Chapter 3
The 8089
Input/Output Processor

CHAPTER 3
THE 8089 INPUT/OUTPUT PROCESSOR

This chapter describes the 8089 Input/Output Processor (IOP). Its organization parallels Chapter 2; that is, sections generally proceed from hardware to software topics as follows:

1. Processor Overview
2. Processor Architecture
3. Memory
4. Input/Output
5. Multiprocessing Features
6. Processor Control and Monitoring
7. Instruction Set
8. Addressing Modes
9. Programming Facilities
10. Programming Guidelines and Examples

As in Chapter 2, the discussion is confined to covering the hardware in functional terms; timing, electrical characteristics and other physical interfacing data are provided in Chapter 4.

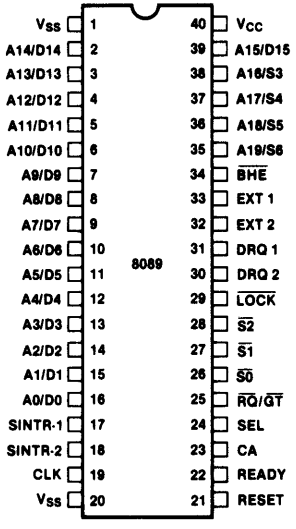


Figure 3-1. 8089 Input/Output Processor Pin Diagram

3.1 Processor Overview

The 8089 Input/Output Processor is a high-performance, general-purpose I/O system implemented on a single chip. Within the 8089 are two independent I/O channels, each of which combines attributes of a CPU with those of a very flexible DMA (direct memory access) controller. For example, channels can execute programs like CPUs; the IOP instruction set has about 50 different types of instructions specifically designed for efficient input/output processing. Each channel also can perform high-speed DMA transfers; a variety of optional operations allow the data to be manipulated (e.g., translated or searched) as it is transferred. The 8089 is contained in a 40-pin dual in-line package (figure 3-1) and operates from a single +5V power source. An integral member of the 8086 family, the IOP is directly compatible with both the 8086 and 8088 when these processors are configured in maximum mode. The IOP also may be used in any system that incorporates Intel's Multibus™ shared bus architecture, or a superset of the Multibus™ design.

Evolution

Figure 3-2 depicts the general trend in CPU and I/O device relationships in the first three generations of microprocessors. First generation CPUs were forced to deal directly with substantial numbers of TTL components, often performing transfers at the bit level. Only a very limited number of relatively slow devices could be supported.

Single-chip interface controllers were introduced in the second generation. These devices removed the lowest level of device control from the CPU and let the CPU transfer whole bytes at once. With the introduction of DMA controllers, high-speed devices could be added to a system, and whole blocks of data could be transferred without CPU intervention. Compared to the previous generation, I/O device and DMA controllers allowed microprocessors to be applied to problems that required moderate levels of I/O, both in terms of the numbers of devices that could be supported and the transfer speeds of those devices.

The controllers themselves, however, still required a considerable amount of attention from the CPU, and in many cases the CPU had to respond to an interrupt with every byte read or written. The CPU also had to stop while DMA transfers were performed.

The 8089 introduces the third generation of input/output processing. It continues the trend of simplifying the CPU's "view" of I/O devices by removing another level of control from the CPU. The CPU performs an I/O operation by building a message in memory that describes the function to be performed; the IOP reads the message, carries out the operation and notifies the CPU when it has finished. All I/O devices appear to the CPU as transmitting and receiving whole blocks of data; the IOP can make both byte- and word-level transfers invisible to the CPU. The IOP assumes all device controller overhead, performs both programmed and DMA transfers, and can recover from "soft" I/O errors without CPU intervention; all of these activities may be performed while the CPU is attending to other tasks.

Principles of Operation

Since the 8089 is a new concept in microprocessor components, this section surveys the basic operation of the IOP as background to the detailed descriptions provided in the rest of the chapter. This summary deliberately omits some operating details in order to provide an integrated overview of basic concepts.

CPU/IOP Communications

A CPU communicates with an IOP in two distinct modes: initialization and command. The initialization sequence is typically performed when the system is powered-up or reset. The CPU initializes the IOP by preparing a series of linked message blocks in memory. On a signal from the CPU, the IOP reads these blocks and determines from them how the data buses are configured and how access to the buses is to be controlled.

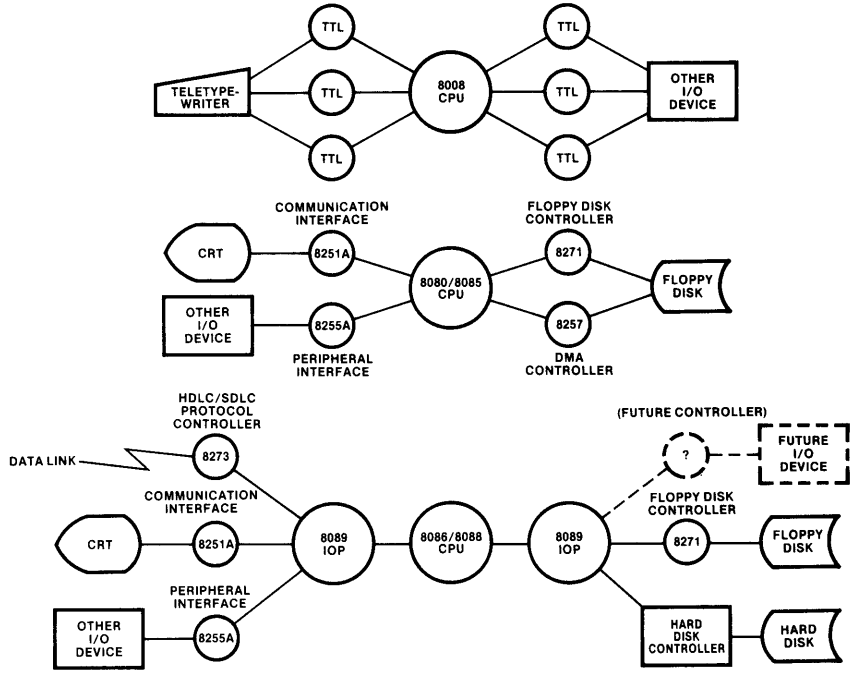


Figure 3-2. IOP Evolution

Following initialization, the CPU directs all communications to either of the IOP's two channels; indeed, during normal operation the IOP appears to be two separate devices—channel 1 and channel 2. All CPU-to-channel communications center on the channel control block (CB) illustrated in figure 3-3. The CB is located in the CPU's memory space, and its address is passed to the IOP during initialization. Half of the block is dedicated to each channel. The channel maintains the BUSY flag that indicates whether it is in the midst of an operation or is available for a new command. The CPU sets the CCW (channel command word) to indicate what kind of operation the IOP is to perform. Six different commands allow the CPU to start and stop programs, remove interrupt requests, etc.

If the CPU is dispatching a channel to run a program, it directs the channel to a parameter block (PB) and a task block (TB); these are also shown in figure 3-3. The parameter block is analogous to a parameter list passed by a program to a subroutine; it contains variable data that the channel program is to use in carrying out its assignment. The parameter block also may contain space for variables (results) that the channel is to return to the CPU. Except for the first two words, the format and size of a parameter block are completely open; the PB may be set up to exchange any kind of information between the CPU and the channel program.

A task block is a channel program—a sequence of 8089 instructions that will perform an operation. A typical channel program might use parameter block data to set up the IOP and a device controller for a transfer, perform the transfer, return the results, and then halt. However, there are no restrictions on what a channel program can do; its function may be simple or elaborate to suit the needs of the application.

Before the CPU starts a channel program, it links the program (TB) to the parameter block and the parameter block to the CB as shown in figure 3-3. The links are standard 8086/8088 doubleword pointer variables; the lower-addressed word contains an offset, and the higher-addressed word contains a segment base value. A system may have many different parameter and task blocks; however, only one of each is ever linked to a channel at any given time.

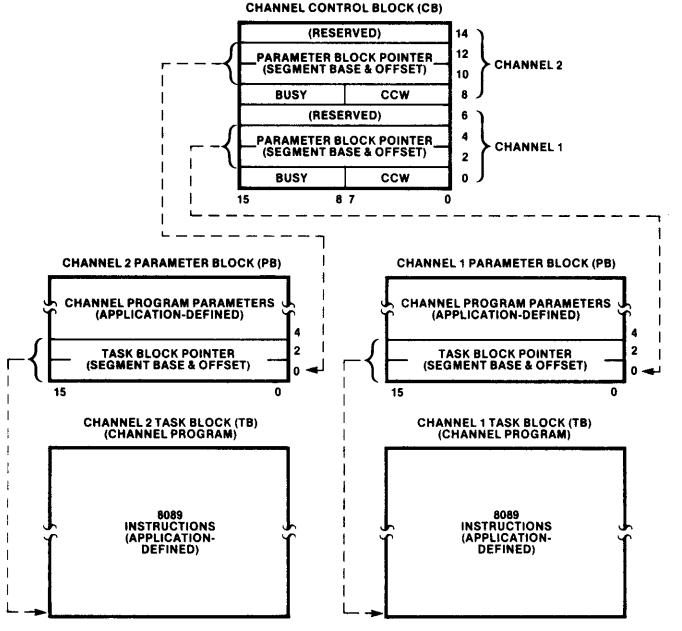


Figure 3-3. Command Communication Blocks

After the CPU has filled in the CCW and has linked the CB to a parameter block and a task block, if appropriate, it issues a channel attention (CA). This is done by activating the IOP's CA (channel attention) and SEL (channel select) pins. The state of SEL at the falling edge of CA directs the channel attention to channel 1 or channel 2. If the IOP is located in the CPU's I/O space, it appears to the CPU as two consecutive I/O ports (one for each channel), and an OUT instruction to the port functions as a CA. If the IOP is memory-mapped, the channels appear as two consecutive memory locations, and any memory reference instruction (e.g., MOV) to these locations causes a channel attention.

An IOP channel attention is functionally similar to a CPU interrupt. When the channel recognizes the CA, it stops what it is doing (it will typically be idle) and examines the command in the CCW. If it is to start a program, the channel loads the addresses of the parameter and task blocks into internal registers, sets its BUSY flag and starts executing the channel program. After it has issued the CA, the CPU is free to perform other processing; the channel can perform its function in parallel, subject to limitations imposed by bus configurations (discussed shortly).

When the channel has completed its program, it notifies the CPU by clearing its BUSY flag in the CB. Optionally, it may issue an interrupt request to the CPU.

The CPU/IOP communication structure is summarized in figure 3-4. Most communication takes place via "message areas" shared in common memory. The only direct hardware communications between the devices are channel attentions and interrupt requests.

Channels

Each of the two IOP channels operates independently, and each has its own register set, channel attention, interrupt request and DMA control signals. At a given point in time, a channel may be idle, executing a program, performing a DMA transfer, or responding to a channel attention. Although only one channel actually runs at a time, the channels can be active concurrently, alternating their operations (e.g., channel 1 may execute instructions in the periods between successive DMA transfer cycles run by channel 2). A built-in priority system allows high-priority activities on one channel to preempt less critical operations on the other channel. The CPU is able to further adjust priorities to handle special cases. The CPU starts the channel and can halt it, suspend it, or cause it to resume a suspended operation by placing different values in the CCW.

Channel Programs (Task Blocks)

Channel programs are written in ASM-89, the 8089 assembly language. About 50 basic instructions are available. These instructions operate on bit, byte, word and doubleword (pointer) variable types; a 20-bit physical address variable type (not used by the 8086/8088) can also be manipulated. Data may be taken from registers, immediate constants and memory. Four memory addressing modes allow flexible access to both memory variables and I/O devices located anywhere in either the CPU's megabyte memory space or in the 8089's 64k I/O space.

The IOP instruction set contains general purpose instructions similar to those found in CPUs as well as instructions specifically tailored for I/O

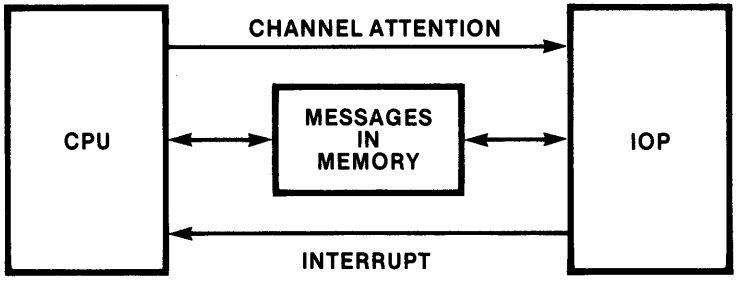


Figure 3-4. CPU/IOP Communication

operations. Data transfer, simple arithmetic, logical and address manipulation operations are available. Unconditional jump and call instructions also are provided so that channel programs can link to each other. An individual bit may be set or cleared with a single instruction. Conditional jumps can test a bit and jump if it is set (or cleared), or can test a value and jump if it is zero (or non-zero). Other instructions initiate DMA transfers, perform a locked test-and-set semaphore operation, and issue an interrupt request to the CPU.

DMA Transfers

The 8089 XFER (transfer) instruction prepares the channel for a DMA transfer. It executes one additional instruction, then suspends program execution and enters the DMA transfer mode. The transfer is governed by channel registers setup by the program prior to executing the XFER instruction.

Data is transferred from a source to a destination. The source and destination may be any locations in the CPU's memory space or in the IOP's I/O space; the IOP makes no distinction between memory components and I/O devices. Thus transfers may be made from I/O device to memory, memory to I/O device, memory to memory and I/O device to I/O device. The IOP automatically matches 8- and 16-bit components to each other.

Individual transfer cycles (i.e., the movement of a byte or a word) may be synchronized by a signal (DMA request) from the source or from the destination. In the synchronized mode, the channel waits for the synchronizing signal before starting the next transfer cycle. The transfer also may be unsynchronized, in which case the channel begins the next transfer cycle immediately upon completion of the previous cycle.

A transfer cycle is performed in two steps: fetching a byte or word from the source into the IOP and then storing it from the IOP into the destination. The IOP automatically optimizes the transfer to make best use of the available data bus widths. For example, if data is being transferred from an 8-bit device to memory that resides on a 16-bit bus (e.g., 8086 memory), the IOP will normally run two one-byte fetch cycles and then store the full word in a single cycle.

Between the fetch and store cycles, the IOP can operate on the data. A byte may be translated to another code (e.g., EBCDIC to ASCII), or compared to a search value, or both, if desired.

A transfer can be terminated by several programmer-specified conditions. The channel can stop the transfer when a specified number (up to 64k) of bytes has been transferred. An external device may stop a transfer by signaling on the channel's external terminate pin. The channel can stop the transfer when a byte (possibly translated) compares equal, or unequal, to a search value. Single-cycle termination, which stops unconditionally after one byte or word has been stored, is also available.

When the transfer terminates, the channel automatically resumes program execution. The channel program can determine the cause of the termination in situations where multiple terminations are possible (e.g., terminating when 80 bytes are transferred or a carriage return character is encountered, whichever occurs first). As an example of post-transfer processing, the channel program could read a result register from the I/O device controller to determine if the transfer was performed successfully. If not (e.g., a CRC error was detected by the controller), the channel program could retry the operation without CPU intervention.

A channel program typically ends by posting the result of the operation to a field supplied in the parameter block, optionally interrupting the CPU, and then halting. When the channel halts, its BUSY flag in the channel control block is cleared to indicate its availability for another operation. As an alternative to being interrupted by the channel, the CPU can poll this flag to determine when the operation has been completed.

Bus Configurations

As shown in figure 3-5, the IOP can access memory or ports (I/O devices) located in a 1-megabyte system space and memory or ports located in a 64-kilobyte I/O space. Although the IOP only has one physical data bus, it is useful to think of the IOP as accessing the system space via a system data bus and the I/O space over an I/O data bus. The distinction between the "two" buses is based on the type-of-cycle signals output

by the 8288 Bus Controller. Components in the system space respond to the memory read and memory write signals, whether they are memory or I/O devices. Components in the I/O space respond to the I/O read and I/O write signals. Thus I/O devices located in the system space are memory-mapped and memory in the I/O space is I/O-mapped. The two basic configuration options differ in the degree to which the IOP shares these buses with the CPU. Both configurations require an 8086/8088 CPU to be strapped in maximum mode.

