "Backward" means the label physically precedes the instruction in the source module, and "forward" means the label follows the instruction in the source text. The distances are from the end of the transfer instruction; the distance to the instruction immediately following the transfer is 0 bytes.

```
ADDI    GA, 5           ; REGISTER, IMMEDIATE
ADD     GC, [GB]        ; REGISTER, MEMORY (BASED)
ADDBI   [PP],10         ; MEMORY (BASED), IMMEDIATE
ADDB    IX, [GB].5      ; REGISTER, MEMORY (OFFSET)
ADDB    BC, [GC].COUNT  ; REGISTER, MEMORY (OFFSET)
ADD     [GC+IX], BC     ; MEMORY (INDEXED), REGISTER
ADDI    [GA+IX+],5      ; MEMORY (INDEXED AUTO-INCREMENT), IMMED
ADDB    [PP].ERROR, [GA] ; MEMORY (OFFSET), MEMORY (BASED)
```

Figure 3-52. ASM-89 Operand Coding Examples

Two important points can be drawn from table 3-19. First, a target must lie within 32k bytes of a transfer instruction; this should not prove restrictive except in very large programs. Second, one byte can be saved in the assembled instruction by writing the short mnemonic when the target is known to be within −128 through +127 assembled bytes of the transfer.

It is also important to note that a program transfer target must reside in the same module as the transferring instruction, i.e., the target address must be known at assembly time.

Procedures

An ASM-89 program may invoke an out-of-line procedure (subroutine) with the CALL/LCALL instruction. The first instruction operand specifies a memory location where the content of TP will be stored as a physical address pointer before control is transferred to the procedure. The procedure may return to the instruction following the CALL/LCALL by using the MOVP instruction to restore TP from the save area. Figure 3-53 illustrates one approach to procedure linkage.

A channel program may use the first two words of its parameter block (pointed to by PP) as a task pointer save area. However, this is not recommended if there is any chance that the CPU will issue a "suspend" command to the channel; this command stores the current value of TP in the same location, possibly overwriting a return address.

As in any program transfer, the target of a CALL/LCALL instruction must be contained in the same module and within 32k bytes of the instruction.

Segment Control

The relocatable object module produced by the ASM-89 assembler consists of a single logical segment. (A segment is a storage unit up to 64k bytes long; for a more complete description, refer to sections 2.3 and 2.7.) The ASM-89 SEGMENT and ENDS directives name the segment as shown in figure 3-54. Typically, all instructions and most directives are coded in between these directives. The END directive, which terminates the assembly, is an exception.

The LOC-86 utility can assign this logical segment to any memory address that is a physical segment boundary (i.e., whose low-order four bits are 0000). In a ROM-based system, variable data (which must be in RAM) can be "clustered" together at one "end" of the program as shown in figure 3-55. The ORG directive can then be used to force assembly of the variables to start at a given offset from the beginning of the segment (2,000 hexadecimal bytes in figure 3-55). As the

Table 3-19. Program Transfer Displacement

Target Location			
Mnemonic Form	**Direction**	**Distance**	**Displacement Sign Bytes**
Short (e.g., JMP)	Backward	≤128	− 1
	Forward	≤127	+ 1
	Backward	≤32,768	− 2
	Forward	≤32,767	Error
	Backward	>32,768	Error
	Forward	>32,767	Error
Long (e.g., LJMP)	Backward	≤128	− 2
	Forward	≤127	+ 2
	Backward	≤32,768	− 2
	Forward	≤32,767	+ 2
	Backward	>32,768	Error
	Forward	>32,767	Error

```
.
.
.
CALL SAVE:   DS   3   ; TP SAVE AREA
.
.
.
; SET UP TP SAVE AREA
;    NOTE: EXAMPLE ASSUMES PROGRAM
;          IS IN I/O SPACE. USE LPDI
;          IF IN SYSTEM SPACE.
;          MOVI   GC, CALLSAVE      ; LOAD ADDRESS TO GC
; CALL IT.
          LCALL   [GC],DEMO
.
.
.
          HLT   ; LOGICAL END OF PROGRAM

; DEFINE THE PROCEDURE.
DEMO:
; PROCEDURE INSTRUCTIONS GO HERE.
; NOTE: PROCEDURE MUST NOT UPDATE GC
;       AS IT POINTS TO THE RETURN ADDRESS.
.
.
.
; RETURN TO CALLER.
          MOVP   TP, [GC]
```

Figure 3-53. ASM-89 Procedure Example

```
CHANNEL1     SEGMENT     ; START OF SEGMENT
               .
               .
               .
             ASM89 SOURCE STATEMENTS
               .
               .
               .
CHANNEL1     ENDS        ; END OF SEGMENT
             END         ; END OF ASSEMBLY
```

Figure 3-54. ASM-89 SEGMENT and ENDS Directives

figure shows, the segment can then be located so that instructions and constants fall into the ROM portion of memory, while the variable part of the segment is located in RAM. The entire segment, including any "unused" portions, of course, cannot exceed 64k bytes.

Intermodule Communication

An ASM-89 module can make some of its addresses available to other modules by defining symbols with the PUBLIC directive. At a minimum, a channel program must make the address of its first instruction available to the CPU module that starts the channel program. Figure 3-56 shows an ASM-89 module that contains three channel programs labelled READ, WRITE and DELETE. The example shows how a PL/M-86 program and an ASM-86 program could define these "entry points" as EXTERNAL and EXTRN symbols respectively. When the modules are linked together, LINK-86 will match the externals with the publics, thus providing the CPU programs with the addresses they need.

