To: Allen Morris
Re: Victor 9000 & DP101

Running DP101 produces the following error message:

Error in number of sectors per cluster

"   "   "   bytes per sector

Illegal number of FATS: (0 on flooppy)
(32 on hard disk)

The data in the boot area is not the same as returned by DOS function 54.

I show function 54 to be "Get verify status - returned in AL" assembling and running:

```
MOV AH,54
INT 21
INT 20
```

returns "0" in AL

Getting out my friendly set of disk tools, I find the following data that may be of help.

	Hard Disk(C)	Floppy(A)
Drive No	2	0
Unit	2	0
Sector Size	512	512
Cluster Size	16	4
Media Description By	00	01
Available space for DATA	2497 Clusters	594 Clusters
	20455424 Bytes	1216512 Bytes
Reserved Sectors	1	1
File Allocation Tables	2	2
Sectors per FAT	8	2
Directory Sectors	20	8
Max. Directory Entries	312	128

****

The following data is from the Victor Tech Ref Manual

MS-DOS allocates space on a single-sided diskette (SS) and a double sided (DS) diskette as follows:

Track 0 Sector 0		Disk Label
	Sectors 1-[illegible]	Two copies of the FAT, two sectors per FAT (SS)
	Sectors 1-4	Two copies of the FAT, two sectors per

	FAT (DS)
Sectors 3-10	Directory (SS)
Sectors 5-12	Directory (DS)
Sectors 11-	Data Region (SS)
Sectors 13-	Data Region (DS)

****

Victor 9000 Hard-DIsk Label Format

FIELD NAME	DATA	TYPE CONTENTS
Label_Type	WORD	0000=unqualified 0001=Current Revision
Device_ID	WORD	0001=current revision
Serial_number	Byte(16)	ASCII
Sector_Size	WORD	512
IPL_VECTOR		
Disk Address	DWORD	Logical Address
Load_Address	WORD	Paragraph Number
Load_Length	WORD	Paragraph Count
Cod_Entry	PTR	Memory Address
Primary_Boot_Volume		
	WORD	Virtual Volume #
Control_Parems	BYTE(16)	(For Tandem TM603SE)
# Cylinders	BYTE(Hi)	00H
	BYTE(lo)	E6H (=230)
# Heads	BYTE	06H (=6)
1st reduced-	BYTE(Hi)	00H
current cyl.	BYTE(Lo)	80H (=128)
1st write-	BYTE(Hi)	00H
precomp cyl.	BYTE(Lo)	80H (=128)
ECC Data burst	BYTE	0BH (=11)
Options	BYTE	02h (=2)
Interleave	BYTE	05H (=5, note that 0 means 5)
Spares	BYTE(6)	00H
Available_Media_List		
Region_Count	BYTE	Number of Regions
Region_Descr	(var)	Variable by region count
Region_PA	DWORD	Phisical Address
Region_Size	DWORD	Block Count
Working_Media_List		
Region_Count	BYTE	Number of Regions
Region_Descr	(var)	Variable by region count
Region_PA	DWORD	Phisical Address
Region_Size	DWORD	Block Count

Virtual_Volume_List		
Volume_Count	BYTE	Number of Virtual Volumes
Volume_Address	DWORD	Virtual volume label Logical Address

The above table describes those elements found in the hard-disk label, following is a discussion of the meaning of the entries themselves:

* Label Type - this defines the state of the driver layout and the revision number of the label

* Device ID - Classification identifying the arrangement, for example, the drive Mfg, controller revision number. This allows for the identification of compatible controller/drives.

* Serial Number - the serial number of the unit is stored here.

* Sector Size - the physical atomical unit of storage on the media

* Initial Program Load Vector (IPL) - this is a descriptor identifying the boot program and it's location on disk. This information is generated from the primary boot volume label via the utility HDSETUP.

  * Disk Address - The logical disk address of the boot program image

  * Load Address - the paragraph address of the memory where the boot program is to load. A zero entry indicates a default load at the highest RAM location.
  * Load Length - The length of the boot program in paragraphs.
  * Code Entry - a long memory address of the starting entry of the boot program. segment of zero defaults to the segment of the loaded program.

* Primary Boot Volume - the logical address of the virtual volume label containing the IPL vector and configueration information.

* Controller Parameters - a list of controller dependent information, for use in device reset and configuration.

* Available Media List - a list of permanent useable areas of the disk. This is derived from the available media list and from the format function of the HDSETUP utility.

* Physical Address - the disk address of the region
* Region Size - the number of physical blocks in the region.

* Working Media List - a list of the working areas of the disk. This is derived from the AVailable Media List and from the format function of the HDSETUP utility.
    * Physical Address - disk address of the region
    * Region Size - the number of physical blocks in the region

* Virtual Volume List - a list of the logical disk addresses of all virtual volume labels.

****

Victor 9000 Hard-Disk Virtual Volume Label Format

The Virtual Volume Label provides information on the structure of the Virtual Volume. Generally the operating system references this label, while the HDSETUP utility will create and reference it. The Virtual Volume Label appears as follows:

```
FIELD NAME           DATA        TYPE CONTENTS

Label_Type           WORD        0000=nul
Volume_Name          BYTE(16)    ASCII
IPL_Vector
     Disk Address    DWORD       Virtual Address
     Load_Address    WORD        Paragraph Number
     Load_Length     WORD        Paragraph_Count
     Code_Entry      PTR         Memory Address

Volume_Capacity      DWORD       # of physical blocks
Data_Start           DWORD       Virtual Address
Host_Block_Size      WORD        MS-DOS    = 512 bytes
Allocation_Unit      WORD        # of physical blocks
Number_Of_Directory_Entries
                     WORD        Entry count
Reserved             BYTE(16)    Future expansion - set to nulls

Configuration_Information
     Assignment_Count     BYTE # of assignment mappings
     Assignment           (var) Variable by assignment count
     Device_Unit     WORD        Physical Unit Number
     Volume_Index    WORD        Index into virtual volume list
```

The above table describes those elements found in the hard-disk Virtual Volume label, following is a discussion of the meanings of the entries themselves.

* Label Type - this defines the type of operating enviorment that the virtual volume is configured for. It is used for type checking when assigning volumes to

drives.

* Volume Name - the name of the virtual volume as defined by the user.

* Initial Program Load Vector - this is a descriptor identifying the boot program and it's location within the virtual volume. This field is used to generate the IPL vector on the drive label when configuering the primary boot volume.
    * Disk Address - the virtual disk address of the boot program image.
    * Load Address - the paragraph address of the memory where the boot program is to load. A zero entry indicates a default load to the highest RAM location.
    * Load Length - the length of the boot program in paragraphs
    * Code Entry - a long memory address to the starting entry of the boot program. Segment of zero defaults to the segment of the loaded program.

* Volume Capacity - the number of actual blocks that comprise the virtual volume.
* Data Start - the offset in blocks into the virtual volume for the start of the data space.
* Host Block Size - The atomical unit used by the host in data transfer operations.
* Allocation Unit (AU) - this operating system dependent field means the storage allocation size used by the host in the virtual volume. It is used in determining disk parameter tables and disk definitions.
* Number of Directory Entries - this operating system dependent field means the number of entries in the hosts directory. It is used in determine disk parameter tables and disk definitions.
* Configueration Information - a list of the drive assignments for a system at boot time. It is used to map logical drives to virtual volumes. This filed is referenced via the label of the booted drive.

****

The above spelling errors are mine - not Victors. Besidex the memap you downloaded, this should give you all the info I can fins on the Victor disks.

Victor does have their own Super-Bios which I will upload also.

If you need any additional information, please let me know and I'll see what I can find. My home tel is 349-3602 or leave word here.

Franz Hirner

Hardware Reference Manual

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

Second VICTOR printing April, 1983.

ISBN 0-88182-065-2

Printed in U.S.A.

CONTENTS

FIGURES

TABLES

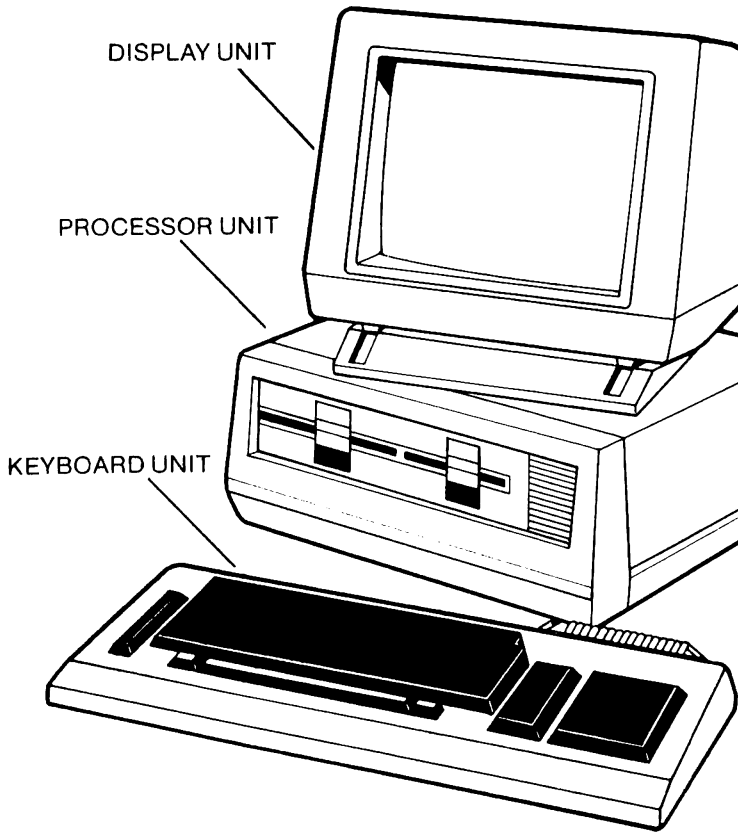
1

SYSTEM DESCRIPTION

1. SYSTEM DESCRIPTION

The system is designed for maximum operator comfort and comfort and ease of use. The system is composed of three modules, and occupies the desk space normally needed for an office typewriter. Its modules are: the processor unit, the display unit, and the keyboard unit. Coiled cables interconnect these stand-alone modules, allowing easy positioning and mobility. A standard configuration is shown in Figure 1.

Figure 1: Typical Arrangement of Main Units



The system can be connected to a wide variety of peripherals and accommodates local and long distance communications. Standard interfaces include a parallel port (Centronics or IEEE-488), programmable RS-232(V-24) channels, an internal control port, and an audio controller for digitized voice and tone output.

PROCESSOR UNIT

The processor unit physically supports the display unit, as shown in Figure 1. The main logic, disk drives, and power supply are housed in the processor unit. The two integral single-sided 5 1/4-inch floppy disk drives store up to 1.2 megabytes of information. The system incorporates a minimum 128K bytes of random access memory (RAM), expandable to 512K bytes.

DISPLAY UNIT

The display unit swivels and tilts to permit optimum adjustment of the viewing angle, and the unit incorporates a 12-inch antiglare screen to prevent eye strain. The display is 25 lines; each line has 80 characters. Characters are formed in a 10-x-16 font cell, providing a high resolution display. A bit-mapped graphics mode with 800-x-400-dot matrix screen resolution is available under software control. Software also controls the overall screen brightness, character contrast, and audio volume.

KEYBOARD UNIT

The keyboard unit is designed for comfort and ease of operation. It is completely software definable and features several keys that are specifically designed for special-function use in application programs. The keyboard contains separate typewriter and numeric/calculator keypad configurations, double-size general-function keys, special-function keys, and editing and cursor-control keys. A cluster of keys is also used to manipulate screen brightness, character contrast, and audio volume.

2

PROCESSOR UNIT

2. PROCESSOR UNIT

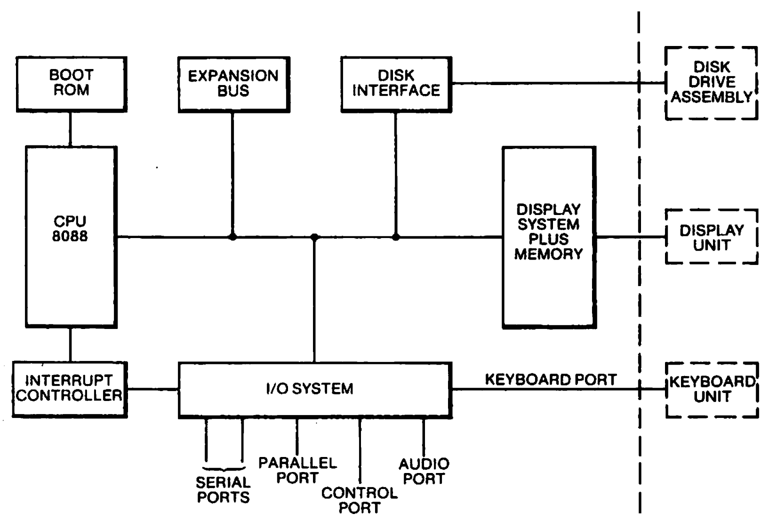
The heart of the processor unit is the Intel 8088 microprocessor. This processor is a version of the Intel 16-bit 8086 processor that contains an 8-bit bus interface. The 8088 is software-compatible with the 8086, and thus supports 16-bit operations, including multiply and divide. The processor has a 20-bit physical address space, providing 1 megabyte of addressable memory I/O.

As indicated earlier, the processor unit is the module that physically supports the display unit. It contains three basic assemblies: the main logic board, the disk drive assembly, and the power supply.

MAIN LOGIC BOARD

As shown in Figure 2, the main logic board is comprised of the central processing unit (CPU) section, the input/output (I/O) section, the display section, the disk interface section, and the expansion bus.

Figure 2: Main Logic Block Diagram



8088 CENTRAL PROCESSING UNIT (CPU)

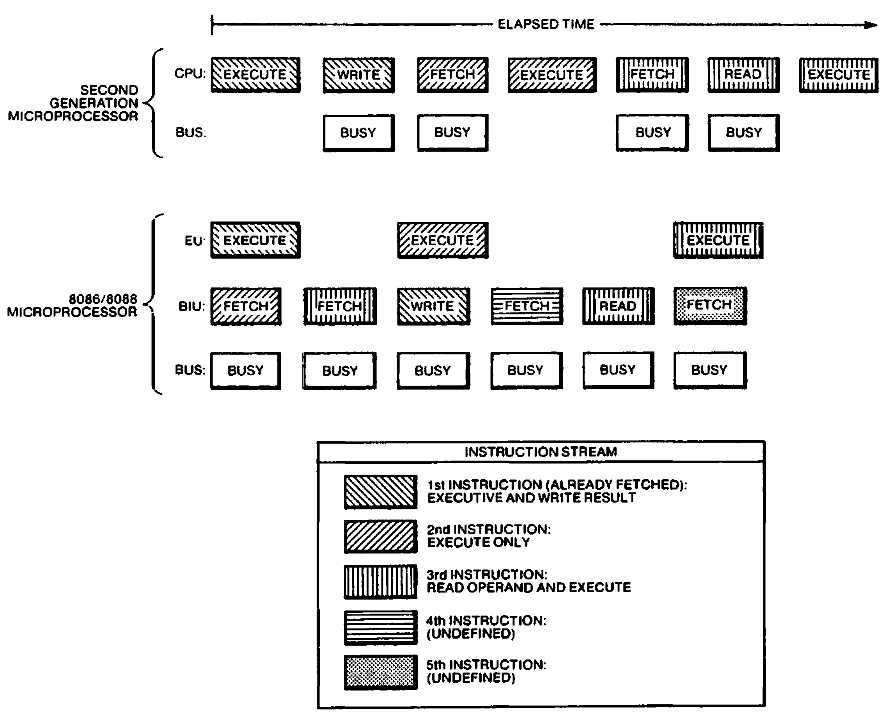
Microprocessors execute programs by cycling through the following four steps:

1. Fetch the next instruction from memory.
2. Read an operand (if required by the instruction).
3. Execute the instruction.
4. Write the result (if required by the instruction).

These steps have historically been performed in a series or with a single bus cycle fetch overlap. The architecture of the 8088 CPU allocates the same steps to two separate processing units within the CPU. The execution unit (EU) executes instructions. The bus interface unit (BIU) fetches instructions, reads operands, and writes results.

The two units operate independently of each other, thus allowing overlap of instruction-fetch activity and instruction-execution activity. The time required to fetch instructions "disappears" because it no longer impacts instruction execution time; the next instruction to be executed by the EU has always already been fetched by the BIU. Figure 3 provides an example which illustrates this overlap and compares it to traditional microprocessor operation. In the example, overlapping reduces the elapsed time required to execute three instructions, and, during that execution time, allows two additional instructions to be fetched.

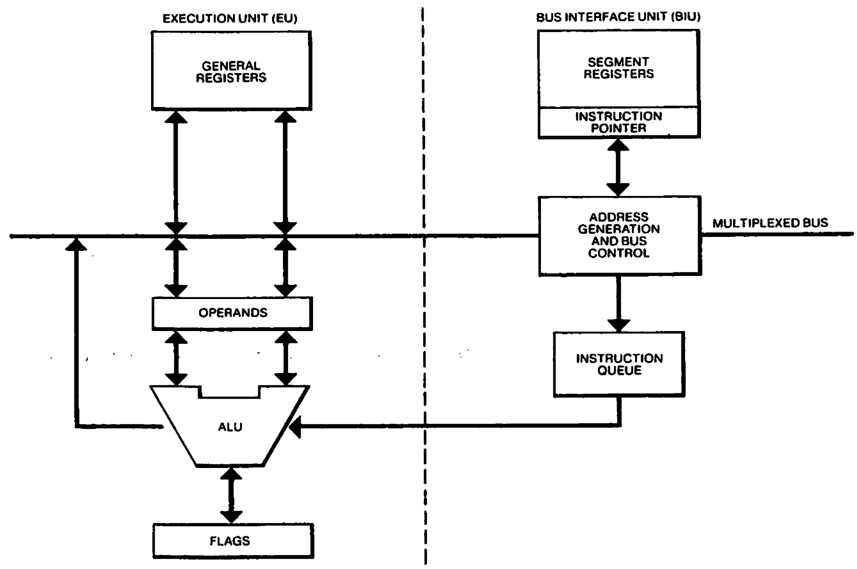
Figure 3: Overlapped Instruction Fetch and Execution



Execution Unit

All registers and data paths in the EU are 16 bits wide, providing for fast internal transfers. CPU status and control flags are maintained in the EU by a 16-bit arithmetic/logic unit (ALU) that manipulates the general registers and the instruction operands (Figure 4).

Figure 4: Execution and Bus Interface Units



The EU is not connected to the outside world via the system bus. It obtains instructions from a queue maintained by the BIU. When an instruction requires access to memory or to a peripheral device, the EU sends a request to the BIU to store or obtain the data. The BIU performs an address relocation that gives the EU access to a full megabyte of memory space.

Bus Interface Unit

The BIU performs all bus operations for the EU. Upon demand from the EU, the BIU transfers data between the CPU and the memory or an I/O device.

While the EU is executing instructions, the BIU fetches instructions from memory. The instructions are stored in an internal RAM array called the instruction stream queue. The 8088 instruction queue holds up to four bytes of the instruction stream. The queue size is sufficient to allow the BIU to keep the EU supplied with fetched instructions without monopolizing the system bus. The BIU fetches another instruction byte whenever: (1) one byte in the queue is empty and (2) there is no active request for bus access (Figure 3).

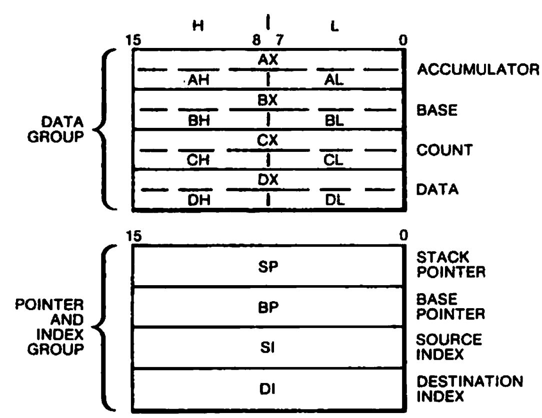
The instruction queue usually contains at least one byte of the instruction stream; the EU does not have to wait for instructions to be fetched. The instructions in the queue are those stored in the memory locations immediately adjacent to and higher than the instruction currently being executed. That is, the queue contains the next logical instructions, as long as execution proceeds serially. If the EU executes an instruction that transfers control to another location, the BIU resets the queue, fetches the instruction from the new address, passes it immediately to the EU, and then begins refilling the queue from the new location.

The BIU suspends instruction fetching whenever the EU requests a memory or I/O read or write. A fetch already in progress is completed before the EU's bus request is executed.

General Registers

The 8088 has eight 16-bit general registers (Figure 5). The general registers are divided into two sets of four registers: the data registers called the H&L group (H&L stands for "high and low"), and the pointer and index registers which are called the P&I group.

Figure 5: General Registers



The data registers are unique in that their upper (high) and lower halves are separately addressable. Each data register can be used interchangeably as a 16-bit register or as two 8-bit registers. However, the CPU registers are always accessed as 16-bit units. Data registers can be used without constraint in most arithmetic and logic operations. Certain instructions use specified registers implicitly (see Table 1), allowing compact, powerful encoding.

Table 1: Implicit Use of General Registers

REGISTER	OPERATIONS
AX	Word multiply, word divide, word I/O
AL	Byte multiply, byte divide, byte I/O, translate, decimal arithmetic
AH	Byte multiply, byte divide
BX	Translate
CX	String operations, loops
CL	Variable shift and rotate
DX	Word multiply, word divide, indirect I/O
SP	Stack operations
SI	String operations
DI	String operations

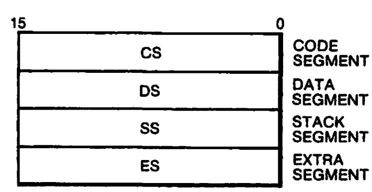
The pointer and index registers can also participate in most arithmetic and logic operations. All eight general registers fit the definition of "accumulator," as used with first and second generation microprocessors. The P&I registers (except for the BP register) are also used implicitly in some instructions, as shown in Table 1.

Segment Registers

One megabyte of memory space is divided into logical segments of up to 64K bytes each. The CPU has direct access to four segments at a time. The starting location (the base address) of each segment, is contained in the segment registers (see Figure 6). The CS register points to the current code segment; instructions are fetched from this segment. The SS register points to the current stack segment; stack operations are performed on locations in this segment. The DS register points to the current data segment and generally contains program variables. The ES register points to the current extra.

The segment registers can be accessed by programs and manipulated with several instructions.

Figure 6: Segment Registers



Instruction Pointer

The 16-bit instruction pointer (IP) is similiar to the program counter (PC) in the 8080/8085 CPUs. The IP points to the next instruction. It is updated by the BIU so that it contains the offset (distance in bytes) of the next instruction from the beginning of the current code segment. During normal execution, the IP contains the offset of the next instruction to be fetched by the BIU. Whenever the IP is saved on the stack, it is automatically adjusted to point to the next instruction to be executed. Programs do not have direct access to the IP; however, instructions cause the IP to change and to be saved on and restored from the stack.

Flags

The 8088 has six 1-bit status flags that the EU posts (Figure 7). The flags reflect specified properties of the result of an arithmetic or logic operation. Different instructions affect the status flags differently. Another group of instructions is available that allows a program to alter its execution, depending on the result of a prior operation. This result is indicated by the state of these flags. Examples of conditions reflected by the flags are described below:

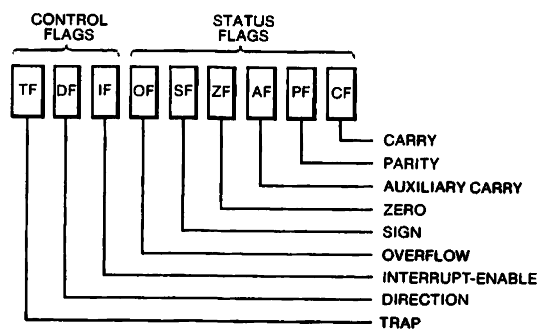
- The auxiliary carry flag (AF) is set when a carry out of the low nibble into the high nibble or a borrow from the high nibble into the low nibble of an 8-bit quantity (low-order byte of a 16-bit quantity) has occurred. This flag is used by decimal arithmetic instructions.
- The carry flag (CF) is set when a carry out of, or a borrow into, the high-order bit of the result (8- or 16-bit) has occurred. This flag is used by instructions that use the CF to add and subtract multibyte numbers. Rotate instructions also isolate a bit in memory or in a register by placing it in the CF.
- The overflow flag (OF) is set when an arithmetic overflow has occurred; that is, a significant digit has been lost (i.e., the size of the result exceeded the capacity of its destination location). An interrupt on overflow instruction is available to generate an interrupt in an arithmetic overflow.
- The sign flag (SF) is set when a result's high-order bit is a 1. Negative binary numbers are represented in the 8088 in standard two's complement notation. SF indicates the sign of the result (0=positive, 1=negative).
- The parity flag (PF) is set when the result has even parity (an even number of 1-bits).
- The zero flag (ZF) is set when the result of the operation is 0.

Three additional control flags (Figure 7) can be set and cleared by programs to alter processor operations:

- Setting the direction flag (DF) causes string instructions to auto-decrement (to process strings from high addresses to low maskable) interrupt requests. Clearing IF disables these interrupts. IF has no affect on nonmaskable interrupts generated externally or internally.

- Setting the trap flag (TF) puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes each instruction.

Figure 7: Flags

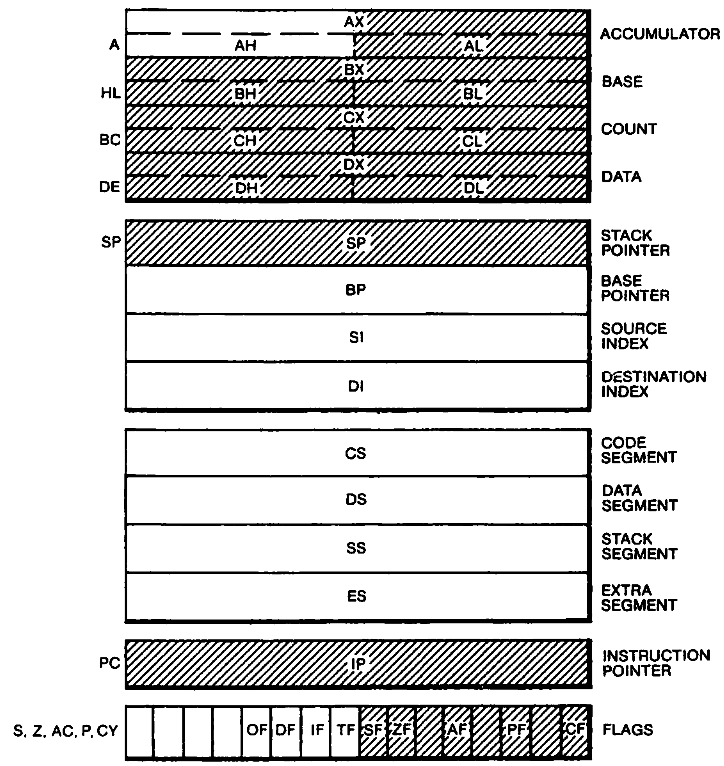


8080/8085 Register and Flag Correspondence

The registers, the flags, and the program counter in the 8080/8085 CPUs have counterparts in the 8088 CPU (see Figure 8). The A register (accumulator) in the 8080/8085 corresponds to the AL register in the 8088. The 8080/8085 H&L, B&C, and D&E registers correspond to registers BH, BL, CH, CL, DH, and DL, respectively, in the 8088. The 8080/8085 stack pointer (SP) and program counter (PC) correspond to the 8088 SP and IP.

The AF, CF, PF, SF, and ZF flags are the same in both CPU families. The remaining 8088 flags and registers are unique to the 8088. The 8080/8085 to 8088 mapping allows direct translation of most existing 8080/8085 program code into 8088 program code.

Figure 8: 8080/8085 Register Subset



Memory

The 8088 has 1,048,576 bytes of address space. This section describes how memory is functionally organized and used.

STORAGE ORGANIZATION The 8088 memory storage space is organized as an array of 8-bit bytes (see Figure 9). Instructions, byte data, and word data may be stored at any byte address, regardless of alignment. This technique saves storage space because code can be densely packed in memory (see Figure 10).

Figure 9: Storage Organization

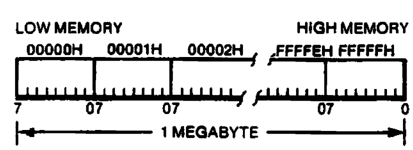
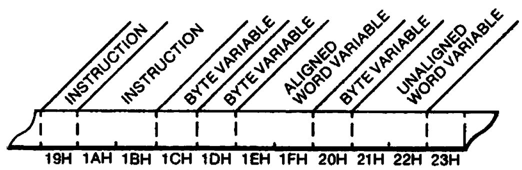


Figure 10: Instruction and Variable Storage



The most-significant byte in word data is always stored in the higher memory location (see Figure 11). This storage convention is "invisible" to the user except when the user monitors the system bus or reads memory dumps. A special class of data is stored as double words (i.e., two consecutive words) called pointers, which are used to address data and code outside the currently-addressable segments. The lower-addressed word of a 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 (see Figure 12).

Figure 11: Storage of Word Variables

724H		725H		
0	2	5	5	HEX
0000	0010	0101	0101	BINARY

VALUE OF WORD STORED AT 724H: 5502H

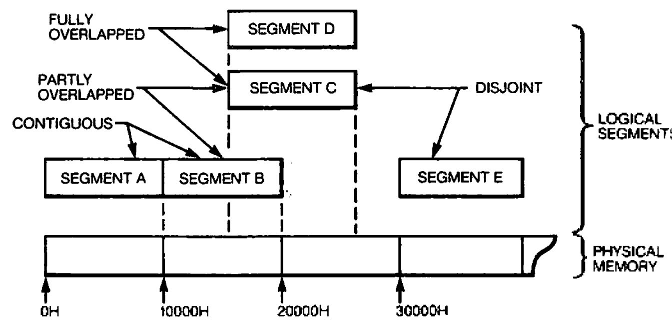
Figure 12: Storage of Pointer Variables

4H		5H		6H		7H		
6	5	0	0	4	C	3	B	HEX
0110	0101	0000	0000	0100	1100	0011	1011	BINARY

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

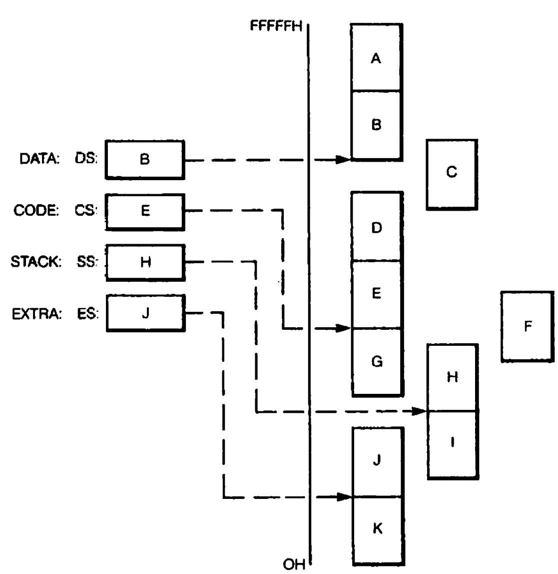
SEGMENTATION 8088 programs view the megabyte of memory space as a group of segments defined by the application. A segment is a logical unit of memory up to 64K bytes long. Each segment contains contiguous memory locations and is an independent, separately-addressable unit. Software assigns each segment a base address, which is the segment's starting location in the memory space. All segments begin on 16-byte memory boundaries. The segments can be disjoint, partially overlapped, or fully overlapped (see Figure 13). A physical memory location can be mapped into (contained in) one or more logical segments.

Figure 13: Segment Locations in Physical Memory



The segment registers contain (point to) the base address values of the four currently addressable segments (see Figure 14). Programs access code and data in other segments by changing the segment registers to point to the segments containing the needed code or data.

Figure 14: Currently Addressable Segments



Individual applications define and use segments differently. The currently-addressable segments provide a generous work space: 64K bytes for code, a 64K byte stack, and 128K bytes of data storage. Many applications can be written that simply initialize the segment registers and then forget them. However, large applications should be designed with careful consideration given to segment definition.

The segmented structure of the 8088 memory space supports modular software design and discourages the development of huge, monolithic programs.

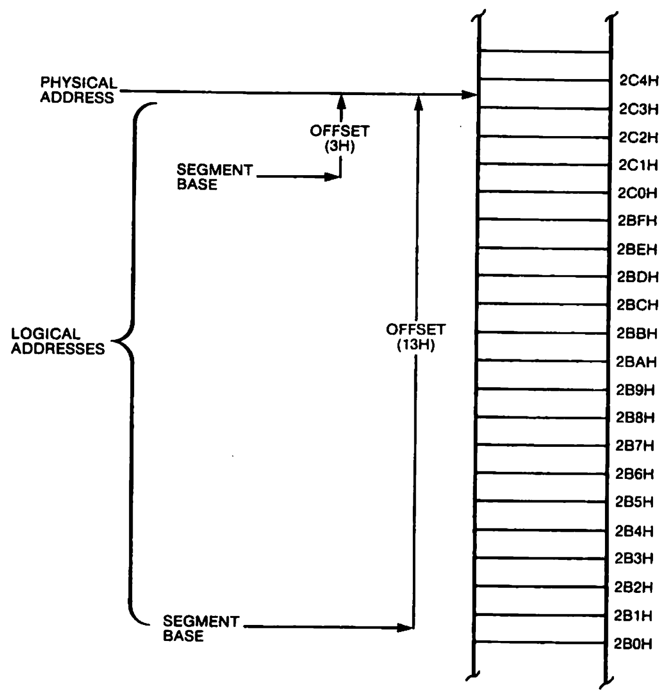
The segments can be used to advantage in many programming situations—for example, when programming an editor for several on-line terminals. A 64K text buffer (probably an extra segment) could be assigned to each terminal. A single program could maintain all the buffers by simply changing register ES to point to the buffer of the terminal requiring service.

PHYSICAL ADDRESS GENERATION There are two kinds of memory location addresses: physical and logical. A physical address is a 20-bit value that identifies each byte location in the megabyte memory space. Physical-address range varies from OH through FFFFFH. All exchanges between the CPU and memory components use physical addresses.

Programs use logical addresses, which allow code to be developed before the code is assigned physical addresses. This technique facilitates dynamic management of memory resources.

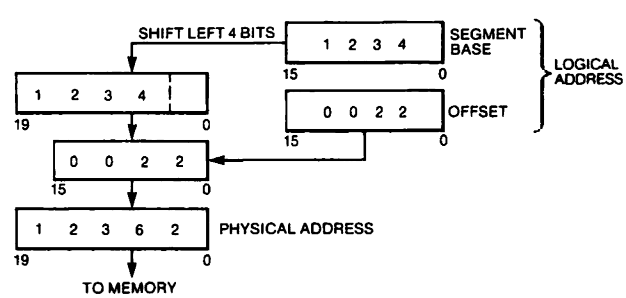
A logical address consists of two values: a segment-base value and an offset value. The segment-base value for any memory location is the value that defines the first byte of the segment. The offset value is the number of bytes from the beginning of the segment to the target location. Segment-base and offset values are unsigned 16-bit quantities. The lowest addressed byte in a segment has an offset value of 0. Different logical addresses can map to the same physical location, as shown in Figure 15. The physical memory location 2C3H shown in Figure 15 is contained in two different overlapping segments, one beginning at 2B0H and the other at 2C0H.

Figure 15: Logical and Physical Addresses



When the BIU accesses memory to fetch an instruction, or to obtain or store a variable, it generates a physical address from a logical address. It does this by (1) shifting the segment-base value four bit positions, and (2) adding the offset value, as illustrated in Figure 16. This addition process results in modulo 64K addressing, which causes addresses to wrap around from the end of a segment to the beginning of the same segment.

Figure 16: Physical Address Generation



The BIU obtains the logical address of a memory location from different sources, depending on the type of reference that is being made (see Table 2). Instructions are always fetched from the current code segment. The IP contains the offset of the target instruction from the beginning of the segment. Stack instructions always operate on the current stack segment. The SP contains the offset of the top of the stack. Most memory operands reside in the current data segment, although the program can instruct the BIU to access a variable in one of the other currently addressable segments. The offset of a memory variable is calculated by the EU; the calculation is based on the addressing mode specified in the instruction, and the result is called the operand's effective address (EA).

Table 2: Logical Address Sources

TYPE OF MEMORY REFERENCE	DEFAULT SEGMENT BASE	ALTERNATE SEGMENT BASE	OFFSET
Instruction fetch	CS	NONE	IP
Stack operation	SS	NONE	SP
Variable (except following)	DS	CS, ES, SS	Effective address
String source	DS	CS, ES, SS	SI
String destination	ES	NONE	DI
BP used as base register	SS	CS, DS, ES	Effective Address

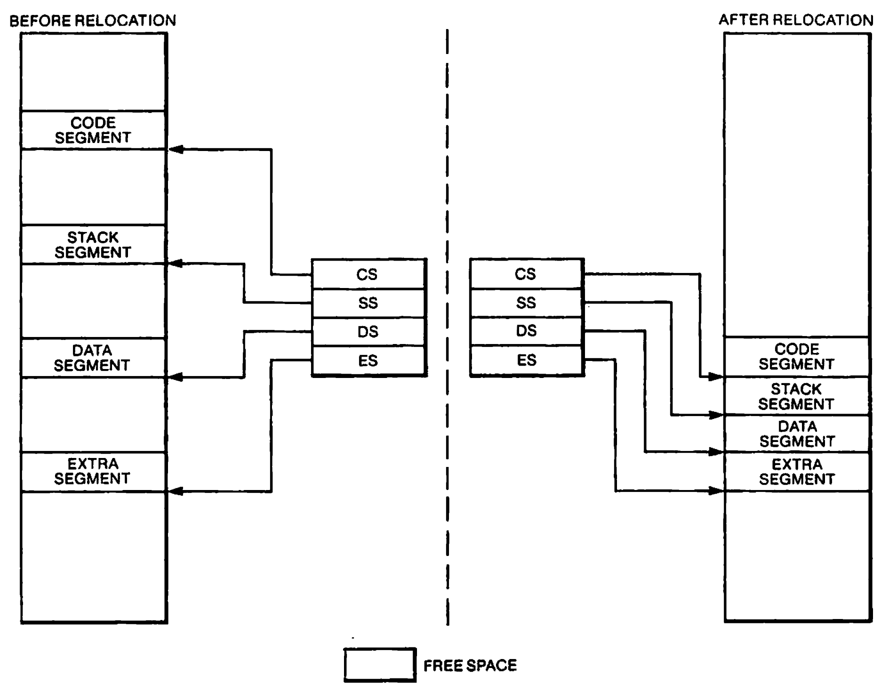
Strings are addressed differently than other variables. The source operand of a string instruction usually lies in the current data segment; however, another currently-addressable data segment may be specified. The source operand's offset is taken from register SI (the source index register). The destination operand of a string instruction always resides in the current extra segment, and its offset is taken from DI (the destination index register). The string instructions automatically adjust SI and DI as they process the strings one byte or word at a time.

When register BP (the base pointer register) is designated as a base register in an instruction, the variable is assumed to reside in the current stack segment. Using register BP is a convenient way to address data on the stack. The BP register can be used to access data in any of the other currently addressable segments.

Programmers usually find the segment assumptions of the BIU convenient to use. A programmer can, however, direct the BIU to access a variable in any of the currently-addressable segments by preceding an instruction with a segment override prefix. This 1-byte machine instruction tells the BIU which segment register to use to access a variable referenced in the following instructions. The only exception to this is a string instruction's destination operand, which must be located in the extra segment.

DYNAMICALLY RELOCATABLE CODE Dynamically relocatable—or position-independent—programming is made possible by the segmented memory structure of the 8088. The dynamic relocation technique makes effective use of available memory by taking advantage of the system's multiprogramming/multitasking capabilities. Inactive programs can be written to disk, making the space they occupied available to other programs. A disk-resident program can be read back into any available memory location and restarted. When a program needs a large contiguous block of storage and only nonadjacent fragments are available, other program segments can be compacted to free up a contiguous space (Figure 17).