In the local configuration, shown in figure 3-6, the IOP (or IOPs if two are used) shares both buses with the CPU. The system bus and the I/O bus are the same width (8 bits if the CPU is an 8088 or 16 bits if the CPU is an 8086). The IOP system space corresponds to the CPU memory space, and the IOP I/O space corresponds to the CPU I/O space. Channel programs are located in the system space; I/O devices may be located in either space. The IOP requests use of the bus for channel program instruction fetches as well as for DMA and programmed transfers. In the local configuration, either the IOP or the CPU may use the buses, but not both simultaneously. The advantage of the local configuration is that intelligent DMA may be added to a system with no additional components beyond the IOP. The disadvantage is that parallel operation of the processors is limited to cases in which the CPU has instruction in its queue that can be executed without using the bus.

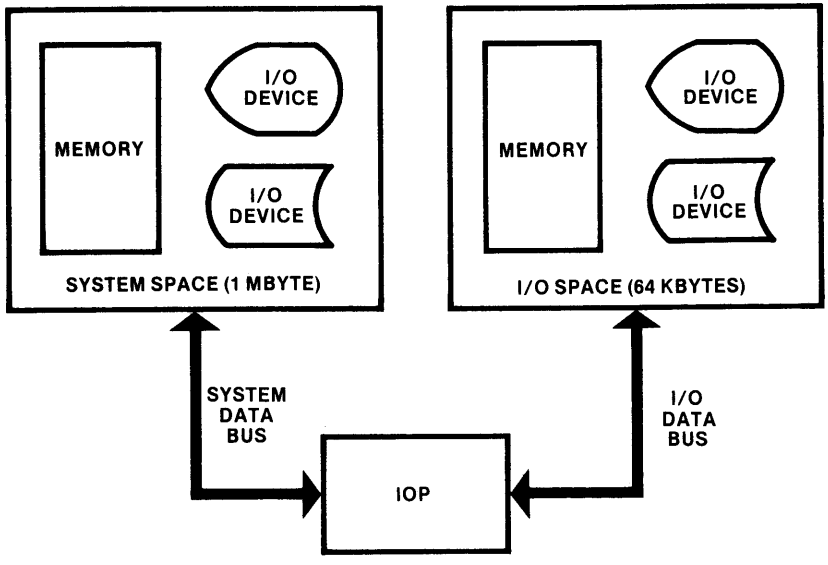


Figure 3-5. IOP Data Buses

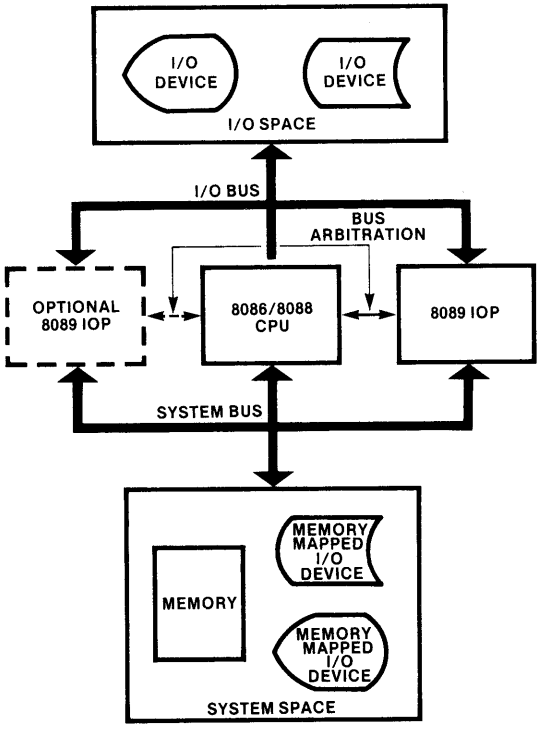


Figure 3-6. Local Configuration

In the remote configuration (figure 3-7), the IOP (or IOPs) shares a common system bus with the CPU. Access to this bus is controlled by 8289 Bus Arbiters. The IOP's I/O bus, however, is physically separated from the CPU in the remote configuration. Two IOPs can share the local I/O bus. Any number of remote IOPs may be contained in a system, configured in remote clusters of one or two. The local I/O bus need not be the same physical width as the shared system bus, allowing an IOP, for example, to interface 8-bit peripherals to an 8086. In the remote configuration, the IOP can access local I/O devices and memory without using the shared system bus, thereby reducing bus contention with the CPU. Contention can further be reduced by locating the IOP's channel programs in the local I/O space. The IOP can then also fetch instructions without accessing the system bus. Parameter, channel control and other CPU/IOP communication blocks must be located in system memory, however, so that both processors can access them. The remote configuration thus increases the degree to which an IOP and a CPU can operate in parallel and thereby increases a system's throughput potential. The price paid for this is that additional hardware must be added to arbitrate use of the shared bus, and to separate the shared and local buses (see Chapter 4 for details).

It is also possible to configure an IOP remote to one CPU, and local to another CPU (see figure 3-8). The local CPU could be used to perform heavy computational routines for the IOP.

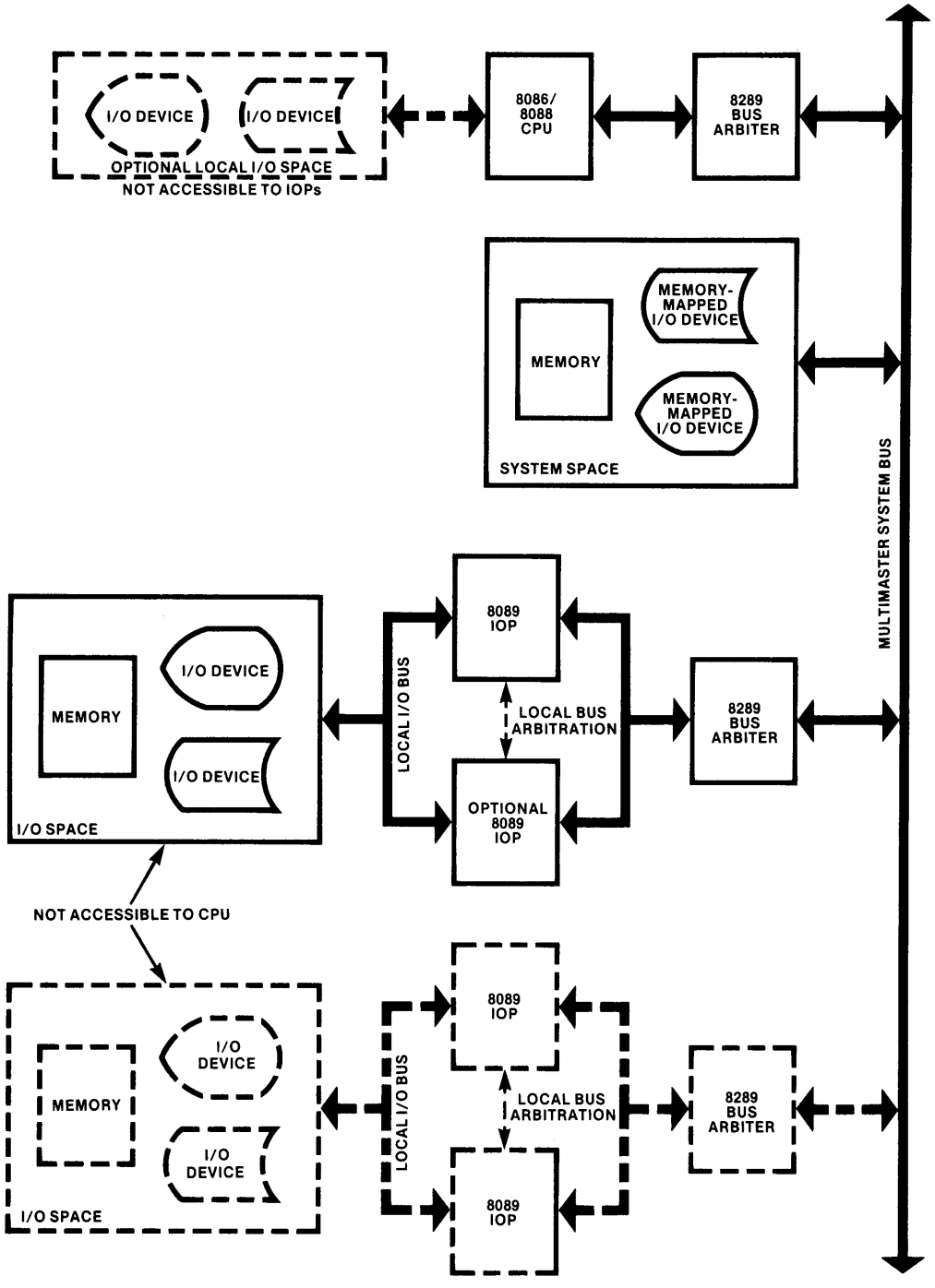


Figure 3-7. Remote Configuration

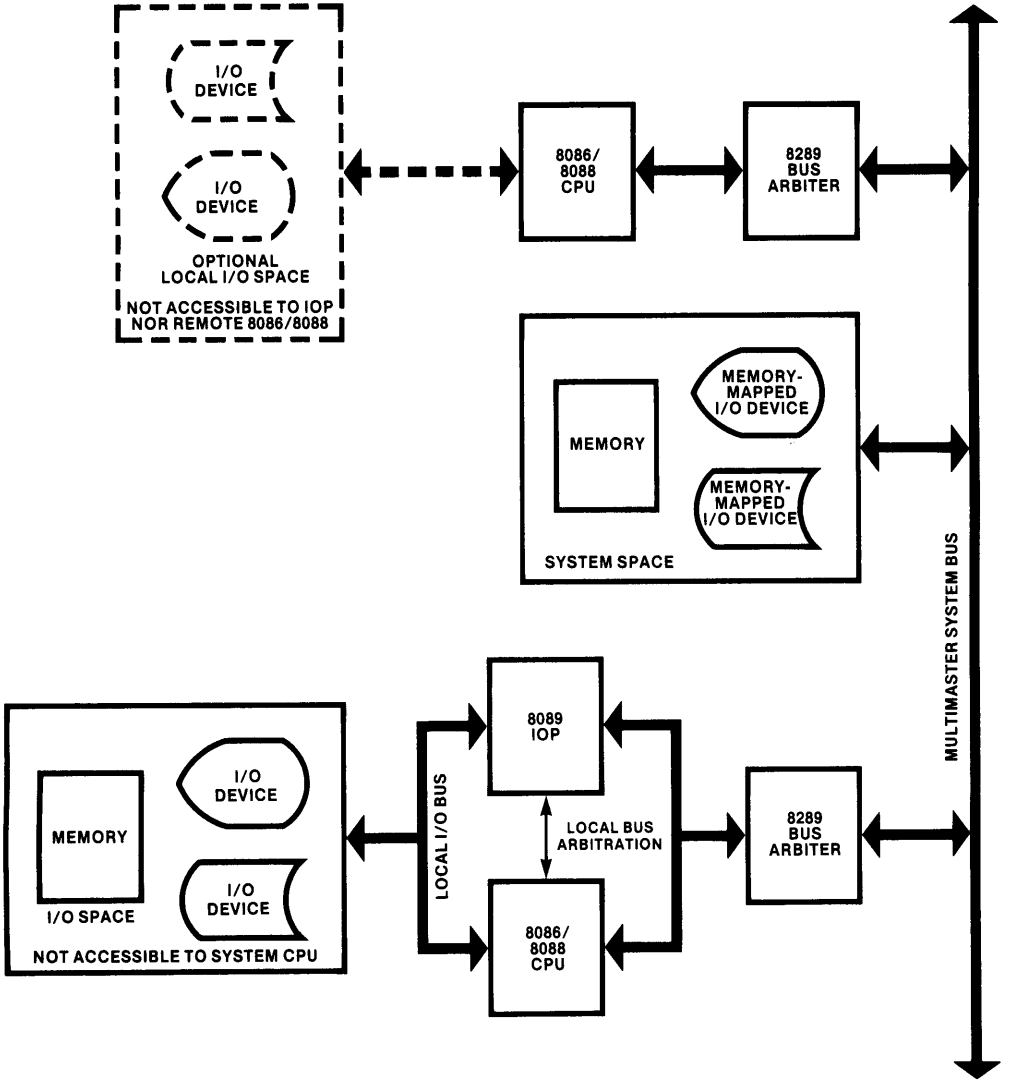


Figure 3-8. Remote IOP Configured With Local 8086/8088

A Sample Transaction

Figure 3-9 shows how a CPU and an IOP might work together to read a record (sector) from a floppy disk. This example is not illustrative of the IOP's full capabilities, but it does review its basic operation and its interaction with a CPU.

The CPU must first obtain exclusive use of a channel. This can be done by performing a "test and set lock" operation on the selected channel's BUSY flag. Assuming the CPU wants to use channel 1, this could be accomplished in PL/M-86 by coding similar to the following:

```
DO WHILE LOCKSET (@CH1.BUSY,0FFH);
    END;
```

In ASM-86 a loop containing the XCHG instruction prefixed by LOCK would accomplish the same thing, namely testing the BUSY flag until it is clear (0H), and immediately setting it to FFH (busy) to prevent another task or processor from obtaining use of the channel.

Having obtained the channel, the CPU fills in a parameter block (see figure 3-10). In this case, the CPU passes the following parameters to the channel: the address of the floppy disk controller, the address of the buffer where the data is to be placed, and the drive, track and sector to be read. It also supplies space for the IOP to return the result of the operation. Note that this is quite a "low-level" parameter block in that it implies that the CPU has detailed knowledge of the I/O system. For a "real" system, a higher-level parameter block would isolate the CPU from I/O device characteristics. Such a block might contain more general parameters such as file name and record key.

After setting up the parameter block, the CPU writes a "start channel program" command in channel 1's CCW. Then the CPU places the address of the desired channel program in the parameter block and writes the parameter block address in the CB. Notice that in this simple example, the CPU "knows" the address of the channel program for reading from the disk, and presumably also "knows" the address of another program for writing, etc. A more general solution would be to place a function code (read, write, delete, etc.) in the parameter block and let a single channel program execute different routines depending on which function is requested.

After the communication blocks have been setup, the CPU dispatches the channel by issuing a channel attention, typically by an OUT instruction for an I/O-mapped 8089, or a MOV or other memory reference instruction for a memory-mapped 8089.

The channel begins executing the channel program (task block) whose address has been placed in the parameter block by the CPU. In this case the program initializes the 8271 Floppy Disk Controller by sending it a "read data" command followed by a parameter indicating the track to be read. The program initializes the channel registers that define and control the DMA transfer.

Having prepared the 8271 and the channel itself, the channel program executes a XFER instruction and sends a final parameter (the sector to be read) to the 8271. (The 8271 enters DMA transfer mode immediately upon receiving the last of a series of parameters; sending the last parameter after the XFER instruction gives the channel time to setup for the transfer.) The DMA transfer begins when the 8271 issues a DMA request to the channel. The transfer continues until the 8271 issues an interrupt request, indicating that the data has been transferred or that an error has occurred. The 8271's interrupt request line is tied to the IOP's EXT1 (external terminate on channel 1) pin so that the channel interprets an interrupt request as an external terminate condition. Upon termination of the transfer, the channel resumes executing instructions and reads the 8271 result register to determine if the data was read successfully. If a soft (correctable) error is indicated, the IOP retries the transfer. If a hard (uncorrectable) error is detected, or if the transfer has been successful, the IOP posts the content of the result register to the parameter block result field, thus passing the result back to the CPU. The channel then interrupts the CPU (to inform the CPU that the request has been processed) and halts.

When the CPU recognizes the interrupt, it inspects the result field in the parameter block to see if the content of the buffer is valid. If so, it uses the data; otherwise it typically executes an error routine.