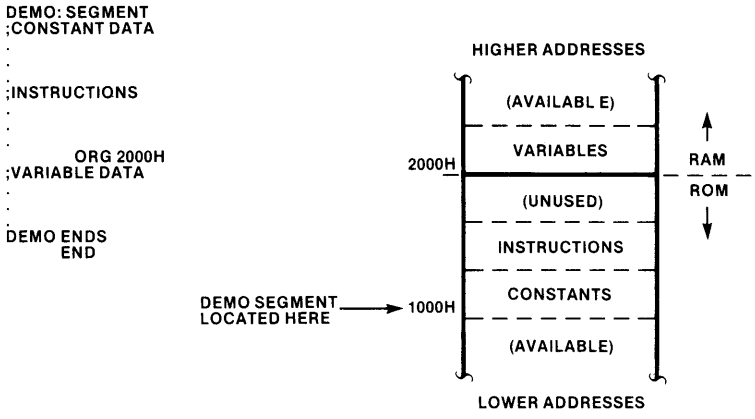


Figure 3-55. Using the ASM-89 ORG Directive

```
                    ASM-89 MODULE DEFINES THREE PUBLIC SYMBOLS
.
.
.
PUBLIC      READ, WRITE, DELETE
.
.
.
.
READ:       ; ASM89 INSTRUCTIONS FOR "READ" OPERATION
.
.
            HLT
WRITE:      ; ASM89 INSTRUCTIONS FOR "WRITE" OPERATION
.
.
            HLT
DELETE:     ; ASM89 INSTRUCTIONS FOR "DELETE" OPERATION
.
.
            HLT
```

Figure 3-56. ASM-89 PUBLIC Directive

PL/M-86 MODULE USES "WRITE" SYMBOL

```
DECLARE   (READ,WRITE,DELETE) POINTER EXTERNAL;
DECLARE   PARM$BLOCK   STRUCTURE
          (TP$START          POINTER,
          BUFFER$ADDR         POINTER,
          BUFFER$LEN          WORD);
.
.
.
/*SET UP "WRITE" CHANNEL OPERATION*/
PARM$BLOCK. TP$START = WRITE;
.
.
.
```

ASM-86 MODULE USES "READ" SYMBOL

```
                .
                .
                .
EXTRN           READ,WRITE,DELETE
.
.
.
READ_PTR        DD  READ
WRITE_PTR       DD  WRITE
DELETE_PTR      DD  DELETE
.
.
.
; PARM_BLOCK
                EVEN        ; FORCE TO EVEN ADDRESS
TP_START        DD ?
BUFFER_ADDR     DD ?
BUFFER_LEN      DW ?
.
.
.
; SET UP "READ" CHANNEL OPERATION
      MOV   AX, WORD PTR READ_PTR          ; 1ST WORD
      MOV   WORD PTR TP_START, AX
      MOV   AX, WORD PTR READ_PTR          ; 2ND WORD
      MOV   WORD PTR TP_START + 2, AX
      .
      .
      .
```

Figure 3-56. ASM-89 PUBLIC Directive (Cont'd.)

Conversely, an ASM-89 module can obtain the address of a public symbol in another module by defining it with the EXTRN directive. An external symbol, however, can only appear as the initial value operand of a DD directive (see figure 3-57). This effectively means that an ASM-89 program's use of external symbols is limited to obtaining the addresses of data located in the system space. Another way of doing this, which may be preferable in many cases, is to have the CPU program place system space addresses in the parameter block.

PL/M-86 PROGRAM DECLARES PUBLIC SYMBOL "BUFFER"

```
.
.
.
DECLARE BUFFER (80) BYTE PUBLIC;
.
.
.
```

ASM-89 PROGRAM OBTAINS ADDRESS OF PUBLIC SYMBOL "BUFFER"

```
.
.
.
EXTRN BUFFER
.
.
.
BUF_ADDRESS   DD   BUFFER
.
.
.
LPD   GA, BUF_ADDRESS   ; POINT TO SYSTEM BUFFER
.
.
.
```

Figure 3-57. ASM-89 EXTRN Directive

Sample Program

Figure 3-58 diagrams the logic of a simple ASM-89 program; the code is shown in figure 3-59. The program reads one physical record (sector) from a diskette drive controlled by an 8271 Floppy Disk Controller. No particular system configuration is implied by the program, except that the 8271 resides in the IOP's I/O space.

Hardware address decoding logic is assumed to be set up as follows:

- reading location FF00H selects the 8271 status register,
- writing location FF00H selects the 8271 command register,
- reading location FF01H selects the 8271 result register
- writing location FF01H selects the 8271 parameter register
- decoding the address FF04H provides the 8271 DACK (DMA acknowledge) signal.

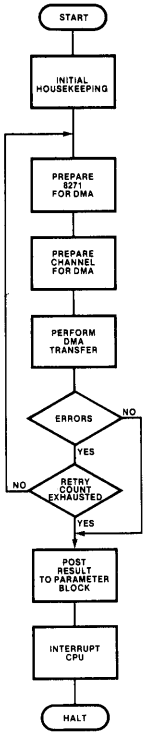


Figure 3-58. ASM-89 Sample Program Flow

The program uses structures to address the parameter block and the 8271 registers. Register PP contains the address of the parameter block, and the program loads GC with FF00H to point to the 8271 registers. The program's entry point (the label START) is defined as a PUBLIC symbol so that the CPU program can place its address in the parameter block when it starts the program.

Register IX is used as a retry counter. If the transfer is not completed successfully (bit 3 of the 8271 result register ≠ 0), the program retries the transfer up to 10 times.