Figure 17: Dynamic Code Relocation



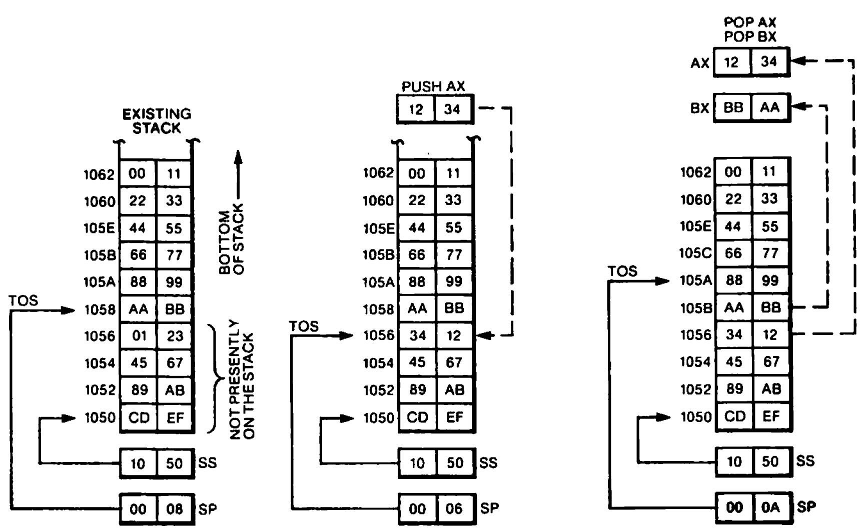
To be dynamically relocatable, all offsets in the program must be relative to fixed values contained in the segment registers. This allows the program to be moved anywhere in memory as long as the segment registers are updated to point to the new base addresses. A dynamically relocatable program must not load or alter its segment registers and must not transfer directly to a location outside the current code segment.

STACK IMPLEMENTATION Stacks in the 8088 are implemented in memory. They are located by the SS (the stack segment register) and the SP (the stack pointer register). A system may have an unlimited number of stacks. Each may be the maximum length of a segment, 64K bytes.

Attempting to expand a stack beyond 64K bytes overwrites the beginning of the stack. Only one stack is directly addressable at a time; this stack is the current stack, often referred to simply as "the" stack. SS contains the base address of the current stack. SP contains the offset of the top of the stack from the stack segment's base address. The stack's base address (contained in SS) is not the "bottom" of the stack.

Stacks are 16 bits wide. Instructions that operate on a stack add and remove stack items one word at a time. An item is pushed onto the stack (see Figure 18) by decrementing SP by 2 and writing the item at the new TOS (top of stack). An item is popped off the stack by copying it from TOS then incrementing SP by 2. In other words, the stack grows down in memory toward its base address. Stack operations never move or erase items on the stack. The TOS changes only as a result of updating the stack pointer.

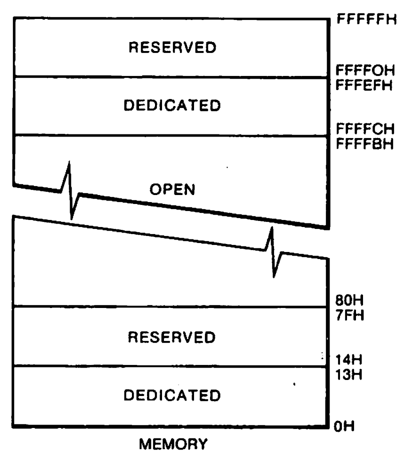
Figure 18: Stack Operation



STACK OPERATION FOR CODE SEQUENCE
PUSH AX
POP AX
POP BX

DEDICATED AND RESERVED MEMORY LOCATIONS Two areas in extremely low and high memory—0H through 7FH (128 bytes) and FFFF0H through FFFFFH (16 bytes)—are dedicated to specific processor functions or are reserved for use by hardware and software products (Figure 19). These areas are reserved for interrupt and system reset processing, and should not be used for any other purpose.

FIGURE 19: Reserved and Dedicated Memory



8086/8088 MEMORY ACCESS The 8088 always accesses memory in bytes. Word operands are accessed in two bus cycles, regardless of their alignment. Instructions are also fetched one byte at a time. Although word operand alignment does not affect performance, locating 16-bit data on even addresses ensures maximum throughput if the system is transferred to an 8086.

Input/Output

MEMORY-MAPPED I/O I/O devices may be placed in the 8088 memory space. The CPU cannot tell the difference between I/O devices as long as each device responds as a memory component.

Memory-mapped I/O provides programming flexibility. Instructions that normally reference memory may be used to access an I/O port located in the memory space. The move (MOV) instruction, for example, can transfer data between any 8088 register and a port. AND, OR, and TEST instructions may be used to manipulate bits in I/O device registers. Memory-mapped I/O takes advantage of the 8088 memory addressing modes. For example, a group of terminals can be treated as an array in memory with an index register selecting a terminal in the array.

However, a price is paid for the added programming flexibility that memory-mapped I/O provides. Dedicating part of the memory space to I/O devices reduces the number of addresses available for memory (although with a megabyte of memory space this should rarely be a constraint). Also, memory reference instructions take longer to execute and are less compact than simpler IN and OUT instructions.

DIRECT MEMORY ACCESS The 8088 provides hold (HOLD) and hold acknowledge (HLDA) signals that are compatible with traditional DMA controllers. By activating HOLD, a DMA controller can request use of the bus for direct transfer of data between an I/O device and memory. The CPU responds by completing the current bus cycle (if one is in progress) and then issuing HLDA, which grants the bus to the DMA controller. The CPU does not attempt to use the bus until HOLD goes inactive.

WAIT AND TEST The 8088 can be synchronized to an external event with the WAIT (wait for TEST) instruction and the TEST input signal. When the EU executes a WAIT instruction, the result depends on the state of the TEST input line. If TEST is not connected to or receiving an external signal, the processor enters an idle state and repeatedly retests the TEST line at 5-clock intervals. If TEST is connected to an external signal source, execution continues with the instruction following the WAIT.

The TEST input is connected to a "byte ready" signal from the floppy disk controller. This allows the processor to synchronize data transfer operations.

Processor Control And Monitoring — Interrupts

Microcomputer system design requires that I/O devices such as keyboards, displays, sensors, and other components receive efficient servicing to ensure that the microcomputer can perform a large number of system tasks with little or no effect on throughput.

One desirable method for ensuring efficient servicing is to allow the microprocessor to execute its main program, stopping to service peripheral devices only when told to do so by the device itself. In effect, this method provides an external asynchronous input which informs the processor to complete whatever instruction is currently being executed and to fetch a new routine to service the requesting device. Once this servicing is complete, the processor resumes exactly where it left off.

The 8088 interrupt system is a simple and versatile interrupt system. Every interrupt is assigned a type code that identifies it to the CPU. The 8088 can handle up to 256 different interrupt types. Interrupts may be initiated by devices external to the CPU, or they may be triggered by software interrupt instructions and, under certain conditions, by the CPU itself, as illustrated in Figure 20. Figure 21 illustrates the basic response of the 8088 to an interrupt. The next sections elaborate on the information presented in Figure 21.

Figure 20: Interrupt Sources

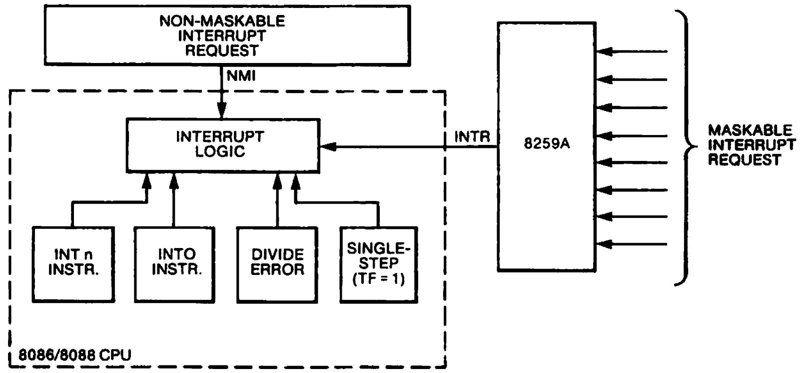
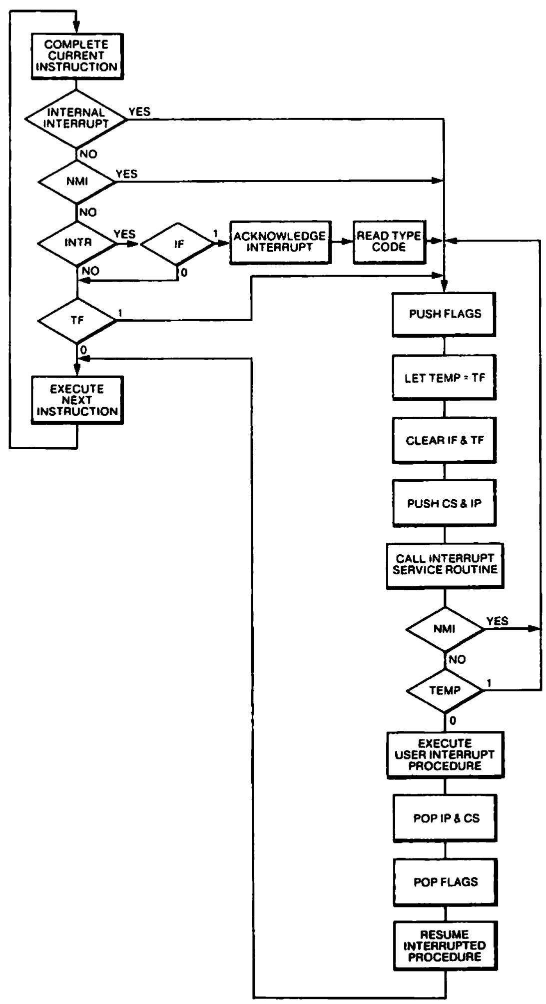


Figure 21: Interrupt Processing Sequence



EXTERNAL INTERRUPTS External devices can use two lines in the 8088 to signal interrrupts: interrupt request (INTR) and nonmaskable interrupt (NMI). The INTR line is driven by an 8259A programmable interrupt controller (PIC). The PIC is a flexible circuit controlled by software commands from the 8088.

The PIC appears as a set of I/O ports to the software and connects to devices that need interrupt services. It accepts interrupt requests from the attached devices and determines which service request has the highest priority. If the device selected for service has a higher priority than the one currently being serviced, the PIC activates the 8088 INTR line.

The CPU response to the active INTR line is based on the state of the interrupt-enable flag (IF). The currently-executing instruction is completed before the interrupt becomes active.

Occasionally, an interrupt request is not recognized until after the following instruction. Repeat, LOCK, and segment override prefixes are considered part of the instructions they prefix. Therefore, no interrupt is recognized between execution of a prefix and an instruction.

A move (MOV) to a segment register instruction and a POP segment register instruction are treated similarly (no interrupt is recognized until after the following instruction). This mechanism protects a program that is changing to a new stack (by updating SS and SP). The processor pushes the CS and IP flags into the wrong area of memory if an interrupt is recognized after SS has been changed, but before SP has been altered.

If a segment register and another value must be updated together, first the segment register must be changed, and then the instruction changing the other value must be given.

An interrupt request is recognized in the middle of an instruction in two instances—WAIT and repeated string instructions. In these cases, interrupts are accepted after any completed primitive operation or wait test cycles.

IF is clear when the interrupts signaled on INTR are masked or disabled, in which case the CPU ignores the interrupt request and processes the next instruction. The INTR signal is not latched by the CPU. It must be held active until a response is received or the request is withdrawn. When IF is set—enabling interrupts on INTR—the CPU recognizes the interrupt request and processes it. Interrupt requests arriving on INTR are enabled by executing a set interrupt-enable flag (STI) instruction, and disabled by executing a clear interrupt-enable flag (CLI) instruction. Writing commands to the 8259A (the PIC chip) selectively masks some of these requests. STI and IRET instructions re-enable interrupts only after the end of the following instruction, which reduces excessive stack buildup.

The CPU acknowledges an interrupt request by executing two consecutive interrupt acknowledge (INTA) bus cycles. Bus hold requests are not honored until INTA cycles are completed. The first INTA cycle signals to the 8259A that the request has been honored. The 8259A responds during the second INTA cycle by placing the interrupt byte containing the interrupt type (0-255) associated with the requesting device on the data bus. (Type assignment is made when the 8259A is initialized by software in the 8088.) The CPU uses this type code to call the indicated interrupt procedure.

A nonmaskable interrupt (NMI) request can arrive on another CPU line from an external source. This edge-triggered line signals to the CPU that a catastrophic event—such as the imminent loss of power, a memory error detection, or a bus parity error—has occurred. Interrupt requests arriving on NMI cannot be disabled. They are latched by the CPU and have a higher priority than an interrupt requested on INTR (level-triggered). NMI is first recognized when an interrupt request arrives on both lines during execution of an instruction. Nonmaskable interrupts are predefined as type 2. The processor does not need a type code to call the NMI procedure and does not run the INTA bus cycles in response to an NMI request.

The time required for the CPU to recognize an external request is determined by the number of clock cycles remaining to complete the instruction currently being executed. This delay is referred to as interrupt latency. The longest possible interrupt latency occurs when an interrupt request arrives during multiplication, divison, variable-bit shift, or rotate instruction execution. In the most extreme case, interrupt latency spans two instructions, rather than one.

INTERNAL INTERRUPTS Execution of an interrupt (INT) instruction generates an immediate interrupt. The interrupt type code identifies the procedure needed to process the interrupt. Since any type code can be specified, software interrupts can be used to test interrupt procedures that are written to service external devices.

When the overflow flag (OF) is set, an interrupt on overflow (INTO) instruction (a type 4 interrupt) is initiated immediately after the completion of the currently executing instruction. The CPU generates a type 0 interrupt following execution of a divide (DIV) instruction or an integer divide (IDIV) instruction when the calculated quotient is larger than the specified destination. When the trap flag (TF) is set, the CPU automatically generates a type 1 interrupt after every instruction. This single-step execution, which is a powerful debugging tool, is discussed in more detail later.

All internal interrupts (INT, INTO, divide-error, and single step) share these characteristics:

- The interrupt type code is contained in the instruction or is predefined.
- No INTA bus cycles are run.

- ▶ Except for single-step interrupts, internal interrupts cannot be disabled.
- ▶ Internal interrupts (except single-step) have higher external interrupts (see Table 3). When interrupt requests arrive on NMI and/or INTR during execution of an instruction that causes an internal interrupt (e.g., a divide error), the internal interrupt is processed first.

Table 3: Interrupt Priorities

INTERRUPT	PRIORITY
Divide error, INT n, INTO	Highest
NMI	
INTR	
Single-step	Lowest

INTERRUPT POINTER TABLE The interrupt pointer (or interrupt vector) table links an interrupt type code and its associated service procedure. The interrupt pointer table occupies the first 1K bytes of low memory. There may be up to 256 entries in the table, one for each interrupt type that can occur in the system. Each entry in the table is a double-word pointer containing the address of the procedure servicing interrupts of that type. The higher-addressed word of the segment contains the procedure. The lower-addressed word contains the procedure's offset from the beginning of the segment. Each entry is four bytes long; the CPU calculates the location of the correct entry for a given interrupt type by simply multiplying the type number by 4.

In applications that do not recognize interrupt types, space at the high end of the table can be used for other purposes.

The 8088 activates an interrupt procedure by executing the equivalent of an intersegment indirect CALL instruction after pushing the flags onto the stack. The address contained in the interrupt pointer table element located at n × 4 (where "n" represents the type number) is the target of the CALL. The CPU saves the address of the next instruction by pushing CS and IP onto the stack. It transfers control to the interrupt procedure by replacing the second and first words of the table element.

The processor activates the interrupt procedures in priority order when multiple interrupt requests arrive simultaneously. Figure 22 shows how procedures would be activated in an extreme case. The processor is running in single-step mode with external interrupts enabled. INTR is activated during execution of a divide instruction. The instruction generates a divide error interrupt. Except for INTR, the interrupts are recognized in the order of their priorities (see Figure 23). INTR is not recognized until after the following instruction because recognition of the earlier interrupts cleared IF. If an earlier response to INTR is desired, interrupts can be re-enabled in any of the interrupt response routines.

All main-line code is executed in single-step mode (Figure 22). The processing speed (full speed or single-step mode speed) can be selected in each occurrence of the single-step routine because of the order of interrupt processing.

Figure 22: Processing Simultaneous Interrupts

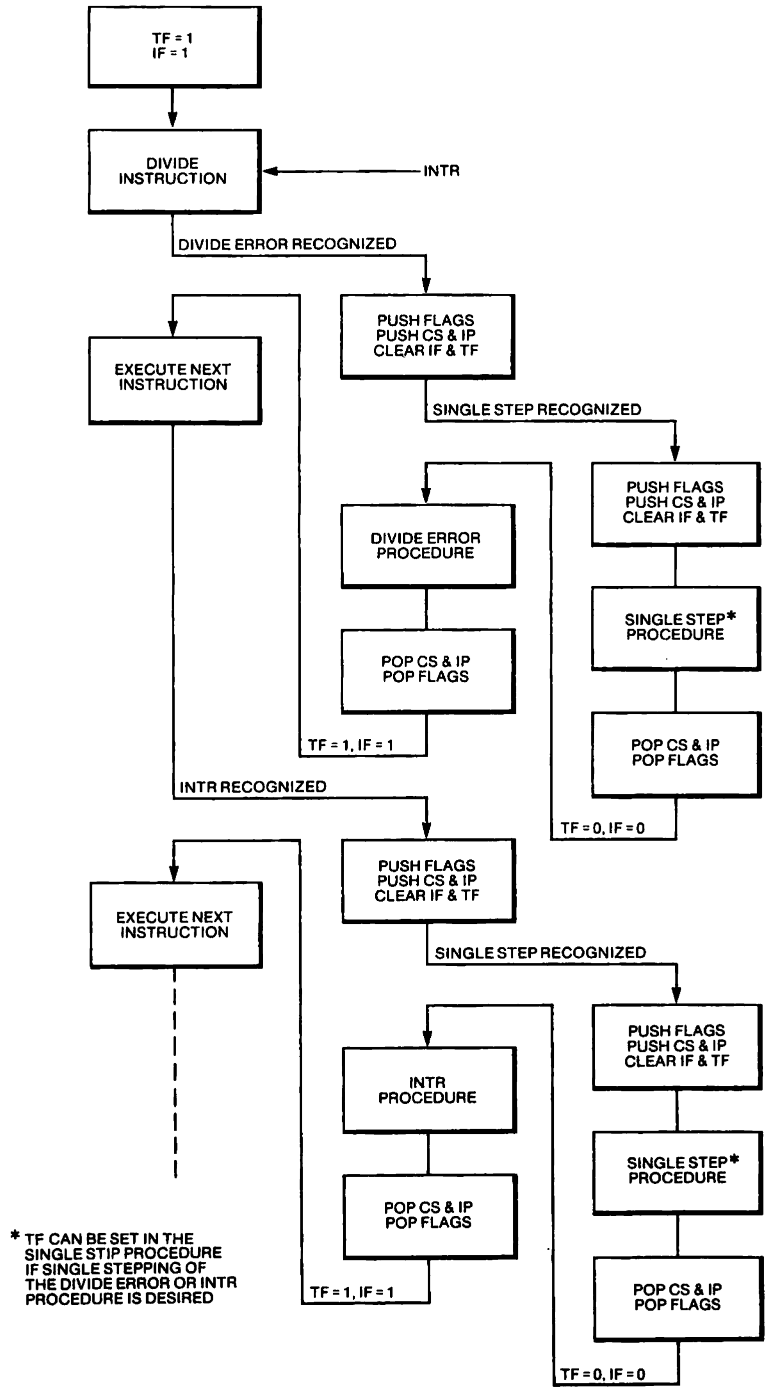
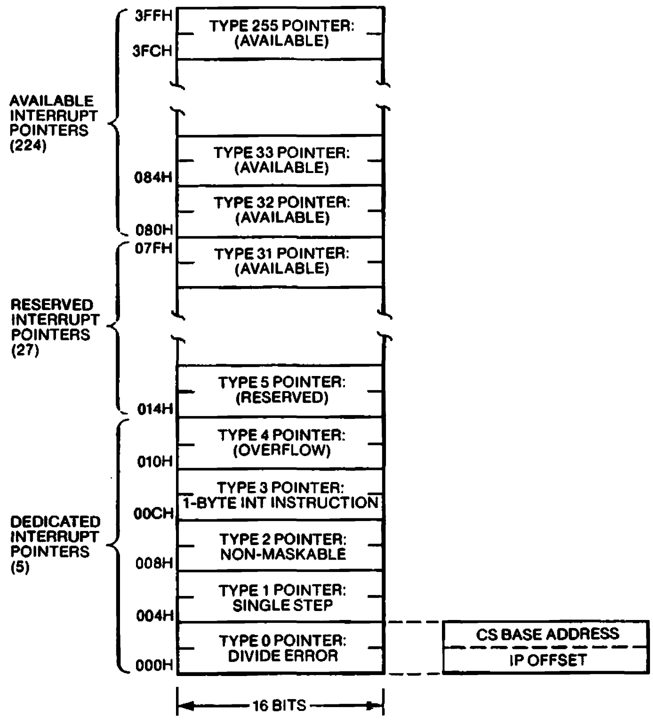


Figure 23: Interrupt Pointer Table



INTERRUPT PROCEDURES Flags CS and IP are pushed onto the stack and flags TF and IF are cleared when an interrupt service procedure is entered. The procedure can re-enable external interrupts with the set-interrupt-enable flag (STI) instruction, allowing itself to be interrupted by a request on INTR. Interrupts are not actually enabled until the instruction following STI has executed. An interrupt procedure can always be interrupted by a request arriving on NMI. The interrupt procedure can also be interrupted by software- or processor-initiated interrupts occuring within the procedure. (Programmers should ensure that the type of interrupt being serviced does not inadvertently occur during the interrupt procedure. For example, attempting to divide by 0 in the divide error (type 0) interrupt procedure results in the procedure being reentered endlessly.) Sufficient stack space must be available to accommodate the maximum depth of interrupt nesting that occurs in the system.

Prior to procedure termination, any registers used by the interrupt procedures should be saved before they are updated and restored. External interrupts for all sections except those sections of code that cannot be interrupted without risking erroneous results should be enabled. Interrupt requests on INTR can be lost if external interrupts are disabled for too long in a procedure.

Interrupt procedures with an interrupt return (IRET) instruction should be terminated. The IRET instruction assumes that the stack is in the same condition as when the procedure was entered. It pops the top three stack words into IP, CS, and the flags, and returns to the instruction that was to be executed when the interrupt procedure was activated.

The actual processing done by the procedure is application dependent. When servicing an external device, the procedure sends a command to the device, instructing it to remove its interrupt request. It can then read status information from the device, determine the cause of the interrupt, and act accordingly.

A software-initiated interrupt procedure can be used as a service routine (supervisor call) for other programs in the system. In this case, the procedure is activated when a program, rather than an external device, needs attention. (The "attention" might be to search a file for a record, send a message to another program, request an allocation of free memory, etc.) Software interrupt procedures can be used to advantage in systems that dynamically relocate programs during execution. Since the interrupt pointer table is at a fixed storage location, procedures can call each other through the table by issuing software interrupt instructions. This provides a stable communication exchange, independent of procedure addresses. Interrupt procedures can be moved if the interrupt pointer table is always updated, providing linkage from the calling program via the interrupt type code.

The 8088 is in single-step mode when the trap flag (TF) is set. In this mode, the processor automatically generates type 1 interrupt processing. The CPU automatically pushes the flags onto the stack and then clears TF and IF. The processor is not in single-step mode when the single-step interrupt procedure is entered. The old flag image is restored from the stack when the single-step procedure terminates, placing the CPU back into single-step mode.

Single stepping is a valuable debugging tool. A single-step procedure acts as a window into the system, through which operations can be observed on an instruction-by-instruction basis. A single-step interrupt procedure prints or displays register contents, instruction pointer values, key memory variables, etc., as they change after each instruction. This permits the exact flow of a program to be traced in detail. The point at which discrepancies occur can be identified by a single-step routine. A single-step routine can be used to accomplish the following:

- Writing a message when a specified memory location or I/O port changes value (or equals a specified value)
- Providing diagnostics selectively (for instance, only for certain instruction addresses)
- Letting a routine execute a number of times before providing diagnostics

---

The 8088 does not have instructions for setting or clearing TF. TF can be changed by modifying the flag image on the stack. The PUSHF and POPF instructions push and pop the flags. (TF can be set by ORing the flag image with 0100H. Clear TF by ANDing it with FEFFH.) After TF is set, the first single-step interrupt occurs after the first instruction following the IRET from the single-step procedure has been executed.

If the processor is single stepping, it processes an interrupt (either internal or external) as follows:

1. Control is passed normally (flags, CS and IP are pushed) to the procedure designated for handling the type of interrupt that has occurred.

2. Before the first instruction of that procedure is executed, the single-step interrupt is recognized and control is passed normally (flags, CS and IP are pushed) to the type 1 interrupt procedure.

3. When single-step procedure terminates, control returns to the previous interrupt procedure. Figure 23 illustrates this process in a case where two interrupts occur when the processor is in single-step mode.

BREAKPOINT INTERRUPT A type 3 interrupt is a breakpoint interrupt. A breakpoint is any place in a program where normal execution is arrested so that some sort of special processing may be performed. Breakpoints are inserted into programs during debugging to display registers, memory locations, etc., at crucial points in the program.

The INT 3 (breakpoint) instruction is one byte long, which facilitates planting a breakpoint anywhere in a program. The processor can be placed in single-step mode by using a breakpoint procedure.

Breakpoint instructions can insert new instructions (patch) into a program without recompiling or reassembling it. This can be done by saving an instruction byte and replacing it with an INT 3 (CCH) machine instruction. The breakpoint procedure contains new machine instructions—code to restore the saved instruction byte and decrement IP on the stack before returning control to the program. The displaced instruction is executed after the patch instructions.

NOTE: Undertake patching a program with caution. This action requires machine-instruction programming and can add new bugs to a program. Also note that a patch is only a temporary measure to be used in exeptional conditions. The affected code should be updated and retranslated as soon as possible.

SYSTEM RESET The 8088 RESET line provides an orderly way to start or restart an executing system. When the processor detects the positive-going edge of a pulse on RESET, it terminates all activities until the signal goes low, at which time it initializes the system as shown in Table 4.

Table 4: CPU State Following Reset

CPU COMPONENT	CONTENT
Flags	Clear
Instruction Register	0000H
CS Register	FFFFH
DS Register	0000H
SS Register	0000H
ES Register	0000H
Queue	Empty

Since the code segment register contains FFFFH and the instruction pointer contains 0H, the processor executes its first instruction following system reset from absolute memory location FFFF0H. This location normally contains an intersegment direct JMP instruction whose target is the actual beginning of the system program. External (maskable) interrupts are disabled by system reset. As soon as the system is initialized, the system software should re-enable interrupts to the point where they can be processed.

PROCESSOR HALT When the halt (HLT) instruction is executed, the 8088 enters the halt state. This condition may be interpreted as "stop all operations until an external interrupt occurs or the system is reset." No signals are floated during the halt state, and the content of the address and data buses is undefined. A bus hold request arriving on the HOLD line is acknowledged normally while the processor is halted.

The halt state can be used when an event prevents the system from functioning correctly. An example might be a power-fail interrupt. After recognizing that loss of power is imminent, the CPU could use the remaining time to move registers, flags and vital variables to a battery-powered CMOS RAM area and then halt until the return of power was signaled by an interrupt or system reset.

Addressing Modes

The 8088 accesses instruction operands in many different ways. Operands can be in registers, instructions, memory, or I/O ports. Memory address and I/O port operands can be calculated several ways. These addressing modes extend the flexibility and convenience of the instruction set. This section briefly describes register and immediate operands, and then covers the 8088 memory and I/O addressing modes in detail.

REGISTER AND IMMEDIATE OPERANDS The quickest, most compact executing instructions specify only register operands. This is because register address is encoded in instructions in a very few bits, and the operation is performed entirely within the CPU (no bus cycles are run). Registers can be source operands and/or destination operands.

Immediate operands are constant data 8- or 16-bits long, contained in an instruction that is available directly from the instruction queue and can be accessed quickly. Like a register operand, no bus cycles are needed to obtain an immediate operand. Immediate operands are limited; they are constant values and can only serve as source operands.

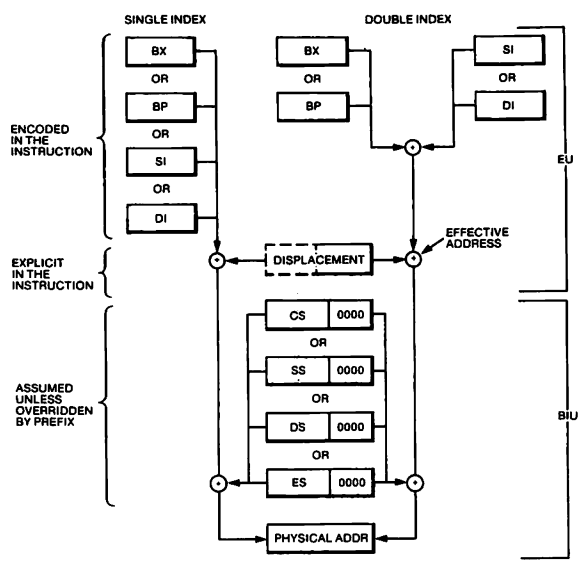
MEMORY ADDRESSING MODES Memory operands must be transferred to or from the CPU over the bus. The EU has direct access to register and immediate operands. When the EU needs to read or write a memory operand, it passes an offset value to the BIU. The BIU adds the offset to the (shifted) content of a segment register, producing a 20-bit physical address. Then it executes the bus cycle(s) needed to access the operand.

EFFECTIVE ADDRESS The operand's effective address (EA) is the offset calculated by EU for a memory operand. EA is an unsigned 16-bit number expressing the operand's distance in bytes from the beginning of the segment in which it resides.

The EU calculates the EA in several different ways. Information encoded in the second byte of the instruction tells the EU how to calculate the EA of each memory operand. A compiler or assembler derives this information from the statement or instruction written by the programmer. Assembly language programmers have access to all addressing modes.

Figure 24 shows that the execution unit calculates the EA by adding a displacement, the content of a base register, and the content of an index register. The variety of 8088 memory addressing modes results from combinations of these three components in a given instruction.

Figure 24: Memory Address Computation



All mnemonics ©Intel Corporation 1981.

The displacement, an 8- or 16-bit number contained in the instruction, is derived from the position of the operand name (a variable or label) in the program. A programmer can modify this value or specify the displacement.

A programmer can specify that BX or BP serve as a base register whose content is to be used in the EA computation. SI or DI can be specified as an index register. The displacement value can change the contents of the base and index registers can change during execution. This makes it possible for one instruction, as determined by current values in the base and/or index registers, to access different memory locations.

It takes time for EU to calculate a memory operand's EA. The more elements in the calculation, the longer it takes. Table 5 shows the time required to compute an effective address for any combination of displacement, base register, and index register.

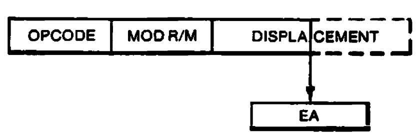
Table 5: Effective Address Calculation Time

EA COMPONENTS		CLOCKS*
Displacement Only		6
Base or Index Only	(BX,BP,SI,DI)	5
Displacement + Base or Index	(BX,BP,SI,DI)	9
Base + Index	BP+DI, BX+SI	7
	BP+SI, BX+DI	8
Displacement + Base + Index	BP+DI+DISP BX+SI+DISP	11
	BP+SI+DISP BX+DI+DISP	12

*Add 2 clocks for segment override.

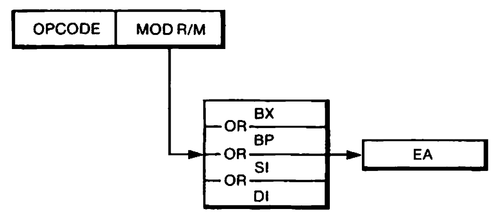
DIRECT ADDRESSING Direct addressing (see Figure 25) is the simplest memory addressing mode. No registers are involved; the EA is taken directly from the displacement field of the instruction. Direct addressing is used to access simple variables (scalars).

Figure 25: Direct Addressing



REGISTER INDIRECT ADDRESSING The effective address of a memory operand can be taken from one of the base or index registers, as shown in Figure 26. When the value in the base of the index register is updated appropriately, one instruction can operate on many different memory locations. The load effective address (LEA) and arithmetic instructions change the register value.

Figure 26: Register Indirect Addressing



NOTE: Any 16-bit general register can be used for register indirect addressing with the JMP or CALL instructions.

BASED ADDRESSING In based addressing (Figure 27), the effective address is the sum of a displacement value and the content of register BX or register BP. Specifying BP as a base register directs the BIU to obtain the operand from the current stack segment (unless a segment override prefix is present). Therefore, based addressing with BP is a convenient way to access stack data.

Based addressing provides a straightforward way of addressing structures located at different places in memory (see Figure 28). A base register can be pointed at the base of the structure, and elements of the structure can be addressed by their displacements from the base. Different copies of the same structure can be accessed by changing the base register.

Figure 27: Based Addressing

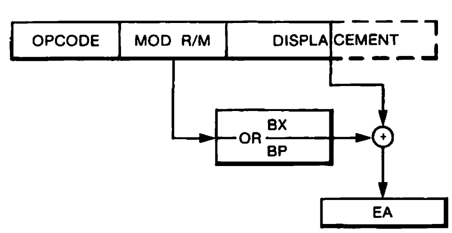
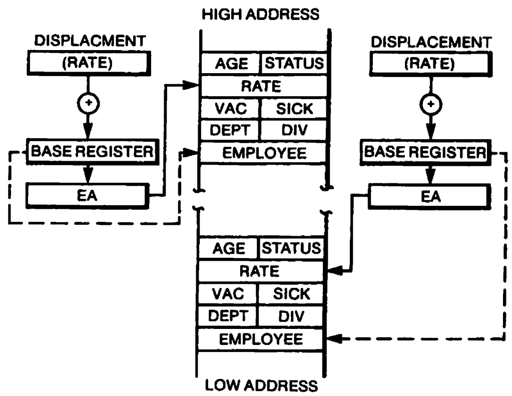


Figure 28: Accessing a Structure with Based Addressing



INDEXED ADDRESSING In indexed addressing, the effective address is calculated by the sum of a displacement plus the content of an index register (SI or DI) as shown in Figure 29. Indexed addressing is often used to access elements in an array (see Figure 30). The displacement locates the beginning of the array, and the value of the index register selects one element (the first element is selected if the index register contains 0). All array elements are the same length, so simple arithmetic on the index register selects any element.

Figure 29: Indexed Addressing

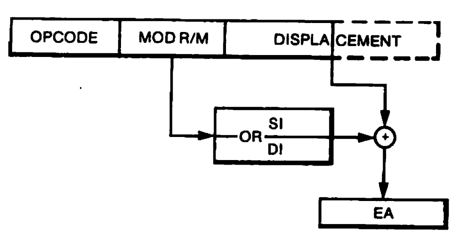
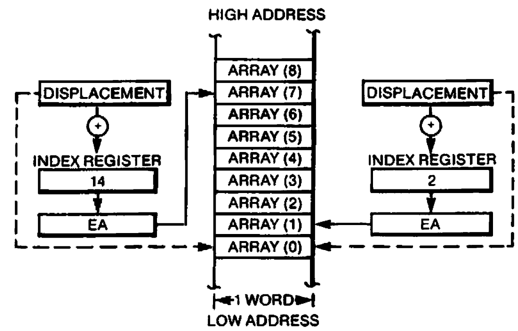


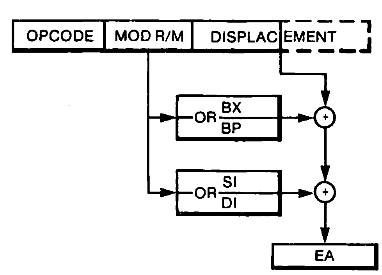
Figure 30: Accessing an Array with Indexed Addressing



BASED INDEXED ADDRESSING Based indexed addressing generates an effective address that is the sum of a base register, an index register, and a displacement (see Figure 31). Two address components can be varied at execution time, making based indexed addressing a very flexible mode. Based indexed addressing provides a convenient way for a procedure to address an array allocated on a stack (see Figure 32). Register BP can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. The offset of the beginning of the array from the reference point can be expressed by a displacement value, and an index register can be used to access individual array elements.

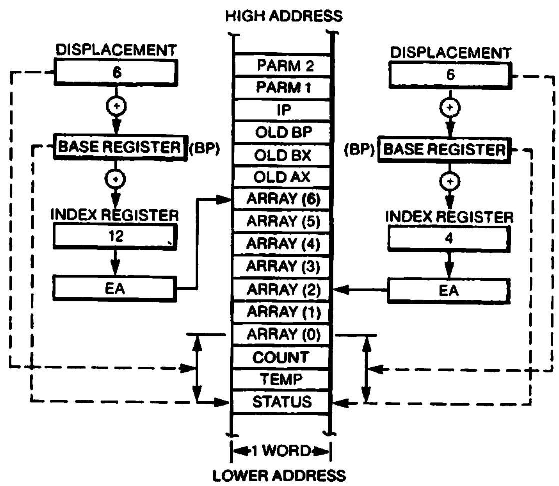
Based indexed addressing can access arrays contained in structures and matrices (two-dimension arrays).

Figure 31: Based Indexed Addressing



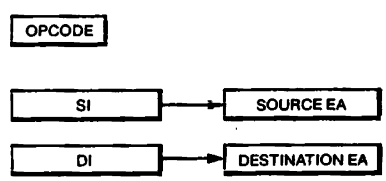
All mnemonics ©Intel Corporation 1981

Figure 32: Addressing a Stack Array with Based Indexed Addressing



STRING ADDRESSING String instructions do not use the normal memory addressing modes to access their operands. Instead, the index registers are used implicitly as shown in Figure 33. When a string instruction is executed, SI is assumed to point to the first byte or word of the source string, and DI is assumed to point to the first byte or word of the destination string. In a repeated string operation, the CPUs automatically adjust SI and DI to obtain subsequent bytes or words.

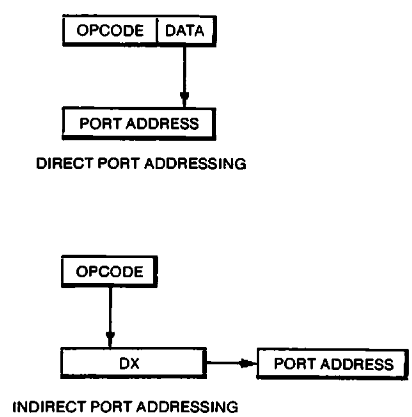
Figure 33: String Operand Addressing



I/O PORT ADDRESSING When an I/O port is memory mapped, any of the memory operand addressing modes can be used to access the port. For example, a group of terminals can be accessed as an array. String instructions can also transfer data to memory-mapped ports with an appropriate hardware interface.

The two addressing modes that can be used to access ports located in the I/O space are illustrated in Figure 34. In direct port addressing, the port number is an 8-bit immediate operand. This allows fixed access to ports numbered 0-255. Indirect port addressing is similar to register indirect addressing of memory operands. The port number is taken from register DX and ranges from 0 to 65,535. By previously adjusting the content of register DX, one instruction can access any port in the I/O space. A group of adjacent ports can be accessed using a simple software loop that adjusts the value in DX.

Figure 34: I/O Port Addressing



Boot ROM

The boot ROM has up to 16K of memory. When the 8088 is reset or powered on, the microprocessor goes to the highest memory area and begins to execute code in the boot ROM. The boot ROM performs basic initialization of all hardware in the machine. It then tries to read the boot software in the disk drives, which contains the operating system. The boot software is loaded into the processor's system random access memory (RAM). When this process is completed, the boot ROM jumps into the operating system and begins executing in the operating system.