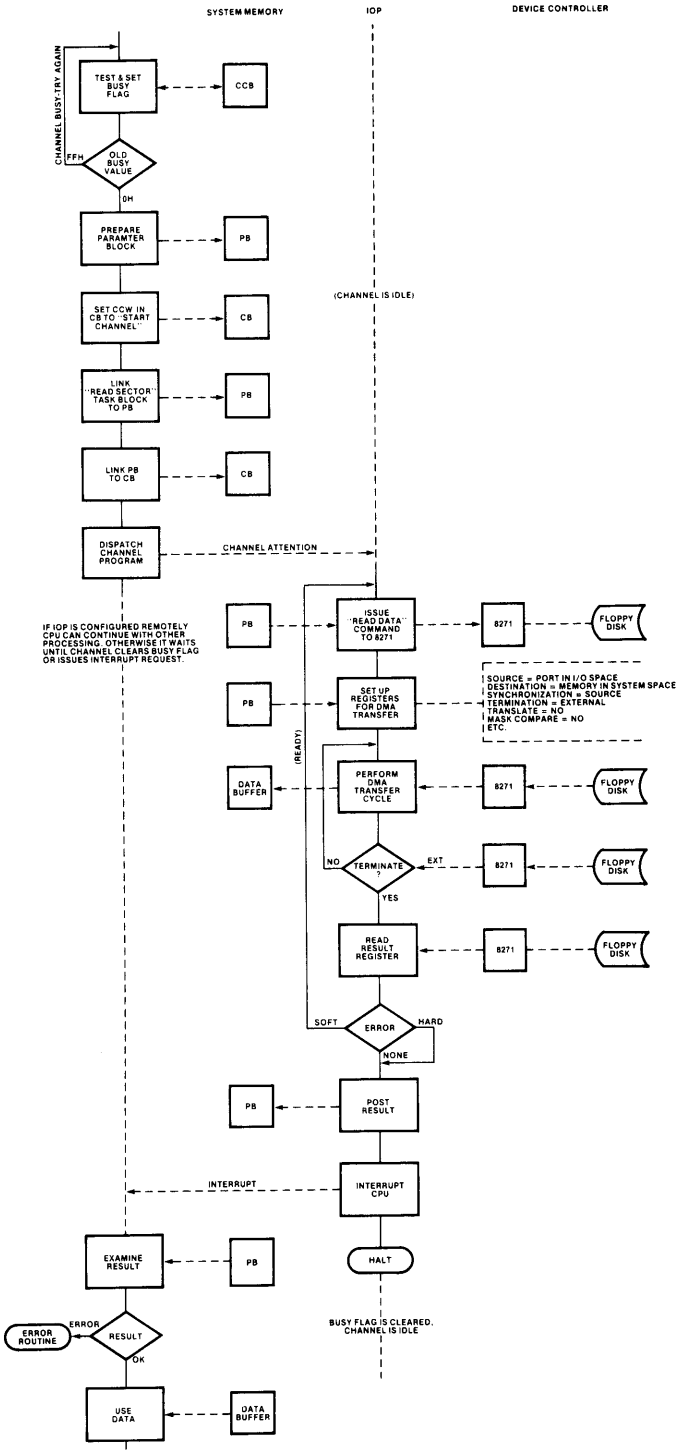


Figure 3-9. Sample CPU/IOP Transaction

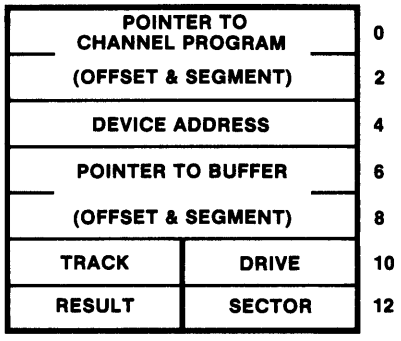


Figure 3-10. Sample Parameter Block

Applications

Combining the raw speed and responsiveness of a traditional DMA controller, an I/O-oriented instruction set, and a flexible bus organization, the 8089 IOP is a very versatile I/O system. Applications with demanding I/O requirements, previously beyond the abilities of microcomputer systems, can be undertaken with the IOP. These kinds of I/O-intensive applications include:

- systems that employ high-bandwidth, low-latency devices such as hard disks and graphics terminals;
- systems with many devices requiring asynchronous service; and
- systems with high-overhead peripherals such as intelligent CRTs and graphics terminals.

In addition, virtually every application that performs a moderate amount of I/O can benefit from the design philosophy embodied in the IOP: system functions should be distributed among special-purpose processors. An IOP channel program is likely to be both faster and smaller than an equivalent program implemented with a CPU. Programming also is more straightforward with the IOP's specialized instruction set.

Removing I/O from the CPU and assigning it to one or more IOPs simplifies and structures a system's design. The main interface to the I/O system can be limited to the parameter blocks. Once these are defined, the I/O system can be designed and implemented in parallel with the rest of the system. I/O specialists can work on the I/O system without detailed knowledge of the application; conversely, the operating system and application teams do not need to be expert in the operation of I/O devices. Standard high-level I/O systems can be used in multiple application systems. Because the application and I/O systems are almost independent, application system changes can be introduced without affecting the I/O system. New peripherals can similarly be incorporated into a system without impacting applications or operating system software. The IOP's simple CPU interface also is designed to be compatible with future Intel CPUs.

Keeping in mind the true general-purpose nature of the IOP, some of the situations where it can be used to advantage are:

- Bus matching - The IOP can transfer data between virtually any combination of 8- and 16-bit memory and I/O components. For example, it can interface a 16-bit peripheral to an 8-bit CPU bus, such as the 8088 bus. The IOP also provides a straightforward means of performing DMA between an 8-bit peripheral and 8086 memory that is split into odd- and even-addressed banks. The 8089 can access both 8- and 16-bit peripherals connected to a 16-bit bus.
- String processing - The 8089 can perform a memory move, translate, scan-for-match or scan-for-nonmatch operation much faster than the equivalent instructions in an 8086 or 8088. Translate and scan operations can be setup so that the source and destination refer to the same addresses to permit the string to be operated on in place.
- Spooling - Data from low-speed devices such as terminals and paper tape readers can be read by the 8089 and placed in memory or on disk until the transmission is complete. The IOP can then transfer the data at high speed when it is needed by an application program. Conversely, output data ultimately destined for a low-speed device such as a printer, can be temporarily spooled to disk and then printed later. This permits batches of data to be gathered or distributed by low-priority programs that run in the background, essentially using up "spare" CPU and IOP cycles. Application programs that use or produce the data can execute faster because they are not bound by the low-speed devices.

- Multitasking operating systems - A multitasking operating system can dispatch I/O tasks to channels with an absolute minimum of overhead. Because a remote channel can run in parallel with the CPU, the operating system's capacity for servicing application tasks can increase dramatically, as can its ability to handle more, and faster, I/O devices. If both channels of an IOP are active concurrently, the IOP automatically gives preference to the higher-priority activity (e.g., DMA normally preempts channel program execution). The operating system can adjust the priority mechanism and also can halt or suspend a channel to take care of a critical asynchronous event.

- Disk systems - The IOP can meet the speed and latency requirements of hard disks. It can be used to implement high-level, file-oriented systems that appear to application programs as simple commands: OPEN, READ, WRITE, etc. The IOP can search and update disk directories and maintain free space maps. "Hierarchical memory" systems that automatically transfer data among memory, high-speed disks and low-speed disks, based on frequency of use, can be built around IOPs. Complex database searches (reading data directly or following pointer chains) can appear to programs as simple commands and can execute in parallel with application programs if an IOP is configured remotely.

- Display terminals - The 8089 is well suited to handling the DMA requirements of CRT controllers. The IOP's transfer bandwidth is high enough to support both alphanumeric and graphic displays. The 8089 can assume responsibility for refreshing the display from memory data; in the remote configuration, the refresh overhead can be removed from the system bus entirely. Linked-list display algorithms may be programmed to perform sophisticated modes of display.

  Each time it performs a refresh operation, the IOP can scan a keyboard for input and translate the key's row-and-column format into an ASCII or EBCDIC character. The 8089 can buffer the characters, scanning the stream until an end-of-message character (e.g., carriage return) is detected, and then interrupt the CPU.

  A single IOP can concurrently support an alphanumeric CRT and keyboard on one channel and a floppy disk on the other channel. This configuration makes use of approximately 30 percent of the available bus bandwidth. Performance can be increased within the available bus bandwidth by adding an 8086 or 8088 CPU to a remote IOP configuration. This configuration can provide scaling, rotation or other sophisticated display transformations.

3.2 Processor Architecture

The 8089 is internally divided into the functional units depicted schematically in figure 3-11. The units are connected by a 20-bit data path to obtain maximum internal transfer rates.

Common Control Unit (CCU)

All IOP operations (instructions, DMA transfer cycles, channel attention responses, etc.) are composed of sequences of more basic processes called internal cycles. A bus cycle takes one internal cycle; the execution of an instruction may require several internal cycles. There are 23 different types of internal cycles each of which takes from two to eight clocks to execute, not including possible wait states and bus arbitration times.

The common control unit (CCU) coordinates the activities of the IOP primarily by allocating internal cycles to the various processor units; i.e., it determines which unit will execute the next internal cycle. For example, when both channels are active, the CCU determines which channel has priority and lets that channel run; if the channels have equal priority, the CCU "interleaves" their execution (this is discussed more fully later in this section). The CCU also initializes the processor.

Arithmetic/Logic Unit (ALU)

The ALU can perform unsigned binary arithmetic on 8- and 16-bit binary numbers. Arithmetic results may be up to 20 bits in length. Available arithmetic instructions include addition, increment and decrement. Logical operations ("and," "or" and "not") may be performed on either 8- or 16-bit quantities.

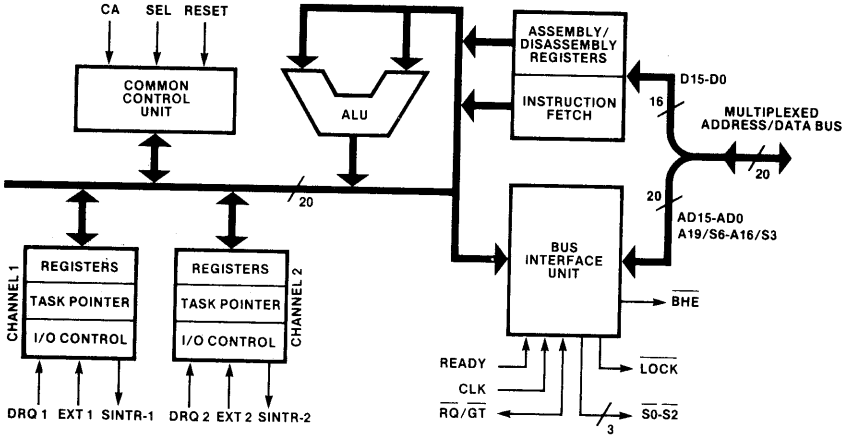


Figure 3-11. 8089 Block Diagram

Assembly/Disassembly Registers

All data entering the chip flows through these registers. When data is being transferred between different width buses, the 8089 uses the assembly/disassembly registers to effect the transfer in the fewest possible bus cycles. In a DMA transfer from an 8-bit peripheral to 16-bit memory, for example, the IOP runs two bus cycles, picking up eight bits in each cycle, assembles a 16-bit word, and then transfers the word to memory in a single bus cycle. (The first and last cycles of a transfer may be performed differently to accommodate odd-addressed words; the IOP automatically adjusts for this condition.)

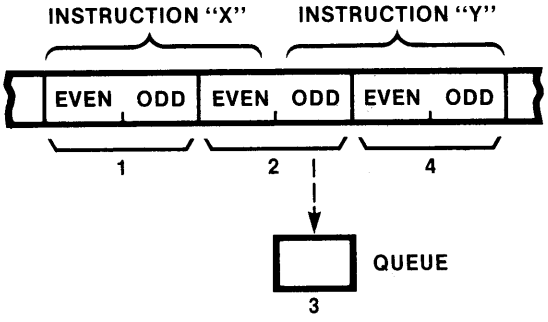
Instruction Fetch Unit

This unit controls instruction fetching for the executing channel (one channel actually runs at a time). If the bus over which the instructions are being fetched is eight bits wide, then the instructions are obtained one byte at a time, and each fetch requires one bus cycle. If the instructions are being fetched over a 16-bit bus, then the instruction fetch unit automatically employs a 1-byte queue to reduce the number of bus cycles. Each channel has its own queue, and the activity of one channel does not affect the other's queue.

During sequential execution, instructions are fetched one word at a time from even addresses; each fetch requires one bus cycle. This process is shown graphically in figure 3-12. When the last byte of an instruction falls on an even address, the odd-addressed byte (the first byte of the following instruction) of the fetched word is saved in the queue. When the channel begins execution of the next instruction, it fetches the first byte from the queue rather than from memory. The queue, then, keeps the processor fetching words, rather than bytes, thereby reducing its use of the bus and increasing throughput.

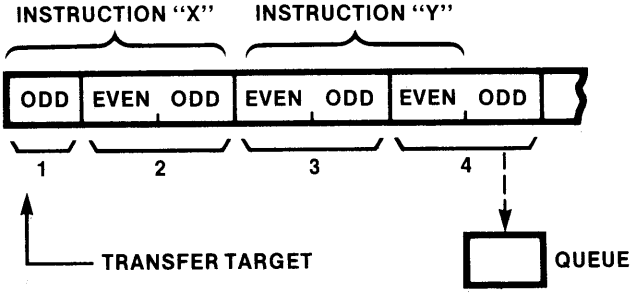
The processor fetches bytes rather than words in two cases. If a program transfer instruction (e.g., JMP or CALL) directs the processor to an instruction located at an odd address, the first byte of the instruction is fetched by itself as shown in figure 3-13. This is because the program transfer invalidates the content of the queue by changing the serial flow of execution.

The second case arises when an LPDI instruction is located at an odd address. In this situation, the six-byte LPDI instruction is fetched: byte, word, byte, byte, byte, and the queue is not used. The first byte of the following instruction is fetched in one bus cycle as if it had been the target of a program transfer. Word fetching resumes with this instruction's second byte.



FETCH	INSTRUCTION BYTES
1	FIRST TWO BYTES OF "X"
2	THIRD BYTE OF "X" PLUS FIRST BYTE OF "Y", WHICH IS SAVED IN QUEUE
3	FIRST BYTE OF "Y" FROM QUEUE—NO BUS CYCLE
4	LAST TWO BYTES OF "Y"

Figure 3-12. Sequential Instruction Fetching (16-Bit Bus)



FETCH	INSTRUCTION BYTES
1	FIRST (ODD-ADDRESSED) BYTE OF "X" (8-BIT BUS CYCLE)
2	SECOND AND THIRD BYTES OF "X"
3	FIRST AND SECOND BYTES OF "Y".
4	THIRD BYTE OF "Y" PLUS FIRST BYTE OF NEXT INSTRUCTION, WHICH IS SAVED IN QUEUE

Figure 3-13. Instruction Fetching Following a Program Transfer to an Odd Address (16-Bit Bus)

Bus Interface Unit (BIU)

The BIU runs all bus cycles, transferring instructions and data between the IOP and external memory or peripherals. Every bus access is associated with a register tag bit that indicates to the BIU whether the system or I/O space is to be addressed. The BIU outputs the type of bus cycle (instruction fetch from I/O space, data store into system space, etc.) on status lines $\overline{S0}$, $\overline{S1}$, and $\overline{S2}$. An 8288 Bus Controller decodes these lines and provides signals that selectively enable one bus or the other (see Chapter 4 for details).

The BIU further distinguishes between the physical and logical widths of the system and I/O buses. The physical widths of the buses are fixed and are communicated to the BIU during initialization. In the local configuration, both buses must be the same width, either 8 or 16 bits (matching the width of the host CPU bus). In the remote configuration, the IOP system bus must be the same physical width as the bus it shares with the CPU. The width of the IOP's I/O bus, which is local to the 8089, may be selected independently. If any 16-bit peripherals are located in the I/O space, then a 16-bit I/O bus must be used. If only 8-bit devices reside on the I/O bus, then either an 8- or a 16-bit I/O bus may be selected. A 16-bit I/O bus has the advantage of easy accommodation of future 16-bit devices and fewer instruction fetches if channel programs are placed in the I/O space.

For a given DMA transfer, a channel program specifies the logical width of the system and the I/O buses; each channel specifies logical bus widths independently. The logical width of an 8-bit physical bus can only be eight bits. A 16-bit physical bus, however, can be used as either an 8- or 16-bit logical bus. This allows both 8- and 16-bit devices to be accessed over a single 16-bit physical bus. Table 3-1 lists the permissible physical and logical bus widths for both locally and remotely configured IOPs. Logical bus width pertains to DMA transfers only. Instructions are fetched and operands are read and written in bytes or words depending on physical bus width.