Since the 8271 automatically requests a DMA transfer upon receipt of the last parameter, this parameter is sent immediately following the XFER command.

```
8089 ASSEMBLER


ISIS-II 8089 ASSEMBLER V1.0 ASSEMBLY OF MODULE FLOPPY
OBJECT MODULE PLACED IN :F0:FLOPPY.OBJ
ASSEMBLER INVOKED BY ASM89 FLOPPY.A89


                                        1
0000                                    2 FLOPPY            SEGMENT
                                        3 ;***
                                        4 ;*** 8089 PROGRAM TO READ SECTOR FROM FLOPPY DISK
                                        5 ;***
                                        6
                                        7 ;*** LAY OUT PARAMETER BLOCK.
0000                                    8 PARM_BLOCK        STRUC
0004                                    9    RESERVED_TP:       DS   4
0008                                   10    BUFF_PTR:          DS   4
0009                                   11    TRACK:             DS   1
000A                                   12    SECTOR:            DS   1
000B                                   13    RETURN_CODE:       DS   1
                                       14    PARM_BLOCK    ENDS
                                       15
                                       16 ;***LAY OUT 8271 DEVICE REGISTERS.
0000                                   17 FLOPPY_REGS      STRUC
0001                                   18    COMMAND_STAT:      DS   1
0002                                   19    PARM_RESULT:       DS   1
                                       20    FLOPPY_REGS   ENDS
                                       21
                                       22 ;***8271 ADDRESSES.
 FF00                                  23 FLOPPY_REG_ADDR  EQU   OFF00H        ;LOW-ADDRESSED REGISTER
 FF04                                  24 DACK_8271        EQU   OFF04H        ;DMA ACKNOWLEDGE
                                       25
                                       26 ;***MAKE PROGRAM ENTRY POINT ADDRESS
                                       27 ;       AVAILABLE TO OTHER MODULES.
                                       28 PUBLIC           START
                                       29
                                       30 ;***CLEAR RETURN CODE IN PARAMETER BLOCK.
0000  0A4F 0A 00                       31 START:           MOVBI   [PP].RETURN_CODE,0
                                       32
                                       33 ;***INITIALIZE RETRY COUNT.
0004  B130 0A00                       34                  MOVI    IX,10
                                       35
                                       36 ;***POINT GC AT LOW-ORDER 8271 REGISTER.
0008  5130 00FF                       37                  MOVI    GC,FLOPPY_REG_ADDR
                                       38
                                       39 ;***SEND COMMAND SEQUENCE TO 8271, HOLDING FINAL PARM.
                                       40 ;***WAIT UNTIL 8271 IS NOT BUSY.
000C  EABA 00 FC                       41 RETRY:           JNBT    [GC].COMMAND_STAT,7,RETRY
                                       42 ;***SEND "READ SECTOR, DRIVE 0" COMMAND.
0010  0A4E 00 12                       43                  MOVBI   [GC].COMMAND_STAT,012H
                                       44 ;***SEND TRACK ADDRESS PARAMETER.
0014  0293 08 02CE 01                  45                  MOVB    [GC].PARM_RESULT,[PP].TRACK
                                       46
                                       47 ;***LOAD CHANNEL CONTROL REGISTER SPECIFYING:
                                       48 ;   FROM PORT TO MEMORY,
                                       49 ;   SYNCHRONIZE ON SOURCE,
                                       50 ;   GA POINTS TO SOURCE,
                                       51 ;   TERMINATE ON EXT,
                                       52 ;   TERMINATION OFFSET = 0.
001A  D130 2088                        53                  MOVI    CC,08820H
                                       54
```

Figure 3-59. ASM-89 Sample Program

```
                                        55 ;***SET SOURCE BUS = 8, DEST BUS = 16.
001E   A000                             56                  WID     8,16
                                        57
                                        58 ;***POINT GB AT DESTINATION, GA AT SOURCE.
0020   238B 04                          59                  LPD     GB,[PP].BUFF_PTR
0023   1130 04FF                        60                  MOVI    GA,DACK_8271
                                        61
                                        62 ;***INSURE THAT 8271 IS READY FOR LAST PARAMETER.
0027   AABA 00 FC                       63 WAIT1:           JNBT    [GC].COMMAND_STAT,5,WAIT1
                                        64
                                        65 ;***PREPARE FOR DMA.
002B   6000                            66                  XFER
                                        67
                                        68 ;***START DMA BY SENDING FINAL PARAMETER TO 8271.
002D   0293 09 02CE 01                  69                  MOVB    [GC].PARM_RESULT,[PP].SECTOR
                                        70
                                        71 ;***PROGRAM RESUMES HERE FOLLOWING EXT.
                                        72
                                        73 ;***IF TRANSFER IS OK THEN EXIT, ELSE TRY AGAIN.
0033   6ABE 01 05                       74                  JBT     [GC].PARM_RESULT,3,EXIT
                                        75
                                        76 ;***DECREMENT RETRY COUNT.
0037   A03C                             77                  DEC     IX
                                        78
                                        79 ;***TRY AGAIN IF COUNT NOT EXHAUSTED.
0039   A840 D0                          80                  JNZ     IX,RETRY
                                        81
                                        82 ;***WAIT UNTIL 8271 IS NOT BUSY.
003C   EABA 00 FC                       83 EXIT:            JNBT    [GC].COMMAND_STAT,7,EXIT
                                        84
                                        85 ;***SEND "READ RESULT" COMMAND TO 8271.
0040   0A4E 00 2C                       86                  MOVBI   [GC].COMMAND_STAT,02CH
                                        87
                                        88 ;***WAIT FOR RESULT.
0044   8ABA 00 FC                       89 WAIT2:           JNBT    [GC].COMMAND_STAT,4,WAIT2
                                        90
                                        91 ;***POST RESULT IN PARAMETER BLOCK FOR CPU.
0048   0292 01 02CF 0A                  92                  MOVB    [PP].RETURN_CODE,[GC].PARM_RESULT
                                        93
                                        94 ;***INTERRUPT CPU.
004E   4000                            95                  SINTR
                                        96
                                        97 ;***STOP EXECUTION.
0050   2048                            98                  HLT
                                        99
0052                                   100 FLOPPY          ENDS
                                       101                 END


SYMBOL TABLE
------------

 DEFN VALUE TYPE  NAME
----- ----- ----  ----

   10  0004  SYM  BUFF_PTR
   18  0000  SYM  COMMAND_STAT
   24  FF04  SYM  DACK_8271
   83  003C  SYM  EXIT
    2  0000  SYM  FLOPPY
   17  0000  STR  FLOPPY_REGS
   23  FF00  SYM  FLOPPY_REG_ADDR
    8  0000  STR  PARM_BLOCK
   19  0001  SYM  PARM_RESULT
    9  0000  SYM  RESERVED_TP
   41  000C  SYM  RETRY
   13  000A  SYM  RETURN_CODE
   12  0009  SYM  SECTOR
   31  0000  PUB  START
   11  0008  SYM  TRACK
   63  0027  SYM  WAIT1
   89  0044  SYM  WAIT2

ASSEMBLY COMPLETE; NO ERRORS FOUND
```

Figure 3-59. ASM-89 Sample Program (Cont'd.)

Linking and Locating ASM-89 Modules

The LINK-86 utility program combines multiple relocatable object modules into a single relocatable module. The input modules may consist of modules produced by any of the 8086 family language translators: ASM-89, ASM-86, or PL/M-86. LINK-86's principal function is to satisfy external references made in the modules. Any symbol that is defined with the EXTRN directive in ASM-89 or ASM-86 or is declared EXTERNAL in PL/M-86 is an external reference, i.e., a reference to an address contained in another module. Whenever LINK-86 encounters an external reference, it searches the other modules for a PUBLIC symbol of the same name. If it finds the matching symbol, it replaces the external reference with the address of the object.