INPUT/OUTPUT (I/O) FUNCTIONS

The I/O function consists of serial ports, a parallel port, a control port, an audio input/output function, and a keyboard port.

Serial Ports

The standard configuration includes two full-duplex, serial communications ports. The serial ports are independent and are controlled by a single chip, the NEC 7201. These ports support the RS-232 standard serial interface and can be programmed for asynchronous and for more advanced protocols (e.g., SDLC and IBM binary synchronous communications). Each port is capable of running with an internally generated bit clock (or clocks) supplied by an external source (usually the MODEM). The clock selection is made under software control. There is a programmable bit clock generator for each channel to provide clocking if the internal mode is selected (channels 0 and 1 of the 8253 timer chip are used for this purpose).

Parallel Port

The parallel port is a dual function port supporting parallel Centronics and IEEE 488 interfaces. It is software configurable so as to support these interfaces. The Centronics interface is an 8-bit parallel output interface to standard printers and other devices; the IEEE 488 interface is an instrumentation interface. Initially developed by Hewlett-Packard, the IEEE 488 interface allows for multiple independent devices and for better control and more advanced functions than does the Centronics port. The parallel port is buffered with the standard IEEE 488 drivers.

Control Port

The control port is a series of stake pins on the main logic assembly that contain I/O lines from a 6522 I/O chip. There are two complete 8-bit I/O control ports. Each pin can be configured for input or output (to drive one standard TTL load).

Each 8-bit port has two handshake control lines. The only pin on the control port dedicated to another function is the most significant bit (MSB) of port B. This pin is dedicated to the audio clock that controls the sample rate for the audio. When the Codec audio is in use, the MSB is active.

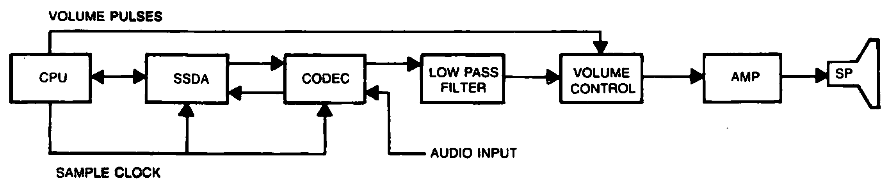
The control port also has a light pen connection which connects to the CRT controller chip and to +12V, -12V, +5V, and ground signals. It supplies minimum power to an external device.

Audio Section

The audio section can generate voice, tones, bells, or other sounds through the speaker in the processor unit. The sounds are stored in a specially coded digitized form in the computer memory. The volume level of sounds generated by the processor unit can be controlled through software or directly with special keys on the keyboard. With additional hardware, the audio section also supports input from external analog sources, allowing digital recording of sounds for future playback.

As shown in Figure 35, the sound output function acts basically as a pipeline from the CPU to the speaker. Sound in digital byte form is stored in the CPU memory. The CPU transfers the sound bytes to the synchronous serial data adapter (SSDA). The SSDA converts the bytes into a serial bit stream of data to feed to the coder/decoder (Codec). The Codec converts the serial data into a varying analog signal. The analog signal is sent through a low pass filter to remove any high frequency noise generated in the digital-to-analog conversion in the Codec. The filtered analog signal is sent into a volume-control section. The volume-control section switches the analog signal at a variable on-to-off rate, allowing the sound level to be controlled. The analog signal is finally sent through an audio amplifier to the speaker in the processor unit.

Figure 35: Audio Section Block Diagram



The synchronous serial data adapter (SSDA) is the major interface between the CPU and the audio section. The main function of the SSDA in playback mode is the buffering and conversion of 8-bit bytes into a serial bit stream for the Codec. In the record mode, the SSDA also converts a serial bit stream from the Codec into bytes for the CPU.

The SSDA is a 6852 I/O chip. The SSDA's control and data resisters are memory-mapped in the CPU's high memory space. The SSDA contains a 3-byte FIFO register buffer. The FIFO allows the CPU to fill the SSDA with three bytes of data and then perform other processing while the SSDA shifts bits out to the Codec. This reduces processor overhead while the processor is playing or recording sounds. The SSDA first shifts the data to the Codec's least significant bit. The SSDA control registers then tell the CPU that the FIFO is ready for more data. The SSDA also provides playback/record (decode/encode) control via its "DTR" output.

The CPU controls the sound quality of the audio section with the shift clock sent to the SSDA and the Codec. The shift clock is generated in one of the CPU's 6522 I/O chips. The PB7 output from the 6522 is controlled by an internal timer, which provides adjustable clock frequency. The higher the frequency of the shift clock, the better the sound quality. Because faster shift clocks require more memory to store the sound bytes, a trade off must be made between sound quality and memory storage. A shift clock of 16Khz will produce telephone quality reproduction of the original sound with each second requiring 2K bytes of storage.

The Codec converts digital data into analog signal in the playback mode and analog signal into digital data in the record mode. The Codec uses a technique known as delta modulation to convert the serial bit stream into analog output. The digital data's 0's and 1's are commands to the integrator in the Codec to make its analog output signal "go up" or "go down" respectively. The serial bit stream represents the direction for the analog output signal.

To increase dynamic range, continuously variable slope delta-modulation (CVSD) is used. An outstanding characteristic of CVSD is its ability, with fairly simple circuitry, to transmit intelligible voice sounds at relatively low data rates. CVSD increases the dynamic range by "companding" (compressing-expanding), which gives small signals a higher relative gain. The CVSD scheme detects three or more consecutive 0's or 1's in the data stream. When this occurs, the gain of the integrator is adjusted to ramp faster to track larger signals. Up to a limit, the more consecutive 1's or 0's, the larger the obtained ramp amplitude, and the better the reproduction of the original sound.

The low pass filter removes unwanted high frequency noise generated in the CODEC. The filter is set for a 3KHz cutoff frequency. This limits sounds to the normal voice bandwidth.

Volume is controlled by varying the duty factor of the analog signal from the filter. The CPU controls the volume level by switching the analog signal on and off at a frequency above the audible range. A minimum of 20KHz is recommended. The CPU uses a 6522's shift register in a recirculating output mode to generate the duty cycle for the volume control. This allows selection of seven different volume levels (and also off).

The final stage is a four watt audio power amplifer which drives the speaker mounted in the disk drive subassembly. A large speaker can be attached to produce more sound output.

Keyboard Interface

There are six signals, or lines, going to the keyboard from the processor. A +5V supply and a ground signal power the keyboard. A shield line shields the keyboard from static and interference. There are three signal lines: ready, data, and acknowledge.

The ready, data, and acknowledge lines control communications between the keyboard and the processor. The keyboard sends data to the microprocesser serially. The keyboard acknowledges or signals to the processor that a key signal has been received and is ready to be sent to the processor. It does this with a keyboard ready line. When the processor is ready, it handshakes the data in via the acknowledge line and the data comes across on the keyboard data line.

The keyboard uses the serial shift register capabilities of a 6522 interface chip to communicate with the microprocessor. This function is handled automatically by the 6522 until the whole key identifier has been received into the shift register. Then the processor reads the key identifier, and handshakes the final check bit sequence.

See Chapter 4, "Keyboard Unit," for a more detailed description of the keyboard interface.

DISK INTERFACE

The signals sent to the disk interface are 8-bit data lines, read/write signals, selection logic signals, and addressing and control signals. They control, send information to, and receive information from the disk drive assembly. A connector on the main logic assembly connects to the drive assembly through a cable. The main logic assembly and microprocessor control the drives with these signals while receiving and sending data to the drive assembly.

EXPANSION BUS

The main logic board supports expansion of the system through four female 50-pin edge connectors. These connectors provide an interface for memory expansion boards and special control boards. Some of the control boards are highspeed network systems, hard disk controller interfaces, and I/O expansion boards for use with science-related applications. The expansion bus has a set of data lines, addressing lines, control lines, and power lines capable of driving any expansion interface. Additional expansion capabilities provide external-device access to memory internal to the main logic assembly.

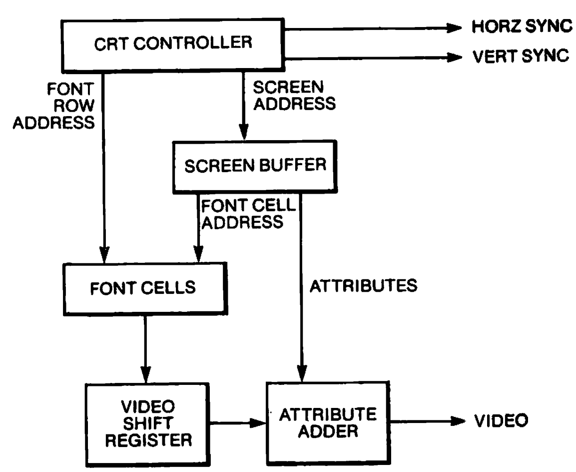
DISPLAY

Standard raster scanning techniques are used to display information on the screen. The most common mode of operation is the text mode, which displays 80 character cells horizontally by 25 lines vertically. This means that an electron beam, scanning horizontally, divides the screen into scan lines. The lines are scanned from left to right and top to bottom.

As the beam scans left to right, the CRT controller generates addresses for the screen buffer RAM. The CRT controller selects words from the screen buffer memory, determining the type of character and the attributes to be displayed. A character cell is 10 dots wide by 16 scan lines high in the text mode. These characters are RAM-mapped and programmable.

The lower 128K bytes of RAM (as well as the 4K bytes from F0000 to F0FFF) is dual port memory. One port of the lower 128K bytes of RAM is used by the display hardware to refresh the raster-scan display. The dual-port memory is managed by an arbitrator circuit that guarantees one refresh access to the display RAM every character cell time. The arbitrator circuit adds a wait-state to any 8088 memory cycle if this is necessary to isolate it from the display-refresh cycle. The display circuit manages the memory-refresh in the dual port on-board dynamic RAM.

Figure 36: Display System Block Diagram

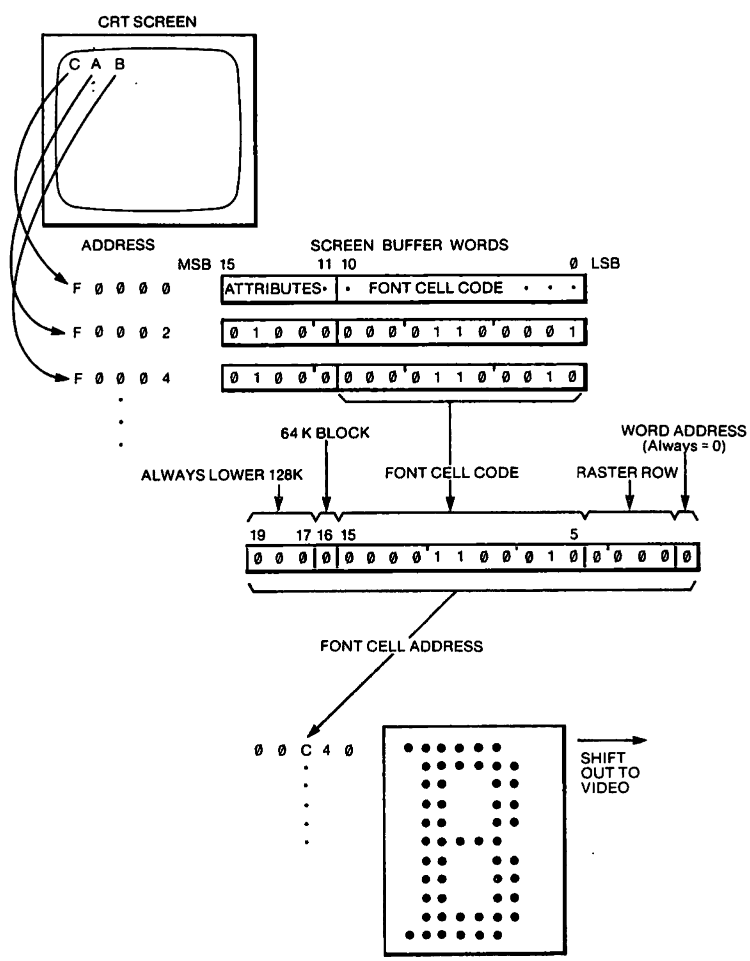


Screen Buffer

The screen buffer is a section of memory 2000 words in length (it is mapped at addresses F0000 through F0FFF).

The words are arranged linearly. The first word in the screen buffer defines the top leftmost character on the screen. The next word in the screen buffer defines the next character on the screen, reading left to right, and etc. All of the characters on the screen are defined in the screen buffer prior to display.

Figure 37: Display Operation



FONT POINTER The words in the screen buffer are broken into two pieces. The lower 11 bits comprise the font pointer. The upper five bits are attribute bits. The font pointer contains binary address information. Up to 2048 characters, or font cells, can be displayed on the screen.

ATTRIBUTE BITS There are five attribute codes associated with each character. Four of these attribute bits are used for reverse video, underline/strikeover, high/low-intensity, and nondisplay. The other bit is available for user software or external hardware.

Each character on the screen is affected by the attributes in the upper 5 bits. Each attribute bit is independent of the other bits.

Reverse Video The reverse video attribute displays black characters on a white background. This affects all the dots in every character, including underline and other modes.

Display High/Low Intensity The high/low intensity attribute displays a character in high intensity (enhanced mode), or in low intensity.

Display Underline/Strikeover The underline/strikeover attribute works in conjunction with the font cell control bit mentioned above. One bit in a font cell word determines where the underline/strikeover occurs (this is discussed later, in "Font Cell"). Underline creates a solid line through the character cell; thus, text underlining is programmable. It can also be used as a strikeover if the underline control bit is in the middle of the character rows. The strikeover is displayed on the screen and superimposed on the character when the attribute is turned on.

Nondisplay Attribute The nondisplay attribute suppresses dot information so that the character is not displayed on the screen.

Software Attribute The software is available for software application program use to identify special fields on the screen, mark the end of lines, or mark special text in an editor. It is not used for display generation functions.

The character and attribute bits are organized into words. The lower 11 bits of each word define which of the 2048 possible characters (font cells) is placed at that location on the screen. The upper five bits identify attributes. These words are on even address boundaries. The 80-character-by-25-line display occupies 2000 words (4000 bytes) of the screen memory.

The five attribute bits are sent to the video control section. The video control section adds the reverse video, intensity, cursor, underline, and nondisplay functions, according to the attribute bits.

The lower 11 bits are the font cell code. The font cell code has other address bits added to it—five lower bits and four upper bits—to generate a font cell address. The first four of the five lower bits, one through four, are the raster row. Using this binary code, 16 raster rows—the number of raster rows in a standard character—can be addressed.

The lower bit, bit 0, is the byte address bit. It is always a zero because words in memory for the font cell are being addressed.

The upper four bits select the 64K block of memory in which the font cells are located. The font cell RAM is limited to the lowest 128K of memory, so bit 17 through bit 19 are always zero.

When bit 16 is zero, it selects the lower 64K of memory. When bit 16 is one, it selects the next block of 64K of memory. This 15-bit address, bits 19 to 5, is the base of the font cell address. The display hardware then appends this address to the raster row being scanned. It takes the addressed word out of the font cell memory and passes it to the video shift register. The word is then processed through attribute control and out to the display.

Font Cell

Characters are generated using a high-density dot matrix technique resulting in a high-resolution display of characters on the screen. This technique uses a font cell as the basic structure within which characters are developed for display. The font cell is a sequential block of 16 words that are accessed to form a dot matrix 16 bits wide and 16 raster rows high.

The first word's least significant bit (LSB) is displayed at the top leftmost position of the font cell display. The second word's LSB is displayed at the leftmost position on the second line, and so forth, through all 16 scan lines. Ten dots of the 16-bit wide cell are displayed on each line. The remaining six dots of each word, which are most significant bits (MSBs), are not displayed.

The underline/strikeover control bit is the MSB of each font.

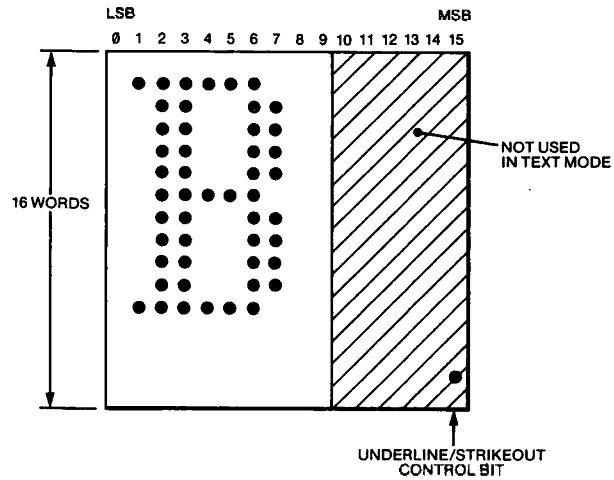
In normal mode, a bit value of 1 displays a white dot, and a bit value of 0 displays a black dot (in reverse video mode, the reverse is displayed). A word, which consists of 16 bits, defines the condition of each dot in the matrix (see Figure 38).

Figure 38: Font Cell Example

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

* = Underline/Strikeover Bit

Figure 39: Block Diagram of a Font Cell



To summarize, the CRT controller chip generates word addresses in the screen buffer memory. A portion of each word contains the attributes, which are passed to the video output section. Another portion of each word is the font cell code, which, when combined with other bits, generates a font cell address. The word at this font cell address is loaded into a video shift register which turns the parallel word into serial bits and passes it to the video output section, where it is combined with the attribute functions. The CRT controller chip also generates the horizontal/vertical signals that go to the display.

Display Brightness

Overall display brightness is software adjustable. Brightness may be adjusted to one of eight different levels by setting the brightness control bits (PB2, PB3, and PE4 of the 6522 at E8040) to the binary value corresponding to the desired level. The binary values range from zero to seven, in order of increasing brightness.

Display Contrast

Display contrast is also software adjustable. The contrast function controls the difference in intensity between high- and low-intensity characters. Only the intensity of the low-intensity characters is varied by the contrast function. Contrast may be adjusted to one of eight levels by setting the binary value of the desired level in the three contrast control bits (PB5, PB6, and PB7 of the 6522 at E8040). The binary values range from zero to seven, in order of increasing contrast (a binary value of zero causes no difference in contrast).

HIGH RESOLUTION MODE

A bit-mapped high-resolution mode is configured for 800 by 400 dots of bit-addressable display. In this mode, the reverse video, high/low-intensity, and nondisplay attributes apply to fixed 16- by 16-dot cells on the screen, and the underline/strikeover attribute is disabled.

The high-resolution mode makes special use of the font cell graphics. The output line (HIRES) controls the font cell width. When high, this line enables the 16-dot matrix, which displays all 16 bits of each font cell word. In this mode, the screen is organized into a 50-column by 25-line display.

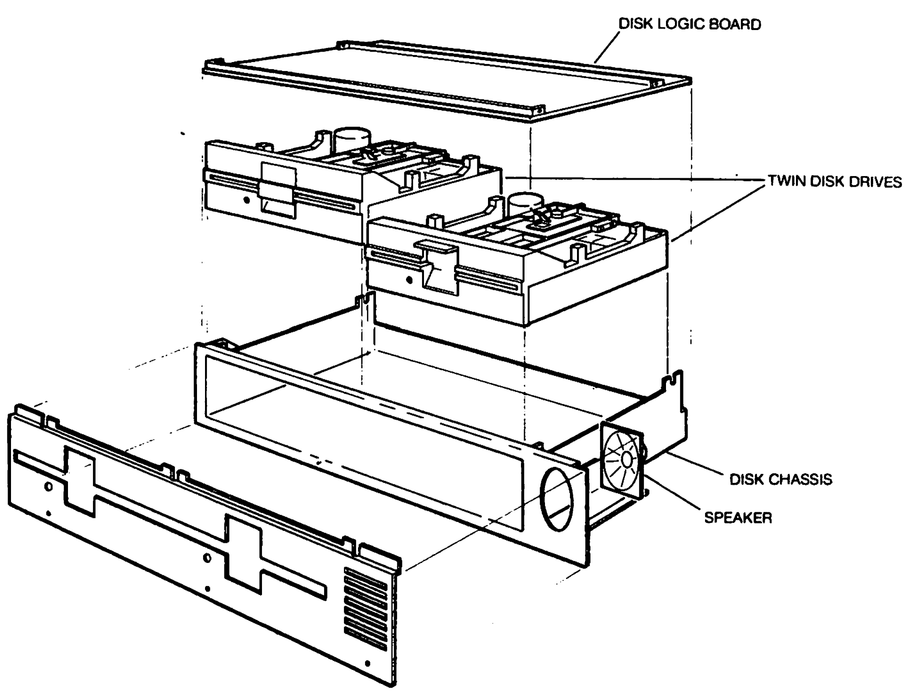
To use the bit-mapped display mode, the screen buffer is filled with font cell pointers which address successive font cells, by column. For example, if line 1/column 1 addresses font cell N, line 2/column 1 would address font cell N+1, and line 25/column 1 would address font cell N+24. Line 1/column 2 would address font cell N+25, and so forth. Line 25/column 50 which would address font cell N+1249. The font cell memory is directly manipulated, without further modification to the screen buffer.

In high-resolution mode, the programmer's view of the screen is 20,000 contiguous words of bit-mapped dots organized into 16-bit wide columns, going from top to bottom, and left to right as word addresses increase.

DISK DRIVE ASSEMBLY

As shown in Figure 40, the disk drive assembly is comprised of two floppy disk drive mechanisms, a disk drive interface board, and a chassis which also contains a speaker. The disk drive assembly provides the system with a minimum of 1.2 million bytes (formatted) of auxillary storage.

Figure 40: Disk Drive Assembly



The standard drive units are 5-1/4 inch, 80-track mechanisms, which operate with single-sided media. Track density is 96 tracks per inch, and recording density is maintained at approximately 8000 bits per inch on all tracks.

FUNCTIONAL DESCRIPTION

The disk drive interface board provides all the low level operations required to convert binary information for storage on and retrieval from diskette. Status and drive control interface to the drives is also provided on the disk drive interface board.

The processing unit maintains functional control of the disk drive assembly.

Reading Data

The 8088 CPU transfers data from the disk to memory as byte-by-byte read operations. Before the data is transferred, the drive motor for the drive containing the disk is started, and the head is positioned to the correct track. The GCR read circuit provides sync detection and separation. (Sync is a special GCR pattern that does not occur in normal data fields. The sync pattern consists of 10 ones during a byte time; other GCR patterns cannot contain more than 8 ones during a byte time.) When the GCR read circuit detects a sync mark, it starts a counter that causes an interrupt to be sent to the CPU, if sync remains present for 6 byte times. This interrupt to the CPU, which is called SYN and is on the highest level interrupt input line to the interrupt controller, informs the CPU that a header sync mark has been detected.

HEADER SEARCH When a sync interrupt occurs while the CPU is searching for a sector, the CPU enters the controller software that will compare the sector header information with the sector requested (the sector header contains the data block ID, track numbers, the sector number, and the checksum). This compare function is performed by the CPU on a byte-by-byte basis. The GCR read circuit provides a data byte every 21.3 microseconds. In order to be able to keep up with the high data rate, the CPU uses a special instruction (WAIT) that stops processing until a byte-ready strobe occurs on the test input. The CPU then continues processing by reading the latched data byte and comparing it with the requested sector information.

If the sector is not the correct sector, the CPU returns from the interrupt and continues processing until the next header sync interrupt. Once the desired sector header has been found, the data transfer can begin.

DATA TRANSFER Before the CPU can read the data block of a sector, the clock recovery circuitry must be resynchronized. This is required because the data block is updated and can be written at any random phase relative to the header information. The data block sync mark is only 5 bytes long and is not detected by the header sync mark detection circuit (header sync marks must be at least 6 bytes in length). The CPU polls the sync input line until the data block sync is detected and then verifies that the byte following sync—the data block IO byte—is correct. If it is not correct, a "not data block IO error" is generated, and no data is transferred. Using the WAIT instruction, the CPU then transfers the following 512 bytes of sector information to the present destination in memory. As the CPU moves the data to memory, it also computes the checksum. This resulting checksum is then compared with the checksum recorded in the data block. If the checksums match, the data transfer is correct; otherwise, error recovery by the CPU is needed.

Writing Data

Data transfer from memory to disk is performed by the CPU in much the same manner as for read operations. The disk drive motor is started and set to the proper speed, and the head is positioned at the correct track by the controller software. The CPU does a header search using the method described earlier in "Reading Data." When the desired header is matched, the CPU starts an update operation of the data portion of the sector and, before turning on the write current, times the GAP1 area. The 5-byte data block sync area is written. Next the 10-byte data block, and then 512 bytes of sector data are written from the preset location in memory. As the data is written, the CPU also creates the 2-byte checksum, which is written at the end of the data section.

The CPU also controls the trim erase timing of the read/write head. The purpose of trim erase is to erase any remaining portion of the old data section that was recorded from the sides of the new data section. At the end of the update, the write current is turned off, and, about 31 byte times later, the trim erase is turned off.

Verification

In order to ensure reliable data storage, all sector updates are followed by a verify operation. A verify operation is similar to a read operation, except that the data in memory is compared to the read disk data being transferred to memory. If any of the bytes do not compare correctly with the data in memory, an error is flagged, and an error recovery is performed by the CPU.

Formatting

A blank or new diskette must be formatted before it can be used. (Some programs, such as DCOPY, perform the formatting function implicitly.) Formatting is done by writing control information and dummy data blocks to all 80 tracks on the disk (see the "Track Format" and "Sector Format" sections under "Physical Description"). The format is a variable number of sectors per zone in soft sectored format. In order to achieve maximum speed tolerance on each diskette, the CPU performs an adaptive format procedure. Diskette speed variation (from unit to unit) causes the number of bytes on a track to vary. During format this problem is solved by always providing a fixed number of unused bytes to allow for the worst case speed. Instead of allowing the unused bytes to be wasted, the format procedure measures the size of the first track in each zone and then adjusts the gap to the size of the sector format. This causes the physical sector size to remain constant regardless of diskette speed during format. This method allows the maximum possible tolerance to speed variation without requiring a gap at the end of the track to allow for speed variation. The technique makes better use of the unused space by distributing it and using the additional intersector time to achieve stabilization of the clock recovery circuitry.

Refer to "Speed Control" and "Motor Speed Control" for more details on speed control.

Positioning

The head positioning mechanism for each drive is a four-phase stepper motor. The disk drive interface has drivers for each stepper motor which are controlled directly by the CPU. By properly sequencing the four phases of the stepper motor, the CPU can move the head of each drive in or out. All timing and control is done in software by the CPU. To reduce power consumption, the stepper motors are energized only when the drive is active; otherwise they are turned off by the CPU. The independent stepper drivers allow the CPU to perform overlapping seeks, resulting in higher system performance.

Speed Control

In order to attain maximum data capacity, the media passes under the head at a constant linear velocity. To attain this, the rotational period is varied as the radius of the track changes. The disk rotational speed is selected by the CPU. The actual speed control is performed by a single chip computer on the disk drive interface board. The CPU communicates with the speed control processor (SCP) by an eight-bit port. On system powerup, the SCP uses a default speed table that allows the system to boot. Once the operating system software is loaded, the CPU writes a new speed table to the SCP that allows it to operate with the current 512-byte sectors. The SCP can be programmed with up to 15 different speeds.

PHYSICAL DESCRIPTION

The disk interface board contains the circuitry necessary to control both of the integrated system disk drives. This circuitry consists of drive motor speed control, read/write head positioning, data decoding and encoding, read channel electronics, and write channel electronics. The interface board receives functional control from the processor unit through a dedicated I/O bus.

Motor Speed Control

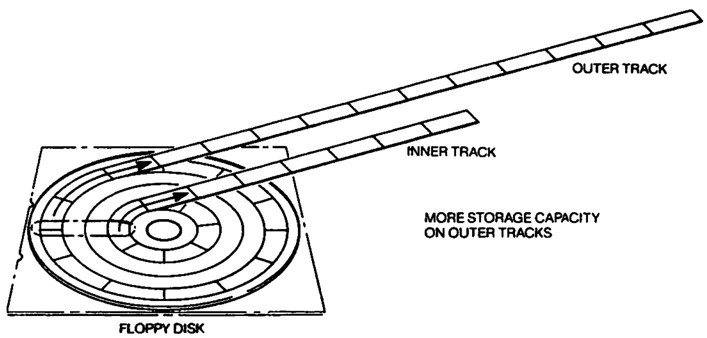
The traditional approach to storing data on floppy disks is to write data (using some encoding scheme) at a fixed rate, while rotating the disk at a constant speed. This results in several undesirable characteristics. Three major undesirable characteristics that were addressed are wasted capacity, large variation in the read signal amplitude, and low system tolerance to motor speed variation.

Since the circumference of the outermost track on the floppy is larger than the circumference of the innermost track (and, in fact, larger than all other tracks) the recording density on the outermost track is lower than on the innermost. The major limiting factor in recording on magnetic media is bit density (actually, flux reversal density), which means that the outer tracks contain less data than the inner tracks, unless adjustment is made to accommodate this problem.

Also, when the disk is rotated at a constant speed or RPM, the linear velocity of the head relative to the media varies from track to track. Since the amplitude of the recorded signal is partly a function of speed, the signal amplitude varies greatly from the outermost track (where it is highest) to the innermost track. This results in a read channel that has a lower signal-to-noise ratio than would be obtainable if all tracks were recorded with a constant amplitude signal.

These two problems are overcome by setting disk rotation speed according to the track circumference. This is done in a way that maintains a nearly constant bit density and a nearly constant linear velocity, hence a constant amplitude signal.

Figure 41: Disk Track and Sector Layout



Data written to the disk is organized into groups of 512 bytes (plus a number of synchronization and control information bytes). These groups are called sectors. Although the circumference of each track differs slightly, it is not possible to take advantage of the potential difference in capacity without using sectors of varying size. Therefore, the speed is changed only when this results in enough additional capacity for an extra sector. The disk is thus divided into groups of tracks, called zones. Each zone, when being read or written, causes the disk to rotate at a slightly different speed.

The third problem—low system tolerance to motor speed variations—is caused by a phenomena called bit shift or pulse crowding. Bit shift occurs during recording at moderately high densities. This introduces timing errors in the data transitions during subsequent reads. The clock recovery circuitry interprets these variations as motor speed error, which reduces the system's tolerance to speed variations of the drive motor.

This problem has been reduced by improving the motor speed control and using an encoding technique that is more tolerant of bit shift error. The disk rotational speed control is accomplished by using a crystal-controlled, closed-loop servo system. The servo system actually consists of two interacting closed servo loops.

The first servo loop is a fast acting inner loop, which is an analog circuit that provides excellent short-term stability. This circuit uses a charge-pump technique, which converts tach pulses from the drive motor to a voltage. This voltage is compared to a reference voltage, and any difference generates a correction in motor speed.

The second servo loop (the outer loop) digitally counts a fixed number of tach pulses from the motor, and measures the period of time that this takes. It then compares this time with the expected time. Any difference results in a modification of the reference voltage for the inner loop. This is accomplished using a single-chip microprocessor (an 8048), which uses the 5 Mhz system clock and two (8-bit) digital-to-analog converters (one per drive). Since this outer loop is crystal-referenced, it provides absolute long term stability and virtually eliminates unit to unit speed differences.

The microprocessor contains a set of speed control tables. These tables are initialized to default values at power-on and are reloadable by the processor unit.

Data Encoding Technique—GCR

To record data on magnetic media, like floppies, the data first has to be converted from the internal computer format into a form that can be stored and retrieved. This is true because data in the internal format may contain long sequences of like bits—either ones or zeroes. If data is recorded with more than a few bit times having no changes (flux reversals), the characteristics of the read channel make it impossible to read back the same signal that was recorded. Also, the data is written at a constant frequency (bit rate), but no clock signal is written. This means that the clock information must be re-created during subsequent read operations. Even though the disk speed is closely controlled (to within 2%), data transitions are required periodically to resynchronize the clock recovery circuitry.

An encoding technique called group code recording (GCR) is used to convert the data from internal representation to an acceptable form. GCR converts each (4-bit) nibble into a 5-bit code that guarantees a recording pattern that never has more than two zeroes together. Then data is recorded on the disk by causing a flux reversal for each "one" bit and no flux reversal for each "zero" bit.

Read Channel

The read channel consists of a magnetic pickup (read/write head), an amplifier section, a clock recovery section, a serial to parallel converter, and a 10-bit to 8-bit (GCR to internal form) conversion section.

The read/write head picks up a low amplitude (approximately 2 to 8 millivolts) signal from the disk. This signal is amplified differentially (to minimize the effects of common mode noise), and pass-band filtered (to reduce noise at frequencies other than those of interest). The linear output from the filter is passed to the differentiator, which generates a wave form whose zero crossovers correspond to the peaks of the read signal (these peaks occur approximately where the flux reversals take place during the write). Then this signal is fed to the comparator and digitizer circuitry. The comparator and digitizer circuitry generate a 1-microsecond read data pulse, corresponding to each peak of the read signal. These pulses serve two purposes: first, each of these pulses represents a "one" bit and so sets the serial data latch (to one); second, these pulses are used by the clock recovery circuit to keep a phase-locked loop (PLL) synchronized to the data being read from the disk. At each clock cycle (bit time), the serial data latch is shifted into the serial to parallel converter, and the serial data latch is reset (to zero).

When 10 bits have been shifted into the serial to parallel converter, the data is converted back into the original 8-bit byte. This data byte is latched, and a signal is sent to the processor unit that a byte is ready to be read.

Write Channel

The write channel consists of an 8-bit to 10-bit (internal form to GCR) code conversion section, a parallel to serial converter, write/erase current control, and the read/write head. The write circuitry is configured so that it is impossible to enable the write current if the diskette is write-protected. The write circuitry also initializes to read mode at power-up, and is prevented from writing until the power has stabilized.

Sector Format

Figure 42 illustrates sector format; Table 6 describes the parts of the sector:

Figure 42: Sector Format

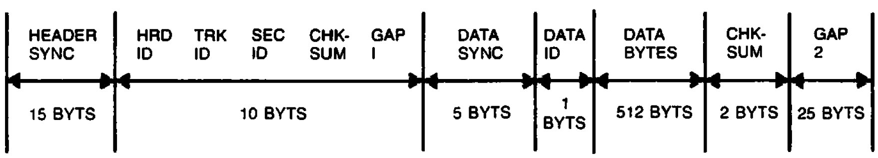


Table 6: Sector Components

COMPONENT	DESCRIPTION
Header sync	This sync mark synchronizes the PLL and causes sync detect interrupts to be sent to the CPU.
Sector header (header ID, track ID, sector ID, and checksum)	This area of 4 bytes contains sector indentification information.
Gap 1	This gap allows time for the CPU to process the sector header in formation and for the read/write head to clear the header for an update.
Data Sync	This sync mark synchronized the PLL and indicates the start of the data field.
Data field (data sync, data ID, data bytes, and checksum)	This is the useful data content of the sector for error detection if a 2-byte checksum is used.
Gap 2	This gap allows for speed variation during an update so that the next sector sync mark is not overwritten.

Track Format

Table 7 presents track format:

Table 7: Track Format

ZONE NUMBER	TRACK NUMBERS LOWER HEAD (STANDARD)	UPPER HEAD	SECTORS PER TRACK	ROTATIONAL PERIOD (MS)
0	0-3	(unused)	19	237.9
1	4-15	0-7	18	224.5
2	16-26	8-18	17	212.2
3	27-37	19-29	16	199.9
4	38-48	30-40	15	187.6
5	49-59	41-51	14	175.3
6	60-70	52-62	13	163.0
7	71-79	63-74	12	149.6
8	unused	75-79	11	144.0

Physical Bus Interface

The disk drive interface board connects to the CPU board via a 50-pin ribbon cable. This cable carries the data bus, address lines, and control signals needed to interface to the three 6522's on the interface board. All the I/O ports of the CPU System are memory-mapped, allowing more efficient I/O operations.

POWER SUPPLY

The power supply for is designed for operational and equipment safety, single-switch operation, and data protection.

The power supply is a 4 voltage regulator with one +5V output, two +12V outputs, and one -12V output. Overall feedback regulates all outputs by sensing the +5V. The -12V output and one of the +12V outputs have independent series regulators.

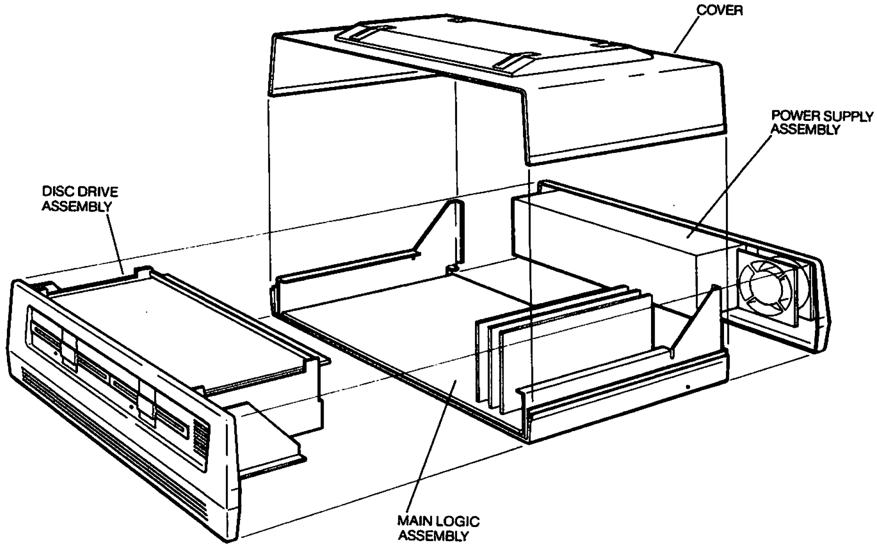
The power supply provides 6 amps of +5V +2%, 2 amps of +12V ±5%, 1.5 amps of +12V ±5%, and .2 amp of -12V ±5%. The operating range is 90-137Vac or 190-270Vac. The range may be selected and strapped by jumper wire. The power supply operates at 47-63 Hz. All power levels are regulated with overvoltage and overcurrent protection.

Line filters provide noise/ripple suppression and conducted/radiated radio frequency energy reduction.

When the power supply is shorted or overloaded, fold-back limiting occurs, preventing overheating. The unit withstands shorted output for an indefinite period and transients of up to 6000V peak. The power supply absorbs transients without causing any deviation at the output.

As shown in Figure 43, the power supply is in a shielded case, housed in the rear of the processor unit. The power supply module contains a fuse, a power switch and a line filter connector which connects to the AC power mains. It powers the processor unit, installed options, the display unit, and the keyboard unit. A 4-inch fan, mounted in the right rear of the processor unit, provides cooling air flow.

Figure 43: Processor Unit



3

DISPLAY UNIT

3. DISPLAY UNIT

The video display unit is supported by a swivel ramp and fits on top of the processor unit. The swivel ramp permits the video display unit to be swiveled right or left and to be tilted up or down. A fabric grid on the face of the CRT reduces glare and reflection and increases character contrast.

A coiled cord with a locking connector plugs the video display unit into the processor unit. The cord carries power and video signals, sync signals, and brightness control signals to the video display unit.

The video display system uses +12V power at approximately 1.2 amps. The horizontal sweep rate is approximately 15KHz. A vertical refresh rate of 76 Hz, or 76 frames per second, prevents visual flicker.

An interlace method of display is used. Each frame contains half the picture. This is very similar to what happens on a conventional television and permits a high-resolution 400-line vertical capability.

Display brightness and contrast are both software adjustable. Brightness, controlled by signals sent from the processor unit's display section, may be varied to two intensities. Contrast is controlled on the main logic board of the processor unit. The user may select eight levels of contrast from the keyboard.

4

KEYBOARD UNIT

4. KEYBOARD UNIT

The function of the keyboard is to generate and send coded electrical signals to the processor unit as each key is depressed or released. The keyboard is entirely reconfigurable.

The keyboard unit is approximately 19 inches wide, 1.8 inches high, and 6.4 inches deep. It is connected to the rear of the processor unit by a coiled cord.

The key switch is a high reliability capacitive-type switch on the keyboard. There is no mechanical contact. The signal is detected electrically, so the switch has a very long life.