In addition to performing transfers, the BIU is responsible for local bus arbitration. In the local configuration, the BIU uses the $\overline{RQ}/\overline{GT}$ (request/grant) line to obtain the bus from the CPU and to return it after a transfer has been performed. In the remote configuration, the BIU uses $\overline{RQ}/\overline{GT}$ to coordinate use of the local I/O bus with another IOP or a local CPU, if present. System bus arbitration in the remote configuration is performed by an 8289 Bus Arbiter that operates invisibly to the IOP. The BIU automatically asserts the $\overline{LOCK}$ (bus lock) signal during execution of a TSL (test and set lock) instruction and, if specified by the channel program, can assert the $\overline{LOCK}$ signal for the duration of a DMA transfer. Section 3.5 contains a complete discussion of bus arbitration.

Table 3-1. Physical/Logical Bus Combinations

Configuration	System Bus Physical:Logical	I/O Bus Physical:Logical
Local	8:8 16:8/16	8:8 16:8/16
Remote	8:8 16:8/16 16:8/16 8:8	8:8 16:8/16 8:8 16:8/16

Channels

Although the 8089 is a single processor, under most circumstances it is useful to think of it as two independent channels. A channel may perform DMA transfers and may execute channel programs; it also may be idle. This section describes the hardware features that support these operations.

I/O Control

Each channel contains its own I/O control section that governs the operation of the channel during DMA transfers. If the transfer is synchronized, the channel waits for a signal on its DRQ (DMA request) line before performing the next fetch-store sequence in the transfer. If the transfer is to be terminated by an external signal, the channel monitors its EXT (external terminate) line and stops the transfer when this line goes active. Between the fetch and store cycles (when the data is in the IOP) the channel optionally counts,

translates, and scans the data, and may terminate the transfer based on the results of these operations. Each channel also has a SINTR (system interrupt) line that can be activated by software to issue an interrupt request to the CPU.

Registers

Figure 3-14 illustrates the channel register set, and table 3-2 summarizes the uses of each register. Each channel has an independent set of registers; they are not accessible to the other channel. Most of the registers play different roles during channel program execution than in DMA transfers. Channel programs must be careful to save these registers in memory prior to a DMA transfer if their values are needed following the transfer.

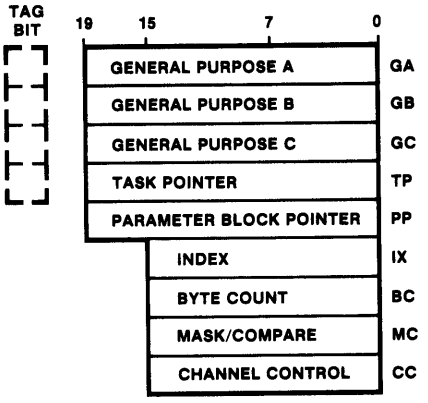


Figure 3-14. Channel Register Set

General Purpose A (GA). A channel program may use GA for a general register or a base register. A general register can be an operand of most IOP instructions; a base register is used to address memory operands (see section 3.8). Before initiating a DMA transfer, the channel program points GA to either the source or destination address of the transfer.

General Purpose B (GB). GB is functionally interchangeable with GA. If GA points to the source of a DMA transfer, then GB points to the destination, and vice versa.

General Purpose C (GC). GC may be used as a general register or a base register during channel program execution. If data is to be translated during a DMA transfer, then the channel program loads GC with the address of the first byte of a translation table before initiating the transfer. GC is not altered by a transfer operation.

Task Pointer (TP). The CCU loads TP from the parameter block when it starts or resumes a channel program. During program execution, the channel automatically updates TP to point to the

Table 3-2. Channel Register Summary

Register	Size	Program Access	System or I/O Pointer	Use by Channel Programs	Use in DMA Transfers
GA	20	Update	Either	General, base	Source/destination pointer
GB	20	Update	Either	General, base	Source/destination pointer
GC	20	Update	Either	General, base	Translate table pointer
TP	20	Update	Either	Procedure return, instruction pointer	Adjusted to reflect cause of termination
PP	20	Reference	System	Base	N/A
IX	16	Update	N/A	General, auto-increment	N/A
BC	16	Update	N/A	General	Byte counter
MC	16	Update	N/A	General, masked compare	Masked compare
CC	16	Update	N/A	Restricted use recommended	Defines transfer options

next instruction to be executed; i.e., TP is used as an instruction pointer or program counter. Program transfer instructions (JMP, CALL, etc.) update TP to cause nonsequential execution. A procedure (subroutine) returns to the calling program by loading TP with an address previously saved by the CALL instruction. The task pointer is fully accessible to channel programs; it can be used as a general register or as a base register. Such use is not recommended, however, as it can make programs very difficult to understand.

Parameter Block Pointer (PP). The CCU loads this register with the address of the parameter block before it starts a channel program. The register cannot be altered by a channel program, but is very useful as a base register for accessing data in the parameter block. PP is not used during DMA transfers.

Index (IX). IX may be used as a general register during channel program execution. It also may be used as an index register to address memory operands (the address of the operand is computed by adding the content of IX to the content of a base register). When specified as an index register, IX may be optionally auto-incremented as the last step in the instruction to provide a convenient means of "stepping" through arrays or strings. IX is not used in DMA transfers.

Byte Count (BC). BC may be used as a general register during channel program execution. If DMA is to be terminated when a specific number of bytes has been transferred, BC should be loaded with the desired byte count before initiating the transfer. During DMA, BC is decremented for each byte transferred, whether byte count termination has been selected or not. If BC reaches zero, the transfer is stopped only if byte count termination has been specified. If byte count termination has not been selected, BC "wraps around" from 0H to FFFFH and continues to be decremented.

Mask/Compare (MC). A channel program may use MC for a general register. This register also may be used in either a channel program or in a DMA transfer to perform a masked compare of a byte value. To use MC in this way, the program loads a compare value in the low-order eight bits of the register and a mask value in the upper eight bits (see figure 3-15). A "1" in a mask bit *selects* the bit in the corresponding position in the compare value; a "0" in a mask bit *masks* the corresponding bit in the compare value. In figure 3-15, a value compared with MC will be considered equal if its low-order five bits contain the value 00100; the upper three bits may contain any value since they are masked out of the comparison.

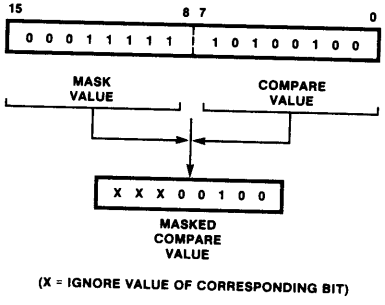


Figure 3-15. Mask/Compare Register

Channel Control (CC). The content of the channel control register governs a DMA transfer (see figure 3-16). A channel program loads this register with appropriate values before beginning the transfer operation; section 3.4 covers the encoding of each field in detail. Bit 8 (the chain bit) of CC pertains to channel program execution rather than to a DMA transfer. When this bit is zero, the channel program runs at normal priority; when it is one, the priority of the program is raised to the same level as DMA (priorities are covered later in this section). Although a channel program may use CC as a general register, such use is not recommended because of the side effects on the chain bit and thus on the priority of the channel program. Channel programs should restrict their use of CC to loading control values in preparation for a DMA transfer, setting and clearing the chain bit, and storing the register.

Program Status Word (PSW)

Each channel maintains its own program status word (PSW) as shown in figure 3-17. Channel programs do not have access to the PSW. The PSW records the state of the the channel so that channel operation may be suspended and then resumed later. When the CPU issues a "suspend" command, the channel saves the PSW, task pointer, and task pointer tag bit in the first four bytes of the channel's parameter block as shown in figure 3-18. Upon receipt of a subsequent

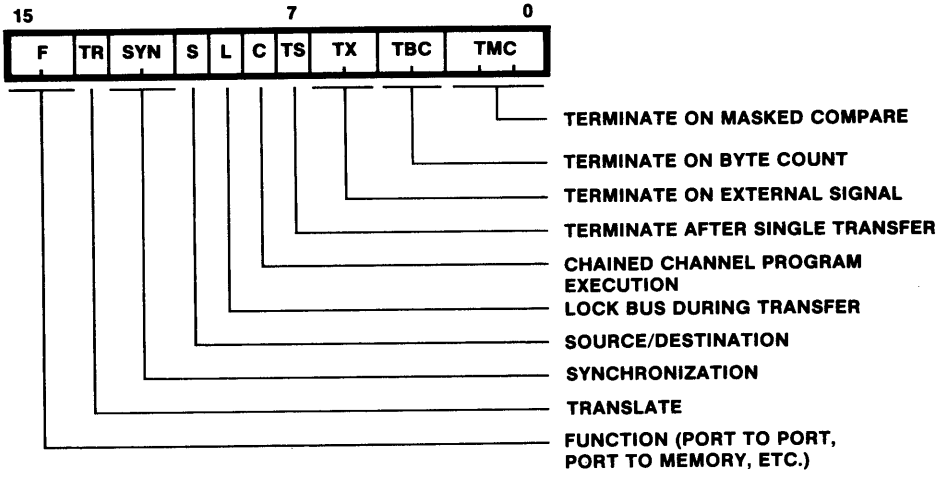


Figure 3-16. Channel Control Register

"resume" command, the PSW, TP, and TP tag bit are restored from the parameter block save area and execution resumes.

Two conditions override the normal channel priority mechanism. If one channel is performing DMA (priority 1) and the channel receives a channel attention (priority 2), the channel attention is serviced at the end of the current DMA transfer cycle. This override prevents a synchronized DMA transfers from "shutting out" a channel attention. DMA terminations and chained channel programs postpone recognition of a CA on the *other* channel; the CA is latched, however, and is serviced as soon as priorities permit.

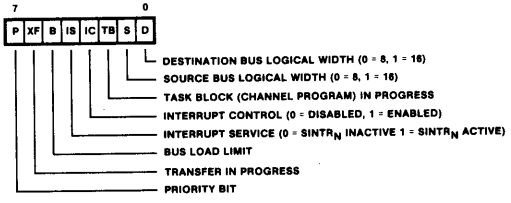


Figure 3-17. Program Status Word

The IOP's $\overline{\text{LOCK}}$ (bus lock) signal also supersedes channel switching. A running channel will not relinquish control of the processor while $\overline{\text{LOCK}}$ is active, regardless of the priorities of the activities on the two channels. This is consistent with the purpose of the $\overline{\text{LOCK}}$ signal: to guarantee exclusive access to a shared resource in a multiprocessing system. Refer to sections 3.5 and 3.7 for futher information on the $\overline{\text{LOCK}}$ signal and the TSL instruction.

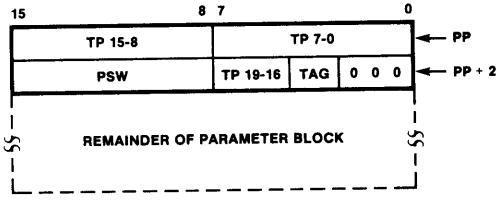


Figure 3-18. Channel State Save Area

Tag Bits

Registers GA, GB, GC, and TP are called pointer registers because they may be used to access, or

point to, addresses in either the system space or the I/O space. The pointer registers may address either memory or I/O devices (IOP instructions do not distinguish between memory and I/O devices since the latter are memory-mapped). The tag bit associated with each register (figure 3-14) determines whether the register points to an address in the system space (tag=0) or the I/O space (tag=1).

The CCU sets or clears TP's tag bit depending on whether the command it receives from the CPU is "start channel program in system space," or "start channel program in I/O space." Channel programs alter the tag bits of GA, GB, GC, and TP by using different instructions for loading the registers. Briefly, a "load pointer" instruction clears a tag bit, a "move" instruction sets a tag bit, and a "move pointer" instruction moves a memory value (either 0 or 1) to a tag bit. Section 3.9 covers these instructions in detail.

If a register points to the system space, all 20 bits are placed on the address lines to allow the full megabyte to be directly addressed. If a register points to the I/O space, the upper four bits of the address lines are undefined; the lower 16 bits are sufficient to access any location in the 64k byte I/O space.

Concurrent Channel Operation

Both channels may be active concurrently, but only one can actually run at a time. At the end of each internal cycle, the CCU lets one channel or the other execute the next internal cycle. No extra overhead is incurred by this channel switching. The basis for making the determination is a priority mechanism built into the IOP. This mechanism recognizes that some kinds of activities (e.g., DMA) are more important than others. Each activity that a channel can perform has a priority that reflects its relative importance (see table 3-3).

Two new activities are introduced in table 3-3. When a DMA transfer terminates, the channel executes a short internal channel program. This DMA termination program adjusts TP so that the user's program resumes at the instruction specified when the transfer was setup (this is discussed in detail in section 3.4). Similarly, when a channel attention is recognized, the channel executes an internal program that examines the CCW and carries out its command. Both of these programs consist of standard 8089 instructions that are fetched from internal ROM. Intel Application Note AP-50, *Debugging Strategies and Considerations for 8089 Systems*, lists the instructions in these programs. Users monitoring the bus during debugging may see operands read or written by the termination or channel attention programs. The instructions themselves, however, will not appear on the bus as they are resident in the chip.

Notice also that, according to table 3-3, a channel program may run at priority 3 or at priority 1.

Table 3-3. Channel Priorities and Interleave Boundaries

Channel Activity	Priority (1 = highest)	Interleave Boundary	
		By DMA	By Instruction
DMA transfer	1	Bus cycle[1]	Bus cycle[1]
DMA termination sequence	1	Internal cycle	None
Channel program (chained)	1	Internal cycle[2]	Instruction
Channel attention sequence	2	Internal cycle	None
Channel program (not chained)	3	Internal cycle[2]	Instruction
Idle	4	Two clocks	Two clocks

[1]DMA is not interleaved while $\overline{\text{LOCK}}$ is active.
[2]Except TSL instruction; see section 3.7.

Channel program priority is determined by the chain bit in the channel control register. If this bit is cleared, the program runs at normal priority (3); if it is set, the program is said to be chained, and it runs at the same priority as DMA. Thus, the chain bit provides a way to raise the priority of a critical channel program.

The CCU lets the channel with the highest priority run. If both channels are running activities with the same priority, the CCU examines the priority bits in the PSWs. If the priority bits are unequal, the channel with the higher value (1) runs. Thus, the priority bit serves as a "tie breaker" when the channels are otherwise at the same priority level. The value of the priority bit in the PSW is loaded from a corresponding bit in the CCW; therefore, the CPU can control which channel will run when the channels are at the same priority level. The priority bit has no effect when the channel priorities are different. If both channels are at the same priority level and if both priority bits are equal, the channels run alternately without any additional overhead.

The CCU switches channels only at certain points called interleave boundaries; these vary according to the type of activity running in each channel and are shown in table 3-3. In table 3-3 and in the following discussion, the terms "channel A" and "channel B" are used to identify two active channels that are bidding for control of an IOP. "Channel A" is the channel that last ran and will run again unless the CCU switches to "channel B." Where the CCU switches from one channel (channel A) to another (channel B) depends on whether channel B is performing DMA or is executing instructions. For this determination, instructions in the internal ROM are considered the same as instructions executed in user-written channel programs (chained or not chained). Table 3-3 shows that a switch from channel A to channel B will occur sooner if channel B is running DMA. DMA, then, interleaves instruction execution at internal cycle boundaries. Since instructions are often composed of several internal cycles, instruction execution on channel A can be suspended by DMA on channel B (when channel A next runs, the instruction is resumed from the point of suspension). DMA on channel A is interleaved by DMA on channel B after any bus cycle (when channel A runs again, the DMA transfer sequence is resumed from the point of suspension). If both channels are executing programs, the interleave boundaries are extended to instruction boundaries: a program on channel B will not run until channel A reaches the end of an instruction. Note that a DMA termination sequence or channel attention sequence on channel A cannot be interleaved by instructions on channel B, regardless of channel B's priority. These internal programs are short, however, and will not delay channel B for long (see Chapter 4 for timing information).

Table 3-4 summarizes the channel switching mechanism with several examples. It is important to remember that channel switching occurs only when both channels are ready to run. In typical applications, one of the channels will be idle much of the time, either because it is waiting to be dispatched by the CPU or because it is waiting for a DMA request in a synchronized transfer. (During a synchronized transfer, the channel is idle between DMA requests; for many peripherals, the channel will spend much more time idling than executing DMA cycles.) The real potential for one channel "shutting out" a priority 1 activity on the other channel is largely limited to unsynchronized DMA transfers and locked transfers (synchronized or unsynchronized). Long, chained channel programs and high-speed synchronized DMA will slow a priority 1 activity on the other channel, but will not shut it out because the channels will alternate (assuming their priority bits are equal). A chained channel program will shut out any lower priority activity on the other channel, including a channel attention. (The channel attention is latched by the IOP, however, so it will execute when the other channel drops to a lower priority.) Chained channel programs should therefore be used with discretion and should be made as short as possible.