The most common occurrence of an external reference in a system that employs one or more 8089s is the channel program address. In order for a CPU program to start a channel program, it must ensure that the address of the first channel program instruction is contained in the first two words of the parameter block. Since the channel program is assembled separately, the translator that processes the CPU program will not typically know its address. If this address is defined as an external symbol (see figure 3-56), LINK-86 will obtain the address from the ASM-89 channel program when the two are linked together. (The ASM-89 program must, of course, define the symbol in a PUBLIC directive.)

Other external references may arise when one module uses data (e.g., a buffer) that is contained in another module, and (in PL/M-86 and ASM-86 modules) when one module executes another module, typically by a CALL statement or instruction.

When an 8089 module (or modules) is to be located in the system space, it may be linked together with PL/M-86 or ASM-86 modules as described above and shown in figure 3-60. LINK-86 resolves external references and combines the input modules into a single relocatable object module. This module can be input to LOC-86 (LOC-86 assigns final absolute memory addresses to all of the instructions and data). This absolute object module may, in turn, be processed by the OH-86 utility to translate the module into the hexadecimal format. This format makes the module readable (the records are written in ASCII characters) and is required by some PROM programmers and RAM loaders. Intel's Universal PROM Programmer (UPP) and iSBC 957™ Execution Package (loader) use the hexadecimal format.

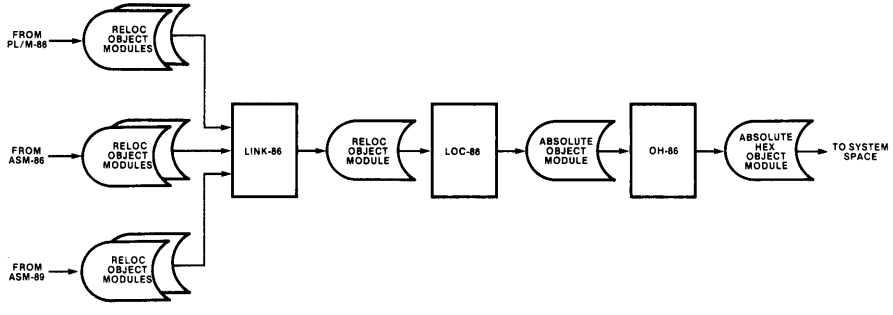


Figure 3-60. Creating a Single Absolute Object Module

If the 8089 code is to reside in its I/O space, a different technique is required since separate absolute object modules must be produced for the system and I/O spaces. Figure 3-61 shows how to link and locate when there are external references between I/O space modules and system space modules.

The normal link and locate sequence is followed and culminates in the production of an absolute module in hexadecimal format. Since the records in this file are human-readable, the file can be edited using the ISIS-II text editor. The editing task involves finding the 8089 I/O space records in the file, writing them to one file, and then writing the 8086/8088 records (destined for the system space) to another file. *MCS-86 Absolute Object File Formats*, Order No. 9800921, available from Intel's Literature Department, describes the records in absolute (including hexadecimal) object modules.

When using the previous method, it is likely that LOC-86 will issue messages warning that segments overlap. For example, the 8089 code would typically be located starting at absolute location 0H of the I/O space. However, the 8086/8088 interrupt pointer table occupies these low memory addresses in the system space. Since LOC-86 has no way to know that the segment will ultimately be located in different address spaces, it will warn of the conflict; the warning may be ignored.

An alternative to linking the modules together and then separating them is to link system space modules separately from I/O space modules as shown in figure 3-62. This approach avoids the manual edit of the absolute object module and the segment conflict messages from LOC-86. It requires, however, that modules in the two spaces not use the EXTRN/PUBLIC mechanism to refer to each other. Modules in the same space can define external and public symbols, however.

External references from I/O space modules to system space modules can be eliminated if the CPU programs pass all system space addresses in parameter blocks. In other words, a channel program can obtain any address in the system space if the address is in the parameter block. Using this approach allows the system space addresses to be changed during execution. If the addresses are constant values, they may also be altered as system development proceeds without relinking the channel programs.

External references from system space modules to addresses in the I/O space may be eliminated by assigning these addresses values that are known at assembly or compilation time. Figure 3-63 illustrates how the ASM-89 ORG directive can be used to force the first instruction (entry point) of a channel program to an absolute address. In the case of the example, one module contains two entry points labelled "READ" and "WRITE." Assuming the module is located at absolute address 0H in the I/O space, the channel programs will begin at 200H and 600H respectively. In the example, these values have been chosen arbitrarily; in a typical application they would be based on the length of the programs and the location of RAM and ROM areas. By starting the programs at fixed addresses that are known to the CPU programs that activate them, the channel programs can be reassembled without needing to relink the CPU programs.

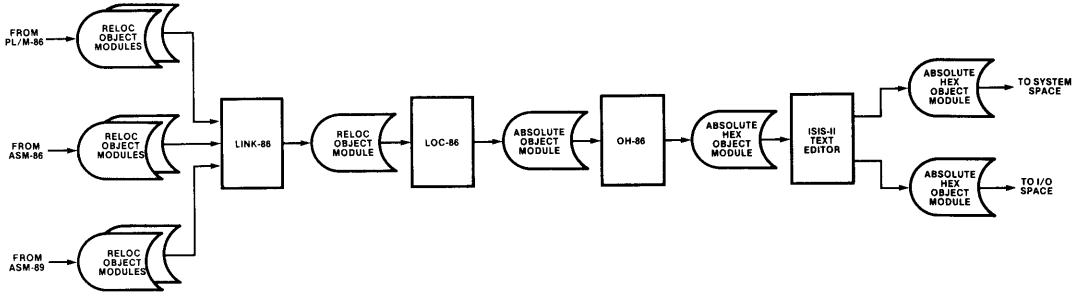


Figure 3-61. Creating Separate Absolute Object Modules—External References in Relocatable Modules