Key surfaces are sculpted for comfortable typing. Key caps are removable and interchangeable, facilitating service and allowing the keyboard to be customized.

The keyboard unit is organized into five key groups. The central key group is arranged in a standard typewriter configuration. A numeric/calculator keypad is located at the far right of the keyboard. The general function keys across the top row are double-sized and can be defined for specific purposes by applications programs. A single column of specific function keys are located on the far left of the keyboard. Editing and cursor-control function keys are located in a double column between the typewriter keyboard and the numeric/calculator keypad groups.

The coiled cord is the conduit for all of the keyboard unit's inputs and outputs. The keyboard unit receives power and ground signals, a shield signal which protects the keyboard from static discharge and radiating noise, and three handshake or data control signals which control data transfer from the keyboard to the processor unit.

The comunication between the processor unit and the keyboard unit is serial. The transmission is in 9-bit words. The first eight bits are the data byte, with the least significant bit transmitted first. The last bit is a stop bit.

The keyboard returns key numbers and key status through the eight data bits. The most significant bit of the key number returned by the keyboard unit is status which flags a key "close" or a key "open." The least significant seven bits are the key number.

A single-chip microprocessor in the keyboard unit scans the keyboard for key closures and communicates with the processor unit. Keyboard status communicated to the processor unit is completely independent of key condition. The microprocessor reports an event, such as a key making or breaking contact, and the processor unit determines what that key's function is, based on application program definition.

The keyboard unit processor has an event buffer. It buffers events in case activity is going on in the processor unit that prevents it from servicing all the event signals coming in.

The communication protocol is accomplished through the use of three signal lines. The first control line passes the data serially. The second control line from the keyboard indicates to the processor unit that an event signal is ready, and the processor unit acknowledges this, using the third signal as a handshake. This return line from the processor unit to the keyboard unit is called the acknowledge line. It tells the keyboard that the processor unit has taken the bit and is making the appropriate handshake.

A protocol is defined for handling overflow problems (when the keyboard unit overflows its buffer). The protocol allows the keyboard to enter a "hold-off" state, thus permitting the processor to complete an activity without losing any event signals.

The keyboard can be made to time-out and retransmit event signals in case of an error or a problem in the handshake. The keyboard processor supports N-key rollover, which means that status is reported as the keys are depressed and as they are released. As long as the event queue doesn't overflow and the processor unit keeps up with the event queue, an unlimited number of keys can be rapidly depressed.

APPENDIXES

INTRODUCTION

The 8086 and 8088 execute exactly the same instructions. This instruction set includes equivalents to the instruction typically found in previous microprocessors, such as the 8080/8085. Significant new operations include:

- Multiplication and division of signed and unsigned binary numbers as well as unpacked decimal numbers
- Move, scan, and compare operations for strings up to 64K bytes in length
- Nondestructive bit testing
- Byte translation from one code to another
- Software-generated interrupts
- A group of instructions that can help coordinate the activities of multiprocessor systems

These instructions treat different types of operands uniformly. Nearly every instruction can operate on either byte or word data. Register, memory, and immediate operands may be specified interchangeably in most instructions (except, of course, that immediate values may only serve as source and not destination operands). In particular, memory variables can be added to, subtracted from, shifted, compared, and so on, in place, without moving them in and out of registers. This saves instructions, registers, and execution time in assembly language programs. In high-level languages, where most variables are memory based, compilers, such as PL/M-86, can produce faster and shorter object programs.

The 8086/8088 instruction set can be viewed as existing at two levels: the assembly level and the machine level. To the assembly language programmer, the 8086 and 8088 appear to have a repertoire of about 100 instructions. One MOV (move) instruction, for example, transfers a byte or a word from a register or a memory location or an immediate value to either a register or a memory location. The 8086 and 8088 CPUs, however, recognize 28 different MOV machine instructions ("move byte register to memory," "move word immediate to register," etc.). The ASM-86 assembler translates the assembly-level instructions written by a programmer into the machine-level instructions that are actually executed by the 8086 or 8088. Compilers such as PL/M-86 translate high-level language statements directly into machine-level instructions.

The two levels of the instruction set address two different requirements: efficiency and simplicity. The numerous—there are about 300 in all—forms of machine-level instructions allow these instructions to make very efficient use of storage. For example, the

---

machine instruction that increments a memory operand is three or four bytes long because the address of the operand must be encoded in the instruction. To increment a register, however, does not require as much information, so the instruction can be shorter. In fact, the 8086 and 8088 have eight different machine-level instructions that increment a different 16-bit register; these instructions are only one byte long.

If a programmer had to write one instruction to increment a register, another to increment a memory variable, etc., the benefit of compact instructions would be offset by the difficulty of programming. The assembly-level instructions simplify the programmer's view of the instruction set. The programmer writes one form of the INC (increment) instruction and the ASM-86 assembler examines the operand to determine which machine-level instruction to generate.

This section presents the 8086/8088 instruction set from two perspectives. First, the assembly-level instructions are described in functional terms. The assembly-level instructions are then presented in a reference table that breaks out all permissible operand combinations with execution times and machine instruction length, plus the effect that the instruction has on the CPU flags.

DATA TRANSFER INSTRUCTIONS

The 14 data transfer instructions (Table A-1) move single bytes and words between memory and register as well as between register AL or AX and I/O ports. The stack manipulation instructions are included in this group as are instructions for transferring flag contents and for loading segment registers.

Table A-1: Data Transfer Instructions

	GENERAL PURPOSE
MOV	Move byte or word
PUSH	Push word onto stack
POP	Pop word off stack
XCHG	Exchange byte or word
XLAT	Translate byte
	INPUT/OUTPUT
IN	Input byte or word
OUT	Output byte or word
	ADDRESS OBJECT
LEA	Load effective adress
LDS	Load pointer using DS
LES	Load pointer using ES
	FLAG TRANSFER
LAHF	Load AH register from flags
SAHF	Store AH register in flags
PUSHF	Push flags onto stack
POPF	Pop flags off stack

GENERAL PURPOSE DATA TRANSFERS

MOV *destination, source*

MOV transfers a byte or a word from the source operand to the destination operand.

PUSH *source*

PUSH decrements SP (the stack pointer) by two and then transfers a word from the source operand to the top of stack now pointed by SP. PUSH often is used to place parameters on the stack before calling a procedure; more generally, it is the basic means of storing temporary data on the stack.

POP *destination*

POP transfers the word at the current top of stack (pointed to by SP) to the destination operand, and then increments SP by two to point to the new top of stack. POP can be used to move temporary variables from the stack to registers or memory.

XCHG *destination, source*

XCHG (exchange) switches the contents of the source and destination (byte or word) operands. When used in conjunction with the LOCK prefix, XCHG can test and set a semaphore that controls access to a resource shared by multiple processors.

XLAT *translate-table*

XLAT (translate) replaces a byte in the AL register with a byte from a 256-byte, user-coded translation table. Register BX is assumed to point to the beginning of the table. The byte in AL is used as an index into the table and is replaced by the byte at the offset in the table corresponding to AL's binary value. The first byte in the table has an offset of 0. For example, if AL contains 5H, and the sixth element of the translation table contains 33H, then AL will contain 33H following the instruction. XLAT is useful for translating characters from one code to another, the classic example being ASCII to EBCDIC or the reverse.

IN *accumulator, port*

IN transfers a byte or a word, respectively, to the AL register or AX register, from an input port. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in the DX register, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535.

OUT *port, accumulator*

OUT transfers a byte or a word from the AL register or the AX register, respectively, to an output port. The port number may be specified either with an immediate byte constant, allowing access to ports numbered 0 through 255, or with a number previously placed in register DDX, allowing variable access (by changing the value in DX) to ports numbered from 0 through 65,535).

ADDRESS OBJECT TRANSFERS

These instructions manipulate the addresses of variables rather than the contents or values of variables. They are most useful for list processing, based variables, and string operations.

LEA *destination, source*

LEA (Load Effective Address) transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a 16-bit general register. LEA does not affect any flags. The XLAT and string instructions assume that certain registers point to operands; LEA can be used to lead these registers (e.g., loading BX with the address of the translate table used by the XLAT instruction).

LDS *destination, source*

LDS (Load pointer using DS) transfers a 32-bit pointer variable from source operand, which must be a memory operand, to the destination operand and register DS. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register DS. Specifying SI as the destination operand is a convenient way to prepare to process a source string that is not in the current data segment (string instructions assume that the source string is located in the current data segment and that SI contains the offset of the string).

LES *destination, source*

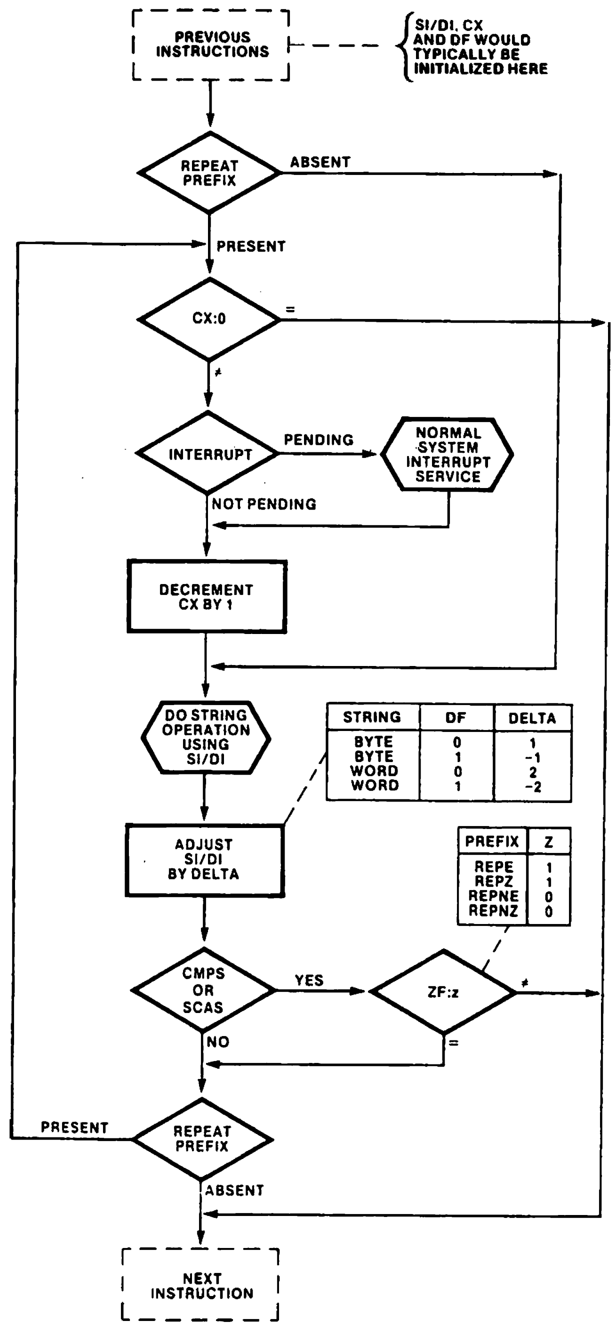
LES (Load pointer using ES) transfers a 32-bit pointer variable from the source operand, which must be a memory operand, to the destination operand and register ES. The offset word of the pointer is transferred to the destination operand, which may be any 16-bit general register. The segment word of the pointer is transferred to register ES. Specifying DI as the destination operand is a convenient way to prepare to process a destination string that is not in the current extra segment. (The destination string must be located in the extra segment, and DI must contain the offset of the string.)

FLAG TRANSFERS

LAHF

LAHF (Load register AH from Flags) copies SF, ZF, AF, PF and CF (the 8080/8085 flags) into bits 7, 6, 4, 2 and 0, respectively, of register AH (see Figure A-1). The content of bits 5, 3 and 1 is undefined; the flags themselves are not affected. LAHF is provided primarily for converting 8080/8085 assembly language programs to run on an 8086 or 8088.

Figure A-1: String Operation Flow



All mnemonics with ©Intel Corporation 1981

SAHF

SAHF (Store register AH into Flags) transfers bits 7, 6, 4, 2, and 0 from register AH into SF, ZF, AF, PF, and CF, respectively, replacing whatever values these flags previously had. OF, DF, IF and TF are not affected. This instruction is provided for 8080/8085 compatibility.

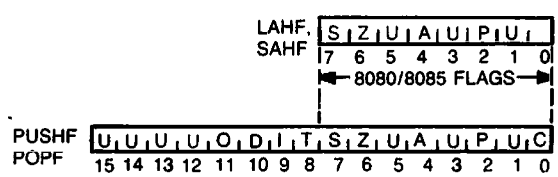
PUSHF

PUSHF decrements SP (the stack pointer) by two and then transfers all flags to the word at the top of stack pointed to be SP (see Figure A-1). The flags themselves are not affected.

POPF

POPF transfers specific bits from the word at the current top of stack (pointed to by register SP) into the 8086/8088 flags, replacing whatever values the flags previously contained (Figure A-2). SP is then incremented by two to point to the new top of stack. PUSHF and POPF allow a procedure to save and restore a calling program's flags. They also allow a program to change the setting of TF (there is no instruction for updating this flag directly). The change is accomplished by pushing the flags, altering bit 8 of the memory image, and then popping the flags.

Figure A-2: Flag Storage Formats



U = UNDEFINED; VALUE IS INDETERMINATE
O = OVERFLOW FLAG
D = DIRECTION FLAG
I = INTERRUPT ENABLE FLAG
T = TRAP FLAG
S = SIGN FLAG
Z = ZERO FLAG
A = AUXILIARY CARRY FLAG
P = PARITY FLAG
C = CARRY FLAG

ARITHMETIC INSTRUCTIONS

ARITHMETIC DATA FORMATS

8086 and 8088 arithmetic operations (Table A-2) may be performed on four types of numbers: unsigned binary, signed binary (integers), unsigned packed decimal and unsigned unpacked decimal (see Table A-3). Binary numbers may be 8 or 16 bits long. Decimal numbers are stored in bytes, two digits per byte for packed decimal and one digit per byte for unpacked decimal. The processor always assumes that the operands specified in arithmetic instructions contain data that represent valid numbers for the type of instruction being performed. Invalid data may produce unpredictable results.

Table A-2: Arithmetic Instructions

	ADDITION
ADD	Add byte or word
ADC	Add byte or word with carry
INC	Increment byte or word by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
	SUBTRACTION
SUB	Subtract byte or word
SBB	Subtract byte or word with borrow.
DEC	Decrement byte or word by 1
NEG	Negate byte or word
CMP	Compare byte or word
AAS	ASCII adjust for subtraction
DAS	Decimal adjust for subtraction
	MULTIPLICATION
MUL	Multiply byte or word unsigned
IMUL	Integer multiply byte or word
AAM	ASCII adjust for multiply
	DIVISION
DIV	Divide byte or word unsigned
IDIV	Integer divide byte or word
AAD	ASCII adjust for division
CBW	Convert byte to word
CWD	Convert word to doubleword

Table A-3: Arithmetic Interpretation of 8-Bit Numbers

HEX	BIT PATTERN	UNSIGNED BINARY	SIGNED BINARY	UNPACKED DECIMAL	PACKED DECIMAL
07	0 0 0 0 0 1 1 1	7	+7	7	7
89	1 0 0 0 1 0 0 1	137	-119	Invalid	89
C5	1 1 0 0 0 1 0 1	197	-59	Invalid	Invalid

Unsigned binary numbers may be either 8 or 16 bits long; all are considered in determining a number's magnitude. The value range of an 8-bit unsigned binary number is 0-255; 16 bits can represent values from 0 through 65,535. Addition, subtraction, multiplication, and division operations are available for unsigned binary numbers.

Signed binary numbers (integers) may be either 8 or 16 bits long. The high-order (leftmost) bit is interpreted as the number's sign: 0 = positive, and 1 = negative. Negative numbers are represented in standard two's complement notation. Since the high-order bit is used for a sign, the range of an 8-bit integer is -128 through +127; 16-bit integers may range from -32,768 through +32,767. The value zero has a positive sign. Multiplication and division operations are provided for signed binary numbers. Addition and subtraction are performed with the unsigned binary instructions. Conditional jump instructions, as well as an "interrupt on overflow" instruction, can be used following an unsigned operation on an integer to detect overflow into the sign bit.

Packed decimal numbers are stored as unsigned byte quantities. The byte is treated as having one decimal digit in each half-byte (nibble); the digit in the high-order half-byte is the most significant. Hexadecimal values 0-9 are valid in each half-byte, and the range of a packed decimal number is 0-99. Addition and subtraction are performed in two steps. First an unsigned binary instruction is used to produce an intermediate result in register AL. Then an adjustment operation is performed which changes the intermediate value in AL to a final correct packed decimal result. Multiplication and division adjustments are not available for packed decimal numbers.

Unpacked decimal numbers are stored as unsigned byte quantities. The magnitude of the number is determined from the low-order half-byte; hexadecimal values 0-9 are valid and are interpreted as decimal numbers. The high-order half-byte must be zero for multiplication and division; it may contain any value for addition and subtraction. Arithmetic on unpacked decimal numbers is performed in two steps. The unsigned binary addition, subtraction, and multiplication operations are used to produce an intermediate result in register AL. An adjustment instruction then changes the value in AL to a final correct unpacked decimal number. Division is performed similarly, except that the adjustment is carried out on the numerator operand in register AL first, and then a following unsigned binary division instruction produces a correct result.

Unpacked decimal numbers are similar to the ASCII character representations of the digits 0-9. Note, however, that the high-order half-byte of an ASCII numeral is always 3H. Unpacked decimal arithmetic may be performed on ASCII numeric characters under the following conditions:

- The high-order half-byte of an ASCII numeral must be set to 0H prior to multiplication or division.
- Unpacked decimal arithmetic leaves the high-order half-byte set to 0H; it must be set to 3H to produce a valid ASCII numeral.

ARITHMETIC INSTRUCTIONS AND FLAGS

The 8086/8088 arithmetic instructions post certain characteristics of the result of the operation to six flags. Most of these flags can be tested by following the arithmetic instruction with a conditional jump instruction; the INTO (interrupt on overflow) instruction also may be used. The various instructions affect the flags differently, as explained in the instruction descriptions. However, they follow these general rules:

- CF (Carry Flag): If an addition results in a carry out of the high-order bit of the result, then CF is set; otherwise CF is cleared. If a subtraction results in a borrow into the high-order bit of the result, then CF is set; otherwise CF is cleared. Note that a signed carry is indicated by CF=OF. CF can be used to detect an unsigned overflow. Two instructions, ADC (add with carry) and SBB (subtract with borrow), incorporate the carry flag in their operations and can be used to perform multibyte (e.g., 32-bit, 64-bit) addition and subtraction.
- AF (Auxiliary Carry Flag): If an addition results in a carry out of the low-order half-byte of the result, then AF is set; otherwise AF is cleared. If a subtraction results in a borrow into the low-order half-byte of the result, then AF is set; otherwise AF is cleared. The auxiliary carry flag is provided for the decimal adjust instructions and ordinarily is not used for any other purpose.
- SF (Sign Flag): Arithmetic and logical instructions set the sign flag equal to the high-order bit (bit 7 or 15) of the result. For signed binary numbers, the sign flag will be 0 for positive results and I for negative results (so long as overflow does not occur). A conditional jump instruction can be used following addition or subtraction to alter the flow of the program depending on the sign of the result. Programs performing unsigned operations typically ignore SF since the high-order bit of the result is interpreted as a digit rather than a sign.
- ZF (Zero Flag): If the result of an arithmetic or logical operation is zero, then ZF is set; otherwise ZF is cleared. A conditional jump instruction can be used to alter the flow of the program if the result is or is not zero.
- PF (Parity Flag): If the low-order eight bits of an arithmetic or logical result contain an even number of 1-bits, then the parity flag is set; otherwise it is cleared.

  PF is provided for 8080/8085 compatibility; it also can be used to check ASCII characters for correct parity.
- OF (Overflow Flag): If the result of an operation is too large a positive number, or too small a negative number to fit in the destination operand (excluding the sign bit), then OF is set; otherwise OF is cleared. OF thus indicates signed arithmetic overflow; it can be tested with a conditional jump or the INTO (interrupt on overflow) instruction. OF may be ignored when performing unsigned arithmetic.

ADDITION

ADD *destination, source*

The sum of the two operands, which may be bytes or words, replaces the destination operand. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADD updates AF, CF, OF, PF, SF, and ZF.

ADC *destination, source*

ADC (Add with Carry) sums the operands, which may be bytes or words, adds one if CF is set, and replaces the destination operand with the result. Both operands may be signed or unsigned binary numbers (see AAA and DAA). ADC updates AF, CF, OF, PF, SF, and ZF. Since ADC incorporates a carry from a previous operation, it can be used to write routines to add numbers longer than 16 bits.

INC *destination*

INC (Increment) adds one to the destination operand. The operand may be a byte or a word and is treated as an unsigned binary number (see AAA and DAA). INC updates AF, OF, PF, SF, and ZF; it does not affect CF.

AAA

AAA (ASCII Adjust for Addition) changes the contents of register AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAA updates AF and CF; the content of OF, PF, SF, and ZF is undefined following execution of AAA.

DAA

DAA (Decimal Adjust for Addition) corrects the result of previously adding two valid packed decimal operands (the destination operand must have been register AL). DAA changes the content of AL to a pair of valid packed decimal digits. It updates AF, CF, PF, SF, and ZF; the content of OF is undefined following execution of DAA.

SUBTRACTION

SUB *destination, source*

The source operand is subtracted from the destination operand, and the result replaces the destination operand. The operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SUB updates AF, CF, OF, PF, SF, and ZF.

SBB *destination, source*

SBB (Subtract with Borrow) subtracts the source from the destination, subtracts one if CF is set, and returns the result to the destination operand. Both operands may be bytes or words. Both operands may be signed or unsigned binary numbers (see AAS and DAS). SBB updates AF, CF, OF, PF, SF, and ZF. Since it incorporates a borrow from a previous operation, SBB may be used to write routines that subtract numbers longer than 16 bits.

DEC *destination*

DEC (Decrement) subtracts one from the destination, which may be a byte or a word. DEC updates AF, OF, PF, SF, and ZF; it does not affect CF.

NEG *destination*

NEG (Negate) subtracts the destination operand, which may be a byte or a word, from 0 and returns the result to the destination. This forms the two's complement of the number, effectively reversing the sign of an interger. If the operand is zero, its sign is not changed. Attempting to negate a byte containing -128 or a word containing -32,768 causes no change to the operand and sets OF. NEG updates AF, CF, OF, PF, SF, and ZF. CF is always set except when the operand is zero, in which case it is cleared.

CMP *destination, source*

CMP (Compare) subtracts the source from the destination, which may be bytes or words, but does not return the result. The operands are unchanged, but the flags are updated and can be tested by a subsequent conditional jump instruction. CMP updates AF, CF, OF, PF, SF, and ZF. The comparison reflected in the flags is that of the destination to the source. If a CMP instruction is followed by a JG (Jump if Greater) instruction, for example, the jump is taken if the destination operand is greater than the source operand.

AAS

AAS (ASCII Adjust for Subtraction) corrects the result of a previous subtraction of two valid unpacked decimal operands (the destination operand must have been specified as register AL). AAS changes the content of AL to a valid unpacked decimal number; the high-order half-byte is zeroed. AAS updates AF and CF; the content of OF, PF, SF, and ZF is undefined following execution of AAS.

DAS

DAS (Decimal Adjust for Subtraction) corrects the result of a previous subtraction of two valid packed decimal operands (the destination operand must have been specified as register AL). DAS changes the content of AL to a pair of valid packed decimal digits. DAS updates AF, CF, PF, SF, and ZF; the content of OF is undefined following execution of DAS.

MULTIPLICATION

MUL *source*

MUL (Multiply) performs an unsigned multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source operand is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. The operands are treated as unsigned binary numbers (see AAM). If the upper half of the result (AH for byte source, DX for word source) is nonzero, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF, and ZF is undefined following execution of MUL.

IMUL *source*

IMUL (Integer Multiply) performs a signed multiplication of the source operand and the accumulator. If the source is a byte, then it is multiplied by register AL, and the double-length result is returned in AH and AL. If the source is a word, then it is multiplied by register AX, and the double-length result is returned in registers DX and AX. If the upper half of the result (AH for byte source, DX for word source) is not the sign extension of the lower half of result, CF and OF are set; otherwise they are cleared. When CF and OF are set, they indicate that AH or DX contains significant digits of the result. The content of AF, PF, SF, and ZF is undefined following execution of IMUL.

AAM

AAM (ASCII Adjust for Multiply) corrects the result of a previous multiplication of two valid unpacked decimal operands. A valid 2-digit unpacked decimal number is derived from the content of AH and AL and is returned to AH and AL. The high-order half-bytes of the multiplied operands must have been 0H for AAM to produce a correct result. AAM updates PF, SF, and ZF; the content of AF, CF, and OF is undefined following execution AAM.

DIVISION

DIV *source*

DIV (divide) performs an unsigned division of accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH. The single-length quotient is returned in AL, and the single-length remainder is returned in AH. If the source operand is a word, it is divided into the double-length dividend in registers AX and DX. The single-length quotient is returned in AX, and the single-length remainder is returned in DX. If the quotient exceeds the capacity of its destination register (FFH for byte source, FFFFFH for word source), as when division by zero is attempted, a type 0 interrupt is generated, and the quotient and remainder are undefined. Nonintegral quotients are truncated to integers. The content of AF, CF, OF, PF, SF, and ZF is undefined following execution of DIV.

IDIV *source*

IDIV (Integer Divide) performs a signed division of the accumulator (and its extension) by the source operand. If the source operand is a byte, it is divided into the double-length dividend assumed to be in registers AL and AH; the single-length quotient is returned in AL, and the single-length remainder is returned in AH. For byte integer division, the maximum positive quotient is +127(7FH) and the minimum negative quotient is 127(81H). If the source operand is a word,it is divided into the double-length dividend in registers AX and DX; the single-length quotient is returned in AX, and the single-length remainder is returned in DX. For word integer division, the maximum positive quotient is +32,767 (7FFFH) and the minimum negative quotient is 32,767 (8001H). If the quotient is positive and exceeds the maximum, or is negative and is less than the minimum, the quotient and remainder are undefined, and a type 0 interrupt is generated. In particular, this occurs if division by 0 is attempted. Nonintegral quotients are truncated (toward 0) to integers, and the remainder has the same sign as the dividend. The content of AF, CF, OF, PF, SF, and ZF is undefined following IDIV.

AAD

AAD (ASCII Adjust for Division) modifies the numerator in AL before dividing two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH must be zero for the subsequent DIV to produce the correct result. The quotient is returned in AL, and the remainder is returned in AH; both high-order half-bytes are zeroed. AAD updates PF, SF, and ZF; the content of AF, CF, and OF is undefined following execution of AAD.

CBW

CBW (Convert Byte to Word) extends the sign of the byte in register AL throughout register AH. CBW does not affect any flags. CBW can be used to produce a double-length (word) dividend from a byte prior to performing byte division.

CWD

CWD (Convert Word to Doubleword) extends the sign of the word in register DX. CWD does not affect any flags. CWD can be used to produce a double-length (doubleword) dividend from a word prior to performing word division.

BIT MANIPULATION INSTRUCTIONS

The 8086 and 8088 provide three groups of instructions (Table A-4) for manipulating bits within both bytes and words: logical, shifts, and rotates.

Table A-4: Bit Manupulation Instructions

	LOGICALS
NOT	"Not" byte or word
AND	"And" byte or word
OR	"Inclusive or" byte or word
XOR	"Exclusive or" byte or word
TEST	"Test" byte or word
	SHIFTS
SHL/SAL	Shift logical/arithmetic left byte or word
SHR	Shift logical right byte or word
SAR	Shift arithmetic right byte or word
	ROTATES
ROL	Rotate left byte or word
ROR	Rotate right byte or word
RCL	Rotate through carry left byte or word
RCR	Rotate through carry right byte or word

LOGICAL

The logical instructions include the boolean operators "not," "and," "inclusive or", and "exclusive or", plus a TEST instruction that sets the flags, but does not alter either of its operands.

AND, OR, XOR and TEST affect the flags as follows: The overflow (OF) and carry (CF) flags are always cleared by logical instructions, and the content of the auxiliary carry (AF) flag is always undefined following execution of a logical instruction. The sign (SF), zero (ZF) and parity (PF) flags are always posted to reflect the result of the operation and can be tested by conditional jump instructions. The interpretation of these flags is the same as for arithmetic instructions. SF is set if the result is negative (high-order bit is 1), and is cleared if the result is positive (high-order bit is 0). ZF is set if the result is zero; it is otherwise cleared. PF is set if the result contains an even number of 1-bits (has even parity) and is cleared if the number of 1-bits is odd (the result has odd parity). Note that NOT has no effect on the flags.

NOT *destination*

NOT inverts the bits (forms the one's complement) of the byte or word operand.

AND *destination, source*

AND performs the logical "and" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if both correspondence bits of the original operands are set; otherwise the bit is cleared.

OR *destination, source*

OR performs the logical "inclusive or" of the two operands (byte or word) and returns the result to the destination operand. A bit in the result is set if either or both corresponding bits in the original operands are set; otherwise the result bit is cleared.

XOR *destination, source*

XOR (Exclusive Or) performs the logical "exclusive or" of the two operands and returns the result to the destination operand. A bit in the result is set if the corresponding bits of the original operands contain opposite values (one is set, the other is cleared); otherwise the result bit is cleared.

TEST *destination, source*

TEST performs the logical "and" of the two operands (byte or word), updates the flags, but does not return the result—i.e., neither operand is changed. If a TEST instruction is followed by a JNZ (Jump if Not Zero) instruction, the jump will be taken if there are any corresponding 1-bits in both operands.

SHIFTS

The bits in bytes and words may be shifted arithmetically or logically. Up to 255 shifts may be performed, according to the value of the count operand coded in the instruction. The count may be specified as the constant 1, or as register CL, allowing the shift count to be a variable supplied at execution time. Arithmetic shifts may be used to multiply and divide binary numbers by powers of two (see note in description of SAR). Logical shifts can be used to isolate bits in bytes or words.

Shift instructions affect the flags as follows: AF is always undefined following a shift operation. PF, SF, and ZF are updated normally, as in the logical instructions. CF always contains the value of the last bit shifted out of the destination operand. The content of OF is always undefined following a multibit shift. In a single-bit shift, OF is set if the value of the high-order (sign) bit was changed by the operation; if the sign bit retains its original value, OF is cleared.

SHL/SAL *destination, count*

SHL and SAL (Shift Logical Left and Shift Arithmetic Left) perform the same operation and are physically the same instruction. The destination byte or word is shifted left by the number of bits specified in the count operand. Zeros are shifted in on the right. If the sign bit retains its original value, then IF is cleared.

SHR *destination, source*

SHR (Shift Logical Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Zeros are shifted in on the left. If the sign bit retains its original value, then OF is cleared.

SAR *destination, count*

SAR (Shift Arithmetic Right) shifts the bits in the destination operand (byte or word) to the right by the number of bits specified in the count operand. Bits equal to the original high-order (sign) bit are shifted in on the left, preserving the sign of the original value. Note that SAR does not produce the same result as the dividend of an equivalent IDIV instruction if the destination operand is negative and 1-bits are shifted out. For example, shifting -5 right by one bit yields -3, while integer division of -5 by 2 yields -2. The difference in the instructions is that IDIV truncates all numbers toward zero, while SAR truncates positive numbers toward zero and negative numbers toward negative infinity.

ROTATES

Bits in bytes and words also may be rotated. Bits rotated out of an operand are not lost as in a shift, but are circled back into the other end of the operand. As in the shift instructions, the number of bits to be rotated is taken from the count operand, which may specify either a constant of 1, or the CL register. The carry flag may act as an extension of the operand in two of the rotate instructions, allowing a bit to be isolated in CF and then tested by a JC (Jump if Carry) or JNC (Jump if Not Carry) instruction.

Rotates affect only the carry and overflow flags. CF always contains the value of the last bit rotated out. On multibit rotates, the value of OF is always undefined. In single-bit rotates, OF is set if the operation changes the high-order (sign) bit of the destination operand. If the sign bit retains its original value, OF is cleared.

ROL *destination, count*

ROL (Rotate Left) rotates the destination byte or word left by the number of bits specified in the count operand.

ROR *destination, count*

ROR (Rotate Right) operates similar to ROL except that the bits in the destination byte or word are rotated right instead of left.

RCL *destination, count*

RCL (Rotate through Carry Left) rotates the bits in the byte or word destination operand to the left by the number of bits specified in the count operand. The carry flag (CF) is treated as "part of" the destination operand; that is, its value is rotated into the low-order bit of the destination, and is itself replaced by the high-order bit of the destination.

RCR *destination, count*

RCR (Rotate through Carry Right) operates exactly like RCL except that the bits are rotated right instead of left.

STRING INSTRUCTIONS

Five basic string operations, called primitives, allow strings of bytes or words to be operated on, one element (byte or word) at a time. Strings of up to 64k bytes may be manipulated with these instructions. Instructions are available to move, compare, and scan for a value, as well as for moving string elements to and from the accumulator (see Table A-5). These basic operations may be preceded by a special one-byte prefix that causes the instruction to be repeated by the hardware, allowing long strings to be processed much faster than would be possible with a software loop. The repetitions can be terminated by a variety of conditions, and a repeated operation may be interrupted and resumed.

Table A-5: String Instructions

REP	Repeat
REPE/REPZ	Repeat while equal/zero
REPNE/REPNZ	Repeat while not equal/not zero
MOVS	Move byte or word string
MOVSB/MOVSW	Move byte or word string
CMPS	Compare byte or word string
SCAS	Scan byte or word string
LODS	Load byte or word string
STOS	Store byte or word string

The string instructions operate quite similarly in many respects; the common characteristics are covered here and in Table A-6 and Figure A-2 rather than in the descriptions of the individual instructions. A string instruction may have a source operand, a destination operand, or both. The hardware assumes that a source string resides in the current data segment; a segment prefix byte may be used to override this assumption. A destination string must be in the current extra segment. The assembler checks the attributes of the operands to determine if the elements of the strings are bytes or words. The assembler does not, however, use the operand names to address the strings. Rather, the content of register SI (source index) is used as an offset to address the current element of the source string, and the content of register DI (destination index) is taken as the offset of the current destination string element. These registers must be initialized to point to the source/destination strings before executing the string instruction; the LDS, LES, and LEA instructions are useful in this regard.

Table A-6: String Instruction Register and Flag Use

SI	Index (offset) for source string
DI	Index (offset) for destination
CX	Repetition counter
AL/AX	Scan value Destination for LODS Source for STOS
DF	0=auto-increment SI, DI 1=auto-decrement SI, DI
ZF	Scan/compare terminator

The string instructions automatically update SI and/or DI in anticipation of processing the next string element. The DF (direction flag) setting determines whether the index registers are auto decremented (DF=1). If byte strings are being processed, SI and/or DI is adjusted by 1; the adjustment is 2 for word strings.

If a Repeat prefix has been coded, then register CX (count register) is decremented by 1 after each repetition of the string instruction; therefore, CX must be initialized to the number of repetitions desired before the string instruction is executed. If CX is 0, the string instruction is not executed, and control goes to the following instruction.

REP/REPE/REPZ/ REPNE/REPNZ

REP (Repeat), REPE (Repeat While Equal), REPZ (Repeat While Zero), REPNE (Repeat While Not Equal), and REPNZ (Repeat While Not Zero) are five mnemonics for two forms of the prefix byte that controls repetition of a subsequent string instruction. The different mnemonics are provided to improve program clarity. The repeat prefixes do not affect the flags.

REP is used in conjunction with the MOVS (Move String) and STOS (Store String) instructions and is interpreted as "repeat while not end-of-string" (CX not 0). REPE and REPZ operate identically and are physically the same prefix byte as REP. These instructions are used with the CMPS (Compare String) and SCAS (Scan String) instructions and require ZF (posted by these instructions) to be set before initiating the next repetition. REPNE and REPNZ are two mnemonics for the same prefix byte. These instructions function the same as REPE and REPZ, except that the zero flag must be cleared or the repetition is terminated. Note that ZF does not need to be initialized before executing the repeated string instruction.

Repeated string sequences are interruptable; the processor will recognize the interrupt before processing the next string element. System interrupt processing is not affected in any way. Upon return from the interrupt, the repeated operation is resumed from the point of interruption. Note, however, that execution does not resume properly if a second or third prefix (i.e., segment override or LOCK) has been specified in addition to any of the repeat prefixes. The processor "remembers" only one prefix in effect at the time of the interrupt—the prefix that immediately precedes the string instruction. After returning from the interrupt, processing resumes at this point, but any additional prefixes specified are not in effect. If more than one prefix must be used with a string instruction, interrupts may be disabled for the duration of the repeated execution. However, this will not prevent a nonmaskable interrupt from being recognized. Also, the time that the system is unable to respond to interrupts may be unacceptable if long strings are being processed.

MOVS *destination-string, source-string*

MOVS (Move String) transfers a byte or a word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element. When used in conjunction with REP, MOVS performs a memory-to-memory block transfer.

MOVSB/MOVSW

MOVSB and MOVSW are alternate mnemonics for the move string instruction. These mnemonics are coded without operands; they explicitly tell the assembler that a byte string (MOVSB) or a word string (MOVSW) is to be moved (when MOVS is coded, the assembler determines the string type from the attributes of the operands). These mnemonics are useful when the assembler cannot determine the attributes of a string—e.g., when a section of code is being moved.

CMPS *destination-string, source-string*

CMPS (Compare String) subtracts the destination byte or word (addressed by DI) from the source byte or word (addressed by SI). CMPS affects flags without altering either operand, updates SI and DI to point to the next string element, and updates AF, CF, OF, PF, SF, and ZF to reflect the relationship of the destination element to the source element. For example, if a JG (Jump if Greater) instruction follows CMPS, the jump is taken if the destination element is greater than the source element. If CMPS is prefixed with REPE or REPZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are equal (ZF=1)." If CMPS is preceded by REPNE or REPNZ, the operation is interpreted as "compare while not end-of-string (CX not zero) and strings are not equal (ZF=0)." Thus, CMPS can be used to find matching or differing string elements.

SCAS *destination-string*

SCAS (Scan String) subtracts the destination string element (byte or word) addressed by DI from the content of AL (byte string) or AX (word string) and updates the flags, but does not alter the destination string or the accumulator. SCAS also updates DI to point to the next string element and AF, CF, OF, PF, SF, and ZF to reflect the relationship of the scan value in AL/AX to the string element. If SCAS is prefixed with REPE or REPZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element scan value (ZF=1)." This form may be used to scan for departure from a given value. If SCAS is prefixed with REPNE or REPNZ, the operation is interpreted as "scan while not end-of-string (CX not 0) and string-element is not equal to scan-value (ZF=0)." This form may be used to locate a value in a string.

LODS *source-string*

LODS (Load String) transfers the byte or word string element addressed by SI to register AL or AX, and updates SI to point to the next element in the string. This instruction is not ordinarily repeated since the accumulator would be overwritten by each repetition, and only the last element would be retained. However, LODS is very useful in software loops as part of a more complex string function built up from string primitives and other instructions.

STOS *destination-string*

STOS (Store String) transfers a byte or word from register AL or AX to the string element addressed by DI and updates DI to point to the next location in the string. As a repeated operation, STOS provides a convenient way to initialize a string to a constant value (e.g., to blank out a print line).

PROGRAM TRANSFER INSTRUCTIONS

The sequence of execution of instructions in an 8086/8088 program is determined by the content of the code segment register (CS) and the instruction pointer (IP). The CS register contains the base address of the current code segment, the 64k portion of memory from which instructions are presently being fetched. The IP is used as an offset from the beginning of the code segment; the combination of CS and IP points to the memory location from which the next instruction is to be fetched. (Recall that under most operating conditions, the next instruction to be executed has already been fetched from memory and is waiting in the CPU instruction queue.) The program transfer instructions operate on the instruction pointer and on the CS register; changing the content of these causes normal sequential execution to be altered. When a program transfer occurs, the queue no longer contains the correct instruction, and the BIU obtains the next instruction from memory using the new IP and CS values, passes the instruction directly to the EU, and then begins refilling the queue from the new location.