3.3 Memory

The 8089 can access memory components located in two different address spaces. The system space, which coincides with the CPU's memory space, may contain up to 1,048,576 bytes. The I/O space, which may either coincide with the CPU's I/O space or be local (private) to the IOP, may contain up to 65,536 bytes. Memory components in the system space should respond to the memory read and write commands issued by the 8288 Bus Controller. Memory components in the I/O space must respond to 8288 I/O read and write commands. Memory in either space may be

Table 3-4. Channel Switching Examples

Channel A (Ran Last)				Channel B			Result
Activity	Chain Bit	Priority Bit	LOCK	Activity	Chain Bit	Priority Bit	
DMA transfer	X	X	Inactive	Idle	X	X	A runs.
DMA transfer	X	X	Inactive	Channel attention	X	X	A runs until end of current transfer cycle; then B runs.
Channel program	X	0	Inactive	Channel program	X	1	B runs.
Channel program	X	0	Inactive	Channel program	X	0	A and B alternate by instruction.
Channel program	1	X	Inactive	Channel program	0	X	A runs.
DMA transfer	X	1	Inactive	Channel program	1	1	B runs one bus or internal cycle following each bus cycle run by A.*
Channel attention	X	X	Inactive	Channel program	1	X	A runs if it has started the sequence; otherwise B runs.
DMA transfer	X	X	Active	Channel attention	X	X	A runs until DMA terminates.
Channel program (TSL instruction)	0	X	Active	DMA transfer	X	X	A completes TSL instruction, LOCK goes inactive and B runs.

*If transfer is synchronized, B also runs when A goes idle between transfer cycles.

implemented like 8086 memory (16-bit words split into even- and odd-addressed 8-bit banks) or 8088 memory (a single 8-bit bank). See Chapter 4 for physical implementation considerations.

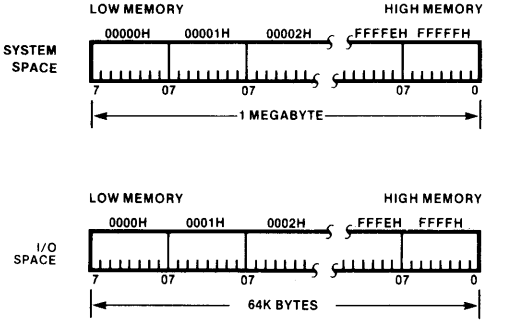


Figure 3-19. Storage Organization

Storage Organization

From a software point of view, both 8089 memory spaces are organized as unsegmented arrays of individually addressable 8-bit bytes (figure 3-19). Instructions and data may be stored at any address without regard for alignment (figure 3-20).

The IOP views the system space differently from the 8086 or 8088 with which it typically shares the space. The 8086 and 8088 differentiate between a location's logical (segment and offset) address and its physical (20-bit) address.

The 8089 does not "see" the logically segmented structure of the memory space; it uses its 20-bit pointer registers to access all locations in the system space by their physical addresses. Memory in the 8089 I/O space is treated similarly except that only 16 bits are needed to address any location.

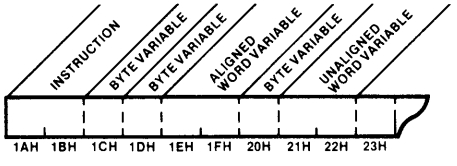


Figure 3-20. Instruction and Variable Storage

Following Intel convention, word data is stored with the most-significant byte in the higher address (see figure 3-21). The 8089 recognizes the doubleword pointer variable used by the 8086 and 8088 (figure 3-22). The lower-addressed word of the pointer contains an offset value, and the higher-addressed word contains a segment base address. Each word is stored conventionally, with the higher-addressed byte containing the most-significant eight bits of the word. The 8089 can convert a doubleword pointer into a 20-bit physical address when it is loaded into a pointer register to address system memory. A special 3-byte variable, called a physical address pointer (figure 3-23), is used to save and restore pointer registers and their associated tag bits.

Dedicated and Reserved Memory Locations

The extreme low and high addresses of the system space are dedicated to specific processor functions or are reserved for use by other Intel hardware and software products; the locations are 0H through 7FH (128 bytes) and FFFF0H through FFFFFH (16 bytes), as shown in figure 3-24. The low addresses are used for part of the 8086/8088 interrupt pointer table. Locations FFFF0H-FFFFBH are used for 8086, 8088 and 8089 startup sequences; the remaining locations are reserved by Intel.

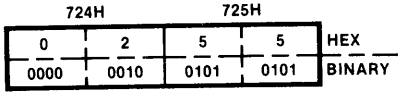
If an IOP is configured locally, its I/O space coincides with the CPU's I/O space, and it must respect the reserved addresses F8H-FFH. The entire I/O space of a remotely-configured IOP may be used without restriction.

Using any dedicated or reserved addresses may inhibit the compatibility of a system with current or future Intel hardware and software products.

Dynamic Relocation

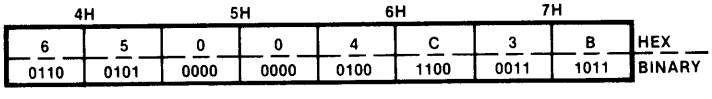
The 8089 is very well-suited to environments in which programs do not occupy static memory locations, but are moved about during execution. Dynamic code relocation allows systems to make efficient use of limited memory resources by transferring programs between external storage and memory, and by combining scattered free areas of memory into larger, more useful, continuous spaces.

IOP channel programs are inherently position-independent, the only restriction being that channel programs that transfer to each other or share data must be moved as a unit. Since the IOP



VALUE OF WORD STORED AT 724H: 5502H

Figure 3-21. Storage of Word Variables



VALUE OF DOUBLEWORD POINTER STORED AT 4H:
SEGMENT BASE ADDRESS: 3B4CH
OFFSET: 65H

Figure 3-22. Storage of Doubleword Pointer Variables

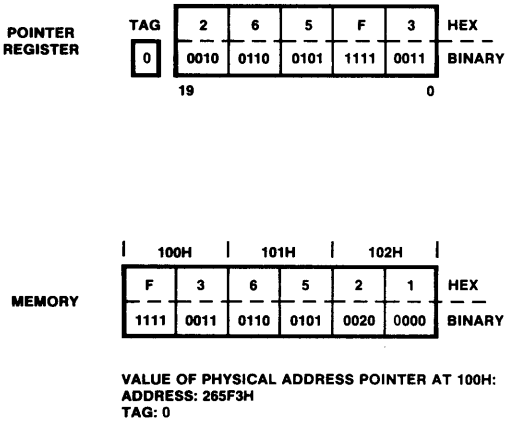


Figure 3-23. Storage of Physical Address Pointer Variables

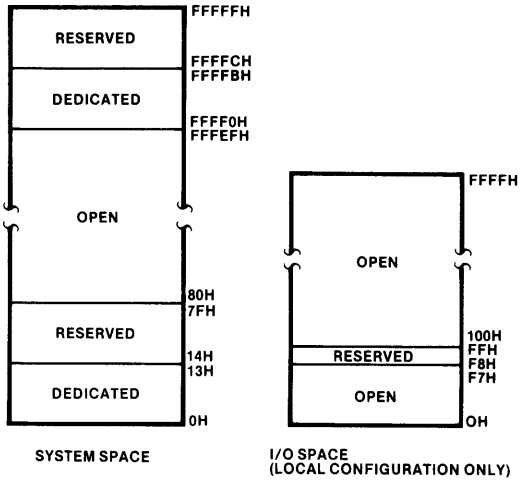


Figure 3-24. Reserved Memory Locations

receives the address of a channel program and its associated parameter block when it is dispatched by the CPU, the location of these blocks is immaterial and can change from one dispatch to the next. (Note, however, that the channel control block cannot be moved without reinitializing the IOP.) Typically, then, the CPU would direct the movement of IOP channel programs and parameter blocks. These blocks, of course, cannot be moved while they are in use.

While the CPU may be in charge of relocation, the IOP is an excellent vehicle for performing the actual transfer of channel programs, parameter blocks, and CPU programs as well. A very simple channel program can transfer code between memory locations by DMA much faster than the equivalent CPU instructions, and transfers between disk and memory also can be performed more efficiently.

Memory Access

Memory accesses are always performed using a pointer register and its associated tag bit. The tag bit indicates whether the access is to the system space (tag=0) or the I/O space (tag=1). The pointer register contains the base address of the location; i.e., the pointer register is used as a base register. Only the low-order 16 bits of the pointer register are used for I/O space locations; all 20 bits are used for system space addresses. Different types of memory accesses use base registers as shown in table 3-5. The 8089 addressing modes allow the base address of a memory operand to be modified by other registers and constant values to yield the effective address of the operand (see section 3.8).

Notice that table 3-5 indicates that memory operands may be addressed using register PP in addition to GA, GB, and GC. PP is maintained by the IOP and can neither be read nor written by a channel program; it can be used, however, to access data in the parameter block. PP has no associated tag bit; a reference to it implies the system space, where a parameter block always resides.

Table 3-5. Base Register Use in Memory Access

Memory Access	Base Register
Instruction Fetch	TP
DMA Source	GA or GB[1]
DMA Destination	GA or GB[1]
DMA Translate Table	GC
Memory Operand	GA or GB or GC or PP[2]

[1]As specified in CC register
[2]As specified in instruction

The IOP is told the physical widths of the system and I/O buses when it is initialized. If a bus is eight bits wide, the IOP accesses memory on this bus like an 8088. Instruction fetches and operand reads and writes are performed one byte at a time; one bus cycle is run for each memory access. Word operands are accessed in two cycles, completely transparent to software. Instruction fetches are made as needed, and the instruction stream is not queued.

The IOP accesses memory on a 16-bit bus like an 8086. As mentioned in the previous section, the instruction stream is generally fetched in words from even addresses with the second byte held in the one-byte queue. If a word operand is aligned (i.e., located at an even address), the 8089 will access it in a single 16-bit bus cycle. If a word operand is unaligned (i.e., located at an odd address), the word will be accessed in two consecutive 8-bit bus cycles. Byte operands are always accessed in 8-bit bus cycles.

For memory on 16-bit buses, performance is improved and bus contention is reduced if word operands are stored at even addresses. The instruction queue tends to reduce the effect of alignment on instructions fetched on a 16-bit bus. In tight loops, performance can be increased by word-aligning transfer targets.

Notice that the correct operation of a program is completely independent of memory bus width. A channel program written for one system that uses an 8-bit memory bus will execute without modification if the bus is increased to 16 bits. It is good practice, though, to write all programs as though they are to run on 16-bit systems; i.e., to align word operands. Such programs will then make optimal use of the bus in whatever system they are run.

3.4 Input/Output

The 8089 combines the programmed I/O capabilities of a CPU with the high-speed block transfer facility of a DMA controller. It also provides additional features (e.g., compare and translate during DMA) and is more flexible than a typical CPU or DMA controller. The 8089 transfers data from a source address to a destination address. Whether the component mapped into a given address is actually memory or I/O is immaterial. All addresses in both the system and I/O spaces are equally accessible, and transfers may be made between the two spaces as well as within either address space.

Programmed I/O

A channel program performs I/O similar to the way a CPU communicates with memory-mapped I/O devices. Memory reference instructions perform the transfer rather than "dedicated" I/O instructions, such as the 8086/8088 IN and OUT instructions. Programmed I/O is typically used to prepare a device controller for a DMA transfer and to obtain status/result information from the controller following termination of the transfer. It may be used, however, with any device whose transfer rate does not require DMA.

I/O Instructions

Since the 8089 does not distinguish between memory components and I/O devices, any instruction that accepts a byte or word memory operand can be used to access an I/O device. Most memory reference instructions take a source operand or a destination operand, or both. The instructions generally obtain data from the source operand, operate on the data, and then place the result of the operation in the destination operand. Therefore, when a source operand refers to an address where an I/O device is located, data is input from the device. Similarly, when a destination operand refers to an I/O device address, data is output to the device.

Most I/O device controllers have one or more internal registers that accept commands and supply status or result information. Working with these registers typically involves:

- reading or writing the entire register;
- setting or clearing some bits in a register while leaving others alone; or
- testing a single bit in a register.

Table 3-6 shows some of the 8089 instructions that are useful for performing these kinds of operations. Section 3.7 covers the 8089 instruction set in detail.

Table 3-6. Memory Reference Instructions Used for I/O

Instruction	Effect on I/O Device
MOV/MOVB	Read or write word/byte
AND/ANDB	Clear multiple bits in word/byte
OR/ORB	Set multiple bits in word/byte
CLR	Clear single bit (in byte)
SET	Set single bit (in byte)
JBT	Read (byte) and jump if single bit =1
JNBT	Read (byte) and jump if single bit =0

Device Addressing

Since memory reference instructions are used to perform programmed I/O, device addressing is very similar to memory addressing. An operand that refers to an I/O device always specifies one of the pointer registers GA, GB, or GC (PP is legal, but an I/O device would not normally be mapped into a parameter block). The base address of the device is taken from the specified pointer register. Any of the memory addressing modes (see section 3.8) may be used to modify the base address to produce the effective (actual) address of the device. The pointer register's tag bit locates the device in the system space (tag=0) or in the I/O space (tag=1). If the device is in the I/O space, only the low-order 16 bits of the pointer register are used for the base address; all 20 bits are used for a system space address. The IOP's system and I/O spaces are fully compatible with the corresponding address spaces of the other 8086 family processors.

I/O Bus Transfers

Table 3-7 shows the number of bus cycles the IOP runs for all combinations of bus size, transfer size (byte or word), and transfer address (even or odd). Bus width refers to the physical bus implementation; the instruction mnemonic determines whether a byte or a word is transferred.

Both 8- and 16-bit devices may reside on a 16-bit bus. All 16-bit devices should be located at even addresses so that transfers will be performed in one bus cycle. The 8-bit devices on a 16-bit bus may be located at odd or even addresses. The internal registers in an 8-bit device on a 16-bit bus must be assigned all-odd or all-even addresses that are two bytes apart (e.g., 1H, 3H, 5H, or 2H, 4H, 6H). All 8-bit peripherals should be referenced with byte instructions, and 16-bit devices should be referenced with word instructions. Odd-addressed 8-bit devices must be able to transfer data on the upper eight bits of the 16-bit physical data bus.

Only 8-bit devices should be connected to an 8-bit bus, and these should only be referenced with byte instructions. An 8-bit device on an 8-bit bus may be located at an odd or even address, and its internal registers may be assigned consecutive addresses (e.g., 1H, 2H, 3H). Assigning all-odd or all-even addresses, however, will simplify conversion to a 16-bit bus at a later date.

Table 3-7. Programmed I/O Bus Transfers

Bus Width:	8				16			
Instruction:	byte		word*		byte		word	
Device Address:	even	odd	even	odd	even	odd	even	odd*
Bus Cycles:	1	1	2	2	1	1	1	2

* not normally used

DMA Transfers

In addition to byte- and word-oriented programmed I/O, the 8089 can transfer blocks of data by direct memory access. A block may be transferred between any two addresses; memory-to-memory transfers are performed as easily as memory-to-port, port-to-memory or port-to-port exchanges. There is no limitation on the size of the block that can be transferred except that the block cannot exceed 64k bytes if byte count termination is used. A channel program typically prepares for a DMA transfer by writing commands to a device controller and initializing channel registers that are used during the transfer. No instructions are executed during the transfer, however, and very high throughput speeds can be achieved.

Preparing the Device Controller

Most controllers that can peform DMA transfers are quite flexible in that they can perform several different types of operations. For example, an 8271 Floppy Disk Controller can read a sector, write a sector, seek to track 0, etc. The controller typically has one or more internal registers that are "programmed" to perform a given operation. Often, certain registers will contain status information that can be read to determine if the controller is busy, if it has detected an error, etc.

An 8089 channel program views these device registers as a series of memory locations. The channel program typically places the device's base address in a pointer register and uses programmed I/O to communicate with the registers.

Some controllers start a DMA transfer immediately upon receiving the last of a series of parameters. If this type of controller is being used, the channel program instruction that sends the last parameter should *follow* the 8089 XFER instruction. (The XFER instruction places the channel in DMA mode after the next instruction; this is explained in more detail later in this section.)