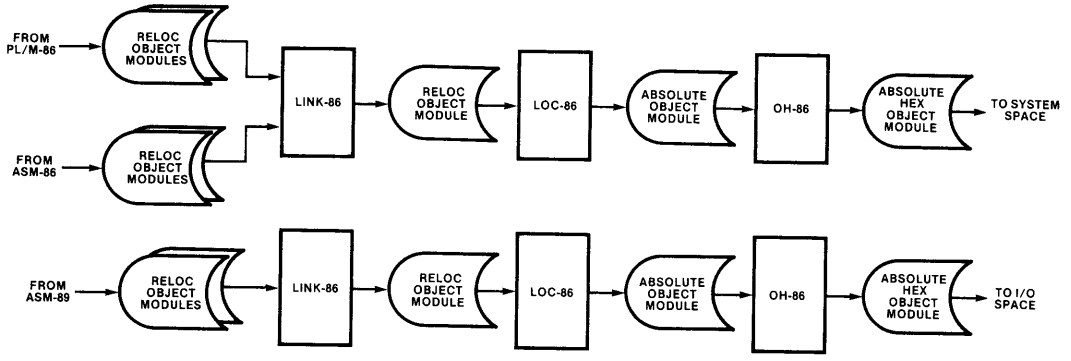


Figure 3-62. Creating Separate Absolute Object Modules—No External References in Relocatable Modules

```
ASM-89 ENTRY POINT DEFINITIONS
.
.
.
        ORG 200H
READ:
.
.
.
; INSTRUCTIONS FOR "READ" CHANNEL PROGRAM
.
.
.
        ORG 600H
WRITE:
.
.
.
; INSTRUCTIONS FOR "WRITE" CHANNEL PROGRAM
.
.
.

ASM-86 DEFINITION OF ENTRY POINT ADDRESSES
.
.
.
READ_ADDR       DD  200H
WRITE_ADDR      DD  600H
.
.
.

PL/M-86 DECLARATION OF ENTRY POINT ADDRESSES
.
.
.
DECLARE READ$ADDR POINTER;
DECLARE WRITE$ADDR POINTER;
READ$ADDR = 200H;
WRITE$ADDR = 600H;
```

Figure 3-63. Using Absolute Entry Point Addresses

3.10 Programming Guidelines and Examples

This section provides two types of 8089 programming information. A series of general guidelines, which apply to system and program design, is presented first. These guidelines are followed by specific coding examples that illustrate programming techniques that may be applied to many different types of applications.

Programming Guidelines

The practices in this section are recommended to simplify system development and, particularly, for system maintenance and enhancement. Software that is designed in accordance with these **guidelines will be adaptable to the changing environment in which most systems operate, and will be in the best position to take advantage of new Intel hardware and software products.**

Segments

Although the IOP does not "see" the segmented organization of system memory, it should respect this logical structure. The IOP should only address the system space through pointers passed by the CPU in the parameter block. It should not perform arithmetic on these addresses or otherwise manipulate them except for the automatic incrementing that occurs during DMA transfers. It is the responsibility of the CPU to pass addresses such that transfer operations do not cross segment boundaries.

Self-Modifying Code

Programs that alter their own instructions are difficult to understand and modify, and preclude placing the code in ROM. They may also inhibit compatibility with future Intel hardware and software products.

Note also that when the 8089 is on a 16-bit bus, its instruction fetch queue can interfere with the attempt of one instruction to modify the next sequential instruction. Although the instruction may be changed in memory, its unmodified first byte will be fetched from the queue rather than memory if it is on an odd address. The processor will thus execute a partially-modified instruction with unpredictable results.

I/O System Design

Section 2.10 notes that I/O systems should be designed hierarchically. Application programs "see" only the topmost level of the structure; all details pertaining to the physical characteristics and operation of I/O devices are relegated to lower levels. Figure 3-64 shows how this design approach might be employed in a system that uses an 8089 to perform I/O. The same concept can be expanded to larger systems with multiple IOPs.

The application system is clearly separated from the I/O system. No application programs perform I/O; instead they send an I/O request to the I/O supervisor. (In systems with file-oriented I/O, the request might be sent to a file system that would then invoke the I/O supervisor.) The I/O request should be expressed in terms of a logical block of data—a record, a line, a message, etc. It should also be devoid of any device-dependent information such as device address, sector size, etc.

The I/O supervisor transforms the application program's request for service into a parameter block and dispatches a channel program to carry out the operation. The I/O supervisor controls the channels; therefore, it knows the correspondence between channels and I/O devices, the locations of CBs and channel programs, and the format of all of the parameter blocks. The I/O supervisor also coordinates channel "events," monitoring BUSY flags and responding to channel-generated interrupt requests. The I/O supervisor does not, however, communicate with I/O devices that are controlled by the channels. If the CPU performs some I/O itself (this should be restricted to devices other than those run by the channels), the I/O supervisor invokes the equivalent of a channel program in the CPU to do the physical I/O. Note that although the I/O supervisor is drawn as a single box in figure 3-64, it is likely to be structured as a hierarchy itself, with separate modules performing its many functions.

The software interface between the CPU's I/O supervisor and an IOP channel program should be completely and explicitly defined in the