Four groups of program transfers are available in the 8086/8088: unconditional transfers, conditional transfers, iteration control instructions and interrupt-related instructions (see Table A-7). Only the interrupt-related instructions affect any CPU flags. As will be seen, however, the execution of many of the program transfer instructions is affected by the states of the flags.

Table A-7: Program Transfer Instructions

UNCONDITIONAL TRANSFERS	
CALL	Call procedure
RET	Return from procedure
JMP	Jump
CONDITIONAL TRANSFERS	
JA/JNBE	Jump if above/not below or equal
JAE/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above or equal
JBE/JNA	Jump if below or equal/not above
JC	Jump if carry
JE/JZ	Jump if equal/zero
JG/JNLE	Jump if greater/not less or equal
JGE/JNL	Jump if greater or equal/not less
JL/JNGE	Jump if less/not greater or equal
JLE/JNG	Jump if less or equal/not greater
JNC	Jump if not carry
JNE/JNZ	Jump if not equal/not zero
JNO	Jump if not overflow
JNP/JPO	Jump if not parity/parity odd
JNS	Jump if not sign
JO	Jump if overflow
JP/JPE	Jump if parity/parity even
JS	Jump if sign
ITERATION CONTROLS	
LOOP	Loop
LOOPE/LOOPZ	Loop if equal/zero
LOOPNE/LOOPNZ	Loop if not equal/not zero
JCXZ	Jump if register CX=0
INTERRUPTS	
INT	Interrupt
INTO	Interrupt if overflow
IRET	Interrupt return

UNCONDITIONAL TRANSFERS

The unconditional transfer instructions may transfer control to a target instruction within the current code segment (intrasegment transfer) or to a different code segment (intersegment transfer). The ASM-86 assembler terms an intrasegment target NEAR and an intersegment target FAR. The transfer is made unconditionally any time the instruction is executed.

CALL
procedure-name

CALL activates an out-of-line procedure, saving information on the stack to permit a RET (return) instruction in the procedure to transfer control back to the instruction following the CALL. The assembler generates one of two types of CALL instruction; the type depends on whether the programmer has defined the procedure name as NEAR or FAR. For control to return properly, the type of CALL instruction must match the type of RET instruction that exits from the procedure. (The potential for a mismatch exists if the procedure and the CALL are contained in separately assembled programs.) Different forms of the CALL instruction allow the address of the target procedure to be obtained from the instruction itself (direct CALL) or from a memory location or register referenced by the instruction (indirect CALL). In the following descriptions, bear in mind that the processor automatically adjusts IP to point to the next instruction to be executed before saving it on the stack.

For an intrasegment direct CALL, SP (the stack pointer) is decremented by two and IP is pushed onto the stack. The relative displacement (up to +32k) of the target procedure from the CALL instruction is then added to the instruction pointer. This form of the CALL instruction is self-relative and is appropriate for position-independent (dynamically relocatable) routines in which the CALL and its target are in the same segment and are moved together.

An intrasegment indirect CALL may be made through memory or through a register. SP is decremented by two and IP is pushed onto the stack. The offset of the target procedure is obtained from the memory word or 16-bit general register referenced in the instruction and replaces IP.

For an intersegment direct CALL, SP is decremented by two, and CS is pushed onto the stack. CS is replaced by the segment word contained in the instruction. SP again is decremented by two. IP is pushed onto the stack and is replaced by the offset word contained in the instruction.

For an intersegment indirect CALL (which only may be made through memory), SP is decremented by two, and CS is pushed onto the stack. CS is then replaced by the content of the second word of the doubleword memory pointer referenced by the instruction. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the content of the first word of the doubleword pointer referenced by the instruction.

RET *optional-pop-value*

RET (Return) transfers control from a procedure back to the instruction following the CALL that activated the procedure. The assembler generates either an intrasegment RET, if the programmer has defined the procedure NEAR, or an intersegment RET, if the procedure has been defined as FAR. RET pops the word at the top of the stack (pointed to by register SP) into the instruction pointer and increments SP by two. If RET is intersegment, the word at the new top of stack is popped into the CS register, and SP is again incremented by two. If an optional pop value has been specified, RET adds that value to SP. This feature may be used to discard parameters pushed onto the stack before the execution of the CALL instruction.

JMP *Target*

JMP unconditionally transfers control to the target location. Unlike a CALL instruction, JMP does not save any information on the stack, and no return to the instruction following the JMP is expected. Like CALL, the address of the target operand may be obtained from the instruction itself (direct JMP) or from memory or a register referenced by the instruction (indirect JMP).

An intrasegment direct JMP changes the instruction pointer by adding the relative displacement of the target from the JMP instruction. If the assembler can determine that the target is within 127 bytes of the JMP, it automatically generates a two-byte form of this instruction called a SHORT JMP; otherwise, it generates a NEAR JMP that can address a target within +32k. Intrasegment direct JMPS are self-relative and are appropriate in position-independent (dynamically relocatable) routines in which the JMP and its target are in the same segment and are moved together.

An intrasegment indirect JMP may be made either through memory or through a 16-bit general register. In the first case, the content of the word referenced by the instruction replaces the instruction pointer. In the second case, the new IP value is taken from the register named in the instruction.

An intersegment direct JMP replaces IP and CS with values contained in the instruction.

An intersegment indirect JMP may be made only through memory. The first word of the doubleword pointer referenced by the instruction replaces IP, and the second word replaces CS.

CONDITIONAL TRANSFERS

The conditional transfer instructions are jumps that may or may not transfer control depending on the state of the CPU flags at the time the instruction is executed. These 18 instructions (see Table A-8) each test a different combination of flags for a condition. If the condition is true, then control is transferred to the target specified in the instruction. If the condition is false, then control passes to the instruction that follows the conditional jump. All conditional jumps are SHORT, that is, the target must be in the current code segment and within -128 to +127 bytes of the first byte of the next instruction (JMP 00H jumps to the first byte of the next instruction). Since the jump is made by adding the relative displacement of the target to the instruction pointer, all conditional jumps are self-relative and are appropriate for position-independent routines.

Table A-8: Interpretation of Conditional Transfers

MNEMONIC	CONDITION TESTED	"JUMP IF. . ."
JA/JNBE	(CF or ZF)=0	above/not below or equal
JAE/JNB	CF=0	above or equal/not below
JB/JNAE	CF=1	below/not above or equal
JBE/JNA	(CF or ZF)=1	below or equal/not above
JC	CF=1	carry
JE/JZ	ZF=1	equal/zero
JG/JNLE	((SF xor OF) or ZF)=0	greater/not less or equal
JGE/JNL	(SF xor OF)=0	greater or equal/not less
JL/JNGE	(SF xor OF)=1	less/not greater or equal
JLE/JNG	((SF xor OF) or ZF)=1	less or equal/not greater
JNC	CF=0	not carry
JNE/JNZ	ZF=0	not equal/not zero
JNO	OF=0	not overflow
JNP/JPO	PF=0	not parity/parity odd
JNS	SF=0	not sign
JO	OF=1	overflow
JP/JPE	PF=1	parity/parity equal
JS	SF=1	sign

NOTE: "above" and "below" refer to the relationship of two unsigned values;
"greater" and "less" refer to the relationship of two signed values.

ITERATION CONTROL

The iteration control instructions can be used to regulate the repetition of software loops. These instructions use the CX register as a counter. Like the conditional transfers, the iteration control instructions are self-relative and may only transfer to targets that are within -128 to +127 bytes of themselves, i.e., they are SHORT transfers.

LOOP *short-label*

LOOP decrements CX by 1 and transfers control to the target operand if CX is not 0; otherwise the instruction following LOOP is executed.

LOOPE/LOOPZ *short-label*

LOOPE and LOOPZ (Loop While Equal and Loop While Zero) are different mnemonics for the same instruction (similar to the REPE and REPZ repeat prefixes). CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and if ZF is set; otherwise the instruction following LOOPE or LOOPZ is executed.

LOOPNE/LOOPNZ *short-label*

LOOPNE and LOOPNZ (Loop While Not Equal and Loop While Not Zero) are also synonyms for the same instruction. CX is decremented by 1, and control is transferred to the target operand if CX is not 0 and ZF is clear; otherwise the next sequential instruction is executed.

JCXZ *short-label*

JCXZ (Jump If CX Zero) transfers control to the target operand if CX is 0. This instruction is useful at the beginning of a loop to bypass the loop if CX has a zero value, i.e., to execute the loop zero times.

INTERRUPT INSTRUCTIONS

The interrupt instructions allow interrupt service routines to be activated by programs as well as by external hardware devices. The effect of software interrupts is similar to hardware-initiated interrupts. However, the processor does not execute an interrupt acknowledge bus cycle if the interrupt originates in software or with an NMI. The effect of the interrupt instructions on the flags is covered in the description of each instruction.

INT *Interrupt-type*

INT (Interrupt) activates the interrupt procedure specified by the interrupt-type operand. INT decrements the stack pointer by two, pushes the flags onto the stack, and clears the trap flag (TF) and interrupt-enable flag (IF) to disable single-step and maskable interrupts. The flags are stored in the format used by the PUSHF instruction. SP is decremented again by two, and the CS register is pushed onto the stack. The address of the interrupt pointer is calculated by multiplying interrupt-type by four; the second word on the interrupt pointer replaces CS. SP again is decremented by two, and IP is pushed onto the stack and is replaced by the first word of the interrupt pointer. If interrupt-type=3, the assembler generates a short (1 byte) form of the instruction, known as the breakpoint interrupt.

Software interrupts can be used as supervisor calls—requests for service from an operating system. A different interrupt-type can be used for each type of service that the operating system could supply for an application program. Software interrupts also may be used to check out interrupt service procedures written for hardware-initiated interrupts.

INTO

INTO (Interrupt on Overflow) generates a software interrupt if the overflow flag (OF) is set; otherwise control proceeds to the following instruction without activating an interrupt procedure. INTO addresses the target interrupt pointer at location 10H; it clears the TF and IF flags and otherwise operates like INT. INTO may be written following an arithmetic or logical operation to activate an interrupt procedure if overflow occurs.

IRET

IRET (Interrupt Return) transfers control back to the point of interruption by popping IP, CS, and the flags from the stack. IRET thus affects all flags by restoring them to previously saved values. IRET is used to exit any interrupt procedure, whether activated by hardware or software.

PROCESSOR CONTROL INSTRUCTIONS

These instructions (see Table A-9) allow programs to control various CPU functions. One group of instructions updates flags, and another group is used primarily for synchronizing the 8086 or 8088 with external events. A final instruction causes the CPU to do nothing. Except for the flag operations, none of the processor control instructions affect the flags.

Table A-9: Processor Control Instructions

	FLAG OPERATIONS
STC	Set carry flag
CLC	Clear carry flag
CMC	Complement carry flag
STD	Set direction flag
CLD	Clear direction flag
STI	Set interrupt-enable flag
CLI	Clear interrupt-enable flag
	EXTERNAL SYNCHRONIZATION
HLT	Halt until interrupt or reset
WAIT	Wait for $\overline{\text{TEST}}$ pin active
ESC	Escape to external processor
LOCK	Lock bus during next instruction
	NO OPERATION
NOP	No operation

FLAG OPERATIONS

CLC

CLC (Clear Carry flag) zeroes the carry flag (CF) and affects no other flags. It (and CMC and STC) is useful in conjunction with the RCL and RCR instructions.

CMC

CMC (Complement Carry flag) toggles CF to its opposite state and affects no other flags.

STC

STC (Set Carry flag) sets CF to 1 and affects no other flags.

CLD

CLD (Clear Direction flag) zeroes DF, causing the string instructions to auto-increment the SI and/or DI index registers. CLD does not affect any other flags.

STD

STD (Set Direction flag) sets DF to 1, causing the string instructions to autodecrement the SI and/or DI index registers. STD does not affect any other flags.

CLI

CLI (Clear Interrupt-enable flag) zeroes IF. When the interrupt-enable flag is cleared, the 8086 and 8088 do not recognize an external interrupt request that appears on the INTR line; in other words, maskable interrupts are disabled. A nonmaskable interrupt appearing on the NMI line, however, is honored, as is a software interrupt. CLI does not affect any other flags.

STI

STI (Set Interrupt-enable flag) sets IF to 1, enabling processor recognition of maskable interrupt requests appearing on the INTR line. Note however, that a pending interrupt will not actually be recognized until the instruction following STI has executed. STI does not affect any other flags.

EXTERNAL SYNCHRONIZATION

HLT

HLT (Halt) causes the 8086/8088 to enter the halt state. The processor leaves the halt state upon activation of the RESET line, upon receipt of a nonmaskable interrupt request on NMI or, if interrupts are enabled, upon receipt of a maskable interrupt request on INTR. HLT does not affect any flags. It may be used as an alternative to an endless software loop in situations where a program must wait for an interrupt.

WAIT

WAIT causes the CPU to enter the wait state while its TEST line is not active. WAIT does not affect any flags.

ESC *external-opcode, source*

ESC (Escape) provides a means for an external processor to obtain an opcode and possibly a memory operand from the 8086 or 8088. The external opcode is a 6-bit immediate constant that the assembler encodes in the machine instruction it builds (see Table A-10). An external processor may monitor the system bus and capture this opcode when the ESC is fetched. If the source operand is a register, the processor does nothing. If the source operand is a memory variable, the processor obtains the operand from memory and discards it. An external processor may capture the memory operand when the processor reads it from memory.

LOCK

LOCK is a 1-byte prefix that causes the 8086/8088 (configured in maximum mode) to assert its bus LOCK signal while the following instruction executes. LOCK does not affect any flags.

NO OPERATION:NOP

NOP (No Operation) causes the CPU to do nothing. NOP does not affect any flags.

INSTRUCTION SET REFERENCE INFORMATION

Appendix I provides detailed operational information for the 8086/8088 instruction set.

The Expansion Bus is basically a buffered extension of the systems 8088 processor plus additional control and timing signals required to interface the system. The expansion bus consists of—

- A multiplexed buffered data bus, BD0-BD7
- A buffered address bus, A8-A19
- Various timing, control, interrupt, and power lines

Table B-1: Expansion Bus Pin Definition

PIN	SIGNAL	I/O	DESCRIPTION
50	A19	IO	Buffered Address Bits 8 to 19: These lines are driven from the 8088 during normal operation and are valid from the falling edge of ALE to the rising edge of the next ALE. If an external device takes control of the system via HOLD and HOLD ACKNOWLEDGE, these lines are tri-stated.
1	A18	IO	
49	A17	IO	
2	A16	IO	
48	A15	IO	
3	A14	IO	
47	A13	IO	
4	A12	IO	
46	A11	IO	
5	A10	IO	
45	A9	IO	
6	A8	IO	
29	BD7	IO	Time Multiplexed Buffered Address/Data Bus: During normal operation, the lower 8 bits of address, AD0-AD7, are valid on the falling edge of ALE.
22	BD6	IO	
28	BD5	IO	
23	BD4	IO	
27	BD3	IO	
24	BD2	IO	
26	BD1	IO	
25	BD0	IO	
9	ALE	O	Buffered Address Latch Enable: Processor signal which inditates BD0-BD7 contain valid addresses. Typically used to latch low-order 8 bits of address.
11	RD	O	Buffered Read Strobe: Processor signal indicating a read cycle.
14	$\overline{\text{WR}}$	O	Buffered Write Strobe: Processor signal indicating a write cycle.
8	$\overline{\text{DEN}}$	O	Buffered Data Enable: Provided by the processor for use as an enable for transceivers.
33	$\overline{\text{DLATCH}}$	O	Data Latch: The falling edge of this signal may be used to strobe data generated from a processor read access.
30	$\overline{\text{EXTIO}}$	I	External IO: Control line which prevents internal data bus buffers from conflicting with external buffers when mapping external IO into address space E0000 to EFFFF. CSEN should be used as a control signal to disable internal buffers via EXTIO and enable external buffers if using address space E0000 to EFFFF. Addresses used by the system cannot be disabled by EXTIO.

19	CSEN	O	Chip Select Enable: This line is synchronized to PHASE2. It is true from a falling edge of PHASE2 to the next falling edge of PHASE2, when address space E0000 to EFFFF is accessed.
40	CLK15B	O	15-Mhz Clock: Signal from which all system timing is derived. Its period is 66.6 nanoseconds with a 50%±10% duty cycle.
38	CLK5	O	5-Mhz Clock: Signal is in phase with the 8088 clock input. Its period is 200 nanoseconds with a 33% duty cycle.
20	PHASE2	O	1-Mhz Clock: Signal is asynchronous with CLK5. Its period is 1 microsecond with a 40/60% duty cycle. Useful to interface 6800-type I/O circuits.
21	XACK	I	External Acknowledge: This line is normally high and may be pulled low by external devices resulting in pulling the 8088 Ready input low, generating wait states. This line is resynchronized by the system logic.
17	$\overline{\text{HOLD}}$	I	Input to the 8088. This is an external request for control of the system buses.

Table B-1: Expansion Bus Pin Definition (Concluded)

PIN	SIGNAL	I/O	DESCRIPTION
18	HLDA	O	Buffered Hold Acknowledge: System response to "HOLD" request. When true (high) the following signals are tri-stated: A8-A19 BD0-BD7 ALE IO/$\overline{\text{M}}$ $\overline{\text{RD}}$ $\overline{\text{WR}}$ DT/$\overline{\text{R}}$ $\overline{\text{DEN}}$ $\overline{\text{SSO}}$ $\overline{\text{INTA}}$ $\overline{\text{DLATCH}}$ is controlled by external logic.
41	READY	O	Status Line: This line reflects the synchronized "ready" input to the 8088.
10	IO/$\overline{\text{M}}$	O	Buffered 8088 Status Line: Distinguishes between a memory or I/O bus cycle.
7	$\overline{\text{SSO}}$	O	Buffered 8088 Status Line.
12	DT/$\overline{\text{R}}$	O	Buffered Data Transmit/Receive: Processor signal typically used to control the direction of system transceivers. The combination of IO/$\overline{\text{M}}$, DT/$\overline{\text{R}}$, and $\overline{\text{SSO}}$ provide current bus cycle status:

IO/$\overline{M}$	DT/$\overline{R}$	$\overline{SSO}$	DESCRIPTION
0	0	0	Instruction fetch
0	0	1	Read from memory
0	1	0	Write from memory
0	1	1	Passive (no bus cycle)
1	0	0	Interrupt acknowledge
1	0	1	Read from I/O
1	1	0	Write to I/O
1	1	1	Halt

15	$\overline{NMI}$	I	Non-Maskable Interrupt: An edge-triggered input which causes a type-2 interrupt. A transition from high to low initiates the interrupt at the end of the current instruction.
16	$\overline{IRQ}$	I	Interrupt Request: This input should be driven with an open collector driver; it is "collector ORed" with five 6522s and one 6852 and is pulled to +5 volts through a 3.3K ohm resistor. A low level on any of these circuits generates a high level input to the system 8259 at IR3 level.
43	IR4	I	Interrupt Request Level 4: Direct access to IR4 of the system 8259.
42	IR5	I	Interrupt Request Level 5: Direct access to IR5 of the system 8259.
13	RESET	O	System Reset: Generated at power on or from the Reset switch.

PIN	SIGNAL	DESCRIPTION
44	Ground	
39	Ground	
35	Ground	
31	Ground	
37	+5volts	250 ma/expansion board
36	+5volts	
34	+12 volts	250 ma/expansion board
32	-12 volts	50 ma/expansion board

Table B-2: Expansion Bus Loading

SIGNAL	NORMAL USAGE I/O	INTERNAL LOAD	EXTERNAL DRIVE
Tri-Stated Lines			
A8-19	O	4	4
BDO-7	IO	5	4
ALE	O	5	4
RD	O	4	4
WR	O	4	4
DEN	O	4	4
IO/M	O	2	4
SSO	O	1	4
DT/R	O	4	4
TTL Outputs			
DLATCH	O	-	4*
CSEN	O	-	4*
C1K15B	O	-	1*
C1K5	O	-	4*
PHASE2	O	-	1*
HLDA	O	-	1*
READY	O	-	4
RESET	O	-	4

NOTE: All loads are 74LSXX loads of .4ma. External drive, as specified, is for each of the four slots available. Care must be taken to ensure adequate drive for other expansion modules which may be installed in the system.

*If required, buffer through one common IC package, such as 74LSO4.

Table B-3: Inputs Driven with Open Collector Drivers

SIGNAL	INTERNAL LOAD	PULLUP PROVIDED
$\overline{\text{EXTIO}}$	2	2.2K
$\overline{\text{XACK}}$	1	2.2K
$\overline{\text{HOLD}}$	1	2.2K
$\overline{\text{NMI}}$	1	2.2K
$\overline{\text{IRQ}}$	1	3.3K

Table B-4: Inputs Direct to System 8259

IR4
IR5

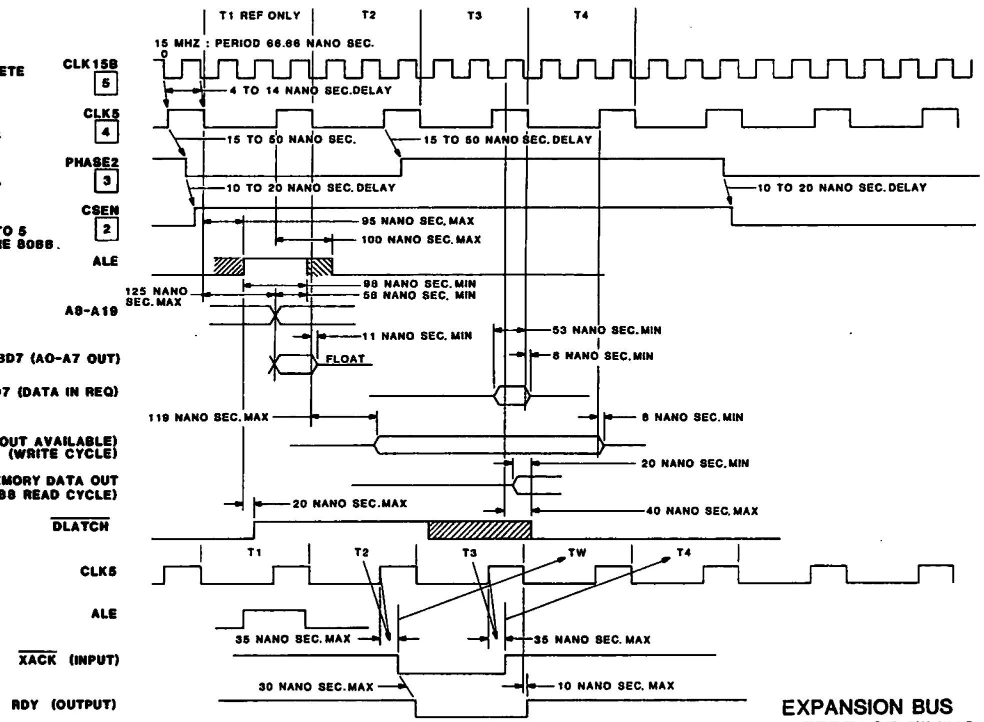
Figure B-1: Expansion Connector

BDO	25	26	BD1
BD2	24	27	BD3
BD4	23	28	BD5
BD6	22	29	BD7
ZACK	21	30	EXTIO
PHASE 2	20	31	Ground
CSEN	19	32	-12 volts
HLDA	18	33	DLATCH
HOLD	17	34	+12 volts
IRQ	16	35	Ground
NMI	15	36	+5 volts
WR	14	37	+5 volts
Reset	13	38	CLK5
DT/R	12	39	Ground
RD	11	40	CLK15B
IO/M	10	41	Ready
ALE	9	42	IR5
DEN	8	43	IR4
SSO	7	44	Ground
A 8	6	45	A 9
A10	5	46	A11
A12	4	47	A13
A14	3	48	A15
A16	2	49	A17
A18	1	50	A19

Figure B-2: Expansion Bus Interface Timing

NOTES

1. BDO-BD7 IS A TIME MULTIPLEXED BI-DIRECTIONAL BUS.
2. CSEN IS ACTIVE HIGH FOR A COMPLETE PHASE 2/PHASE 2 CYCLE WHEN ACCESSING ADDRESS SPACE E0000 TO EFFFF.
3. HI-600 ± 36 NANO SEC PHASE 2 IS NOT PHASE LOCKED TO ALE.
   LO-400 ± 36 NANO SEC
   IF USED FOR TIMING I/O CIRCUITS, THE 8088 MUST BE SYNCRONIZED VIA XACK/READY.
4. IN PHASE WITH AND TRAILING, BY 3 TO 5 NANO SEC, THE INPUT CLOCK TO THE 8088.
5. 50 ± 10% DUTY CYCLE.



EXPANSION BUS INTERFACE TIMING

Appendix C MEMORY MAPPED I/O ADDRESS AND BIT ASSIGNMENTS

Table C-1: 8259A (PIC IOD0)
Address: E0000-E0001

INTERRUPT LEVEL	SIGNAL NAME	DESCRIPTION
IR0	SYN	SYNC DETECT
IR1	COMM	SERIAL COMMUNICATIONS (7201)
IR2	TIMER	8253 TIMER
IR3	PARALLEL	ALL 6522 IRQ (INCLUDING DISK)
IR4	IR4	EXPANSION IR4
IR5	IR5	EXPANSION IR5
IR6	KBINT	KEYBOARD DATA READY
IR7	VINT	VERTICAL SYNC OR NONSPECIFIC INTERRUPT

Table C-2: 8253 (TIMER-IOD1)
Address: E0020-E0023

I/O NAME	SIGNAL NAME	DESCRIPTION
CLK2	100KHZ	CLOCK INPUT (FOR TIME OF DAY)
GATE2	+5 V	
OUT2	TIMER	INTERRUPT FOR TIME OF DAY
GLK1	1.25 MHZ	CLOCK INPUT FOR SERIAL PORT B
GATE1	+5 V	
OUT1	MUX SERIAL B	TO SERIAL PORT B MUX
CLK0	1.25 MHZ	CLOCK INPUT FOR SERIAL PORT A
GATE0	+5 V	
OUT0	MUX SERIAL A	TO SERIAL PORT A MUX

Table C-3: 7201(COMM.CTLR IOD2)
Address: E0040-E0043

I/O NAME	SIGNAL NAME	DESCRIPTION
RXCA	J8-17	RECEIVE CLK A
TXCA	J8-15	TRANSMIT CLK A
RXDA	J8-3	RECEIVE DATA A
TXDA	J8-2	TRANSMIT DATA A
CTSA	J8-5	CLEAR TO SEND A
RTSA	J8-4	REQUEST TO SEND A
DCDA	J8-8	DATA CARRIER DETECT A INPUT
DTRA	J8-20	DATA TERMINAL READY A
RXCB	J9-17	RECEIVE CLK B
TXCB	J9-15	TRANSMIT CLK B
RXDB	J9-3	RECEIVE DATA B
TXDB	J9-2	TRANSMIT DATA B
CTSB	J9-5	CLEAR TO SEND B
RTSB	J9-4	REQUEST TO SEND B
DCDB	J9-8	DATA CARRIER DETECT B INPUT
DTRB	J9-20	DATA TERMINAL READY B

Table C-4: HD46505S (CRTC CSO)
Address: E8000-E8001

INTERRUPT LEVEL	SIGNAL NAME	DESCRIPTION
MA13	HIRES	HIRES ENABLE OUTPUT
MA12	DOT ADDR	32K WORD PAGE SELECT OUTPUT (1=UPPER)

Table C-5: 6522 (VIA 1 CS1)
Address: E8020-E802F

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	DIO1	Parallel data bit 0, IN/OUT
PA1	DIO2	Parallel data bit 1, IN/OUT
PA2	DIO3	Parallel data bit 2, IN/OUT
PA3	DIO4	Parallel data bit 3, IN/OUT
PA4	DIO5	Parallel data bit 4, IN/OUT
PA5	DIO6	Parallel data bit 5, IN/OUT
PA6	DIO7	Parallel data bit 6, IN/OUT
PA7	DIO8	Parallel data bit 7, IN/OUT
CA1	NRFD	Parallel NRFD interrupt input
CA2	NDAC	Parallel NDAC interrupt input
PB0	DAV	Parallel DAV, IN/OUT
PB1	EOI	Parallel EOI, IN/OUT
PB2	REN	Parallel REN, IN/OUT
PB3	ATN	Parallel ATN, IN/OUT
PB4	IFC	Parallel IFC, IN/OUT
PB5	SRQ	Parallel SRQ, IN/OUT
PB6	NRFD	Parallel NRFD, IN/OUT
PB7	NDAC	Parallel NDAC, IN/OUT
CB1	N.C.	
CB2	CODEC VOL	Pulse width control CODEC Vol output (TZ)

Table C-6: 6522 (VIA 2 CS2)
Address: E8040-E804F

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	$\overline{\text{INT}}$/EXTA	Serial A clock select (LOW=INT)
PA1	INT/EXTB	Serial B clock select (LOW=INT)
PA2	RIA	Serial A ring indicate (J8-22)
PA3	DSRA	Serial A data set ready (J8-6)
PA4	RIB	Serial B ring indicate (J9-22)
PA5	DSRB	Serial B data set ready (J9-6)
PA6	KBDATA	Data from keyboard
PA7	$\overline{\text{VERT}}$	Vertical signal input (from CRTC)
CA1	NC	
CA2	SRQ/BUSY	Parallel port IN/OUT
PB0	TALK/LISTEN	Parallel port direction, control, output
PB1	KBACKCTL	Keyboard acknowledge, control, output
PB2	BRT0	LSB of brightness control, output
PB3	BRT1	Intermediate bit of brightness control, output
PB4	BRT2	MSB of brightness control, output
PB5	CONT0	LSB of contrast control, output
PB6	CONT1	Intermediate bit of contrast control, output
PB7	CONT2	MSB of contrast control, output
CB1	KBRDY	Key data ready, input
CB2	KBDATA	Shift register input

Table C-7: 6852 (SSDA CS3)
Address: E8060-E806F

I/O NAME	SIGNAL NAME	DESCRIPTION
RXCLK		Inverted input from PB7 of VIA3 (CODEC CLOCK)
TXCLK		Inverted input from PB7 of VIA3 (CODEC CLOCK)
RXDATA		Input digital data from CODEC
TXDATA		Digital data output to CODEC
SM/DTR		Encode/Decode control for CODEC (Low=Decode, or transmit)
DCD		Inverted input from SM/DTR of this chip
CTS		Input from SM/DTR of this chip

Table C-8: 6522 (VIA 3 CS4)
Address: E8080-E808

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	J5-16	Control Port
PA1	J5-18	Control Port
PA2	J5-20	Control Port
PA3	J5-22	Control Port
PA4	J5-24	Control Port
PA5	J5-26	Control Port
PA6	J5-28	Control Port
PA7	J5-30	Control Port
CA1	J5-12	Control Port
CA2	J5-14	Control Port
PB0	J5-32	Control Port
PB1	J5-34	Control Port
PB2	J5-36	Control Port
PB3	J5-38	Control Port
PB4	J5-40	Control Port
PB5	J5-42	Control Port
PB6	J5-44	Control Port
PB7	J5-46	CODEC Clock Output
CB1	J5-48	Control Port
CB2	J5-50	Control Port

Table C-9: 6522 (VIA 4 CS5)
Address: E80A0-E80AF

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	L0MS0	Drive 0 motor speed, outputs (also
PA1	L0MS1	used as a data bus to load 8048
PA2	L0MS2	parameters during motor speed
PA3	L0MS3	controller initialization)
PA4	ST0A	
PA5	ST0B	Drive 0 stepper phase, outputs
PA6	ST0C	
PA7	ST0D	
CA1	DS0	Door 0 sense interrupt, input
CA2	MODE	Write sync
PB0	L1MS0	
PB1	L1MS1	Drive 1 motor speed, outputs
PB2	L1MS2	
PB3	L1MS3	
PB4	ST1A	
PB5	ST1B	
PB6	ST1C	Drive 1 stepper phase, outputs
PB7	ST1D	
CB1	DS1	Door 1 sense interrupt, input
CB2	N.C.	

Table C-10: 6522 (VIA 6 CS6)
Address: E80C0-E80CF

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	LED0A	LED, drive A, output
PA1	TRK0D0	Track 0, drive A sense, input
PA2	LED1A	LED, drive B, output
PA3	TRK0D1	Track 0, drive B sense, input
PA4	Side Select	Dual side select, output
PA5	Drive Select	Select drive A/B, output
PA6	WPS	Write protect sense, input
PA7	SYNC	Disk sync detect, input
CA1	GCRERR	GCR error input
CA2	DRW	Disk read/write CTRL, output
*PB0	RDY0	Motor speed status, drive A
*PB1	RDY1	Motor speed status, drive B
PB2	SCRESET	Motor speed controller (8048) reset, output
PB3	DS1	Door B sense, input
PB4	DS0	Door A sense, input
PB5		Single/Double sided
PB6		Stepper enable A
PB7		Stepper enable B
CB1	N.C.	
CB2	Erase	Erase head On/Off, output

*Also used as handshake lines during speed controller initialization.

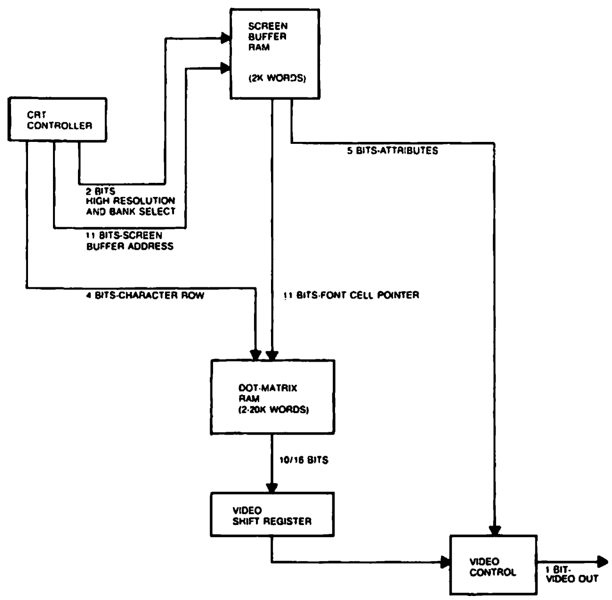
Table C-11: 6522 (VIA 5 CS7)
Address: E80E0-E80EF

I/O NAME	SIGNAL NAME	DESCRIPTION
PA0	E0	
PA1	E1	
PA2	I2	
PA3	E2	Disk data inputs
PA4	E4	
PA5	E5	
PA6	I7	
PA7	E6	
CA1	BRDY	Byte ready input
CA2	RDY0	Motor speed status interrupt, drive 0
PB0	WD0	
PB1	WD1	
PB2	WD2	
PB3	WD3	Disk data outputs
PB4	WD4	
PB5	WD5	
PB6	WD6	
PB7	WD7	
CB1	N.C.	
CB2	RDY1	Motor speed status interrupt, drive 1

INTRODUCTION

The display hardware is a memory-mapped raster scan system. The display RAM physically occupies 4K bytes, starting at F0000H, plus from 4K to 40K bytes of the lower 128 bytes in the 8088 memory map. The display RAM is organized in two separate banks, which operate in a pipelined fashion (see Figure D-1). The first bank is the screen bufgfer; it contains the characters which are to be displayed on the screen. The screen buffer also contains attribute information for each character location. The character selection code (called the font cell pointer), together with the character row number (0-15) is used as the address for the second bank, which contains patterns for the characters (font cells). To generate video, the font cell patterns are accessed and latched into the video shift register.

Figure D-1: Display System Organization



The display hardware is capable of 80 columns by 25 lines of text. The text character cells are 10 dots wide by 16 lines high. These character cells are RAM-mapped and programmable. There is also a 5-bit attribute code associated with each character. Four of these attribute bits are used for reverse-video, underline/strikeover, highlight, and nondisplay. The other bit is available for user software or external hardware. The display hardware can also be configured for a high-resolution mode: 800 by 400 dots of bit-addressable display. In this mode, the reverse-video, double intensity, and nondisplay attributes apply to fixed (16-by-16-dot) cells on the screen, and the underline/strikeover attribute is not operative.

The character and attribute bits are organized into words called the screen buffer. The lower 11 bits of each word define which of the 2048 possible characters is to be placed at that location of the screen. These 11 bits are collectively called the font cell pointer. The upper five bits of the word are the attributes. The MSB (bit 15) is the reverse-video bit. Bit 14 is the low-intensity bit; bit 13 is the underline bit; and bit 12 is the nondisplay bit. The remaining bit is uncommitted.

The screen buffer words are on even-address boundaries. The physical memory of the screen buffer is located, in system address space, at F0000 to F0FFF. The 80-character by 25-line display occupies 2000 words (4000 bytes) of the available 2048 words in the screen buffer. Logically, the screen buffer is mapped to include locations F0000 to F1FFF. Therefore, addressing location F0000 accesses the same physical word as addressing location F1000. The logical beginning of the display screen is selected by a pair of registers in the CRT controller chip (this is a word address). This register pair may be programmed to move the starting address (line one, column one) of the display to any word of the screen buffer. When the control register pair is used in this manner, the screen buffer functions as a 2048-word circular buffer. Using this technique, line scrolling in the text mode may be accomplished by adding 80 to the contents of the screen start register and blanking the 80 words following the previous end of screen. In both these operations, to keep the address within the screen buffer address space, it is also necessary to logically AND the resulting address with F1 FFF.

Figure D-2: Screen Buffer World Format

```
15                                                        0

X  X  X  X  X  X  X  X  X  X  X  X  X  X  X  X
|  |  |  |  |  |                                          |
|  |  |  |  |  |_______________FONT CELL POINTER__________|
|  |  |  |  |
|  |  |  |  |-- RESERVED
|  |  |  |
|  |  |  |-- NON-DISPLAY
|  |  |
|  |  |-- UNDERLINE
|  |
|  |-- LOW-INTENSITY
|
|-- REVERSE-VIDEO
```

The actual dot patterns of each character are stored in the font cell memory. Each 10-dot-by-16-line character cell is stored in 16 consecutive words. This group of 16 words is called a font cell. The lower 10 bits of each word contain the 10 dots of a scan line of the character picture. The upper-left bit of a character would be the LSB of the first word in the 16 consecutive words that define a font cell. Bit 15 of each font cell word is reserved for the underline/strikeover flag bit (in text mode, only). If bit 15 is set and the underline/strikeover attribute (bit 13) from the screen buffer is set, then that scan line will be white; otherwise, the lower 10 bits in that word will be displayed. The nondisplay bit can be used to create "secret" (nondisplayed) characters or fields. If a minimum (128-character) set is defined, the font cells would occupy 4K bytes of memory. The font cells can be located anywhere within the first 128K bytes of RAM, but may not cross the 64K boundary.

HIGH RESOLUTION MODE

The 800-by-400-dot, bit-mapped, high-resolution display is a special-case use of the cell graphics. The output line, called HIRES (from the CRT controller), controls the character cell width. When this line is high, the character cells are 16 dots wide instead of the usual 10 dots. The screen is then organized as 50 columns by 25 lines of 16-by-16-dot font cells. This is accomplished by writing new values into the control registers of the CRT controller. The full 16 bits of each font cell word are used to describe the picture of each character. The screen buffer is organized so that each of the 1250 characters on the screen is a different character, as described earlier in this manual. High-resolution software then operates directly on the font cell memory for display bit manipulation.

> Programming Note: The HIRES/TEXT control and the DOTSEL control (which select whether the beginning address of the font cell memory is to be in the first or the second 64K of system memory) are manipulated via the two high-order address bits in the CRTC display address register pair, R12 and R13. This address interacts with the cursor register pair, R14 and R15, and the light pen register pair, R16 and R17. Specifically, if the light pen register pair is used and/or the cursor-display function is desired, then the software must (1) add the cursor address to the current settings of HIRES/TEXT and DOTSEL and (2) subtract or mask these bits when interpreting a light pen interrupt.

BRIGHTNESS AND CONTRAST CONTROL

The overall display brightness and the contrast between high and low intensity characters are software adjustable.