Preparing the Channel

For a channel to perform a DMA transfer, it must be provided with information that describes the operation. The channel program provides this information by loading values into channel registers and, in one case, by executing a special instruction (see table 3-8).

Source and Destination Pointers. One register is loaded to point to the transfer source; the other points to the destination. A bit in the channel control register is set to indicate which register is the source pointer. If a register is pointed at a memory location, it should contain the address where the transfer is to begin — i.e., the lowest address in the buffer. The channel automatically increments a memory pointer as the transfer proceeds. If the tag bit selects the I/O space, the upper four bits of the register are ignored; if the tag selects the system space, all 20 bits are used. The source and destination may be located in the same or in different address spaces.

Translate Table Pointer. If the data is to be translated as it is transferred, GC should be pointed at the first (lowest-addressed) byte in a 256-byte translation table. The table may be located in either the system or I/O space, and GC

Table 3-8. DMA Transfer Control Information

Information	Register or Instruction	Required or Optional
Source Pointer	GA or GB	Required
Destination Pointer	GA or GB	Required
Translate Table Pointer	GC	Optional
Byte Count	BC	Optional
Mask/Compare Values	MC	Optional
Logical Bus Width	WID	Optional*
Channel Control	CC	Required

*Must be executed once following processor RESET.

should be loaded by an instruction that sets or clears its tag bit as appropriate. The translate operation is only defined for byte data; source and destination logical bus widths must both be set to eight bits.

The channel translates a byte by treating it as an unsigned 8-bit binary number. This number is added to the content of register GC to form a memory address; GC is not altered by the operation. If GC points to the I/O space, its upper four bits are ignored in the operation. The byte at this address (which is in the translate table) is then fetched from memory, replacing the source byte. Figure 3-25 illustrates the translate process.

Byte Count. If the transfer is to be terminated on byte count— i.e., after a specific number of bytes have been transferred—the desired count should be loaded into register BC as an unsigned 16-bit number. The channel decrements BC as the transfer proceeds, whether or not byte count termination has been specified. There are cases (discussed later in this section) where the difference between BC's value before and after the transfer does not accurately reflect the number of bytes transferred to the destination.

Mask/Compare Values. If the transfer is to be terminated when a byte (possibly translated) is found equal or unequal to a search value, MC should be loaded as described in section 3.2. MC is not altered during the transfer. Normally, the logical destination bus width is set to eight bits when transferred data is being compared. If the logical destination width is 16 bits, only the low-order byte of each word is compared.

Logical Bus Width. The 8089 WID (logical bus width) instruction is used to set the logical width of the source and destination buses for a DMA transfer. Any bus whose physical width is eight bits can only have a logical width of eight bits. A 16-bit physical bus, however, can have a logical width of 8 or 16 bits; i.e., it can be used as either an 8-bit or 16-bit bus in any given transfer. Logical bus widths are set independently for each channel.

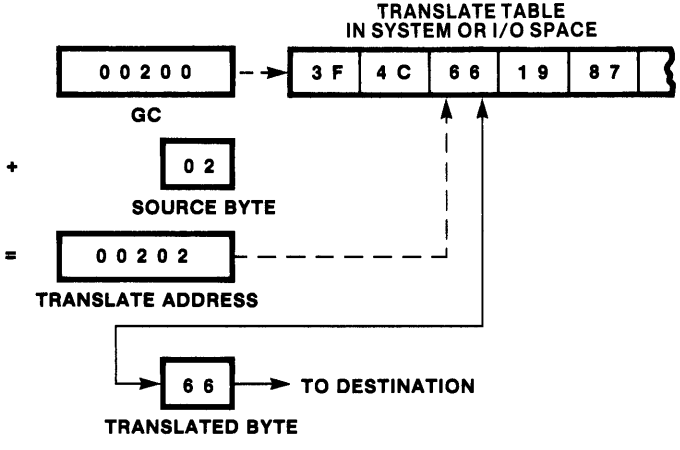


Figure 3-25. Translate Operation

For a transfer to or from an I/O device on a 16-bit physical bus, the logical bus width should be set equal to the peripheral's width; i.e., 8 or 16 bits. Transfers to or from 16-bit memory will run at maximum speed if the logical bus width is set to 16 since the channel will fetch/store words. In the following cases, however, the logical width should be set to 8:

- the data is being translated,
- the data is being compared under mask, and the 16-bit memory is the destination of the transfer.

The WID instruction sets both logical widths and remains in effect until another WID instruction is executed. Following processor reset, the settings of the logical bus widths are unpredictable. Therefore, the WID instruction must be executed before the first DMA transfer.

Channel Control. The 16 bits of the CC register are divided into 10 fields that specify how the DMA transfer is to be executed (see figure 3-26). A channel program typically sets these fields by loading a word into the register.

The *function field* (bits 15-14) identifies the source and destination as memory or ports (I/O devices). During the transfer, the channel increments source/destination pointer registers that refer to memory so that the data will be placed in successive locations. Pointers that refer to I/O devices remain constant throughout the transfer.

The *translate field* (bit 13) controls data translation. If it is set, each incoming byte is translated using the table pointed to by register GC. Translate is defined only for byte transfers; the destination bus must have a logical width of eight.

The *synchronization field* (bits 12-11) specifies how the transfer is to be synchronized. Unsynchronized ("free running") transfers are typically used in memory-to-memory moves. The channel begins the next transfer cycle immediately upon completion of the current cycle (assuming it has the bus). Slow memories, which cannot run as fast as the channel, can extend bus cycles by signaling "not ready" to the 8284 Clock Generator, which will insert wait states into the bus cycle. A similar technique may be used with peripherals whose speed exceeds the channel's ability to execute a synchronized transfer: in effect, the peripheral synchronizes the transfer through the use of wait states. Chapter 4 discusses synchronization in more detail.

Source synchronization is typically selected when the source is an I/O device and the destination is memory. The I/O device starts the next transfer cycle by activating the channel's DRQ (DMA request) line. The channel then runs one transfer cycle and waits for the next DRQ.

Destination synchronization is most often used when the source is memory and the destination is an I/O device. Again, the I/O device controls the transfer frequency by signaling on DRQ when it is ready to receive the next byte or word.

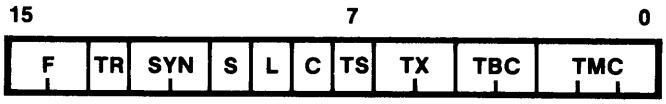
The *source field* (bit 10) identifies register GA or GB as the source pointer (and the other as the destination pointer).

The *lock field* (bit 9) may be used to instruct the channel to assert the processor's bus lock ($\overline{\text{LOCK}}$) signal during the transfer. In a source-synchronized transfer, $\overline{\text{LOCK}}$ is active from the time the first DMA request is received until the channel enters the termination sequence. In a destination-synchronized transfer $\overline{\text{LOCK}}$ is active from the first fetch (which precedes the first DMA request) until the channel enters the termination sequence.

The *chain field* (bit 8) is not used during the transfer. As discussed previously, setting this bit raises channel program execution to priority level 1.

The *terminate on single transfer field* (bit 7) can be used to cause the channel to run one complete transfer cycle only—i.e., to transfer one byte or word and immediately resume channel program execution. When single transfer is specified, any other termination conditions are ignored. Single transfer termination can be used with low-speed devices, such as keyboards and communication lines, to translate and/or compare one byte as it transferred.

The *three low-order fields* in register CC instruct the channel when to terminate the transfer, assuming that single transfer has not been selected. Three termination conditions may be specified singly or in combination.



F	FUNCTION
00	PORT TO PORT
01	MEMORY TO PORT
10	PORT TO MEMORY
11	MEMORY TO MEMORY

TR	TRANSLATE
0	NO TRANSLATE
1	TRANSLATE

SYN	SYNCHRONIZATION
00	NO SYNCHRONIZATION
01	SYNCHRONIZE ON SOURCE
10	SYNCHRONIZE ON DESTINATION
11	RESERVED BY INTEL

S	SOURCE
0	GA POINTS TO SOURCE
1	GB POINTS TO SOURCE

L	LOCK
0	NO LOCK
1	ACTUATE LOCK DURING TRANSFER

C	CHAIN
0	NO CHAINING
1	CHAINED: RAISE TB TO PRIORITY 1

TS	TERMINATE ON SINGLE TRANSFER
0	NO SINGLE TRANSFER TERMINATION
1	TERMINATE AFTER SINGLE TRANSFER

TX	TERMINATE ON EXTERNAL SIGNAL
00	NO EXTERNAL TERMINATION
01	TERMINATE ON EXT ACTIVE; OFFSET = 0
10	TERMINATE ON EXT ACTIVE; OFFSET = 4
11	TERMINATE ON EXT ACTIVE; OFFSET = 8

TBC	TERMINATE ON BYTE COUNT
00	NO BYTE COUNT TERMINATION
01	TERMINATE ON BC = 0; OFFSET = 0
10	TERMINATE ON BC = 0; OFFSET = 4
11	TERMINATE ON BC = 0; OFFSET = 8

TMC	TERMINATE ON MASKED COMPARE
000	NO MASK/COMPARE TERMINATION
001	TERMINATE ON MATCH; OFFSET = 0
010	TERMINATE ON MATCH; OFFSET = 4
011	TERMINATE ON MATCH; OFFSET = 8
100	(NO EFFECT)
101	TERMINATE ON NON-MATCH; OFFSET = 0
110	TERMINATE ON NON-MATCH; OFFSET = 4
111	TERMINATE ON NON-MATCH; OFFSET = 8

Figure 3-26. Channel Control Register Fields

External termination allows an I/O device (typically, the one that is synchronizing the transfer) to stop the transfer by activating the channel's EXT (external terminate) line. If byte count termination is selected, the channel will stop when BC=0. If masked compare termination is specified, the channel will stop the transfer when a byte is found that is equal or unequal (two options are available) to the low-order byte in MC as masked by MC's high-order byte. The byte that stops the termination is transferred. If translate has been specified, the translated byte is compared.

When a DMA transfer ends, the channel adds a value called the termination offset to the task pointer and resumes channel program execution at that point in the program. The termination offset may assume a value of 0, 4, or 8. Single transfer termination always results in a termination offset of 0. Figure 3-27 shows how the termination offsets can be used as indices into a three-element "jump table" that identifies the condition that caused the termination.

As an example of using the jump table, consider a case in which a transfer is to terminate when 80 bytes have been transferred or a linefeed character is detected, whichever occurs first. The program would load 80H into BC and 000AH into MC (ASCII line feed, no bits masked). The channel program could assign byte count termination an offset of 0 and masked compare termination an offset of 4. If the transfer is terminated by byte count (no linefeed is found), the instruction at location TP+0 will be executed first after the termination. If the linefeed is found before the byte count expires, the instruction at TP+4 will be executed first. The LJMP (long unconditional jump, see section 3.7) instruction is four bytes long and can be placed at TP+0 and TP+4 to cause the channel program to jump to a different routine, depending on how the transfer terminates.

If the transfer can only terminate in one way and that condition is assigned an offset of 0, there is no need for the jump table. Code which is to be unconditionally executed when the transfer ends can immediately follow the instruction after XFER. This is also the case when single transfer is specified (execution always resumes at TP+0).

It is possible, however, for two, or even three, termination conditions to arise at the same time. In

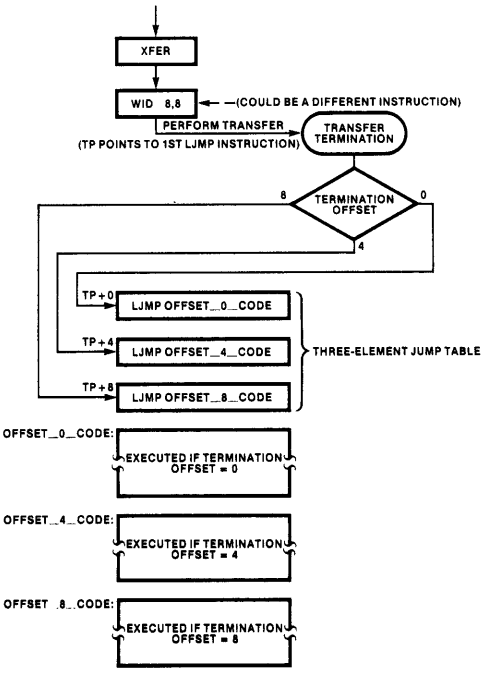


Figure 3-27. Termination Jump Table

the preceding example, this would occur if the 80th character were a linefeed. When multiple terminations occur simultaneously, the channel indicates that termination resulted from the condition with the largest offset value. In the preceding example, if byte count and search termination occur at the same time, the channel program resumes at TP+4.

Beginning the Transfer

The 8089 XFER (transfer) instruction puts the channel into DMA transfer mode after the *following instruction* has been executed. This technique gives the channel time to set itself up when it is used with device controllers, such as the 8271 Floppy Disk Controller, that begin transferring immediately upon receipt of the last in a series of parameters or commands. If the transfer is to or from such a device, the last parameter should be sent to the device after the XFER instruction. If this type of device is not being used, the instruction following XFER would

typically send a "start" command to the controller. If a memory-to-memory transfer is being made, any instruction may follow XFER except one that alters GA, GB, or CC. The HLT instruction should normally not be coded after the XFER; doing so clears the channel's BUSY flag, but allows the DMA transfer to proceed.

DMA Transfer Cycle

A DMA transfer cycle is illustrated in figure 3-28; a complete transfer is a series of these cycles run until a termination condition is encountered. The figure is deliberately simplified to explain the general operation of a DMA transfer; in particular, the updating of the source and destination pointers (GA and GB) can be more complex than the figure indicates. Notice that it is possible to start an unending transfer by not specifying a termination condition in CC or by specifying a condition that never occurs; it is the programmer's responsibility to ensure that the transfer eventually stops.

If the transfer is source-synchronized, the channel waits until the synchronizing device activates the channel's DRQ line. The other channel is free to run during this idle period. The channel fetches a byte or a word, depending on the source address (contained in GA or GB) and the logical bus width. Table 3-9 shows how a channel performs the fetch/store sequence for all combinations of addresses and bus widths. If the destination is on a 16-bit logical bus and the source is on an 8-bit logical bus, and the transfer is to an even address, the channel fetches a second byte and assembles a word internally. During each fetch, the channel decrements BC according to whether a byte or word is obtained. Thus BC always indicates the number of bytes fetched.

The channel samples its EXT line after every bus cycle in the transfer. If EXT is recognized after the first of two scheduled fetches, the second fetch is not run. After the fetch sequence has been completed, the channel translates the data if this option is specified in CC.

If a word has been fetched or assembled, and bytes are to be stored (destination bus is eight bits or transfer is to an odd address), the channel disassembles the word into two bytes. If the transfer is destination-synchronized (only one type of synchronization may be specified for a given transfer), the channel waits for DRQ before running a store cycle. It stores a word or the lower-addressed byte (which may be the only byte or the first of two bytes). Table 3-9 shows the possible combinations of even/odd addresses and logical bus widths that define the store cycle. Whenever stores are to memory on a 16-bit logical bus, the channel stores words, except that bytes may be stored on the first and last cycles.

Table 3-9. DMA Transfer Assembly/Disassembly

Address (Source→ Destination)	Logical Bus Width (Source→Destination)			
	8→8	8→16	16→8	16→16
EVEN→EVEN	B→B	B/B→W	W→B/B	W→W
EVEN→ODD	B→B	B→B	W→B/B	W→B/B
ODD→EVEN	B→B	B/B→W	B→B	B/B→W
ODD→ODD	B→B	B→B	B→B	B→B

B= Byte Fetched or Stored in 1 Bus Cycle
W= Word Fetched or Stored in 1 Bus Cycle
B/B= 2 Bytes Fetched or Stored in 2 Bus Cycles

The channel samples EXT again after the first store cycle and, if it is active, the channel prevents the second store cycle from running. If specified in the CC register, the low-order byte is compared to the value in MC. A "hit" on the comparison (equal or unequal, as indicated in CC) also prevents the second of two scheduled store cycles from running. In both of these cases, one byte has been "overfetched," and this is reflected in BC's value. It would be unusual, however, for a synchronizing device to issue EXT in the midst of a DMA cycle. Note also that EXT is valid only when DRQ is inactive. Chapter 4 covers the timing requirements for these two signals in detail.

GA and GB are updated next. Only memory pointers are incremented; pointers to I/O devices remain constant throughout the transfer.