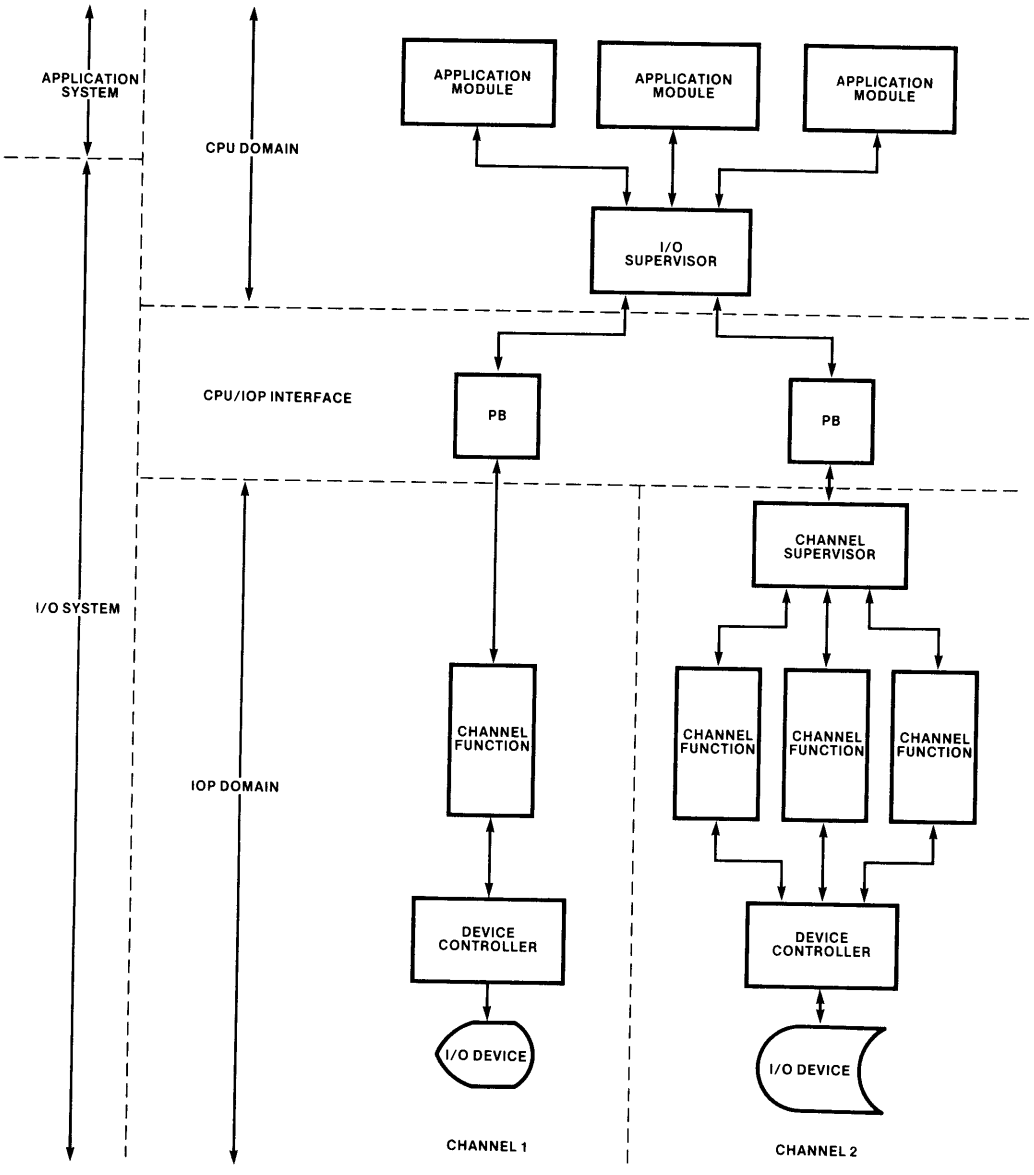


Figure 3-64. 8089-Based I/O System Design

parameter block. For example, the I/O supervisor should pass the addresses of all system memory areas that the channel program will use. The channel program should not be written so that it "knows" any of these addresses, even if they are constants. Concentrating the interface into one place like this makes the system easier to understand and reduces the likelihood of an undesirable side effect if it is modified. It also generalizes the design so that it may be used in other application systems.

Figure 3-64 shows a simple channel program running on channel 1 and a more complex program running on channel 2. Channel 1's program performs a single function and is therefore designed as a simple program. The program on channel 2 performs three functions (e.g., "read," "write," "delete") and is structured to separate its functions. The functions might be implemented as procedures called by the "channel supervisor" depending on the content of the parameter block. Notice that to the I/O supervisor, both programs appear alike; in particular, both have a single entry point.

In some channel programs, different functions will need different information passed to them in the parameter block. Figure 3-65 shows one technique that accommodates different formats while still allowing the channel supervisor to determine which procedure to call from the PB. The parameter block is divided into fixed and variable portions, and a function code in the fixed area indicates the type of operation that is to be performed. Part of the fixed area has been set aside so that additional parameters can be added in the future.

Programming Examples

The first example in this section illustrates how a CPU can initialize a group of IOPs and then dispatch channel programs. This code is written in PL/M-86.

The remaining examples, written in ASM-89, demonstrate the 8089 instruction set and addressing modes in various commonly-encountered programming situations. These include:

- memory-to-memory transfers
- saving and restoring registers

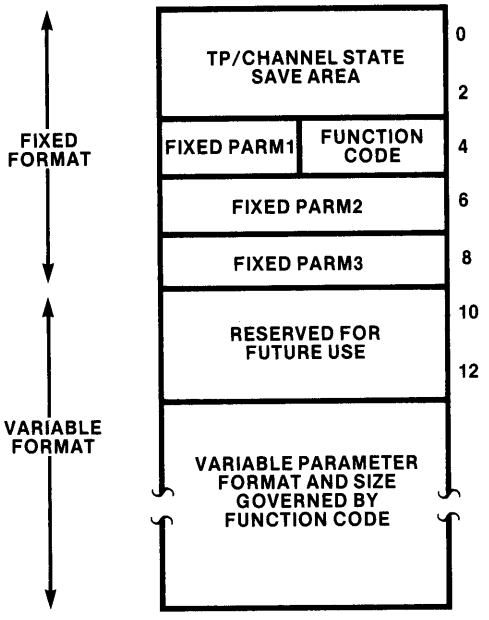


Figure 3-65. Variable Format Parameter Block

Initialization and Dispatch

The PL/M-86 code in figure 3-66 initializes two IOPs and dispatches two channel programs on one of the IOPs. The same general technique can be used to initialize any number of IOPs. The hypothetical system that this code runs on is configured as follows:

- 8086 CPU (16-bit system bus);
- two remote IOPs share an 8-bit local I/O bus via the request/grant lines operating in mode 1;
- 8089 channel attentions are mapped into four port addresses in the CPU's I/O space;
- channel programs reside in the 8089 I/O space;
- one 8089 controls a CRT terminal, one channel running the display, the other scanning the keyboard and building input messages;
- the function of the second 8089 is not defined in the example.

The code declares one CB (channel control block) for each 8089. The CBs are declared as two-element arrays, each element defining the structure of one channel's portion of the CB. The SCB (system configuration block) and SCP (system configuration pointer) are also declared as structures. The SCP is located at its dedicated system space address of FFFF6H. The other structures are not located at specific addresses since they are all linked together by a chain of pointers "anchored" at the SCP.