Brightness may be adjusted to one of eight different levels by setting the brightness control bits (PB2, PB3, and PB4 of the 6522 at E8040) to the binary value corresponding to the desired level. The binary-value range from zero to seven selects increasing brightness levels.

The contrast function controls the difference in intensity between highlighted characters and normal intensity characters. Only the intensity of the normal intensity characters is varied by the contrast function. The contrast function selects one of eight levels by setting the binary value of the desired level in the three contrast control bits

(PB5, PB6, and PB7 of the 6522 at E8040). A value range of zero to seven selects increasing differences between the normal and highlighted characters, with zero causing no difference.

CIRCUIT DESCRIPTION

The lower 128K bytes of RAM is a dual-port memory system. One port is used by the display hardware to refresh the raster-scan display. The other port is used by the 8088 microprocessor for read and write operations. The dual-port memory is managed by an arbitrator circuit that guarantees one refresh access to the display RAM every character cell time. The arbitrator circuit adds a wait state to any 8088 memory cycle if this is necessary to isolate it from the display-refresh cycle. This results in an average of one wait state (200 nsec) for every five processor memory access cycles. Processor and memory cycles are normally four clock periods (200 nsec). This could cause a decrease of approximately 5% in system bus performance. However, due to the 8088 instruction lookahead queue, this decrease in bus performance rarely translates into decreased system performance.

The display-refresh addresses are generated by the HD46505S CRT-controller chip (CRTC). Of the 14 address lines from the CRTC, 11 (MA0-MA10) are used to address the 2K words of screen buffer RAM. The 16 data lines output by the screen buffer are latched and divided into 11 lines of character address information and 5 lines of character attributes. The attribute bits are sent, via a set of character sync registers, to the video control section. The 11 lines of the character address are combined with 4 lines of character-row address and MA12 (DOTSEL) from the CRTC. This address is then multiplexed down to 8 font cell address lines. The 14th character address line (MA13) is used to select the high-resolution mode. The 16-bit data output word from each font cell word is latched and sent to a 16-bit shift register. Either 10 or 16 dots of the shift register are shifted out to the video control section. The video control section adds the reverse video, highlight, underline, and nondisplay attribute bits and the cursor output from the CRTC. The result is sent to the video display, along with horizontal and vertical sync pulses.

The display circuit manages the memory refresh in the 128K bytes of on-board dynamic RAM. The horizontal and vertical retrace intervals are used for memory refresh. Display-refresh cycles occuring during retrace intervals cause 8 bits from the refresh-address counter to be sent to all 128K of dynamic RAM, rather than the normal display-address lines. The display CAS signal is inhibited for a RAS-only memory refresh. The memory-refresh counter is clocked after each refresh cycle. In every 64 microsecond horizontal display period, 15 memory-refresh cycles occur. Every 2 ms, 480 memory-refresh addresses are generated, exceeding the 128-address-per-2ms specified requirement of 16K dynamic RAM.

CRTC DEVICE OPERATION

The CRTC consists of an internal register group, horizontal and vertical timing circuits, a linear address generator, a cursor-control circuit, and a light-pen-detection circuit. Horizontal and vertical timing circuits generate RA0-RA4 , DISPTMG, SYNC, and VSYNC. RA0-RA4 are raster (row) address signals and are used as address bits 1 to 4 for the font cell accesses. DISPTMG, HSYNC, and VSYNC signals are sent to the video control circuit. This horizontal and vertical timing circuit consists of an internal counter and comparator circuit.

The linear address generator generates refresh memory address MA0-MA11 to be used for refreshing the screen. The light-pen-detection circuit detects the light pen position on the screen. When the light pen strobe signal is received, the light pen register latches the address generated by the linear address generator to save the position of the pen on the screen. The cursor control circuit controls the position of the cursor, its height, and its blink rate.

The CRTC provides 13 interface signals to the CPU and 25 interface signals to the display circuits.

Table D-1: Recommended Values For CRTC Register Initialization

REGISTER	CHARACTER MODE	HIGH RESOLUTION MODE
R0	5C	3A
R1	50	32
R2	51	34
R3	CF	C9
R4	19	19
R5	06	06
R6	19	19
R7	19	19
R8	03	03
R9	0E	0E
R10	60	20
R11	0F	0F
R12	00	20
R13	00	00
R14	00	00
R15	00	00

NOTE: All values are in hexadecimal.

INTERFACE SIGNALS TO THE CPU

Bidirectional Data Bus (ID0-ID7)

The bidirectional data bus is used for data transfer betweeen the CRTC and the 8088. The data bus outputs are 3-state buffers and remain in the high-impedance state except when the 8088 performs a CRTC read operation.

Read/Write (R/W)

The R/W signal controls the direction of data transfer between the CRTC and the 8088. When R/W is high, CRTC data is transferred to the 8088. When R/W is low, 8088 data is transferred to the CRTC.

Chip Select (CS)

The CS signal is used to address the CRTC. When CS is low, it enables R/W operation to CRTC internal registers. This signal is derived from decoded address signals of the the 8088.

Register Select (RS)

The RS signal is used to select the address register and the 18 control registers of the CRTC. When RS is low, the address register is selected; when RS is high, control registers are selected. This signal is the lowest bit (A0) of the 8088 address bus.

Enable (E)

The E signal is used as strobe signal in 8088 R/W operations with the CRTC internal registers. This signal is PHASE2.

Reset (RES)

The Reset signal (RES) is an input signal used to reset the CRTC. When RES is low, it forces the CRTC into the following status:

- ▶ All the counters in the CRTC are cleared, and the device stops the display operation
- ▶ All the outputs go low
- ▶ Control registers in the CRTC are not affected

INTERFACE SIGNALS TO DISPLAY CIRCUITS

Character Clock (CLK)

CLK is a standard clock input signal which defines character timing for the CRTC display operation. This signal is provided by the memory controller.

Horizontal Sync (HSYNC)

HSYNC is an active high-level signal which provides horizontal synchronization for the display device.

Vertical Sync (VSYNC)

VSYNC is an active high-level signal which provides vertical synchronization for the display device.

Display Timing (DISPTMG)

DISPTMG is an active high-level signal which defines the display period in horizontal and vertical raster scanning. It is necessary to enable the video signal only when DISPTMG is high.

Refresh Memory Address MA0-MA13

MA0-MA11 are refresh memory address signals which are used to access the screen buffer in order to refresh the CRT screen periodically.

MA11 is unused.

MA12 selects the 64K memory bank to be used for font cell memory.

When MA12 equals 0, it selects system RAM starting at location 0; when MA12 equals 1, it selects system RAM starting at location 10000H.

When MA13 equals 0, it selects text mode when MA13 equals one, it selects bit-mapped HIRES mode.

Raster Address (RA0-RA4)

RA0-RA4 are row-address signals which are used to select the row of the current character in the font cell memory to be displayed.

Cursor Display (CUDISP)

CUDISP is an active high-level video signal which is used to display the cursor on the CRT screen at the current display location. This output is inhibited while DISPTMG is low. This output is mixed with the video signal and is provided to the CRT display circuits.

Light Pen Strobe (LPSTB)

LPSTB is an active high-level input signal which accepts a strobe pulse detected by the light pen and control circuit. When this signal is activated, the memory address (MA0-MA11), along with the current settings of HIRES and DOTADR, are stored in the 14bit light-pen register. The stored memory address needs to be corrected in software, taking the delay time of the display device, light pen, and light-pen-control circuits into account.

INTERNAL REGISTERS

ADDRESS REGISTER (AR)

AR is a 5-bit register used to select among the 18 internal control registers (R0-R17). The value of AR is the address of one of 18 internal control registers. Data values from 18 to 31 do nothing. Access to R0-R17 requires writing the address of the corresponding control register into this register.

HORIZONTAL TOTAL REGISTER (R0)

The contents of R0 program the total number of horizontal character-clock periods per line, including the retrace period. The data is 8-bit, and its value should be programmed according to the selected mode of the display. The programmed value must be one less than the number of character intervals required. When programming for interlace mode, the value must be even.

HORIZONTAL DISPLAYED REGISTER (R1)

R1 is used to program the number of displayed characters per horizontal line. Data is 8-bit, and any value smaller than that in R0 is valid.

HORIZONTAL SYNC POSITION REGISTER (R2)

The contents of R2 program the horizontal sync position in units of the character-clock period. Data is 8-bit, and any value less than R0 is valid. The value programmed should be one less than the sync position desired. The effect of increasing the value in R2 is to shift all characters displayed on the CRT screen to the left. When the value is decreased, character positions shift to the right.

SYNC WIDTH REGISTER (R3)

The contents of R3 set the horizontal sync pulse width and the vertical sync pulse width. The horizontal sync pulse width is programmed in the lower 4 bits, in units of the character-clock period (0 is invalid). The vertical sync pulse width is programmed in the upper 4 bits, in units of the horizontal period. When 0 is programmed in the upper 4 bits, 16 horizontal periods are specified.

VERTICAL TOTAL REGISTER (R4)

R4 is used to program the total number of horizontal scans per frame, including the vertical retrace period. This is a 7-bit value and should be programmed according to the selected display mode. The programmed value should be one less than the number desired.

VERTICAL TOTAL ADJUST REGISTER (R5)

The contents of R5 select the total number of horizontal scans per field. This register allows fine control of the deflection frequency.

VERTICAL DISPLAYED REGISTER (R6)

R6 is used to determine the number of displayed character rows on the CRT screen. This is a 7-bit value, and any number that is smaller than that in R5 is valid.

Table D-2: Pulse Width of Vertical Sync Signal

VSW 2^7	2^6	2^5	2^4	PULSE WIDTH (# Rows)
0	0	0	0	16H
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

NOTE: H=horizontal period.

Table D-3: Pulse Width of Horizontal Sync Signal

HSW 2^3	2^2	2^1	2^0	PULSE WIDTH (# Characters)
0	0	0	0	(not used)
0	0	0	1	1CH
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14
1	1	1	1	15

NOTE: CH=character period; HSW=0 cannot be used.

VERTICAL SYNC POSITION REGISTER (R7)

The contents of R7 set the vertical sync position on the screen, in units of the horizontal character line period. Data is 7-bit, and any number that is equal to or less than the vertical total register can be programmed. The value programmed should be one less than the position desired. Increasing the value shifts the display upward. Decreasing the values shifts the display downward.

INTERLACE AND SKEW REGISTER (R8)

R8 programs the raster-scan mode and the skew (delay) of CUDISP and DISPTMG.

INTERLACE MODE PROGRAM BITS (V, S)

The raster-scan mode is selected by the V and S bits.

Table D-4: Interlace Mode (D0, D1)

V BIT	S BIT	RASTER-SCAN MODE
0	0	Noninterlace mode
1	0	Noninterlace mode
0	1	Interlace sync mode
1	1	Interlace sync and video mode

SKEW PROGRAM BIT (C1, C0, D1, D0)

The C1, C0, D1, and D0 bits are used to program the skew (delay) of CUDISP and DISPTMG.

The skews of the two signals are programmed separately.

Table D-5: DISPTMG Skew Bit (D7, D6)

D1 BIT	D0 BIT	DISPTMG SIGNAL
0	0	Zero skew
0	1	One-character skew
1	0	Two-character skew
1	1	No output

Table D-6: Cursor Skew Bit (D5, D4)

C1 BIT	C0 BIT	NON SKEW
0	0	Zero skew
0	1	One-character skew
1	0	Two-character skew
1	1	No output

The skew function is used to delay the CUDISP and DISPTMG signals for optimum screen-memory access, dot-matrix memory, and video signal timing.

MAXIMUM RASTER ADDRESS REGISTER (R9)

R9 is used to program the maximum raster address (5 bits). This register defines the number of rasters (lines) per character, including intercharacter spaces. Programming is as follows:

- ▶ Noninterlace Mode

  In the following tabulation, the value parameter is set at 4.

RASTER ADDRESS	RESULTING FORMAT
0	- - - - - - - - - -
1	- - - - - - - - - -
2	- - - - - - - - - -
3	- - - - - - - - - -
4	- - - - - - - - - -

NOTE: The number of rasters produced in the character format is 5 (one more than the value programmed).

- ▶ Interlace Sync Mode

  In the following tabulation, the value parameter is 4.

RASTER ADDRESS	RESULTING FORMAT
0	- - - - - - -
0	. . . .
1	- - - - - - -
1	. . . .
2	- - - - - - -
2	. . . .
3	- - - - - - -
3	. . . .
4	- - - - - - -
4	. . . .

NOTE: - - - - - - - and . . . . denote alternate fields.

The total number of rasters in the character is 10. The number is found by doubling the sum of one plus the value programmed.

- ▶ Interlace Sync and Video Mode

The total number of rasters in the character format is one more than the value parameter, as in the noninterlace mode, but the rasters alternate fields. In the following tabulation, a value parameter of 4 is set.

RASTER ADDRESS	RESULTING FORMAT
0	- - - - - - -
1	. . . .
2	- - - - - - -
3	. . . .
4	- - - - - - -

NOTE: - - - - - - - and . . . . denote alternate fields.

CURSOR START RASTER REGISTER (R10)

RIO programs the cursor-start raster (line) address and the cursor-display mode. The lower 5 bits (D0-D4) are cursor-start, and the next 2 bits (D5, D6) are cursor-mode.

Table D-7: Cursor Display Mode (D6, D5)

D5	D6	CURSOR DISPLAY MODE
0	0	Steady cursor
0	1	Cursor off
1	0	Blinking cursor, 16-field period
1	1	Blinking cursor, 32-field period

CURSOR END RASTER REGISTER (R11)

R11 sets the cursor-end raster (line) address.

START ADDRESS REGISTERS (R12, R13)

R12 and R13 are used to program the first (word) address of the screen buffer memory to be displayed. This word will display as line one/column one on the display screen.

CURSOR REGISTERS (R14, R15)

The two read/write registers R14 and R15 store the cursor location. The upper 2 bits (D6, D7) of R14 must always be set to "0".

LIGHT PEN REGISTERS (R16, R17)

The read-only registers R16 and R17 are used to latch the detection-time address of the light pen. The upper 2 bits (D6, D7) of R16 are always "0". The value latched may need to be corrected by software to allow for light pen system delays.

RESTRICTIONS ON PROGRAMMING INTERNAL REGISTERS

The following restrictions on programming internal registers apply:

- 0†Nhd†(Nht + 1)†=256
- 0†Nvd†(Nvt + 1)†=128
- 0†=Nhsp†=Nht
- 0†=Nvsp†=Nvt*
- 0=†NCSTART=†NCEND=†Nr (noninterlace, interlace sync mode)
  0=†NCSTART†NCEND=†Nr+1 (interlace sync and video mode)
- 2=†Nr=†30
- 3=†Nht (except non interlace mode)
  5=†Nht (noninterlace mode only)

*In interlace mode, pulse width is changed +1/2 by the raster time when the vertical sync signal extends over two fields.

NOTES: The values programmed in the internal registers of the CRTC are used directly to control the CRT. Consequently, the display may flicker if the contents of the registers are changed asynchronously to the display operation. The registers should be changed only during the horizontal or vertical retrace period.

NONINTERLACE MODE DISPLAY

Alternate fields are identical. The values of raster addresses (RA0-RA4) are counted, starting at zero.

INTERLACE SYNC MODE DISPLAY

In the interlace sync mode, raster addresses in the even field and the odd field are the same. The same character pattern is displayed in both fields with the displayed position in the odd field 1/2 raster space down from that in the even field.

INTERLACE SYNC AND VIDEO MODE DISPLAY

In interlace sync and video mode, when the raster number is even, the output raster address is different from when the raster number is odd.

Table D-8: Programmed Values into the Registers

REGISTER	REGISTER NAME	VALUE
R0	Horizontal total	Nht
R1	Horizontal displayed	Nhd
R2	Horizontal sync position	Nhsp
R3	Sync width	Nvsw, Nhsw
R4	Vertical total	Nvt
R5	Vertical total adjust	Nadj
R6	Vertical displayed	Nvd
R7	Vertical sync position	Nvsp
R8	Interlace and skew	
R9	Maximum raster address	Nn
R10	Cursor start raster	
R11	Cursor end raster	
R12	Start address (H)	0
R13	Start address (L)	0
R14	Cursor (H)	
R15	Cursor (L)	
R16	Light pen (H)	
R17	Light pen (L)	

NOTE: Nhd†Nht, Nvd†Nvt

Table D-9: Output Raster Address in Interlace Sync and Video Mode

TOTAL NUMBER OF RASTERS IN THE CHARACTER FORMAT	FIELD EVEN	FIELD ODD
Even	Even Address	Odd Address
Odd		
Even Line	Even Address	Odd Address
Odd Line	Odd Address	Even Address

NOTE: Internal line address begins from zero.

NOTE: A wide disparity in the number of ON dots in even fields versus that in odd fields causes unequal average beam currents during alternate fields. This causes CRT final-anode voltage to differ during alternate fields. Since the deflection factor is a function of this voltage, the two fields will have somewhat different widths. Characters will be distorted, particularly near the edges of the screen. Programming for an odd number of rasters per character line is a good way to reduce this type of problem.

CURSOR CONTROL

Figure D-3 shows display patterns in which various values are stored in the cursor-start-raster register and the cursor-end-raster register. Values in the cursor-start-raster register and the cursor-end-raster register must meet the following conditions: cursor-start-raster†= cursor end raster register†= maximum raster address.

Figure D-3: Cursor Control

```
0-----------------------------------        0-----------------------------------
1-----------------------------------        1-----------------------------------
2-----------------------------------        2-----------------------------------
3-----------------------------------        3-----------------------------------
4-----------------------------------        4-----------------------------------
5-----------------------------------        5-----------------------------------
6-----------------------------------        6-----------------------------------
7-----------------------------------        7-----------------------------------
8-----------------------------------        8-----------------------------------
9 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 -  9 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 -
10----------------------------------        10 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 -

Cursor Start Address = 9                    Cursor Start Address = 9
Cursor End Address = 9                      Cursor End Address = 10

0-----------------------------------
1 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 -
2 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 -
3 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 -
4 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 -
5 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 - 0 -
6-----------------------------------
7-----------------------------------
8-----------------------------------
9-----------------------------------
10----------------------------------
```

Audio output from and (optionally) input to the system are provided by a built-in coder/decoder (CODEC), which uses a Continuously-Variable-Slope Delta modulation (CVSD) technique. This device produces audio output by converting a single-bit, digital-bit stream to an analog output.

The bit-stream interface is provided by the 6852-SSDA chip which converts 8-bit data bytes from the processor to a bit-serial data stream for the CODEC. The SSDA also provides encode/decode control, via the DTR output, and a 3-byte FIFO buffer which reduces the real-time processor servicing requirements.

Additional control of the audio section is provided by VIA 1 and VIA 3. The signals provided are Codec Clock and Volume Control. The encode/decode line, controlled by DTR from the SSDA, selects the desired audio function (input or output). Codec clock is a PB7 output (of VIA 3), a timer-generated signal which determines Codec sampling rate (normally about 16KHz). Volume control, a CB2 output (of VIA 1), is a timer-controlled recirculating shift register output and is an eight-step, pulse-width-modulated ultra-audio signal.

INPUT SIGNAL CONDITIONING

The microphone amplifier utilizes half of an LM358 and a JFET in a variable-gain amplifier used as a compressor. The attack time of the compressor is about 50 milliseconds; release time is 250 mS. Input signal amplitude range for acceptable record quality is about 5 to 75 mVRMS. The second stage, 1/2 of a LM358, is a 3-pole butterworth low-pass filter with a cutoff-frequency of about 3 KHz. This filter eliminates "aliasing" in the CVSD modulator.

OUTPUT CONDITIONING AND POWER AMPLIFIER

Following the CVSD, the output (playback) signal is low-pass filtered by another active, 3 KHz cutoff butterworth filter (1/4 LM324). Following this stage, a CA4066B and its attendant drivers provide software-controlled volume control by varying the duty-factor of signal CODEC VOL. The frequency of this signal (including the produced sidebands) must be high enough to be above audible range; a minimum of 20 KHz is recommended. Playback power amplification is provided by an LM383. This stage also provides some roll-off to alleviate the above problem. The power stage will produce 4 watts of audio; thus, an external speaker should be used if above-normal sound levels are programmed, since the internal speaker is rated at only 300 milliwatts.

SSDA DEVICE OPERATION

OVERVIEW

At the bus interface, the SSDA appears as two addressable memory locations. Internally, there are seven registers: two read-only and five write-only registers. The read-only registers are Status and Receive Data; the write-only registers are Control 1, Control 2, Control 3, Sync-Code and Transmit Data. The serial interface consists of serial input and output lines with independent clocks and four peripheral/modem control lines.

Data to be transmitted is transferred directly into the 3-byte Transmit Data First-In First-Out (FIFO) register from the data bus. Availability of the input to the FIFO is indicated by a bit in the Status register; once data is entered, it moves through the FIFO to the last empty location. Data at the output of the FIFO is automatically transferred from the FIFO to the Transmitter Shift register as the shift register becomes available to transmit the next character. If data is not available from the FIFO (underflow condition), the Transmitter Shift register is automatically loaded with either a sync code or an all 1's character. The transmit section should be programmed to not append parity onto the transmitted word.

For use in the S1 audio system, the SSDA should normally be programmed to use 8-bit, no parity, and External Sync mode. Then the DTR control selects the input or output function. However, for completeness and any special functions, all modes of SSDA operation are discussed in the following sections.

The method of serial data accumulating in the receiver depends on the synchronization mode selected. In External Sync mode, used for parallel-serial operation, the receiver is synchronized by the DCD (Data Carrier Detect) input and transfers successive bytes of data to the input of the Receiver FIFO. The Single-Sync-Character mode requires a match between the Sync-Code register and one incoming character before data transfer to the FIFO begins. In Two-Sync-Character mode, two sync codes must be received in sequence to establish synchronization. Subsequent to synchronization in any mode, data is accumulated in the shift register. Availability of a word at the FIFO output is indicated by a bit in the Status register.

The SSDA and its internal registers are selected by the address bus and the Read/Write (R/W) and Enable control lines. To configure the SSDA, Control registers are selected and the appropriate bits set. The Status register can be selected to read status.

The transmitter and receiver clock inputs are tied together. Signals to the microprocessor are the Data bus and Interrupt Request (IRQ).

INITIALIZATION

During a power-up sequence, system reset sets the SSDA in an internally-latched reset condition to prevent erroneous output transitions. The Sync-Code register, Control register 2, and Control register 3 should be loaded prior to the programmed release of the Transmitter and/or Receiver Reset bits. The bits in Control register 1 should be cleared after the Reset line has gone high.

TRANSMITTER OPERATION

Data is transferred to the transmitter section in parallel form via the data bus and the Transmit Data FIFO. The Transmit Data FIFO is a 3-byte register whose status is indicated by the Transmitter Data Register Available status bit (TDRA) and its associated interrupt enable bit. Data is transferred through the FIFO on negative edges of PHASE2 pulses. Two data transfer modes are provided in the SSDA: the 1-byte transfer mode provides for writing data to the transmitter section (and reading from the receiver section) one byte at a time; the 2-byte transfer mode provides for writing two data characters in succession.

Data automatically transfers from the last register location in the Transmit Data FIFO (when it contains data) to the Transmitter Shift register during the last half of the last bit of the previous character. A character is transferred into the Shift register by the Transmitter Clock. Data is transmitted LSB first.

When the Shift register becomes empty and data is not available for transfer from the Transmit Data FIFO, an underflow results, and a character is inserted into the transmitter data stream. This character will be either all 1's or the contents of the Sync-Code register, depending on the state of the Transmit Sync-Code-On-Underflow control bit.

Transmission is initiated by clearing the Transmitter Reset bit in Control register 1. When the Transmitter Reset bit is cleared, the first full positive half-cycle of the Transmit Clock initiates the transmit cycle; the transmission of data (or underflow characters) begins on the negative edge of the Transmit Clock pulse which started the cycle. If the Transmit Data FIFO has not been loaded, an underflow character is transmitted. When the Transmitter Reset bit (Tx Rs) is set, the Transmit Data FIFO is cleared and the TDRA status bit is cleared. After one PHASE2 clock has occurred, the Transmit Data FIFO becomes available for new data and TDRA is inhibited.

RECEIVER OPERATION

Data and a pre-synchronized clock are provided to the SSDA receiver section by means of the Receive Data (Rx Data) and Receive Clock (Rx Clk) inputs. The data is a continuous bit stream; character boundaries cannot be identified within the stream. The Receiver Shift register outputs are high when it is in the reset state.

SYNCHRONIZATION

The SSDA provides three operating modes related to character synchronization: One-Sync-Character mode, Two-Sync-Character mode, and External Sync mode. The External Sync mode requires synchronization and control of the receiving section through the Data Carrier Detect (DCD) input. The external synchronization source could consist of a direct control line from the transmitting end of the serial data link or from external logic designed to detect the start of a message block. The One-Sync-Character mode searches on a bit-by-bit basis until a match is achieved between the data in the Shift register and the Sync-Code register. A match indicates that character synchronization is complete and will be retained for the message block. In the Two-Sync-Character mode, the receiver searches for the first sync-code match on a bit-by-bit basis and then looks for a second successive sync-code character prior to establishing character synchronization. If the second sync-code character is not received, the bit-by-bit search for the first sync-code resumes.

Sync-codes received prior to the completion of synchronization (one or two character) are not transferred to the Receive Data FIFO. Redundant sync-codes received during the preamble or sync-codes which occur as fill characters can automatically be stripped from the data by setting the Strip-Sync control bit to minimize system loading. Character synchronization is retained until cleared by means of the Clear-Sync bit. This bit also inhibits the synchronization search routine.

RECEIVING DATA

Once synchronization has been achieved, subsequent characters are automatically transferred into the Receive Data FIFO and clocked through the FIFO to the last empty location by PHASE2 pulses. The Receiver Data Available status-bit (RDA) indicates when data is available to be read from the last FIFO location (number 3) when in the 1-byte transfer mode. The 2-byte transfer mode causes the RDA status bit to indicate that data is available when the last two FIFO register locations are full. Available data in the Receive Data FIFO triggers an interrupt request if the Receiver Interrupt Enable bit (RIE) is set. The CPU should then read the SSDA Status register, which indicates whether data is available for the CPU to read from the Receive Data FIFO register. The IRQ and RDA status bits are reset by a read from the FIFO.

If more than one character has been received and is resident in the Receive Data FIFO, subsequent PHASE2 clocks cause the FIFO to update and the RDA and IRQ status-bits to again be set. The read-data operation for the 2-byte transfer mode requires a PHASE2 clock intervening between reads to allow the FIFO data to shift.

The other status bit which pertains to the receiver section is Receiver Overrun. The Overrun status bit is automatically set when a character is transferred to the Receive Data FIFO while the first register of the Receive Data FIFO is full. Overrun causes an interrupt if Error Interrupt Enable (EIE) has been set. The transfer of the overrunning character into the FIFO causes the previous character in the FIFO input register location to be lost. The Overrun status bit is cleared by reading the Status register (when the overrun condition is present) followed by a Receive Data FIFO register read. Overrun cannot occur and be cleared without providing an opportunity to detect its occurrence via the Status register.

INPUT/OUTPUT FUNCTIONS

SSDA INTERFACE SIGNALS FOR CPU

The SSDA interfaces to the CPU with an 8-bit bidirectional data bus (ID0-ID7), a chip-select line, a register-select line, an interrupt-request line, a read/write line, an enable line, and a reset line. These signals permit the CPU to have complete control over the SSDA.

SSDA Bidirectional Data (ID0-ID7)

The bidirectional data lines (D0-D7) allow for data transfer between the SSDA and the CPU. The data bus output drivers are three-state devices that remain in the high-impedance (off) state except when the CPU performs an SSDA read operation.

SSDA Enable (PHASE2)

The Enable signal, PHASE2, is a high impedance TTL-compatible input that enables the bus input/output data buffers, clocks data to and from the SSDA, and moves data through the FIFO Registers. This signal is the continuous System PHASE2 1 Mhz clock.

Read/Write (R/W)

The Read/Write line is a high-impedance input that is TTL-compatible and is used to control the direction of data flow through the SSDA's input/output data bus interface. When Read/Write is high (CPU read cycle), SSDA output drivers are turned on if the chip is selected and a selected register is read. When it is low, the SSDA output drivers are turned off and the CPU writes into a selected register. The Read/Write signal is also used to select read-only or write-only registers within the SSDA.

Chip Select (CS)

The Chip Select line is a high impedance TTL-compatible input line used to address the SSDA. The SSDA is selected when CS is low. Transfers of data to and from the SSDA are performed under the control of the Enable signal, Read/Write, and Register Select.

Register Select (RS)

The Register Select line is a high impedance input that is TTL-compatible. A high level is used to select Control registers C2 and C3, the Sync Code register, and the Transmit/Receive Data registers. A low level selects the Control 1 and Status registers (see Table 1). This line is driven by the A0 bit of the system address bus.

Interrupt Request (IRQ)

Interrupt Request is a TTL-compatible, open-drain (no internal pullup), active-low output that is used to interrupt the CPU. The Interrupt Request remains low until cleared by the CPU.

Reset Input

The Reset input provides a means of resetting the SSDA from an external source. In the low state, the Reset input causes the following:

- The Receiver Reset (Rx Rs) and Transmitter Reset (Tx Rs) bits are set, causing both the receiver and transmitter sections to be held in a reset condition.
- Peripheral Control bits PC1 and PC2 are reset to zero, causing the SM/DTR output to be high.
- The Error Interrupt Enable (EIE) bit is reset.
- An internal synchronization mode is selected.
- The Transmitter Data Register Available (TDRA) status bit is cleared and inhibited.

When Reset returns high (the inactive state), the transmitter and receiver sections remain in the reset state until the Receiver Reset and Transmitter Reset bits are cleared via the bus under software control. The Control Register bits affected by Reset (Rx Rs, Tx Rs, PC1, PC2, EIE, and E/I Sync) cannot be changed when Reset is low.

CLOCK INPUTS

Separate high impedance TTL-compatible inputs are driven by a common source for clocking transmitted and received data. The source is the CB2 signal from the Control Port VIA.

Transmit Clock (Tx Clk)

The Transmit clock input is used to clock out of transmitted data. The transmitter shifts data on the negative transition of the clock.

Receive Clock (Rx Clk)

The Receive clock input is used to clock in received data. The clock and data must be synchronized externally. The receiver samples the data on the positive transition of the clock.

SERIAL INPUT/OUTPUT LINES

Receive Data (Rx Data)

The Receive Data line is a high impedance TTL-compatible input through which data is received in a serial format. Data rates may be from 0 to 600 kbs.

Transmit Data (Tx Data)

The Transmit Data output line transfers serial data to a modem or other peripheral. Data rates may be from 0 to 600 kbs.

SSDA REGISTERS

Seven registers in the SSDA can be accessed by means of the bus. The registers are defined as read-only or write-only according to the direction of information flow. The Register Select input (RS) selects two registers in each state, one being read-only and the other write-only. The Read/Write input (R/W) defines which pair is actually accessed. Four registers (two read-only and two write-only) can be addressed via the bus at any particular time. These registers and the required adressing are defined in Table E-1.

Table E-1: SSDA Programming Model

	INPUTS		CONTROL		REGISTER CONTENT							
REGISTER	RS	R/W	AC2	AC1	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
Status (S)	0	1	X	X	Interrupt Request (IRQ)	Receiver Parity Error	Receiver Overrun (RX Ovrn)	Transmitter Underflow (TUF)	Clear-to-Send ($\overline{\text{CTS}}$)	Data Carrier Detect ($\overline{\text{DCD}}$)	Transmitter Data Register Available (TDRA)	Receiver Data Available (RDA)
Control (C1)	0	0	X	X	Address Control 2 (AC 2)	Address Control 1 (AC 1)	Receiver Interrupt Enable (RIE)	Transmitter Interrupt (TIE)	Clear Sync	Strip Sync Characters (Strip Sync)	Transmitter Reset (Tx Rs)	Receiver Reset (Rx Rs)
Receive Data FIFO	1	1	X	X	D7	D6	D5	D4	D3	D2	D1	D0
Control 2 (C2)	1	0	0	0	Error Interrupt Enable (EIE)	Transmit Sync Code on Underflow (TX Sync)	Word Length Select 3 (WS 3)	Word Length Select 2 (WS 2)	Word Length Select 1 (WS 1)	1-Byte/2-Byte Transfer (1-Byte/ 2-Byte)	Peripheral Control 2 (PC 2)	Peripheral Control 1 (PC 1)
Control 3	1	0	0	1	Not Used	Not Used	Not Used	Not Used	Clear Transmitter Underflow Status (CTUF)	Clear $\overline{\text{CTS}}$ Status (Clear $\overline{\text{CTS}}$)	One-Sync-Character/ Two-Sync-Character Mode Control (1 Sync/ 2 Sync)	External/ Internal Sync Mode Control (E/I Sync)
Sync Code	1	0	1	0	D7	D6	D5	D4	D3	D2	D1	D0
Transmit	1	0	1	1	D7	D6	D5	D4	D3	D2	D1	D0

X = Don't care.

CONTROL REGISTER 1 (C1)

Control register 1 is an 8-bit write-only register that can be directly addressed from the data bus. Control register 1 is addressed when RS equals zero.

Receiver Reset (Rx Rs), C1 Bit 0

The Receiver Reset control bit provides both a reset and inhibit function to the receiver section. When Rx Rs is set, it clears the receiver control logic, sync logic, error logic, Rx Data FIFO Control, Parity Error status bit, and DCD interrupt. The Receiver Shift register is set to "ones." The Rx Rs bit must be cleared after the occurrence of a low level on Reset in order to enable the receiver section of the SSDA.

Transmitter Reset (Tx Rs), C1 Bit 1

The Transmitter Reset control bit provides both a reset and inhibit to the transmitter section. When Tx Rs is set, it clears the transmitter control section, Transmitter Shift register, Tx Data FIFO Control (the Tx Data FIFO can be reloaded after one PHASE2 clock pulse), the Transmitter Underflow status bit, and the CTS interrupt, and inhibits the TDRA status bit (in the one-sync-character and two-sync-character models). The Tx Rs bit must be cleared after the occurrence of a low level on Reset in order to enable the transmitter section of the SSDA. If the Tx FIFO is not preloaded, it must be loaded immediately after the Tx Rs release to prevent a transmitter underflow condition.

Strip Synchronization Characters (Strip-Sync), C1 Bit 2

If the Strip-Sync bit is set, the SSDA automatically strips all received characters which match the contents of the Sync-Code register. The characters used for synchronization (one or two characters of sync) are always stripped from the received data stream.

Clear Synchronization (Clear-Sync), C1 Bit 3

The Clear-Sync control bit provides the capability of dropping receiver character synchronization and inhibiting resynchronization. The Clear-Sync bit is set to clear and inhibit receiver synchronization in all modes and is reset to zero to enable resynchronization.

Transmitter Interrupt Enable (TIE), C1 Bit 4

TIE enables both the Interrupt Request output (IRQ) and Interrupt Request status bit to indicate a transmitter service request. When TIE is set and the TDRA status bit is high, the IRQ output goes low (the active state), and the IRQ status bit goes high.

Receiver Interrupt Enable (RIE), C1 Bit 5

RIE enables both the Interrupt Request output (IRQ) and the Interrupt Request status bit to indicate a receiver service request. When RIE is set and the RDA status bit is high, the IRQ output goes low (the active state), and the IRQ status bit goes high.

Address Control 1 (AC1) and Address Control 2 (AC2), C1 Bits 6 and 7

AC1 and AC2 select one of the write-only registers (Control 2, Control 3, Sync-Code, or Tx Data FIFO), as shown in Table G-1, when RS equals one and R/W equals zero.

CONTROL REGISTER 2 (C2)

Control register 2 is an 8-bit write-only register which can be programmed from the bus when the Address Control bits in Control register 1 (AC1 and AC2) are reset and RS equals one and R/W equals zero.

Peripheral Control 1 (PC1) and Peripheral Control 2 (PC2), C2 Bits 0 and 1

The Peripheral Control 1 bit (PC1) and the Peripheral Control 2 bit (PC2) control the direction of data transfer and the selected CODEC function (Encode for receive; Decode for transmit). Control is accomplished by setting PC2 and setting PC1 to 00 for enabling the input (receive) function or to a 01 to enable the output (transmit) function. The DTR output is connected directly to the CTS input of the SSDA. Its complement is connected to the DCD input of the SSDA, as well as to the Encode/Decode select (pin 10) of the CODEC.

1-Byte/2-Byte Transfer (1-Byte/2-Byte), C2 Bit 2

When 1-Byte/2-Byte is set, the TDRA and RDA status bits indicate the availability of their respective data FIFO registers for a single byte data transfer. If 1 Byte/2 Byte is reset, the TDRA and RDA status bits indicate when two bytes of data can be moved without a second status-read. An Enable pulse must occur between data transfers.

Word Length Selects (WS1, WS2, WS3), C2 Bits 3, 4, and 5

Word Length Select bits WS1, WS2, and WS3 select the word length (including parity) for the 7, 8, and 9 bits, as shown in Table G-1.

Transmit Sync-Code on Underflow (Tx Sync), C2 Bit 6

When Tx Sync is set, the transmitter automatically sends a sync-character when data is not available for transmission. If Tx Sync is reset, the transmitter transmits a Mark character (including the parity bit position) on underflow. If the Tx Sync bit is set when the underflow is detected, a pulse approximately the width of a Tx Clk high-period occurs on the underflow output. Internal parity generation is inhibited during underflow except for sync-code fill-character transmission in 8-bit-plus parity word lengths.

Error Interrupt Enable (EIE), C2 Bit 7

When EIE is set, the IRQ status bit goes high and the IRQ output goes low if —

- A receiver overrun occurs. The interrupt is cleared by reading the Status Register and reading the Rx Data FIFO.
- The transmitter has underflowed (in the Tx Sync On Underflow Mode). The interrupt is cleared by writing a "1" into the Clear Underflow, C3 bit 3, or Tx Reset.

When EIE is a 0, the IRQ status bit and the IRQ output are disabled for the preceding error conditions. A low level on the Reset input resets EIE to "0."

CONTROL REGISTER 3 (C3)

Control register 3 is a 4-bit write-only register that can be programmed from the bus when RS equals one and R/W equals zero and when Address Control bits AC1 equals one and AC2 equals zero.

External/Internal Sync Mode Control (E/1 Sync), C3 Bit 0

When the E/1 Sync Mode bit is high, the SSDA is in External Sync mode, and the receiver synchronization logic is disabled. Synchronization can be achieved by means of the DCD input. The DCD input is controlled directly by the DTR output, whose operation is described earlier in "Control Register 2, bits PCO and PC1." Both the transmitter and receiver sections operate as parallel-to-serial converters in External Sync mode. The Clear-Sync bit in Control register 1 acts as a receiver sync inhibit when high to provide a bus-controllable inhibit. The Sync-Code Register can serve as a transmitter fill-character register and a receiver match register in this mode. A low on the Reset input resets the E/1 Sync Mode bit, placing the SSDA in Internal Sync mode.

One-Sync-Character/Two-Sync-Character Mode Control (1 Sync/2 Sync), C3 Bit 1

When the 1 Sync/2 Sync bit is set, the SSDA synchronizes on a single match between the received data and the contents of the Sync-Code register. When the 1 Sync/2 Sync bit is reset, two successive sync characters must be received prior to receiver synchronization. If the second sync character is not detected, the bit-by-bit search resumes from the first bit in the second character. Refer to the section of the Sync Code register for more detailed description.

Clear CTS Status (Clear CTS), C3 Bit 2

When a "1" is written into the Clear CTS bit, the stored status and interrupt are cleared. Subsequently, the CTS status bit reflects the state of the CTS input. The Clear CTS control bit does not affect the CTS input or its inhibit of the transmitter section. The Clear CTS command bit is self-clearing, so writing a "0" into this bit accomplishes nothing.

Clear Transmit Underflow Status (CTUF), C3 Bit 3

When a "1" is written into the CTUF status bit, the CTUF bit and its associated interrupt are reset. The CTUF command bit is self-clearing.