If any termination condition has occurred during this cycle, the channel stops the transfer. It uses the content of the CC register to assign a value to the termination offset, to reflect the cause of the termination. The channel adds this offset to TP and resumes channel program execution at the location now addressed by TP. This offset will

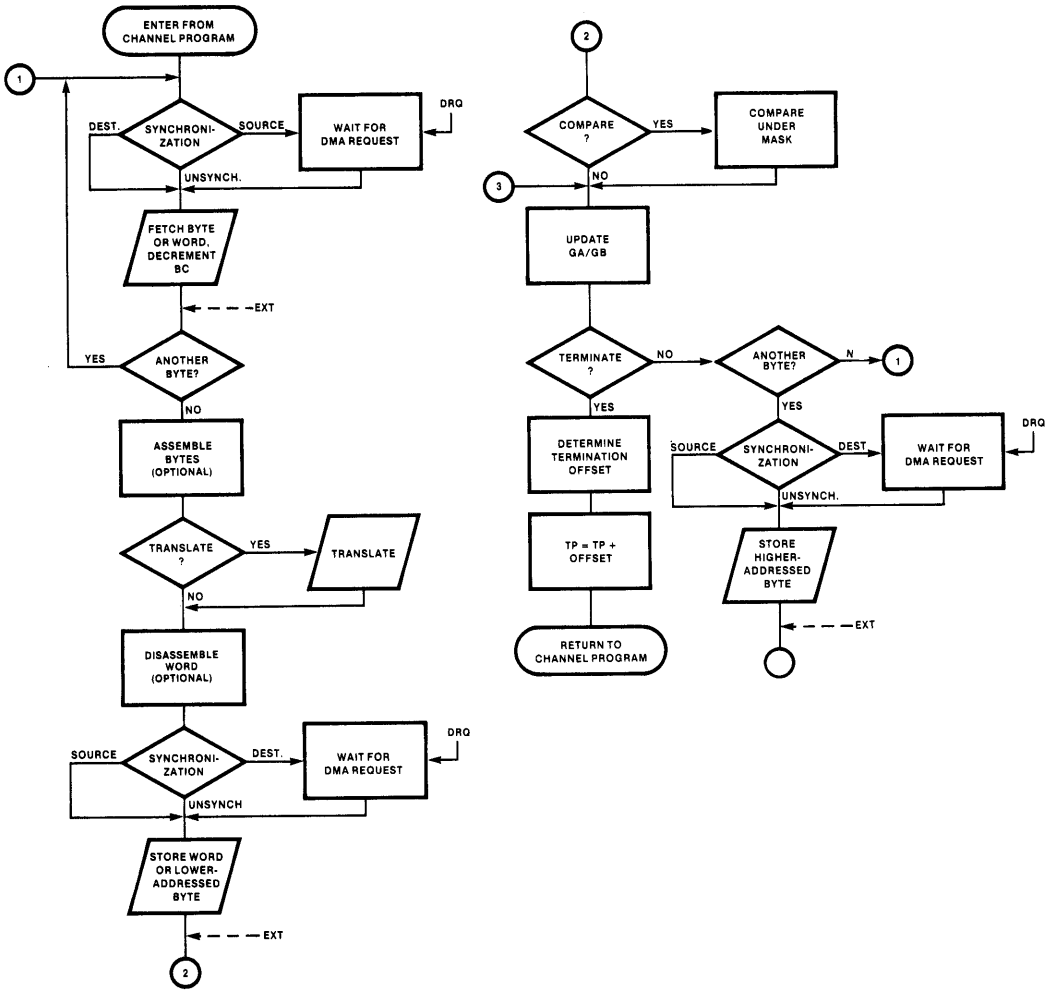


Figure 3-28. Simplified DMA Transfer Flowchart

always be zero, four, or eight bytes past the end of the instruction following the XFER instruction.

If no termination condition is detected and another byte remains to be stored, the channel stores this byte, waiting for DRQ if necessary, and updates the source and destination pointers. After the store, it again checks for termination.

Following the Transfer

A DMA transfer updates register BC, register GA (if it points to memory), and register GB (if it points to memory). If the original contents of these registers are needed following the transfer, the contents should be saved in memory prior to executing the XFER instruction.

A program may determine the address of the last byte stored by a DMA transfer by inspecting the pointer registers as shown in table 3-10. The number of bytes stored is equal to:

last_byte_address — first_byte_address + 1.

For port-to-port transfers, the number of bytes transferred can be determined by subtracting the final value of BC from its original value provided that:

- the original BC > final BC,
- a transfer cycle is not "chopped off" before it completes by a masked compare or external termination.

In general, programs should not use the contents of GA, GB and BC following a transfer except as noted above and in table 3-10. This is because the contents of the registers are affected by numerous conditions, particularly when the transfer is terminated by EXT. In particular, when a program is performing a sequence of transfers, it should reload these registers before each transfer.

3.5 Multiprocessing Features

The 8089 shares the multiprocessing facilities common to the 8086 family of processors. It has on-chip logic for arbitrating the use of the local bus with a CPU or another IOP; system bus arbitration is delegated to an 8289 Bus Arbiter. The 8089's TSL (test and set while locked) instruction enables it to share a resource, such as a buffer, with other processors by means of semaphore (see section 2.5 for a discussion of the use of semaphores to control access to shared resources). Finally, the 8089 can lock the system bus for the duration of a DMA transfer to ensure that the transfer completes without interference from other processors on the bus.

In the remote configuration, the 8089 is electrically compatible with Intel's Multibus™ multimaster bus design. This means that the power and convenience of 8089 I/O processing can be used in 8080- or 8085-based systems that implement the Multibus protocol or a superset of it. This includes single-board computers such as Intel's iSBC 80/20™ and iSBC 80/30™ boards. In addition, the IOP can access other iSBC board products such as memory and communications controllers.

Bus Arbitration

The 8089 shares its system bus with a CPU, and may also share its I/O bus with an IOP or another CPU. Only one processor at a time may drive a bus. When two (or more) processors want to use a shared bus, the system must provide an arbitration mechanism that will grant the bus to one of the processors. This section describes the bus arbitration facilities that may be used with the 8089 and covers their applicability to different IOP configurations.

Table 3-10. Address of Last Byte Stored

Termination	Source	Destination	Synchronization	Last Byte Stored
byte count	memory memory port	memory port memory	any any any	destination pointer[1] source pointer destination pointer
masked compare	memory memory port	memory port memory	any any any	destination pointer source pointer destination pointer
external	memory memory port	memory port memory	unsynchronized destination source	destination pointer source pointer[2] destination pointer

[1]Source pointer may also be used.
[2]If transfer is B/B→W, source pointer must be decremented by 1 to point to last byte transferred.

Request/Grant Line

When an 8089 is directly connected to another 8089, an 8086 or an 8088, the $\overline{RQ}/\overline{GT}$ (request/grant) lines built into all of these processors are used to arbitrate use of a local bus. In the local mode, $\overline{RQ}/\overline{GT}$ is used to control access to both the system and the I/O bus.

As discussed in section 2.6, the CPU's request/grant lines ($\overline{RQ}/\overline{GT0}$ and $\overline{RQ}/\overline{GT1}$) operate as follows:

- an external processor sends a pulse to the CPU to request use of the bus;
- the CPU finishes its current bus cycle, if one is in progress, and sends a pulse to the processor to indicate that it has been granted the bus; and
- when the external processor is finished with the bus, it sends a final pulse to the CPU, to indicate that it is releasing the bus.

The 8089's request/grant circuit can operate in two modes; the mode is selected when the IOP is initialized (see section 3.6). Mode 0 is compatible with the 8086/8088 request/grant circuit and must be specified when the 8089's $\overline{RQ}/\overline{GT}$ line is connected to $\overline{RQ}/\overline{GT0}$ or $\overline{RQ}/\overline{GT1}$ of one of these CPUs. Mode 0 *may* be specified when $\overline{RQ}/\overline{GT}$ of one 8089 is tied to $\overline{RQ}/\overline{GT}$ of another 8089. When mode 0 is used with a CPU, the CPU is designated the master, and the IOP is designated a slave. When mode 0 is used with another IOP, one IOP is the master, and the other is the slave. Master/slave designation also is made at initialization time as discussed in section 3.6. The master has the bus when the system is initialized and keeps the bus until it is requested by the slave. When the slave requests the bus, the master grants it if the master is idle. In this sense, the CPU becomes idle at the end of the current bus cycle. An IOP master, on the other hand, does not become idle until both channels have halted program execution or are waiting for DMA requests. Once granted the bus, the slave (always an IOP) uses it until both channels are idle, and then releases it to the master. In mode 0, the master has no way of requesting the slave to return the bus.

Mode 1 operation of the request/grant lines may only be used to arbitrate use of a private I/O bus between two IOPs. In this case, one IOP is designated the master, and the other is designated the slave. However, the only difference between a master and a slave running in mode 1 is that the master has the bus at initialization time. Both processors may request the bus from each other at any time. The processor that has the bus will grant it to the requester as soon as one of the following occurs on either channel:

- an unchained channel program instruction is completed, or
- a channel goes idle due to a program halt or the completion of a synchronized transfer cycle (the channel waits for a DMA request).

Execution of a chained channel program, a DMA termination sequence, a channel attention sequence, or a synchronized DMA transfer (i.e., a high-priority operation) on either channel prevents the IOP from granting the bus to the requesting IOP.

The handshaking sequence in mode 1 is:

- the requesting processor pulses once on $\overline{RQ}/\overline{GT}$;
- the processor with the bus grants it by pulsing once; and
- if the processor granting the bus wants it back immediately (for example, to fetch the next instruction), it will pulse $\overline{RQ}/\overline{GT}$ again, two clocks after the grant pulse.

The fundamental difference between the two modes is the frequency with which the bus can be switched between the two processors when both are active. In mode 0, the processor that has the bus will tend to keep it for relatively long periods if it is executing a channel program. Mode 1 in effect places unchained channel programs at a lower priority since the processor will give up the bus at the end of the next instruction. Therefore, when both processors are running channel programs or synchronized DMA, they will share the bus more or less equally. When a processor changes to what would typically be considered a higher-priority activity such as chained program execution or DMA termination, it will generally be able to obtain the bus quickly and keep the bus for the duration of the more critical activity.

8289 Bus Arbiter

When an IOP is configured remotely, an 8289 Bus Arbiter is used to control its access to the shared system bus (the CPU also has its own 8289). In a remote cluster of two IOPs or an IOP and a CPU, one 8289 controls access to the system bus for both processors in the cluster. The 8289 has several operating modes; when used with an 8089, the 8289 is usually strapped in its IOB (I/O Peripheral Bus) mode.

The 8289 monitors the IOP's status lines. When these indicate that the IOP needs a cycle on the system bus, and the IOP does not presently have the bus, the 8289 activates a bus request signal. This signal, along with the bus request lines of other 8289s on the same bus, can be routed to an external priority-resolving circuit. At the end of the current bus cycle, this circuit grants the bus to the requesting 8289 with the highest priority. Several different prioritizing techniques may be used; in a typical system, an IOP would have higher bus priority than a CPU. If the 8289 does not obtain the bus for its processor, it makes the bus appear "not ready" as if a slow memory were being accessed. The processor's clock generator responds to the "not ready" condition by inserting wait states into the IOP's bus cycle, thereby extending the cycle until the bus is acquired.

Bus Arbitration for IOP Configurations

When the CPU initializes an IOP, it must inform the IOP whether it is a master or a slave, and which request/grant mode is to be used. This section covers the requirements and options available for each IOP configuration; section 3.6 describes how the information is communicated at initialization time.

Table 3-11 summarizes the bus arbitration requirements and options by IOP configuration. In the local configuration, all bus arbitration is performed by the request/grant lines without additional hardware. One IOP may be connected to each of the CPU's $\overline{RQ}/\overline{GT}$ lines. The IOP connected to $\overline{RQ}/\overline{GT}0$ will obtain the bus if both processors make simultaneous requests.

Since a single IOP in a remote configuration does not use $\overline{RQ}/\overline{GT}$, its mode may be set to 0 or 1 without affect. The single remote IOP, however, must be initialized as a master. If two remote IOPs share an I/O bus, one must be a master and the other a slave; both must be initialized to use the same request/grant mode. Normally, mode 1 will be selected for its improved responsiveness, and the designation of master will be arbitrary. If one IOP must have the I/O bus when the system comes up, it should be initialized as the master.

When a remote IOP shares its I/O bus with a local CPU, it must be a slave and must use request/grant mode 0.

Bus Load Limit

A locally configured IOP effectively has higher bus priority than the CPU since the CPU will grant the bus upon request from the IOP. One or two local IOPs can potentially monopolize the bus at the expense of the CPU. Of course, if the IOP activities are time-critical, this is exactly what should happen. On the other hand, there may be low-priority channel programs that have less demanding performance requirements.

In such cases, the CPU may set a CCW bit called bus load limit to constrain the channel's use of the bus during normal (unchained) channel program

Table 3-11. Bus Arbitration Requirements and Options

IOP	Local		Remote		Remote With Local CPU	
	Master/ Slave	$\overline{RQ}/\overline{GT}$ Mode	Master/ Slave	$\overline{RQ}/\overline{GT}$ Mode	Master/ Slave	$\overline{RQ}/\overline{GT}$ Mode
IOP1	Slave	0	Master	0 or 1	Slave	0
IOP2	Slave	0	Slave	Same as Master	N/A	N/A

execution. When this bit is set, the channel decrements a 7-bit counter from 7F (127) to 0H with each instruction executed. Since the counter is decremented once per clock period, the channel waits a minimum of 128 clock cycles before it executes the next instruction. By forcing the execution time of all instructions to 128 clocks, the use of the bus is reduced to between 3 and 25 percent of the available bus cycles.

Setting the bus load limit effectively enables a CPU to slow the execution of a normal channel program, thus freeing up bus cycles. This is of most use in local configurations, but also may be effective in remote configurations, particularly when channel programs are executed from system memory. Bus load limit has no effect on chained channel programs, DMA transfers, DMA termination, or channel attention sequences.

Bus Lock

Like the 8086 and 8088, the 8089 has a $\overline{\text{LOCK}}$ (bus lock) signal which can be activated by software. The $\overline{\text{LOCK}}$ output is normally connected to the $\overline{\text{LOCK}}$ input of an 8289 Bus Arbiter. When $\overline{\text{LOCK}}$ is active, the bus arbiter will not release the bus to another processor regardless of its priority. A channel automatically locks the bus during execution of the TSL (test and set while locked) instruction and may lock the bus for the duration of a DMA transfer.

If bit 9 of register CC is set, the 8089 activates its $\overline{\text{LOCK}}$ output during a DMA transfer on that channel. If the transfer is synchronized, $\overline{\text{LOCK}}$ is active from the time that the first DRQ is recognized. If the transfer is unsynchronized, $\overline{\text{LOCK}}$ is active throughout the entire transfer (there are no idle periods in an unsynchronized transfer). $\overline{\text{LOCK}}$ goes inactive when the channel begins the DMA termination sequence.

A locked transfer ensures that the transfer will be completed in the shortest possible time and that the transferring channel has exclusive use of the bus. Once the channel obtains the bus and starts a locked transfer, the channel, in effect, becomes the highest-priority processor on that bus.

The 8089 TSL (test and set while locked) instruction can be used to implement a semaphore. (See section 2.5 for a discussion of how a semaphore may be used to control the access of multiple processors to a shared resource.) The instruction activates $\overline{\text{LOCK}}$ and inspects the value of a byte in memory. If the value of the byte is 0H, it is changed (set) to a value specified in the instruction and the following instruction is executed. If the byte does not contain 0H, control is transferred to another location specified in the instruction. The bus is locked from the time the byte is read until it is either written or control is transferred to ensure that another processor does not access the variable after TSL has read it, but before it has updated it (i.e., between bus cycles). The following line of code will repeatedly test a semaphore pointed to by GA until it is found to contain zero:

```
TEST_FLAG: TSL [GA], 0FFH, TEST_FLAG
```

When the semaphore is found to be zero, it is set to FFH and the program continues with the next instruction.

3.6 Processor Control and Monitoring

This section focuses on IOP/CPU interaction, i.e., how the CPU initializes the IOP and subsequently sends commands to channels, and how the channels may interrupt the CPU. It also covers the channels' DMA control signals and the status signals that external devices can use to monitor IOP activities.