Two simple parameter blocks define messages to be transmitted between the PL/M-86 program and the CRT. Each PB contains a pointer to the beginning of the message area and the length of the message. In the case of the keyboard (input) message, the channel program builds the message in the buffer pointed to by the pointer in the PB and returns the length of the message in the PB.

The code initializes one IOP at a time since the chain of control blocks read by the IOP during initialization must remain static until the process is complete. To initialize the first IOP, the code fills in the SYSBUS and SOC fields and links the blocks to each other using the PL/M-86 @ (address) operator. It sets channel 1's BUSY flag to FFH so that it can monitor the flag to determine when the initialization has been completed (the IOP clears the flag to 0H when it has finished). Channel 2's BUSY flag is cleared, although this could just as well have been done after the initialization (the IOP does not alter channel 2's BUSY flag during initialization). The code starts the IOP by issuing a channel attention to channel 1 to indicate that the IOP is a bus master. PL/M-86's OUT function is used to select the port address to which the IOP's CA and SEL lines have been mapped. The data placed on the bus (0H) is ignored by the IOP. It then waits until the IOP clears the channel 1 BUSY flag.

The second IOP is initialized in the same manner, first changing the pointer in the SCB to point to the second IOP's channel control block. If this IOP were on a different I/O bus, the SOC field would have been altered if a different request/grant mode were being used or if the IOP had a 16-bit I/O bus. The second IOP is a slave so its initialization is started by issuing a CA to channel 2 rather than channel 1.

After both IOPs are ready, the code dispatches two channel programs (not coded in the example); one program is dispatched to each channel of one of the IOPs. To avoid external references, the system has been set up so that the PL/M-86 code "knows" the starting addresses of these channel programs (200H and 600H). The code uses the PL/M-86 LOCKSET function to:

- lock the system bus;
- read the BUSY flag;
- set the BUSY flag to FFH if it is clear;
- unlock the system bus.

This operation continues until the BUSY flag is found to be clear (indicating that the channel is available). Setting the flag immediately to FFH prevents another processor (or another task in this program activated as a result of an interrupt) from using the channel. The code fills in the parameter block with the address and length of the message to be displayed, sets the CCW and then links the channel program (task block) start address to the parameter block and links the parameter block to the CB. The channel is dispatched with the OUT function that effects a channel attention for channel 1.

A similar procedure is followed to start channel 2 scanning the terminal keyboard. In this case, the code allows channel 2 to generate an interrupt request (which it might do to signal that a message has been assembled). An interrupt procedure would then handle the interrupt request.

```
/*ASSIGN NAMES TO CONSTANTS*/
DECLARE     CHANNEL$BUSY        LITERALLY '0FFH';
DECLARE     CHANNEL$CLEAR       LITERALLY '0H';
DECLARE     CR /*CARR. RET.*/   LITERALLY '0DH';
DECLARE     LF /*LINE FEED*/    LITERALLY '0AH';
DECLARE     DISPLAY$TB          LITERALLY '200H';
DECLARE     KEYBD$TB            LITERALLY '600H';
```

Figure 3-66. Initialization and Dispatch Example

```
DECLARE  /*IOP CHANNEL ATTENTION ADDRESSES*/
IOP$A$CH1      LITERALLY          '0FFE0H',
IOP$A$CH2      LITERALLY          '0FFE1H',
IOP$B$CH1      LITERALLY          '0FFE2H',
IOP$B$CH2      LITERALLY          '0FFE3H';

DECLARE  /*CHANNEL CONTROL BLOCK FOR IOP$A)
           CB$A(2)          STRUCTURE
           (BUSY            BYTE,
           CCW              BYTE,
           PB$PTR           POINTER,
           RESERVED         WORD);

DECLARE  /*CHANNEL CONTROL BLOCK FOR IOP$B*/
           CB$B(2)          STRUCTURE
           (BUSY            BYTE,
           CCW              BYTE,
           PB$PTR           POINTER,
           RESERVED         WORD);

DECLARE  /*SYSTEM CONFIGURATION BLOCK*/
           SCB              STRUCTURE
           (SOC             BYTE,
           RESERVED         BYTE,
           CB$PTR           POINTER);

DECLARE  /*SYSTEM CONFIGURATION POINTER*/
           SCP              STRUCTURE
           (SYSBUS          BYTE,
           SCB$PTR          POINTER) AT (0FFFF6H);

DECLARE  MESSAGE$PB STRUCTURE
           (TB$PTR          POINTER,
           MSG$PTR          POINTER,
           MSG$LENGTH       WORD);

DECLARE  KEYBD$PB STRUCTUE
           (TP$PTR          POINTER,
           BUFF_PTR         POINTER,
           MSG$SIZE         WORD);

DECLARE  SIGN$ON BYTE (*) DATA
         (CR, LF, 'PLEASE ENTER USER ID');

DECLARE  KEYBD$BUFF BYTE (256);

/*
 *INITIALIZE IOP$A, THEN IOP$B
*/

/*PREPARE CONTROL BLOCKS FOR IOP$A*/
SCP.SCB$PTR = @ SCB;
SCP.SYSBUS = 01H; /*16-BIT SYSTEM BUS*/
SCB.SOC = 02H; /*RQ/GT MODE1, 8-BIT I/O BUS*/
SCB.CB$PTR = @ CB$A(0);
CB$A(0).BUSY = CHANNEL$BUSY
CB$A(1).BUSY = CHANNEL$CLEAR;
```