SYNC-CODE REGISTER

The Sync-Code register is an 8-bit register for storing the programmable sync code required for received data character synchronization in the One-Sync-Character and Two-Sync-Character modes. The Sync-Code register also provides for stripping the sync/fill characters from the received data (a programmable option) and for automatic insertion of fill characters in the transmitted data stream. The Sync-Code register is not used for receiver character synchronization in the External Sync mode; instead, it provides storage of receiver match and transmit fill characters.

The Sync-Code register can be loaded when AC2 and AC1 are a "1" and a "0", respectively, and if R/W equals zero and RS equals one.

The Sync-Code Register may be changed after the detection of a match with the received data (the first sync-code having been detected) to synchronize with a double-word sync pattern. (This sync-code change must occur prior to the completion of the second character.) The sync-match (SM) output can be used to interrupt the CPU system to indicate that the first eight bits have matched. The service routine would then change the Sync Code register to the second half of the pattern. Alternately, One Sync-Character mode can be used for sync-codes of more than 8 bits by using software to check the second and subsequent bytes after reading them from the FIFO.

PARITY FOR SYNC CHARACTER

The Transmitter does not generate parity for the sync character except in 9-bit mode:

Transmitter

9-bit (8-bit + parity) generates an 8-bit sync character + parity

8-bit (7-bit + parity) generates an 8-bit sync character (no parity)

7-bit (6-bit + parity) generates a 7-bit sync character (no parity)

Receiver

DURING SYNCHRONIZATION The Receiver automatically strips the sync character(s) (there are two sync characters if 2-sync mode is selected) used to establish synchronization. Parity is not checked for these sync characters.

AFTER SYNCHRONIZATION IS ESTABLISHED When the "strip-sync" bit is selected, the sync characters (fill characters) are stripped, and parity is not checked for the stripped sync (fill) characters. When the strip-sync bit is not selected (low), the sync character is assumed to be normal data and is transferred into FIFO after parity checking (if a parity format is selected).

Table E-2: Strip Sync Control Bit

STRIP SYNC (C1 BIT 2)	WSO-WS2 (DATA FORMAT; C2 BIT 3-5)	OPERATION
1	X	No transfer of sync-code. No parity check of sync-code.
0	With parity	*Transfer data and sync-codes. Parity check.
0	Without parity	*Transfer data and sync-codes. No parity check.

*Subsequent to synchronization.

Care should be exercised in selecting the sync character in the following situations:

- ▶ When Data format is (6 + parity) or (7 + parity)
- ▶ When Strip sync is not selected (low)
- ▶ When sync code is used as a fill character, and synchronization is established

The transmitter sends a sync character with parity, but the receiver checks the parity as if it were normal data. Therefore, the sync character should be chosen to match the parity check selected for the receiver in the special cases described in Table E-2.

RECEIVE DATA FIRST-IN FIRST-OUT REGISTER (Rx Data FIFO)

The Receive Data FIFO register consists of three 8-bit registers and is used for buffer storage of received data. Each 8-bit register has an internal status bit that monitors its full or empty condition. Data is always transferred from a full register to an adjacent empty register. The transfer from register to register occurs on PHASE2 pulses. The RDA status bit is high when data is available in the last location of the Rx Data FIFO.

In an Overrun condition, the overrunning character is transferred into the full first stage of the FIFO register and causes the loss of that data character. Successive overruns continue to overwrite the first register of the FIFO. This destruction of data is indicated by the Overrun status bit. The Overrun bit is set when the overrun occurs and remains set until the Status Register is read and a read of the Rx Data FIFO occurs.

Unused data bits for short word lengths (including the parity bit) appear as zeros on the data bus when the Rx Data FIFO is read.

TRANSMIT DATA FIRST-IN FIRST-OUT REGISTER (TX DATA FIFO)

The Transmit Data FIFO register consists of three Shift registers used for buffer storage of data to be transmitted. Each 8-bit register has an internal status bit which monitors its full or empty condition. Data is always transferred from a full register to an adjacent empty register. The transfer is clocked by pulses. The TDRA status bit is high if the Tx Data FIFO is available for data.

Unused data bits for short word lengths are handled as "don't cares." The parity bit is not transferred over the data bus since the SSDA generates parity at transmission.

When an Underflow occurs, the Underflow character is either the contents of the sync-code register or an all-ones character. The Underflow is stored in the Status register until cleared and appears on the Underflow output as a pulse approximately the width of a Tx Clk high period.

STATUS REGISTER

The Status register is an 8-bit read-only register. It provides the real-time status of the SSDA and the associated serial data channel. Reading the Status register is nondestructive. The method of clearing status bits depends upon the function each bit represents and is treated separately for each bit in the register, as described in the following sections.

Receiver Data Available (RDA), S Bit 0

The Receiver Data Available status bit indicates when receiver data can be read from the Rx Data FIFO. The presence of Receiver data is in the last register (#3) of the FIFO causes RDA bit to be high for the 1-byte transfer mode. In the 2-byte transfer mode, a high RDA bit indicates that the last two registers (#2 and #3) are full. The second character can be read without a second status read (to determine whether the character is available). Status must be read on a byte-by-byte basis if receiver data error checking is desired. The RDA status bit is reset automatically when data is not available.

Transmitter Data Register Available (TDRA), S Bit 1

The TDRA status bit indicates that data can be loaded into the Tx Data FIFO register. An empty first register (#1) of the Tx Data FIFO is indicated by a high-level TDRA status bit in the 1-byte transfer mode. The first two registers (#1 and #2) must be empty for TDRA to be high when in the 2-byte transfer mode. The Tx Data FIFO can be loaded with two bytes without an intervening status read. TDRA is inhibited by the Tx reset or reset. Upon Tx Reset, the Tx Data FIFO is cleared and then released on the PHASE2 clock pulse. The Tx Data FIFO can then be loaded with up to three data characters, even though TDRA is inhibited. This feature allows preloading data prior to the release of Tx Reset. A high-level CTS input inhibits the TDRA status bit in either sync mode (One-Sync-Character mode or Two-Sync-Character mode). CTS does not affect TDRA in External Sync mode. Thus the SSDA is allowed to operate under the control of the

CTS input with TDRA indicating the status of the Tx Data FIFO. The CTS input does not clear the Tx Data FIFO in any operating mode.

Data Carrier Detect (DCD), S Bit 2

A positive transition on the DCD input is stored in the SSDA until cleared by reading both Status and Rx Data FIFO. A "1" written into Rx Rs also clears the stored DCD status. The DCD status bit, when true, indicates that the DCD input has gone high. The reading of both Status and Receive Data FIFO allows Bit 2 of subsequent Status reads to indicate the state of the DCD input until the next positive transition.

Clear-to-Send (CTS), S Bit 3

A positive transition on the CTS input is stored in the SSDA until cleared by writing a "1" into the Clear CTS control bit or the Tx Rs bit. The CTS status bit, when true, indicates that the CTS input has gone high. The Clear CTS command (a "1" into C3 Bit 2) allows Bit 3 of subsequent Status reads to indicate the state of the CTS input until the next positive transition.

Transmitter Underflow (TUF), S Bit 4

When data is not available for the transmitter, an underflow occurs and is so indicated in the Status register (in the Tx Sync on underflow mode). The underflow status bit is cleared by writing a "1" into the Clear Underflow (CTUF) control bit or the Tx Rs bit. TUF indicates that a sync character will be transmitted as the next character. A TUF is indicated on the output only when the contents of the Sync-Code Register is to be transferred (transmit sync code on underflow equals one).

Receiver Overrun (Rx Ovrn), S Bit 5

Overrun indicates that data has been received when the Rx Data FIFO is full, resulting in data loss. The Rx Ovrn status bit is set when Overrun occurs. The Tx Ovrn status bit is cleared by reading Status followed by reading the Rx Data FIFO or by setting the Rx Rs control bit.

Receiver Parity Error (PE), S Bit 6

The Parity Error status bit indicates that parity for the character in the last register of the Rx Data FIFO did not agree with selected parity. The parity error is cleared when the character to which it pertains is read from the Rx Data FIFO or when Rx Rs occurs. The DCD input does not clear the Parity Error or Rx Data FIFO status bits.

Interrupt Request (IRQ), S Bit 7

The Interrupt Request status bit indicates when the IRQ output is in the active state (IRQ output equals zero). The IRQ status bit is subject to the same interrupt enables (RIE, TIE, and EIE) as the IRQ output. The IRQ status bit simplifies status inquiries for polling systems by providing a single-bit indication of service requests.

STATUS REGISTER

IRQ Bit 7

The IRQ flag is cleared when the source of the IRQ is cleared. The source is determined by the enables in the Control registers. TIE, RIE, EIE.

Bits 6 to 0

Indicate the SSDA status at a point in time, and can be reset as follows:

PE Bit 6 Read Rx Data FIFO, or a "1" into Rx Rs (C1 Bit 0).

Rx Ovrn Bit 5 Read Status and then Rx Data FIFO or a "1" into Rx Rs (C1 Bit 0).

TUF Bit 4 A "1" into CTUF (C3 Bit-3) or into Tx Rs (C1 Bit 1).

CTS Bit 3 A "1" into Clear CTS (C3 Bit 2) or a "1" into Tx Rs (C1 Bit 1).

DCD Bit 2 Read Status and then Rx Data FIFO or a "1" into Rx Rs (C1 Bit 0).

TDRA Bit 1 Write into Tx Data FIFO.

RDA Bit 0 Read Rx Data in FIFO.

CONTROL REGISTER 1

AC2, AC1 Bits 7, 6 Used to access other registers, as shown above.

RIE Bit 5 When "1", enables interrupt on RDA (S Bit 0).

TIE Bit 4 When "1", enables interrupt on TDRA (S Bit 1).

Clear Sync Bit 3 When "1", clears receiver character synchronization.

Strip Sync Bit 2 When "1", strips all sync codes from the received data stream.

Tx Rs Bit 1 When "1", resets and inhibits the transmitter section.

Rx Rs Bit 0 When "1", resets and inhibits the receiver section.

CONTROL REGISTER 2

CTUF Bit 3 When "1", clears TUF (S Bit 4), and IRQ if enabled.

Clear CTS Bit 2 When "1", clears CTS (S Bit 3), and IRQ if enabled.

1 Sync/2 Sync Bit 1 When "1", selects the one-sync-character mode; when "0", selects the two-sync character mode.

E/1 Sync Bit 0 When "1", selects the external sync mode; when "0", selects the internal sync mode.

CONTROL REGISTER 2

EIE Bit 7 When "1", enables the PE, Rx Ovrn, TUF, CTS, and DCD interrupt flags (S Bits 6 through 2).

Tx Sync Bit 6 When "1", allows sync code contents to be transferred on underflow, and enables the TUF Status bit and output. When "0", an all mark character is transmitted on underflow.

WS3, 2, 1 Bits 5 to 3

Table E-3: Word Length Select

BIT 5 WS3	BIT 4 WS2	BIT 3 WS1	WORD LENGTH
0	0	0	6 bits + even parity
0	0	1	6 bits + odd parity
0	1	0	7 bits, no parity
*0	1	1	8 bits, no parity
1	0	0	7 bits + even parity
1	0	1	7 bits + odd parity
1	1	0	8 bits + even parity
1	1	1	8 bits + odd parity

*This is the mode which should always be used.

1-Byte/2-Byte, Bit 2

When "1", enables the TDRA and RDA bits to indicate when a 1-byte transfer can occur; when "0", the TDRA and RDA bits indicate when a 2-byte transfer can occur.

PC2, PC1, Bits 1 and 0

Table E-4: SM/DTR Output Control

BIT 1 PC2S	BIT 0 PC1	SM/DTR OUTPUT AT PIN 5
0	0	1 Select audio output
1	0	0 Select audio input

CODEC DEVICE OPERATION

The Continuously-Variable-Slope-Delta modulator (CVSD) is a simple alternative to more complex conventional conversion techniques in systems requiring digital communication of analog signals. The human voice is analog, but digital transmission of any signal over great distance is attractive. Signal/noise ratios do not vary with distance in digital transmission, and multiplexing, switching, and repeating hardware is more economical and easier to design. However, instrumentation Analog-to-Digital converters do not meet the communications requirements. The CVSD Analog-to-Digital is well suited to the requirements of digital communications and is an economically efficient means of digitizing voice inputs for transmission.

THE DELTA MODULATOR

The innermost control loop of a CVSD converter is a simple delta modulator. A delta modulator consists of a comparator in the forward path and an integrator in the feedback path of a simple control loop. The inputs to the comparator are the analog input signal and the integrator output. The comparator output reflects the sign of the difference between the input voltage and the integrator output. That sign bit is the digital output and also controls the direction of ramp in the integrator. The comparator is normally clocked, producing synchronous and band-limited digital bit-stream.

If the clocked serial bit-stream is transmitted, received, and delivered to a similar integrator at a remote point, the remote integrator output is a copy of the transmitting control loop integrator output. To the extent that the transmitting integrator tracks the input signal, the remote receiver reproduces that input signal. Low-pass filtering at the receiver output eliminates most of the quantizing noise if the clock rate of the bit stream is an octave or more above the upper band limit of the input signal. Input bandwidth cuts off above 3 kHz, so clock rates from 8 kHz up are possible. Thus, the delta modulator digitizes and transmits the analog input to a remote receiver. The serial, unframed nature of the data is ideal for communications networks. With no input at the transmitter, a continuous one/zero alternation is transmitted. If the two integrators are made leaky, then, during any loss of contact, the receiver output decays to zero and receive restart begins without framing when the receiver re-acquires. Similarly, a delta modulator is tolerant of sporadic bit errors.

THE COMPANDING ALGORITHM

The fundamental advantages of the delta modulator are its simplicity and the serial format of its output. Its limitations are those caused by a limited digital bit rate. The analog input must be band-limited and amplitude-limited. The frequency limitations are governed by the Nyquist information rate relationships, and the amplitude capabilities are set by the gain and dynamic range of the integrators.

The frequency limits are bounded on the upper end; that is, for any input bandwidth there exists a clock frequency larger than that bandwidth transmits the signal with a specific noise level. However, the amplitude limits are bounded on both upper and lower ends. For any given signal level, one specific gain achieves an optimum noise level. Unfortunately, the basic delta modulator has a small dynamic range over which the noise level is constant.

The continuously-variable-slope circuitry provides increased dynamic range by adjusting the gain of the integrator. For a given clock frequency and input bandwidth, the additional circuitry increases the delta modulator's dynamic range. External to the basic delta modulator is an algorithm which monitors the past few outputs of the delta modulator in a simple shift register. The register is 2 bits long. The accepted CVSD algorithm simply monitors the contents of the shift register and indicates if it contains all ones or zeros. This condition is called coincidence. When it occurs, it indicates that the gain of the integrator is too small. The coincidence output charges a single pole low-pass filter. The voltage output of this "syllabic filter" controls the integrator gain through a pulse amplitude modulator whose other input is the sign bit or up/down control.

The simplicity of the all-ones/all-zeros algorithm should not be taken lightly. Many other control algorithms using shift registers have been tried. The key to the accepted algorithm is that it provides a measure of the average power or level of the input signal. Other techniques provide more instantaneous information about the shape of the input curve. The purpose of the algorithm is to control the gain of the integrator and to increase the dynamic range. Thus, a measure of the average input level is what is needed.

The algorithm is repeated in the receiver, and thus the level data is recovered in the receiver. Because the algorithm only operates on the past serial data, it changes the nature of the bit stream without changing the channel bit rate.

The effect of the algorithm is to compand the input signal. If the bit stream from a CVSD encoder is played into a basic delta modulator, the output of the delta modulator reflects the shape of the input signal, but all of the output will be at an equal level. Thus, the algorithm is needed at the output to restore the level variations. The bit stream on the channel behaves as if it came from a standard delta modulator with a constant level input.

The delta modulator encoder with the CVSD algorithm provides an efficient method for digitizing voice signals in a manner which is especially convenient for digital communications requirements.

Table E-5: Definitions and Functions of Pins

PIN NUMBER	PIN FUNCTION
Pin 1	VDD (+5 volts)
Pin 2	Audio Ground. Connection to D/A ladders and comparator.
Pin 3	Audio Out. Recovered audio out. Presents approximately 100 kilo-ohm source. Zero signal reference is VDD/2.
Pin 4	AGC (not used). A logic "low" level appears at this output when the recovered signal excursion reaches one-half of full scale value.
Pin 5	Audio Input. Externally AC coupled.
Pin 6	N/C
Pin 7	N/C
Pin 8	Ground Logic Ground
Pin 9	Clock Input
Pin 10	Encode/Decode. A low level selects the encode mode; a high level, the decode mode.
Pin 11	Alternate Plain Text (not used). A low level at this input causes a quieting pattern to be transmitted without affecting the internal operation of the CVSD.
Pin 12	Digital Data Input
Pin 13	Force Zero (not used). A low level at this input forces the transmitted output, the internal logic, and the recovered audio output of the CVSD into the "quieting" condition.
Pin 14	Digital Data Output

MECHANICAL SPECIFICATIONS

KEY TOTAL TRAVEL	Range	.150 in-.200 in ±.010 (3.8 mm-5 mm)	
	Preferred	.170 in (4.3 mm)	
	Key Pretravel (when applicable)	.100 in minimum (2.5 mm)	
ACTUATION FORCE	Standard Key	Range	1.5-2.5 oz ±30% (42.5-70 grams)
		Preferred	1.5 oz ±30% (42.5 grams)
	Special Key	Range	3-5 oz (85-142 grams)
		Preferred	3 oz (85 grams)
RELIABILITY	>100 million cycles		
KEY SPACING	Range	.70-.80 in (18-20 mm)	
	Preferred	.75 in (19 mm)	
KEY SIDEPLAY	.018 in (.5 mm) 2° rotational		
KEY TOP DIMENSION	Range	.47-.60 in (12-15 mm)	
	Preferred	.51 in (13 mm)	
KEY SURFACE	Concave, textured (mat) unless position marked otherwise, low reflection, low glare.		
KEY SWITCH PRESSURES	Keytop shall be capable of withstanding 3 lbs (1.4 kg) pull without coming loose and 11 lbs (5 kg) in the direction of actuation without any damage to the key switch.		

ELECTRICAL SPECIFICATIONS

INPUT POWER	+5VDC ± 5% @ 250 ma
ROLLOVER	N Key
CONNECTOR	Type: AMP 87551-7 or equivalent Spacing: 0.1 in, 7 pin header

Table F-1: Pin Assignment

PIN(S)	NAME	FUNCTION
1, 7	+5V	+5 volts at 250 ma
2, 3	GROUND	System Ground
4	KBACK	TTL Input. Driven by terminal processor. Transitions indicate acknowledgement of KBRDY transitions.
5	$\overline{\text{KBRDY}}$	TTL Output. Driven low by the keyboard to initiate handshake of each data bit of a transmission. Driven high after receipt of the negative edge of the KBACK line.
6	KBDATA	TTL Output. Changed after the positive edge of the KBACK line. Data must change no later than the negative edge of KBRDY. The exception to this is the stop bit. Transfer of the stop bit is as follows: 1) Data line driven low at or before negative edge of KBRDY. 2) Data line and $\overline{\text{KRBDY}}$ driven high following the negative edge of $\overline{\text{KBACK}}$. 3) Keyboard enters the Idle state afterthe positive edge of $\overline{\text{KBACK}}$.

LOGICAL SPECIFICATIONS

PROTOCOL DEFINITION

The communication between the terminal processor and the keyboard is serial. The transmission is in 9-bit words. The first eight bits are the data byte, transmitted LSB first. The last bit is a stop bit.

The keyboard will return key numbers and key status through the eight data bits. The MSB of the key number returned by the keyboard is status which flags a key close or key open. An MSB of one indicates a key close, and an MSB of zero indicates a key open. The least significant 7 bits are the key number.

The stop bit is a zero from $\overline{\text{KBRDY}}$ low to $\overline{\text{KBACK}}$ low. The stop bit goes high before $\overline{\text{KBRDY}}$ goes high and remains high until the next transfer.

The keyboard indicates it has an event in its buffer with the $\overline{\text{KBRDY}}$ line. If transmission is idle, the keyboard can signal an event by taking the $\overline{\text{KBRDY}}$ line low. The high to low transition of $\overline{\text{KBRDY}}$ should flag an interrupt in the terminal processor. The keyboard should raise the $\overline{\text{KBRDY}}$ line on the negative transition of the $\overline{\text{KBACK}}$ line. Each event in the keyboard buffer will cause a transition of the $\overline{\text{KBRDY}}$ line. The keyboard transmission becomes idle after the positive edge of the $\overline{\text{KBACK}}$ line following the stop bit.

The keyboard times out the processor response to $\overline{\text{KBRDY}}$ low for 250 milliseconds. If the processor does not respond with a negative transition of KBACK clock within this time, the keyboard will drive $\overline{\text{KBRDY}}$ high and then restart the current transmission. This will allow the terminal processor to resynchronize to the keyboard data stream.

Table F-2: Switching Characteristics

PARAMETER	FUNCTION DESCRIPTION	REQUIRED TIMING	
		MAX	MIN
TDVRL	KB data valid to KBRDY low	—	0
TRLCL	KBRDY low to KBACK low	250ms	—
TAHKL	KBACK high to KBRDY low (except after stop bit)	1ms	0

RESERVED KEYBOARD CODES

HEX	FUNCTION	DESCRIPTION
FEH	Overflow	Key queue overflow. Keys have been lost.
FFH	Dead	Keyboard dead or not connected.

ENVIRONMENTAL SPECIFICATIONS

OPERATING TEMPERATURE 0° C-50° C

STORAGE TEMPERATURE -40° C-+60° C

HUMIDITY 0-95% noncondensing

MATERIAL Self-extinguishable

KEYBOARD APPROVALS Keyboard meets UL and VDE requirements for approval.

VIBRATION To be determined

SHOCK Operating: 10G peak 1/2 sinusoid: 10ms duration
Nonoperating: 100G peak 1/2 sinusoid: 10ms duration

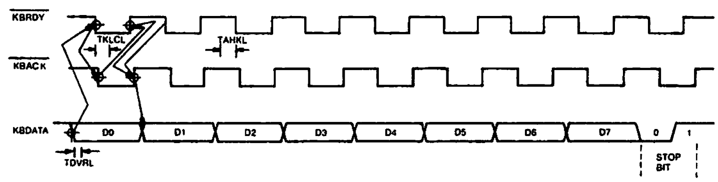
KEYBOARD LAYOUT

Key layouts vary from model to model in relation to the targeted application. The layout is broken into typewriter keys, command keys, and calculator keys. The typewriter pad has 58 possible key positions. The whole keyboard has a total of 104 possible key positions. The typewriter pad is sculptured; other pads are sloped. The layout uses one common PC Board, while the actual number of key positions occupied varies from model to model.

KEYBOARD TIMING DIAGRAM

Figure F-1 illustrates keyboard timing.

Figure F-1: Keyboard Timing Diagram



APPENDIX G COMMUNICATIONS CONTROLLER SPECIFICATIONS

G-1 INTRODUCTION

The NEC uPD7201 Multiprotocol Serial Communications Controller ($MPSC^2$) is a versatile device designed to give you high-level control of your data communication protocols with maximum flexibility and minimum processor overhead. The $MPSC^2$ contains two complete full duplex channels in a 40-pin package and incorporates a variety of sophisticated features to simplify your protocol management.

G-1.1 FEATURES

- Implements the three basic data/communications protocols
  - Asynchronous
  - Character-oriented synchronous (monosync, bisync, external sync)
  - Bit-oriented synchronous (SDLC/HDLC)
- Provides extensive error checking
  - Parity
  - CRC-16
  - CRC-CCITT
  - Break/Abort detection
  - Framing Error detection
- Enhanced data reliability
  - Double-buffered transmitters
  - Quadruply-buffered receivers
  - Programmable transmitter underrun handling
- Simplified system design
  - Simple interface to most microprocessors
  - Automatic Interrupt vectoring for most microprocessors
  - Four DMA channels for maximum throughput with standard 8237/8257-type DMA controllers
  - Single-phase TTL clock
  - Single +5 volt supply

G-2 PIN DESCRIPTION

This section describes the various pin functions available on the MPSC². Some pin numbers are used twice because of their programmability and dual functionality. Those pins that have more than one function are marked with an * in the following descriptions. Refer to Section G-5 for detailed information on selecting pin functions.

Figure G-2.1 Functional Pinout

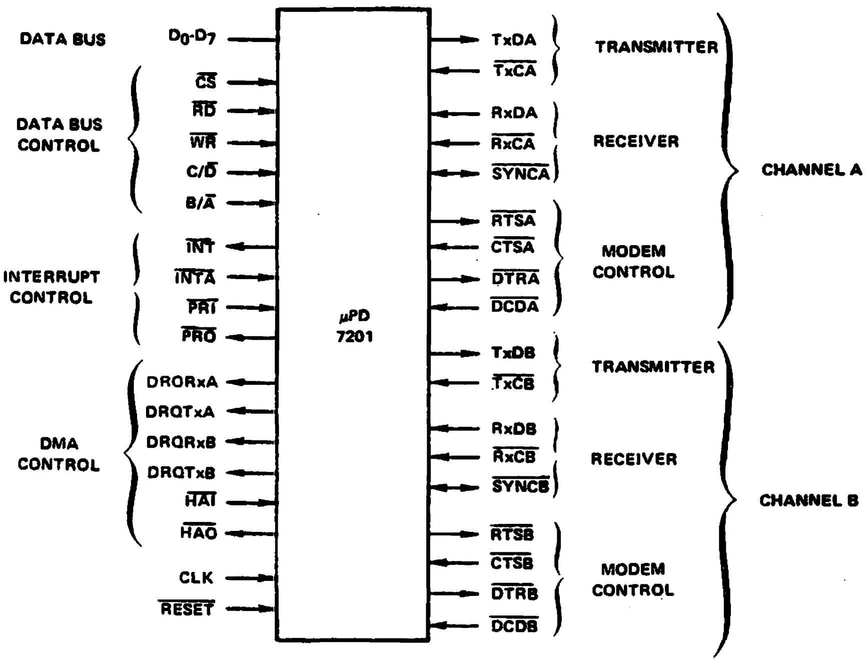
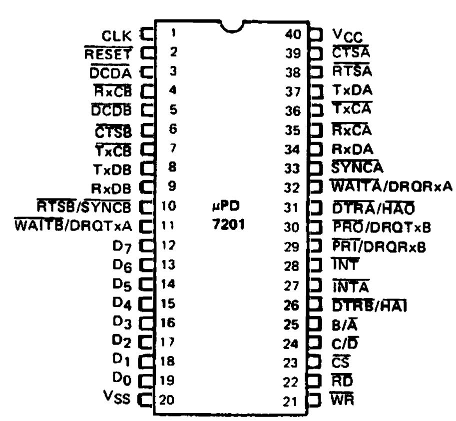


Figure G-2.2 Pin Configuration



12-19 D_0-D_7 Data Bus (bidirectional three-state)
The data bus lines are connected to the system data bus. Data or status from the MPSC² is output on these lines when CS and RD are active and data or commands are latched into the MPSC² on the rising edge of WR when CS is active.

23 CS Chip Select (input, active low)
Chip select allows the MPSC² to transfer data or commands during a read or write cycle.

25 B/A Channel Select (input)
A low selects channel A and a high selects channel B for access during a read or write cycle.

24 C/D Control/Data Select (input)
This input, with RD, WR and B/A, selects the data registers (C/D = 0) on the control and status registers (C/D = 1) for access over the data bus.

22 RD Read (input, active low)
This input (with either CS during a read cycle or HAI during a DMA cycle) notifies the MPSC² to read data or status from the device.

21 WR Write (input, active low)
This input (with either CS during a read cycle or HAI during a DMA cycle) notifies the MPSC² to write data or control information to the device.

2 RESET Reset (input, active low)
A low on this input (one complete CLK cycle minimum) initializes the MPSC² to the following conditions: receivers and transmitters disabled, TxDA and TxDB set to marking (high), and Modem Control Outputs DTRA, DTRB, RTSA, RTSB set high. Additionally, all interrupts are disabled, and all interrupt and DMA requests are cleared. After a reset, you must rewrite all control registers before restarting operation.

1 CLK System Clock (input)
A TTL-level system clock signal is applied to this input. The system clock frequency must be at least 4.5 times the data clock frequency applied to any of the data clock inputs TxCA, TxCB, RxCA or RxCB.

28 INT Interrupt Request (output, open drain, active low)
INT is pulled low when an internal interrupt request is accepted.

27 INTA Interrupt Acknowledge (input, active low)
The processor generates two or three INTA pulses (depending on the processor type) to signal all peripheral devices that an interrupt acknowledge sequence is taking place. During the interrupt acknowledge sequence, the MPSC², if so programmed, places information on the data bus to vector the processor to the appropriate interrupt service location.

29* PRI Interrupt Priority In (input, active low)
This input informs the MPSC² whether the highest priority device is requesting interrupt and is used with PRO to implement a priority resolution "daisy chain" when there is more than one interrupting

device. The state of PRI and the programmed interrupt mode determine the MPSC²'s response to an interrupt acknowledge sequence.

30* PRO Interrupt Priority Out (output, active low)
This output is active when HAI is active and the MPSC² is not requesting interrupt (INT is inactive). The active state informs the next lower priority device that there are no higher priority interrupt requests pending during an interrupt acknowledge sequence.

11*, 32* WAITA WAITB Wait (outputs, open drain)
These outputs synchronize the processor with the MPSC² when block transfer mode is used. You may program it to operate with either the receiver or transmitter, but not both simultaneously. WAIT is normally inactive. For example, if the processor tries to perform an inappropriate data transfer such as a write to the transmitter when the transmitter buffer is full, the WAIT output for that channel is active until the MPSC² is ready to accept the data. The CS, C/D, B/A, RD, and WR inputs must remain stable while WAIT is active.

11*, 29*, 30*, 32* DRQTxA, DRQTxB, DRQRxA, DRQRxB
DMA Request (outputs, active high)
When these lines are active, they indicate to a DMA controller that a transmitter or receiver is requesting a DMA data transfer.

26* HAI Hold Acknowledge In (input, active low)
This input notifies the MPSC² that the host processor has acknowledged the DMA request and has placed itself in the hold state. The MPSC² then performs a DMA cycle for the highest priority outstanding DMA request, if any.

31* HAO Hold Acknowledge Out (output, active low)
This output, with HAI, implements a priority daisy chain for multiple DMA devices. HAO is active when HAI is active and there are no DMA requests pending in the MPSC².

8, 37 TxDA, TxDB Transmit Data (outputs, marking high)
Serial data from the MPSC² is output on these pins.

7, 36 TxCA, TxCB Transmitter Clocks (inputs, active low)
The transmit clock controls the rate at which data is shifted out at TxD. You may program the MPSC² so that the clock rate is 1x, 16x, 32x, or 64x the data rate. Data changes on the falling edge of TxC. TxC features a Schmitt-trigger input for relaxed rise and fall time requirements.

9, 34 RxDA, RxDB Receiver Data (inputs, marking high)
Serial data to the MPSC² is input on these pins.

4, 35 RxCA, RxCB Receiver Clocks (inputs, active low)
The receiver clock controls the sampling and shifting of serial data at RxD. You may program the MPSC² so that the clock rate is 1x, 16x, 32x, or 64x the data rate. RxD is sampled on the rising edge of RxC. RxC features a Schmitt-trigger input for relaxed rise and fall time requirements.

26*, 31* DTRA, DTRB Data Terminal Ready (outputs, active low)
The DTR pins are general-purpose outputs which may be set or reset with commands to the MPSC².

10, 38* RTSA, RTSB Request to Send (outputs, active low)
When you operate the MPSC² in one of the synchronous modes, RTSA and RTSB are general-purpose outputs that you may set or reset with commands to the MPSC². In asynchronous mode, RTS is active immediately as soon as it is programmed on. However, when programmed off, RTS remains active until the transmitter is completely empty. This feature simplifies the programming required to perform modem control.

3, 5 DCDA, DCDB Data Carrier Detect (inputs, active low)
Data carrier detect generally indicates the presence of valid serial data at RxD. You may program the MPSC² so that the receiver is enabled only when DCD is low. You may also program the MPSC² so that any change in state that lasts longer than the minimum specified pulse width causes an interrupt and latches the DCD status bit to the new state.

6, 39 CTSA, CTSB Clear to Send (inputs, active low)
Clear to send generally indicates that the receiving modem or peripheral is ready to receive data from the MPSC². You may program the MPSC² so that the transmitter is enabled only when CTS is low. As with DCD, you may program the MPSC² to cause an interrupt and latch the new state when CTS changes state for longer than the minimum specified pulse width.

10, 33* SYNCA, SYNCB Synchronization (inputs/outputs, active low)
The function of the SYNC pin depends upon the MPSC² operating mode. In asynchronous mode, SYNC is an input that the processor can read. It can be programmed to generate an interrupt in the same manner as DCD and CTS.

In external sync mode, SYNC is an input which notifies the MPSC² that synchronization has been achieved (see Figure G-2.3 for detailed timing). Once synchronization is achieved, hold SYNC low until synchronization is lost or a new message is about to start.

In internal synchronization modes (monosync, bisync, SDLC), SYNC is an output which is active wherever a SYNC character match is made (see Figure G-2.4 for detailed timing). There is no qualifying logic associated with this function. Regardless of character boundaries, SYNC is active on any match.

Figure G-2.3 SYNC Output, External Synchronization

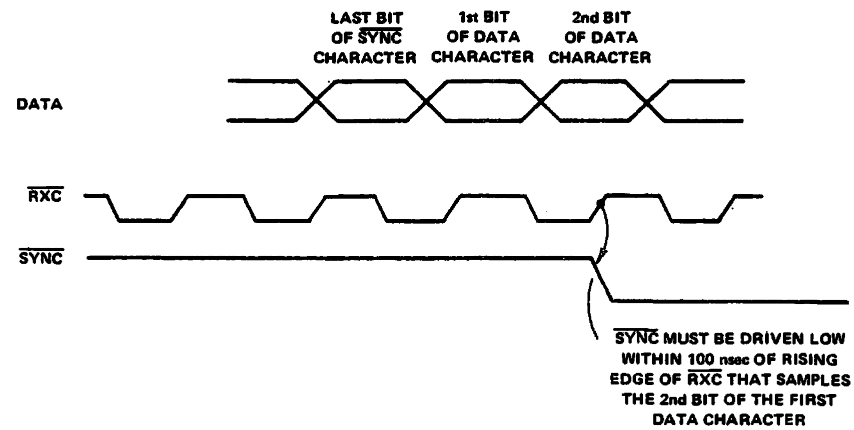
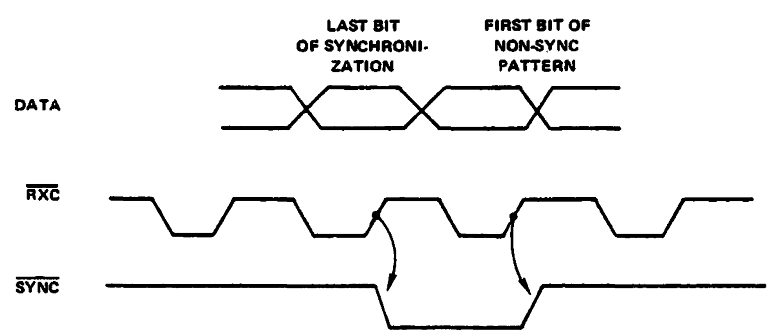


Figure G-2.4 SYNC Output, Internal Synchronization



G-3 PROTOCOLS

A protocol defines a set of rules for transmitting information and control from one place to another. In parallel protocols as you might find on a microprocessor bus, dedicated "control" lines handle functions such as timing, type of information, and error checking. Since the object of serial data communications is to minimize the number of wires, the protocol used must place all of this control information in the serial data stream.

The basic protocol unit or frame can be built into increasingly complex protocols by defining special control characters and fields, and by grouping frames together into larger units. Virtually all communications protocols currently in use are based on one of three basic protocols: Asynchronous, Synchronous Character- or Count-Oriented Protocols (COPs), and Bit-Oriented Protocols (BOPs).

G-3.1 ASYNCHRONOUS PROTOCOL

In asynchronous protocol, each character transmitted has its own framing information in the form of a start and stop bit(s). Each character is a "message" in itself and may be asynchronous with respect to any others. You can implement error detection by adding a parity bit to each character. The transmitter makes the parity bit 1 or 0 so that the character plus parity contains an even or odd number of ones for even parity or odd parity, respectively. Figure G-3.1 illustrates the asynchronous data format.

G-3.2 SYNCHRONOUS CHARACTER ORIENTED PROTOCOLS

In synchronous character-oriented protocols (COPs), the start and stop bits associated with each character are eliminated. A synchronization (sync) character that is not part of the data is transmitted before the data to establish proper framing. The synchronization character is usually 8 or 16 bits long. Monosync and IBM Bisync are typical examples of COPs (Figure G-3.2). Since the framing information is presented only at the beginning, the transmitter must insert fill characters to maintain synchronization. Sync characters are commonly used for this purpose.

As with the asynchronous protocol, a parity bit may be used with each character to provide error checking. A more reliable check is performed by calculating a special 16-bit block check character called a Cyclic Redundancy Check (CRC) for the entire data block and transmitting this character at the end of the data. The most commonly used CRC polynomial for COPs is called CRC-16.

A disadvantage of the character-oriented protocol is having to use special characters such as SYNC to define various portions of a message when you send non-character binary data ("transparent data" in bisync terminology). To do this, you must transmit special DLE sequences and selectively exclude certain characters from the CRC calculation for both the transmitter and receiver. The MPSC^2 features special circuitry to simplify this operation.

G-3.3 SYNCHRONOUS BIT-ORIENTED PROTOCOLS

Synchronous Bit-Oriented Protocols (BOPs) use a special set of rules to distinguish between data and framing characters. This eliminates some of the problems associated with COPs. The most common BOPs in use are the almost-identical HDLC and SDLC protocols shown in Figure G-3.3.

The rules for SDLC (henceforth we will refer only to SDLC although the same information applies to HDLC as well) are quite simple. The basic transmission unit is called a frame and is delineated by a special flag character 01111110 (flags cannot be used as filler like the COP sync character). The data or information field may consist of any number of bits; not necessarily an integral number of n-bit characters. Since data could contain the 01111110 pattern, the transmitter performs the following operation: if five consecutive ones are transmitted, the transmitter inserts a zero bit before the next data bit. Likewise, the receiver must delete any zero that follows five consecutive ones. Six consecutive ones indicate a flag character and eight or more ones indicate a special abort condition.

Error checking is done with a 16-bit CRC character inserted between the end of the information field and the End Of Frame flag. The CRC-CCITT polynomial is generally used. The end of a frame is determined by counting 16 bits (CRC) back from the End Of Frame flag. Special circuitry in the receiver must inform the processor of the boundary between the end of the information field and the beginning of the CRC when the information field is not an integral number of n-bit characters. The MPSC² performs all of the above functions necessary to implement Bit-Oriented Protocols.

Figure G-3.1 Asynchronous Data Character Format

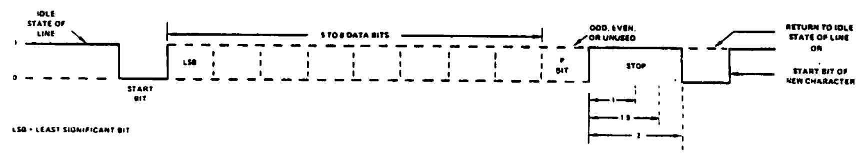


Figure G-3.2 BISYNC Message Format

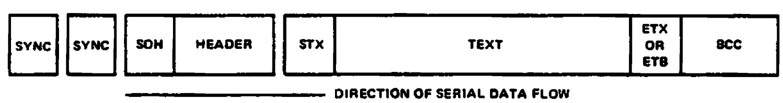
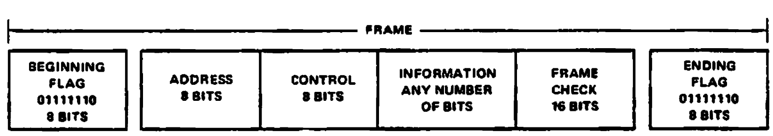


Figure G-3.4 Basic SDLC Frame



G-4 FUNCTIONAL DESCRIPTION

The MPSC² provides two complete serial communications controllers in a single package implementing the following functions:

Parallel-to-Serial and Serial-to-Parallel data conversion.