Initialization

Before the 8089 channels can be dispatched to perform I/O tasks, the IOP must be initialized. The initialization sequence (figure 3-29) provides the IOP with a definition of the system environment: physical bus widths, request/grant mode, and the location of the channel control block.

The sequence begins when the IOP's RESET line is activated. This halts any operation in progress, but does not affect any registers. Upon the first

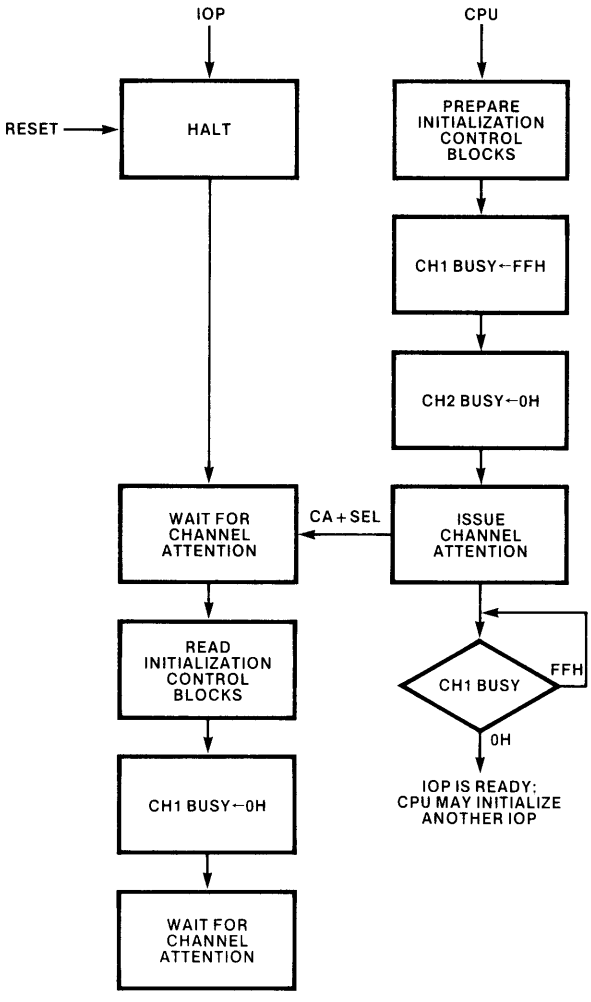


Figure 3-29. Initialization Sequence

RESET after power-up, the content of all IOP registers is undefined. Register contents are preserved if the IOP is subsequently RESET, except that RESET always clears the chain bit in register CC.

The IOP initializes itself by reading information from initialization control blocks located in the system space (see figure 3-30). The three blocks are the SCP (system configuration pointer), SCB (system configuration block) and the CB (channel control block). The CB is normally RAM-based; the SCP and the SCB may be in RAM or ROM. It is the CPU's responsibility to properly setup the control blocks.

The CPU starts the initialization sequence by issuing a channel attention to channel 1 (SEL low) or to channel 2 (SEL high). The CPU typically accesses the channels as two consecutive addresses in its I/O or memory space. An OUT instruction (for an I/O-mapped IOP) or a memory reference instruction (such as MOV) then issues the channel attention.

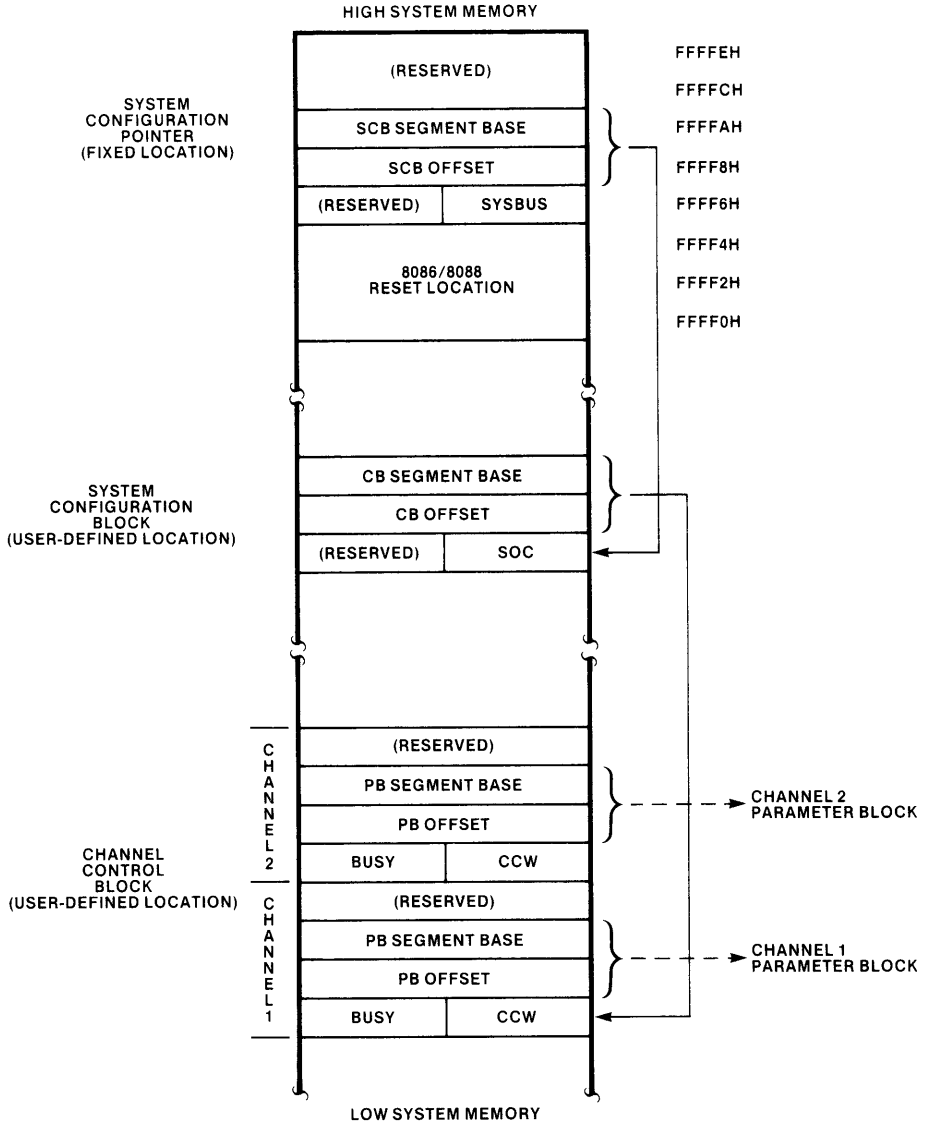


Figure 3-30. Initialization Control Blocks

If channel 1 is selected (SEL=low), the IOP considers itself a master (as discussed in section 3.5). If channel 2 is selected (SEL=high), the IOP operates as a slave. The IOP ignores, and does not latch, any subsequent channel attentions that occur during initialization.

If the IOP is a master, it assumes that it has the bus immediately. If it is a slave, it pulses $\overline{RQ}/\overline{GT}$ to request the bus from the CPU (local configuration) or the other IOP (remote configuration). When the IOP has obtained the bus, it assumes that the system bus is eight bits wide and reads the

SYSBUS field (figure 3-31) from location FFFF6H in system memory. This byte tells the IOP the actual physical width of the system bus; all subsequent accesses take advantage of a 16-bit bus if it is available; i.e., even-addressed words are fetched in single bus cycles. It is therefore advantageous to word-align the control blocks.

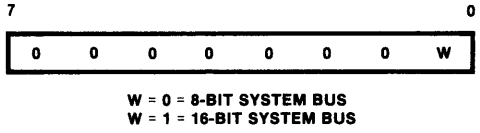


Figure 3-31. SYSBUS Encoding

Next, the IOP reads the SCB address located at FFFF8H. This is a standard doubleword pointer, and the IOP constructs a 20-bit physical address from it by shifting the segment base left four bits and adding the offset word of the pointer.

Having obtained the SCB address, the IOP reads the SOC (system operation command). This byte (see figure 3-32) tells the IOP the request/grant mode and the width of the I/O bus.

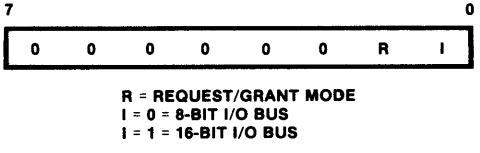


Figure 3-32. SOC Encoding

Then the IOP reads the doubleword pointer to the channel control block, converts the pointer into a 20-bit physical address, and stores it in an internal register. This register is not accessible to channel programs and is only loaded during initialization. The CB, therefore, cannot be moved during execution except by reinitializing the IOP.

After loading the address of the CB, the IOP clears the channel 1 BUSY flag to 0H. The other fields in the CB are used when a channel is dispatched and are not read or altered in the initialization sequence.

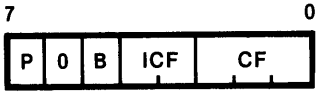
After the CPU has started the initialization sequence, it should monitor channel 1's BUSY flag in the CB to determine when the sequence has been completed. When the BUSY flag has been cleared, the CPU can dispatch either channel. It also can begin the initialization of another IOP. Since each IOP normally has a separate CB, the CPU must allocate the CB and update the pointer in the SCB before initializing the next IOP. Alternatively, multiple SCBs could be employed, each pointing to a different CB area. In this case the CPU would update the pointer in the SCP before initializing the next IOP. It follows from this that in multi-IOP systems, either the SCB or SCP, or both, must be RAM-based. When all IOPs have been initialized, the CPU may use RAM occupied by the SCB for another purpose.

Channel Commands

After initialization, any channel attention is interpreted as a command to channel 1 (SEL=low) or to channel 2 (SEL=high). As discussed in section 3.2, the channel attention, depending on the activities of both channels, may not be recognized immediately. The channel attention is latched, however, so that it will be serviced as soon as priorities allow.

When the channel recognizes the CA, it sets its BUSY flag in the CB to FFH. This does not prevent the CPU from issuing another CA, but provides status information only. In its response to a CA, the channel reads various control fields from system memory. It is the responsibility of the CPU to ensure that the appropriate fields are properly initialized before issuing the CA.

After setting its BUSY flag, the channel reads its CCW from the CB. It examines the command field (see figure 3-33) and executes the command encoded there by the CPU.



CF COMMAND FIELD
000 UPDATE PSW
001 START CHANNEL PROGRAM LOCATED IN I/O SPACE.
010 (RESERVED)
011 START CHANNEL PROGRAM LOCATED IN SYSTEM SPACE.
100 (RESERVED)
101 RESUME SUSPENDED CHANNEL OPERATION
110 SUSPEND CHANNEL OPERATION
111 HALT CHANNEL OPERATION

ICF INTERRUPT CONTROL FIELD
00 IGNORE, NO EFFECT ON INTERRUPTS.
01 REMOVE INTERRUPT REQUEST; INTERRUPT IS ACKNOWLEDGED.
10 ENABLE INTERRUPTS.
11 DISABLE INTERRUPTS.

B BUS LOAD LIMIT
0 NO BUS LOAD LIMIT
1 BUS LOAD LIMIT

P PRIORITY BIT

Figure 3-33. Channel Command Word Encoding

Figure 3-34 illustrates the channel's response to each type of command. Note that if CF contains a reserved value (010 or 100), the channel's response is unpredictable.

The CPU can use the "update PSW" command to alter the bus load limit and priority bits in the PSW (see figure 3-17) without otherwise affecting the channel. This command also allows the CPU to control interrupts originating in the channel; this topic is discussed in more detail later in this section.

The two "start program" commands differ only in their affect on the TP tag bit. If CF=001, the channel sets the tag to 1 to indicate that the program resides in the I/O space. If CF=011, the tag is cleared to 0, and the program is assumed to be in the system space. The channel converts the doubleword parameter block pointer to a 20-bit physical address and loads this into PP. It loads the doubleword task block (channel program) pointer into TP, updates the PSW as specified by the ICF, B and P fields of the CCW and starts the program with the instruction pointed to by TP.

The CPU may suspend a channel operation (either program execution or DMA transfer) by setting CF to 110. The channel saves its state (TP, its tag bit, and PSW) in the first two words of the parameter block (see figure 3-18 for format) and clears its BUSY flag to 0H. Note the following in regard to a suspended operation:

- The content of the doubleword pointer to the beginning of the channel program is replaced by the channel state save data. Therefore, a suspended operation may be resumed, but cannot be started from the beginning without recreating the doubleword pointer.

- TP is the only register saved by this operation. If another channel program is started on this channel, the other registers, including PP, are subject to being overwritten. In general, suspend is used to temporarily halt a channel, not to "interrupt" it with another program. Section 3.10 provides an example of a program that can be used to save another program's registers.

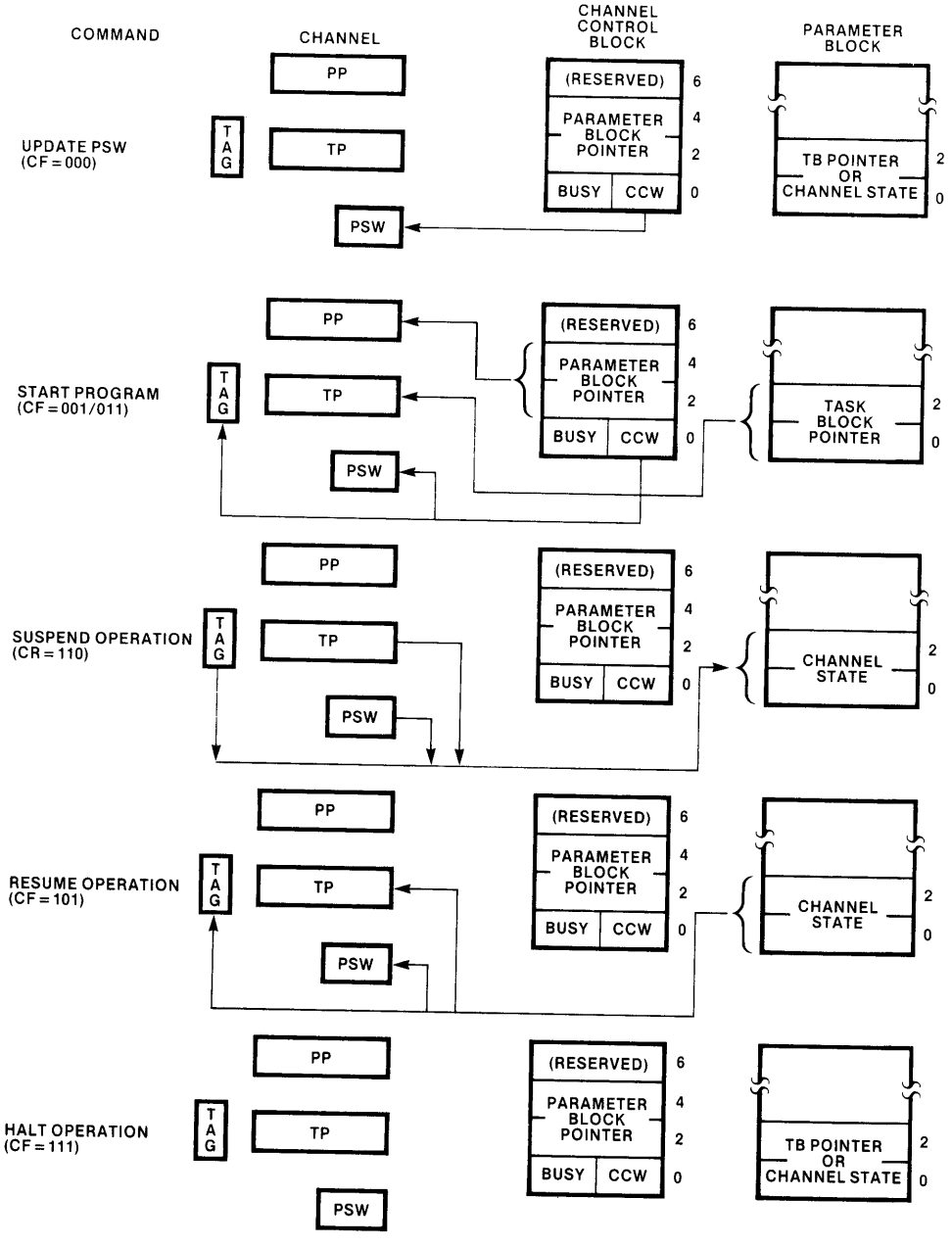


Figure 3-34. Channel Commands