Figure 3-66. Initialization and Dispatch Example (Cont'd.)

```
/*ISSUE CA FOR CHANNEL1, INDICATING IOP IS MASTER*/
OUT (IOP$A$CH1) = 0H;

/*WAIT UNTIL FINISHED*/
DO WHILE CB$A(0).BUSY = CHANNEL$BUSY;
    END;

/*PREPARE CONTROL BLOCKS FOR IOP$B*/
SCB.CB$PTR = @CB$B(0);
CB$B(0).BUSY = CHANNEL$BUSY;
CB$B(1).BUSY = CHANNEL$CLEAR;

/*ISSUE CA FOR CHANNEL2, INDICATING SLAVE STATUS*/
OUT (IOP$B$CH2) = 0H;

/*WAIT UNTIL IOP IS READY*/
DO WHILE CB$B(0).BUSY = CHANNEL$BUSY;
    END;



/*
 *SEND SIGN ON MESSAGE TO CRT CONTROLLED
 *BY CHANNEL 1 OF IOP$A
 */
/*WAIT UNTIL CHANNEL IS CLEAR, THEN SET TO BUSY*/
DO WHILE LOCKSET (@CB$A(0).BUSY, CHANNEL$BUSY);
    END;

/*SET CCW AS FOLLOWS:
 *      PRIORITY = 1,
 *      NO BUS LOAD LIMIT,
 *      DISABLE INTERRUPTS,
 *      START CHANNEL PROGRAM IN I/O SPACE*/
CB$A(0).CCW = 10011001B;

/*LINK MESSAGE PARAMETER BLOCK TO CB*/
CB$A(0).PB$PTR = @ MESSAGE$PB;

/*FILL IN PARAMETER BLOCK*/
MESSAGE$PB.TB$PTR = DISPLAY$TB;
MESSAGE$PB.MSG$PTR = @SIGN$ON;
MESSAGE$PB. MSB$LENGTH = LENGTH (SIGN$ON);

/*DISPATCH THE CHANNEL*/
OUT (IOP$A$CH1) = 0H;

/*
 *DISPATCH CHANNEL 2 OF IOP$A TO
 *CONTINUOUSLY SCAN KEYBOARD, INTERRUPTING
 *WHEN A COMPLETE MESSAGE IS READY
 */
/*WAIT UNTIL CHANNEL IS CLEAR, THEN SET TO BUSY*/
DO WHILE LOCKSET (@ CB$A(1).BUSY, CHANNEL$BUSY);
   END;
```

Figure 3-66. Initialization and Dispatch Example (Cont'd.)

```
/*SET CCW AS FOLLOWS:
  *     PRIORITY = 0
  *     BUS LOAD LIMIT,
  *     ENABLE INTERRUPTS,
  *     START CHANNEL PROGRAM IN I/O SPACE*/
CB$A(1).CCW = 00110001B;
/*LINK KEYBOARD PARAMETER BLOCK TO CB*/
CB$A(1).PB$PTR = @ KEYBD$PB;
/*FILL IN PARAMETER BLOCK*/
KEYBD$PB.TB$PTR = KEYBD$TB;
KEYBD$PB.BUFF$PTR = @ KEYBD$BUFF;
KEYBD$PB.MSG$SIZE = 0H;
/*DISPATCH THE CHANNEL*/
OUT (IOP$A$CH2) = 0H;
```

Figure 3-66. Initialization and Dispatch Example (Cont'd.)

Memory-to-Memory Transfer

Figure 3-67 shows a channel program that performs a memory-to-memory block transfer in seven instructions. The program moves up to 64k bytes between any two locations in the system space. A 16-bit system bus is assumed, and the CPU is assumed to be monitoring the channel's BUSY flag to determine when the program has finished.

To attain maximum transfer speed, the program locks the bus during each transfer cycle. This ensures that another processor does not acquire the bus in the interval between the DMA fetch and store operations. By setting this channel's priority bit in the CCW to 1 and the other channel's to 0, the CPU could effectively prevent the other channel from running during the transfer. Byte count termination is selected so that the transfer will stop when the number of bytes specified by the CPU has been moved. Since there is only a single termination condition, a termination offset of 0 is specified. The transfer begins after the WID instruction, and the HLT instruction is executed immediately upon termination.

Saving and Restoring Registers

A CPU program can "interrupt" a channel program by issuing a "suspend" channel command. The channel responds to this command by saving the task pointer and PSW in the first two words of the parameter block. The suspended program can be restarted by issuing a "resume" command that loads TP and the PSW from the save area.

If the CPU wants to execute another channel program between the suspend and resume operations, the suspended program's registers will usually have to be saved first. If the "interrupting" program "knows" that the registers must be saved, it can perform the operation and also restore the registers before it halts.

A more general solution is shown in figure 3-68. This is a program that does nothing but save the contents of the channel registers. The registers are saved in the parameter block because PP is the only register that is known to point to an available area of memory. A similar program could be written to restore registers from the same parameter block.

Using this approach, the CPU would "interrupt" a running program as follows:

- suspend the running program,
- run the register save program,
- run the "interrupting" program,
- run the register restore program,
- resume the suspended program.

```
MEMEXAMP        SEGMENT
;**MEMORY-TO-MEMORY TRANSFER PROGRAM**
PB              STRUC
TP_RESERVED:    DS      4
FROM_ADDR:      DS      4
TO_ADDR:        DS      4
SIZE:           DS      2
PB              ENDS

;POINT GA AT SOURCE, GB AT DESTINATION.
                LPD             GA, [PP].FROM_ADDR
                LPD             GB, [PP].TO_ADDR
;LOAD BYTE COUNT INTO BC.
                MOV             BC, [PP].SIZE
;LOAD CC SPECIFYING:
;       MEMORY TO MEMORY,
;       NO TRANSLATE,
;       UNSYNCHRONIZED,
;       GA POINTS TO SOURCE,
;       LOCK BUS DURING TRANSFER,
;       NO CHAINING,
;       TERMINATING ON BYTE COUNT,OFFSET = 0.
                MOV             CC, 0C208H
;PREPARE CHANNEL FOR TRANSFER.
                XFER

;SET LOGICAL BUS WIDTH.
                WID             16,16

;STOP EXECUTION AFTER DMA.
                HLT
MEMEXAMP        ENDS
                END
```

Figure 3-67. Memory-to-Memory Transfer Example

```
SAVEREGS        SEGMENT
;SAVE ANOTHER CHANNEL'S REGISTERS IN PB
PB              STRUC
TP_RESERVED:    DS      4
GA_SAVE:        DS      3
GB_SAVE:        DS      3
GC_SAVE:        DS      3
IX_SAVE:        DS      2
BC_SAVE:        DS      2
MC_SAVE:        DS      2
CC_SAVE:        DS      2
PB              ENDS

                MOVP    [PP].GA_SAVE, GA
                MOVP    [PP].GB_SAVE, GB
                MOVP    [PP].GC_SAVE, GC
                MOV     [PP].IX_SAVE, IX
                MOV     [PP].BC_SAVE, BC
                MOV     [PP].MC_SAVE, MC
                MOV     [PP].CC_SAVE, CC
                HLT
SAVEREGS        ENDS
                END
```

Figure 3-68. Register Save Example