Buffering of outgoing and incoming data, allowing the processor time to respond.

Insertion and deletion of framing bits and characters.

Calculation and checking of Parity and CRC error checking.

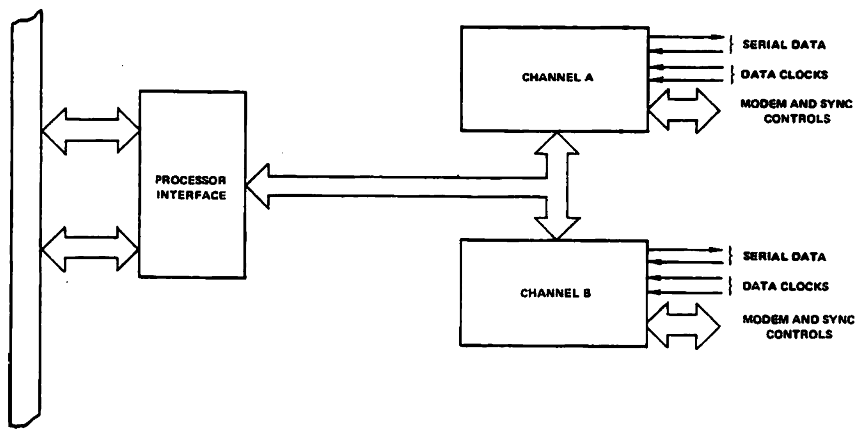
Informing the processor when and what action needs to be taken.

Interfacing with the outside world over discrete modem control lines.

The MPSC² can be logically divided into the following functional groups (Figure G-4.1):

Two identical serial I/O controller channels, each consisting of a Transmitter section and a Receiver section, and a common Processor Interface that connects the MPSC² with the host processor and provides overall device control.

Figure G-4.1 Block Diagram



G-4.1 TRANSMITTER

The MPSC² Transmitter performs all the functions necessary to convert parallel data to the appropriate serial bit streams required by various protocols. The major components of the transmitter are shown in Figure G-4.2. Control and status register fields pertinent to the operation of the transmitter are summarized in Table G-4.1.

The primary data flow through the transmitter begins at the internal data bus. There, characters written to the MPSC² are placed in the buffer register. When any character present in the shift register has been transferred out, or if the shift register is empty, the contents of the buffer register are transferred to the shift register and output with the least significant bit first. Then, a Transmitter Buffer Becoming

Empty indication (flag) is given. This double buffering allows the processor one full character time from this flag to respond with the next character without interrupting data transmission. You should note that it is the transfer of a character from the data buffer to the shift register rather than the empty condition itself that causes the Transmitter Buffer Becoming Empty indication. At initialization or after a Reset Transmitter Interrupt/DMA Pending Command is issued to control register 0 (CR0) you must write one character to the buffer to reset this flag. The Transmitter Buffer Empty bit in status register 0 (SR0), always reflects the presence or absence of a character in the buffer.

After a hardware or software reset, the transmitter data output (TxD) is in high (marking) state. You can pull TxD low (spacing) any time by setting the Send Break bit (CR5 bit 4). TxD remains low until the Send Break bit is reset and any data currently being transmitted is destroyed.

Figure G-4.2 Block Diagram MPSC² Transmitter

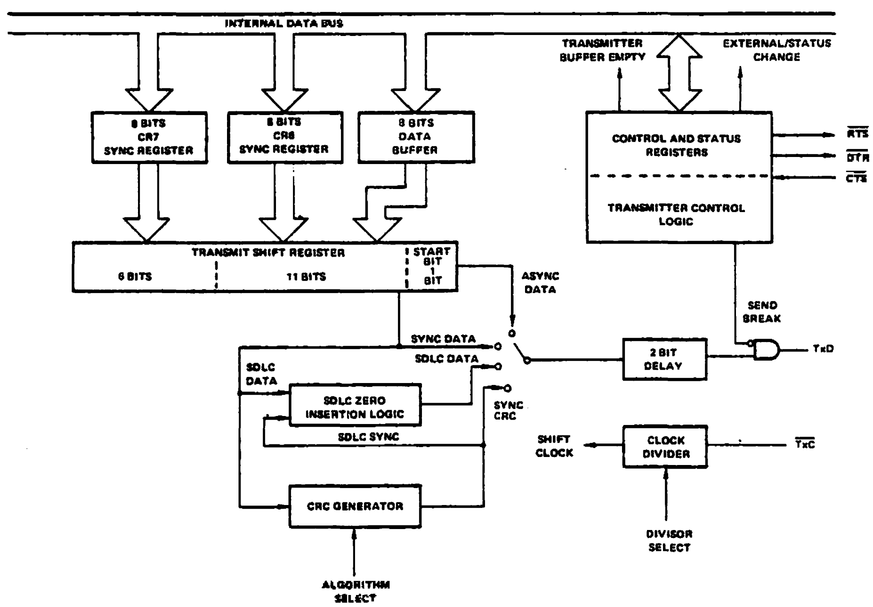


Table G-4.1 Transmitter Control and Status Registers

CONTROL REGISTER	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	CRC CONTROL		COMMAND			REGISTER POINTER		
1							Trans. Int Enable	Ext/Status Int Enable
4	Clock Mode		Sync Format Select		Sync/Async Mode Select		Parity Control	
5	DTR	Bits/Char		Send Break	Transmitter Enable	CRC Type	RTS	CRC Enable
6	SYNC 1							
7	SYNC 2							
3			Auto Enables					

STATUS REGISTER	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0		Trans Underrun/ $\overline{EOM}$	CTS			Trans Buffer Empty		
1								All Async Characters Sent

You can change the number of bits transmitted for each character at any time by modifying the bits/char field (CR5, D_5-D_6) before you load the character into the buffer.

The rate at which data is shifted out is determined by the transmitter clock input (TxC) and the clock mode field (CR4 Bits 6-7). You can select a clock divisor so that the data clock (TxC) rate is equal to 1x, 16x, 32x, or 64x the actual data rate. This field also controls the receiver clock and must be set to 1x for synchronous modes (see Section G-4.2.2 for use in asynchronous mode). Each new bit is shifted out on the falling edge of TxC.

The following is a general discussion of the operation of the MPSC² in various protocol modes. For a detailed description of the registers and examples, see Chapter G-5.

G-4.1.1 Asynchronous Mode

After you select asynchronous mode, initialize the various parameters (number of bits/character, number of stop bits, etc.) and enable the transmitter (CR5 bit 3 = 1). TxD remains in the high (marking) state. When the first character is written to the data buffer, it is transferred to the shift register and the Transmitter Buffer Becoming Empty flag is set. A parity bit, if enabled, and the specified number of stop bits (1, 1½ or 2) are appended to the character. The character plus the start bit are shifted out serially through a one-bit delay. After the character has been completely sent, the next character is loaded into the shift register and the process continues. When no more characters are available, TxD remains high and the All Async

Characters Sent flag (SR1 bit 0) is set until the next character is loaded. The transmitter may be disabled at any time (CR5 bit 3 = 0); however, transmission of the character currently being sent, if any, is completed. Disabling the transmitter does not reset the Transmitter Buffer Becoming Empty flag or any resultant interrupts or DMA requests. You can clear this flag either by writing a character to the data buffer for later transmission or by issuing a Reset Transmitter Interrupt/DMA Pending Command.

The modem control output RTS (Request To Send) may be set or reset at any time with CR5 bit 1. RTS immediately goes to the active state (low) when this bit is set. When reset, RTS does not go high until the shift register and the data buffer are empty.

The function of the modem control input, CTS (Clear To Send), depends upon the Auto Enables Control (CR3 bit 5). When Auto Enables is reset, any transition of CTS sets the External/Status Change flag but has no affect upon transmission. When Auto Enables is set, character transmission cannot begin until CTS goes low. If CTS goes high, any character currently being transmitted is completed and the transmitter is then disabled until CTS again goes low. The CTS flag, SR0 bit 5, reflects the inverted state of the external CTS pins, that is, CTS flag = 1 when CTS = low.

G-4.1.2 COP Synchronous Modes

The MPSC² gives you three distinct COP operating modes: monosync (8-bit sync character), bisync (16-bit sync character), and external sync (the transmitter operates in the same manner as Monosync). When bisync mode is selected, you should program the eight least significant bits (first byte) of the sync character into CR6 and the eight most significant bits (second byte) into CR7. For monosync and external sync modes you should program CR6 with the 8-bit sync character.

During operation in COP modes, the MPSC² transmitter may be in any one of the following phases:

Disabled Phase:	Transmitter Enable is off (CR5, D3=0) or CTS is low when the auto enables function is used;
Idle Phase:	Sync characters are being sent;
Data Phase:	Data from the processor is being transmitted;
CRC Phase:	(If CRC is used) when the CRC check characters are being transmitted.

After selecting the desired protocol and initializing parameters, the transmitter enters and remains in the Disabled Phase, with TxD high until the Transmitter Enable bit is set. Once this is done the transmitter enters the Idle Phase, transmits the first sync character and continues to send sync characters until a character is written into the transmit buffer. When the first data character is loaded into the data buffer and the current sync character has been sent, the trasnmitter enters Data Phase and sends data characters while setting the Transmitter Buffer Becoming Empty flag each time it is ready for the next character.

During the Data Phase, the transmitter may run out of data to send for one of two reasons: (1) The processor is busy and is not able to provide the next data characters within a message, or (2) the data portion of the message is complete and it is time to enter the CRC Phase (or the Idle Phase if CRC is not used). The $MPSC^2$ automatically handles both of these conditions through a mechanism called the Idle/CRC Latch, the state of which may be read from SR0 D_6.

When the transmitter is initialized the Idle/CRC Latch is set, indicating that the transmitter will enter the Idle Phase and begin sending sync characters when there is no data to send. Entering this phase also sets the Transmitter Buffer Becoming Empty flag (if not already set) to indicate with SRO $D_6 = 1$, that the Idle Phase has been entered.

However, if you reset the Idle/CRC Latch with a Reset Idle/CRC Latch command to CRO, a lack of data causes the $MPSC^2$ to enter the CRC Phase and begin sending the 16-bit CRC character calculated up to that point. Entering the CRC Phase sets the Idle/CRC Latch which, in turn, sets the External/Status Change flag indicating that the $MPSC^2$ is sending CRC. After you reset the flag, you may send the next data character to the transmitter and it will be sent immediately following the CRC, or you may do nothing. In either case, the Idle/CRC Latch is now set again so the transmitter enters the Idle Phase when no further data is available.

You can disable the transmitter during any phase of operation. If the transmitter is disabled during the Idle or Data Phases the $MPSC^2$ finishes sending the current character and goes to the Disabled Phase (TxD high). If disabled during the CRC Phase, a 16-bit CRC is sent; however, the remainder of the CRC is supplanted by sync with bit positions matching.

The CRC Generator may be programmed to either of two polynomials, CRC-16 ($x^{16} + x^{15} + x^2 + 1$) or CRC-CCITT ($x^{16} + x^{12} + x^5 + 1$). The CRC Generator may be reset to 0 at any time by issuing a Reset CRC Generator Command to CR0. Since it is sometimes necessary to exclude certain characters from the CRC calculation, the $MPSC^2$ features a CRC enable/disable control (CR5 D_0) that may be changed just prior to loading a character into the transmitter buffer to include or exclude that and subsequent characters in the CRC calculation.

G-4.1.3 SDLC (/HDLC BOP Synchronous) Mode

In SDLC mode, the $MPSC^2$ transmitter operates similarly to monosync transmission with the following exceptions:

WR6 is not used for the transmitter sync character. SDLC flags (sync) are generated internally.

Data and CRC are passed through zero insertion logic before transmission. This logic inserts a 0 bit after transmitting five contiguous ones to distinguish information from framing flags.

A special Send SDLC Abort Command is available in CR0. Issuing this command causes at least 8 but less than 14 ones to be transmitted, destroying any data in the transmitter shift register and buffer. After sending the abort, the transmitter enters Idle Phase.

Resetting the CRC generator initializes it to all ones rather than zeroes and the result bits are inverted before transmission.

G-4.2 RECEIVER

The MPSC² receiver reverses the process performed by the transmitter. It converts the serial data stream of the various protocols back to parallel data for the processor. The major components of the receiver are shown in Figure G-4.4. Control and status registers pertinent to the operation of the receiver are summarized in Table G-4.2.

The primary data path through the receiver begins at the receiver data input RxD. Data passes through a two-bit time delay and into the receiver shift register (the sync data path is described later). The point of entry into the shift register and hence the number of bits per character is determined by the mode of operation and the Bits/Character field of CR3 (D_6-D_7). You can change this field at any time provided that the character that is currently being assembled has not yet reached the new number of bits/character. If the number of bits/character specified is less than eight, the character appears right-justified in the data buffer (with the parity bit, if parity is enabled) and the left side is filled with ones (see Figure G-4.3).

Figure G-4.3 Data Format Example for Less Than 8 Bits/Character

1	1	P	D_4	D_3	D_2	D_1	D_0

5 BITS/CHARACTER; PARITY ENABLED

Figure G-4.4 Block Diagram MPSC² Receiver

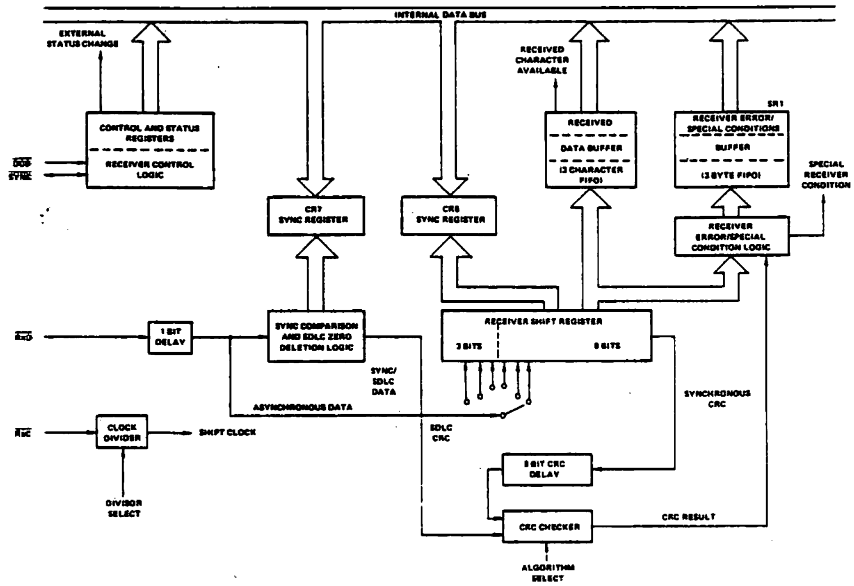


Table G-4.2 Receiver Control and Status Registers

CONTROL REGISTER	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	CRC CONTROL		COMMAND			REGISTER POINTER		
1				Receiver Interrupt Control				Ext/Status Interrupt Enable
3	Bits/Char		Auto Enables	Enter Sync Hunt Phase	Receiver CRC Enable	SDLC Address Search mode	Sync Char Load Inhibit	Receiver Enables
4	Clock Mode		Sync Format Select		Sync/Async Mode Select		Parity Control	
5						CRC Type		
6	SYNC 1							
7	SYNC 2							

STATUS REGISTER	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	Break/ Abort			Sync/Hunt Mode	DCD			Received Character Available
1	SDLC End of Frame	CRC/ Framing Error	Receiver Overrun Error	Parity Error	SDLC I-Field Residue Code			

Once the character has been assembled in the shift register, it is passed to a three-character First In-First Out buffer (FIFO) and the Received Character Available flag (and SR0 D_0) is set to inform the processor that a character is available. The three-character buffer allows the processor up to four character times to service the receiver without losing data. This feature enhances data reliability at high speeds while relaxing software timing requirements. The Received Character Available flag is reset when all characters in the buffer have been read, i.e., the buffer is empty.

As each character is transferred to the buffer, it is checked for errors or special conditions and that information is placed in a parallel FIFO error buffer so that the status associated with each character can be read with that character through status register 1. Reading a character from the data buffer moves the next character and its status to the top of the FIFO. You should read the status first, if it is of interest, and then the data.

The rate at which data is shifted into the receiver is controlled by the receiver clock input (RxC) and the clock mode field (CR4 D_6-D_7). This field also controls the transmitter clock mode. In any of the synchronous modes, you must select the 1x clock mode. In asynchronous mode you may select a divisor such that clock rate (RxC) equals 1x, 16x, 32x, or 64x the actual data rate. However, if you select the 1x mode, the clock must be externally synchronized with the data (see Section G-4.1.3). RxD is always sampled on the rising edge of RxC.

The data carrier detect (DCD) input works the same way as CTS except that it enables the receiver when auto enables is set.

G-4.2.1 Asynchronous Mode

After initializing and enabling the MPSC² Receiver, the receiver logic begins sampling the RxD input for a high-to-low (marking-to-spacing) transition on each rising edge of RxC. When the transition is found, the receiver waits ½ bit time, (for example, eight clock periods if the clock mode is 16x) and samples again to ensure that RxD is still low, improving the MPSC²'s noise immunity. If RxD is still low, the MPSC² assumes this is the middle of the start bit and one bit time later begins to sample RxD to assemble the required number of data and parity (if enabled) bits.

Once the character is assembled, the MPSC² waits one more bit time and again samples RxD. If RxD is not high, the stop bit is missing and a Framing Error is indicated when the character is passed to the data buffer. If a Framing Error has occurred, the MPSC² receiver waits ½ bit time before beginning to sample again to avoid interpreting the Framing Error as a new start bit.

Note that in the 1x Clock mode, the receiver simply waits one clock period after the first high-to-low transition is detected and then begins assembling the character. It is for this reason that data and clock must be synchronized in this mode.

The Break/Abort bit, D_7 of SR0 is set when a null character plus Framing Error is detected (i.e. RxD is low for more than one full character time). Break detection also sets the External/Status Change flag. When RxD returns high and the break has ended, D_7 is reset to 0 and the External Status Change flag is once again set. After the break, a single null character is present in the data buffer. It should be read and discarded.

The following errors may occur during operation and are flagged in status register 1.

Framing Error	See above discussion.
Parity Error	If parity is enabled and a parity error occurs, the Parity Error bit D_4 is set. Once a Parity Error has occurred, the Parity Error bit remains set for subsequent characters until reset by an Error Reset command to CR0. You need only check the end of a message or block to determine if a parity error occurred.
Overrun Error	If the data buffer is full with three characters and a fourth character is received, the last character in the buffer is overwritten and the Overrun Error bit D_5 is set. Like Parity Error, Overrun Error remains set until the Error Reset command is issued.

G-4.2.2 COP Synchronous Modes

The $MPSC^2$ gives you three distinct COP operating modes: (1) monosync (8-bit sync character), (2) bisync (16-bit character), and (3) external sync (the SYNC pin is used as an input to inform the $MPSC^2$ that synchronization has been achieved externally).

When monosync mode is selected, CR7 should be programmed with the 8-bit sync character to be matched by the receiver.

In bisync mode CR6 should contain the least significant bits (first byte) and CR7 should contain the most significant bits (second byte) of the 16-bit character to be matched.

In external sync mode, no sync character is required by the receiver. During operation in the COP modes, the $MPSC^2$ receiver is in one of two phases: (1) Sync Hunt Phase or (2) Data Phase. The receiver automatically enters Sync Hunt Phase when it is enabled (CR3, D_0).

In monosync mode, the incoming data stream passes through and is compared to the sync character in CR7. When a match is found, the receiver switches to Data Phase and begins to pass data to the shift register. If you determine at any time that synchronization has been lost, you may re-enter the Sync Hunt Phase by setting the Enter Hunt Phase bit (D_4) in CR3. When the Hunt Phase is entered or left, the External/Status Change flag is set. When SR0 D_4 (Sync/Hunt) = one, it indicates that the receiver is in Hunt Phase.

Operation is similar in bisync mode, however, when a match is found, CR6 is also checked against the shift register contents and the Hunt Phase is left only if the bytes match. In both monosync and bisync modes, the SYNC pin is used as an output which goes momentarily low any time a sync pattern is detected whether the receiver is in Hunt or Data Phase. See Figure G-2.3 for a detailed timing diagram.

You can inhibit the transfer of sync characters to the data register by setting the Sync Char Load Inhibit bit (CR3, D_1). Since the CRC calculation on sync is not inhibited by this bit, you should use it only to strip leading sync characters from a message if you are using CRC Block Check.

Because of the 8-bit delay between the shift register and the CRC checker, CRC status (SR1, D_6) is not valid immediately after the CRC character is received. CRC status is valid 16 bit times after the last CRC character is transferred to the receive buffer, or 20 bit times after the last CRC bit is shifted in at RxD.

G-4.2.3 SDLC (/HDLC BOP Synchronous) Mode

The MPSC² provides you with high-level processing capability for handling bit-oriented protocols. When you select SDLC Mode, CR7 must be programmed with the SDLC Flag character 01111110.

When operating in SDLC mode, the receiver can be in one of three phases: Hunt Phase, Address Search Phase, or Data Phase.

The receiver automatically enters Hunt Phase when first enabled. The incoming data stream passes through the one-bit delay and enters the Sync Comparison/Zero Deletion logic where the following three operations are performed.

First, whenever a 0 bit follows five consecutive ones, that 0 is deleted from the data stream. Second, if six consecutive ones are received, a Flag Character Received indication is given internally. Third, if eight or more ones are received, an abort is indicated and the External/Status Change Flag is set. Flags and aborts are not transferred to the receiver shift register.

Once a flag is detected, the receiver leaves Hunt Phase (setting the External/Status Change Flag) and, if Address Search Mode (CR3-D_2) is enabled, it enters Address Search Phase. Once this phase is entered, the MPSC² receiver compares the first 8-bit non-flag character with the contents of control register 6. If the two values match, or the received character is the global address 11111111, the receiver immediately enters Data Phase and character assembly begins with this character. If no match is found and the value is not the global address, the receiver remains in Address Search Phase and no data characters are assembled until a flag followed by the correct address is encountered. If Address search Mode is not enabled, Data Phase is entered immediately and character assembly begins with the first non-flag character. Since all messages are framed with flag characters, you can skip an incoming message at any time simply by setting the Enter Hunt Phase bit (D_4) in CR3.

Once in Data Phase, characters are assembled according to the number of bits or characters specified until the next End of Frame flag is encountered. The receiver then sets the Special Receive Condition flag and transfers the character currently being assembled to the receiver buffer regardless of the number of bits actually assembled. A special residue code placed in the status buffer (SR1) uses the number of bits assembled to indicate the boundary between the data and CRC characters (see Section G-5.1 for a more detailed description of the residue code). If Address Search Mode is enabled, the receiver once again enters Address Search Phase.

Unlike the COP mode of operation, data from the Sync Comparison/Zero Deletion logic passes directly to the CRC checker. As a result, when the End of Frame Flag is detected, the CRC calculation is complete and the error status is passed to the status buffer along with the residue code. The CRC checker is automatically reset to all ones at this time.

G-4.3 BUS INTERFACE CONTROLLER

The bus interface controller is the interface between the transmitter and receiver sections and the processor bus. The major components of this section are shown in Figure G-4.5. The control and status registers pertinent to the operation of the control section are illustrated in Table G-4.4.

The bus interface controller can be divided into four major components:

Bus Control Logic
Interrupt Control Logic
DMA Control Logic
Clock and Reset Control Logic

All of these components interact to provide a flexible high-performance interface between the bus architecture defined by your processor and application and the various internal elements that make up the $MPSC^2$.

G-4.3.1 Bus Control Logic

The bus control logic determines the direction and internal source or destination of data and control transfers between the $MPSC^2$ and the processor bus. During operation of the $MPSC^2$, the bus control logic may operate in any of three distinct modes: Processor Read/Write, Interrupt Acknowledge, and DMA Cycle. These last two modes are described in detail in Sections G-4.3.2 and G-4.3.3.

Processor Read/Write mode is the normal mode of operation. The processor transfers data or commands and status to or from the $MPSC^2$ with its instruction set. The $MPSC^2$ is enabled for Processor Read/Write mode when the chip select (CS) input is made active (low). The direction of the transfer is controlled by enabling either the read (RD) or write (WR) inputs. The B/A input determines the source/destination channel for the transfer and the C/D input specifies whether the transfer is character data or control/status information. These inputs are generally connected to the two low-order address lines. Figure 6.1 illustrates a typical connection between a processor and the $MPSC^2$.

Table G-4.3 Read/Write Selection

$\overline{CS}$	B/$\overline{A}$	C/$\overline{D}$	$\overline{RD}$	$\overline{WR}$	OPERATION
1	X	X	X	X	NO OPERATION. THE MPSC^2 IS DESELECTED.
0	X	X	1	1	NO OPERATION. THE MPSC^2 IS DESELECTED.
0	0	0	1	0	WRITE A CHAR TO CHANNEL A TRANSMITTER.
0	0	0	0	1	READ A CHAR FROM CHANNEL A RECIVER.
0	0	1	1	0	WRITE A CONTROL BYTE TO CHANNEL A.
0	0	1	0	1	READ A STATUS BYTE FROM CHANNEL A.
0	1	0	1	0	WRITE A CHAR TO CHANNEL B TRANSMITTER.
0	1	0	0	1	READ A CHAR FROM CHANNEL B RECEIVER.
0	1	1	1	0	WRITE A CONTROL BYTE TO CHANNEL B.
0	1	1	0	1	READ A STATUS BYTE FROM CHANNEL B.
0	X	X	0	0	ILLEGAL.

G-4.3.2 Interrupt Control Logic

The interrupt control logic performs two functions: it prioritizes various internal input requests, and places the appropriate information on the data bus during an Interrupt Acknowledge cycle (if you enabled the MPSC²s vectored interrupt feature).

Figure G-4.5 Bus Interface Controller

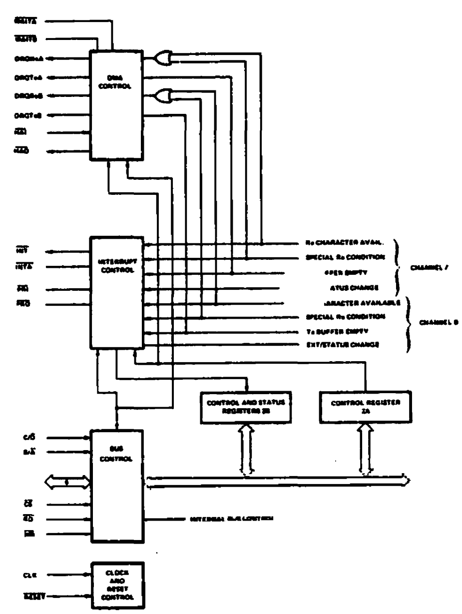


Table G-4.4 Bus Interface Controller Control and Status Registers

CONTROL REGISTER	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
CR0			COMMAND*			REGISTER POINTER		
CR2A		0	Vector Mode Select			Priority	DMA Mode Select	
CR2B	INTERRUPT VECTOR							

*Relevant commands
Channel Reset
End of Interrupt

STATUS REGISTER	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
CR0							Interrupt Pending	
CR2B	INTERRUPT VECTOR							

Each MSPC² channel can generate four different types of interrupt requests:

Received Character Available

Special Received Condition (character received but with an error or SDLC End of Frame flag received)

Transmitter Buffer Empty

External input (CTS, DCD, SYNC, Internal Status (Sync,Idle/CRC Latch) Change)

When any of these requests occurs, the interrupt control logic determines whether to accept the request at that time, issue an interrupt request by setting the INT output low when the request is accepted, and, if Vectored Interrupt mode is enabled, place the interrupt information on the data bus during the times that the interrupt acknowledge input (INTA) is activated by the processor.

As an example, assume that the channel A DCD input has just changed state causing an External/Status Change interrupt request. The following sequence occurs:

If all the following conditions are true:

External/Status Change interrupts are enabled

No higher priority interrupt requests are pending

PRI is active

The MPSC² is not acknowledging a pending lower priority interrupt request

Then the interrupt control logic accepts the interrupt request and sets INT active and PRO inactive.

If Vectored Interrupt mode is enabled, the MPSC² may place information on the data bus in response to a series of INTA pulses as shown in the following chart.

Table G-4.5 Vectored Interrupt Mode

Interrupt Mode Select	PRI	INTA Cycle 1	INTA Cycle 2	INTA Cycle 3
8080/5 Master	0	CD HEX (CALL OP)	VECTOR	0
	1	CD HEX (CALL OP)	HI-Z	HI-Z
8080/5 Slave	0	HI-Z	VECTOR	0
	1	HI-Z	HI-Z	HI-Z
8086	0	HI-Z	VECTOR	*
	1	HI-Z	HI-Z	*

*The 8086 issues 2 Interrupt Acknowledge pulses rather than 3.

When operating in the 8080/5 modes, the MPSC² issues an 8080-type CALL CD vv Hex instruction where vv is the contents of control register 2B (modified by the cause of the interrupt if the Status Affects Vector feature is enabled). In particular, an MPSC² programmed for 8085 Master mode always places the CALL opcode on the data bus regardless of whether that MPSC² has a pending interrupt request. To avoid problems caused by momentary bus contention, you should never program more than one device to operate in this mode.

In 8086 mode, the MPSC² places the vector on the data bus during the second interrupt acknowledge to vector the processor to the approximate location in low memory.

Figure G-4.6 MPSC² Interrupt Conditions

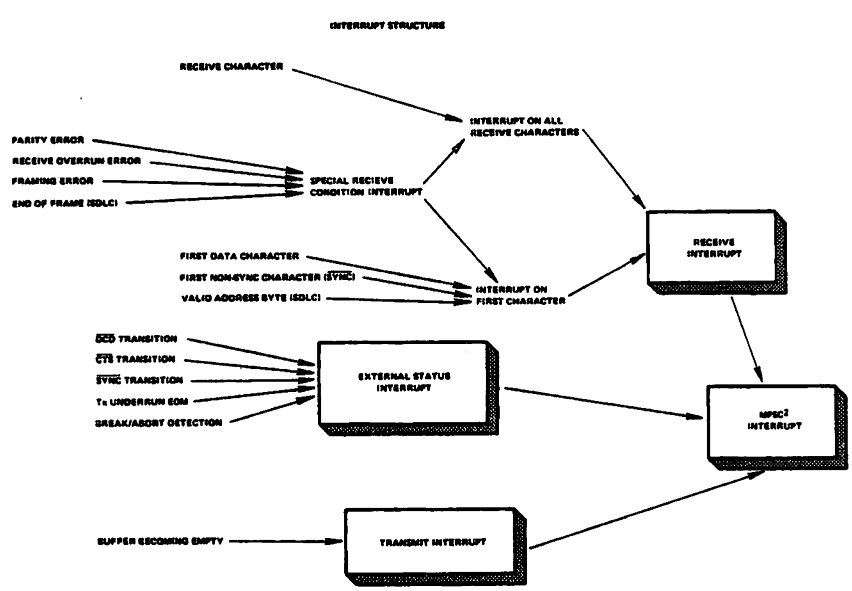
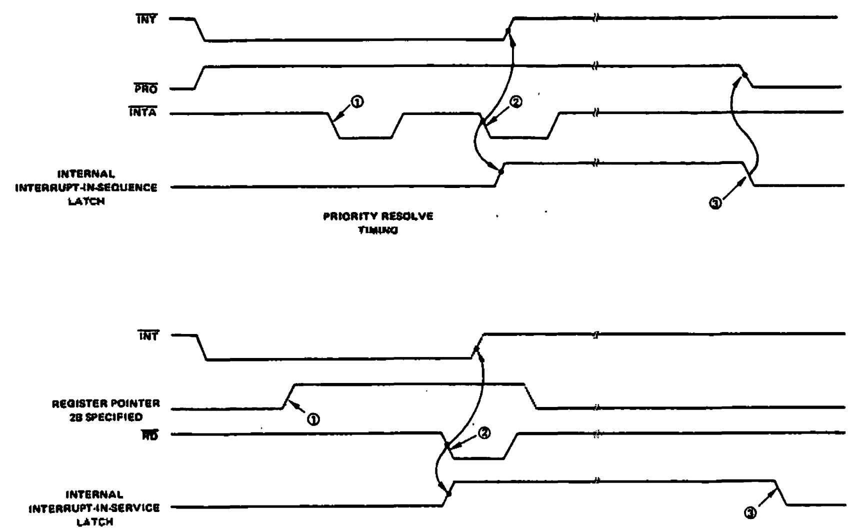


Figure G-4.7 illustrates the action of the interrupt control logic during an interrupt acknowledge sequence.

Figure G-4.7 Interrupt Timing



At the beginning of the first Interrupt Acknowledge cycle, the interrupt prioritization logic is frozen to permit any late interrupt requests by higher priority devices to ripple through and resolve internal priorities before the second interrupt pulse.

At the end of the second INTA pulse, the INT output is released by the acknowledging device and the interrupt prioritization logic is re-enabled with an Interrupt In Service flag set. As long as this flag is set, PRO is held high and only internal interrupt requests with a priority higher than the one currently being serviced are accepted.

While the interrupt is being serviced, the processor issues an End of Interrupt (EOI) command to the MPSC² to reset the interrupt control logic to its previous state. This scheme permits nested interrupts to be serviced and the priority daisy chain to be properly maintained.

When the MPSC² is operated in Non-vectored Interrupt mode, the interrupt control logic operates in a similar manner except that INTA is not used and no vector information is placed on the data bus. Rather, the interrupt acknowledge sequence is simulated by reading the vector (modified if Status Affects Vector is enabled) in status register 2B.

G-4.3.3 DMA Control Logic

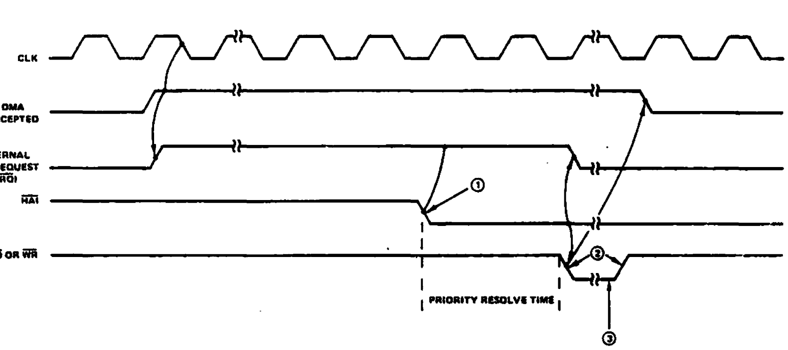
The function of the DMA logic is somewhat similar to that of the interrupt control logic in that service requests must be accepted, prioritized, and information placed on (or, in this case, accepted from as well) the data bus at the appropriate times. However, the purpose of the DMA control logic is to enable the MPSC^2 to avoid interrupting the processor to make a data transfer. This is accomplished by activating an external controller to move the data directly from the MPSC^2 to memory, or vice versa.

The DMA control logic accepts requests from four sources: (1) Received Data Available in channel A, (2) Transmitter Buffer Becoming Empty in channel A., (3) Data Available in channel B, and (4) Transmitter Buffer Becoming Empty in channel B. When an internal DMA request is made by one of the above sources and DMA mode is enabled for that channel, the appropriate DMA request output (e.g. DRQRxA when received data is available in channel A) is made active. This causes the external DMA controller to request control of the processor bus with a hold request. The MPSC^2's daisy chain output, HAO, is at this point locked in the inactive (high) state.

Some time later, the external DMA controller gains control of the processor bus as the processor asserts its hold acknowledge output.

The DMA Controller now places the source or destination address on the address bus and asserts the I/O read or write control line for a data transfer from or to the MPSC^2, respectively. The MPSC^2 also receives the processor hold acknowledge signal possibly through higher priority MPSC^{2}s not requesting DMA, at its HAI input. When HAI is asserted, the DMA control logic freezes all internal requests, determines which one has the highest priority, and performs the transfer when I/O read or write is received from the DMA controller at RD or WR. Once the transfer is complete, the prioritization logic is re-enabled and new or pending requests can be serviced. Figure G-4.8 illustrates some of the timing details of a DMA transfer.

Figure G-4.8 DMA Data Transfer Timing

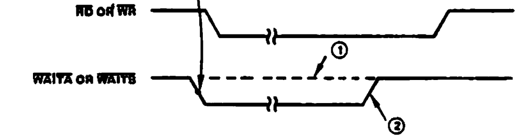


From the above explanation you should note two points. First, in the case of multiple DMA requests from one MPSC², both the MPSC² and the external DMA controller establish priorities independently to determine which request to service first. As a result, you MUST connect the MPSC²'s DMA request outputs to the DMA controller so that both make the same priority decisions. For example, when using the MPSC² with an 8257-type DMA controller and the priority bit (CR2A-D_2) = 0, you must set the controller to the fixed priority mode (as opposed to rotating priority), and connect the MPSC²s DRQRxA output to the 8257's DRQ 0 input, DRQTxA to DRQ 1, and so on.

The second point is that many DMA controllers, such as the 8257, may begin the transfer by asserting RD or WR before the MPSC² can receive HAI through the daisy chain and resolve request priorities. Because of this, you should always derive HLDA to the DMA Controller from HAI of the MPSC²(s) to which it is connected. Additionally, a delay circuit from HAI to HLDA is recommended. Figure G-6.5 shows a typical MPSC²/DMA interface which conforms to these points.

The mechanism that controls the WAIT outputs of the MPSC² is related to the DMA logic. When enabled, the wait logic pulls the WAIT line active when the processor attempts to perform a data transfer operation at an inappropriate time. If WAIT is connected to the processor's WAIT (or READY) input, it waits until the line is released by the MPSC² before completing the data transfer. Since the processor is dedicated to either a read or write operation at any one time, only one WAIT output is required for each channel. You may assign it to operate with either the transmitter or the receiver. Figure G-4.9 illustrates the basic wait feature timing.

Figure G-4.9 Wait Mode Timing



G-4.3.4 Clock and Reset Control Logic

The clock input of the MPSC² controls the various timing states of the MPSC² and is usually connected to the processor clock. The clock is not used by the bus control logic and data transfers need not be synchronized to it in any way. The receiver and transmitter sections use the clock, and it must be at least 4.5x the highest data clock frequency you plan to use. The DMA control logic also uses the clock, and it should be the same clock seen by the external DMA Controller.

The RESET input is used at power-up and at any other time that you wish to reset the MPSC² to its initial state. After a reset, all transmitters and receivers are disabled, any pending interrupt and DMA requests are cleared, and the modem control outputs DTR and RTS are reset (high). When you reset the MPSC², you must hold the RESET input low for at least one complete clock cycle.

G-5 PROGRAMMING THE MPSC²

The software operation of the MPSC² is very straightforward. Its consistent register organization and high-level command structure help to minimize the number of operations required to implement complex protocol designs. Programming is further simplified by the MPSC²s extensive interrupt and status reporting capabilities.

This section is divided into two parts. The first is a detailed description of the commands, bits, and fields in the various MPSC² control and status registers. The second part provides programming examples and flowcharts for the MPSC²'s various operating modes to assist you in developing software for your specific application.

G-5.1 THE MPSC² REGISTERS

The MPSC² interfaces to the system software with a number of control and status registers associated with each channel. Commonly used commands and status bits are accessed directly through control and status registers 0. Other functions are accessed indirectly with a register pointer to minimize the address space that must be dedicated to the MPSC².

Table G-5.1 Control Registers

CONTROL REGISTER	FUNCTION
0	FREQUENTLY USED COMMANDS AND REGISTER POINTER CONTROL
1	INTERRUPT CONTROL
2	PROCESSOR/BUS INTERFACE CONTROL
3	RECEIVER CONTROL
4	MODE CONTROL
5	TRANSMITTER CONTROL
6	SYNC/ADDRESS CHARACTER
7	SYNC CHARACTER

Table G-5.2 Status Registers

STATUS REGISTER	FUNCTION
0	BUFFER AND "EXTERNAL/STATUS" STATUS
1	RECEIVED CHARACTER ERROR AND SPECIAL CONDITION STATUS
2 (CHANNEL B ONLY)	INTERRUPT VECTOR

All control and status registers except CR2 are separately maintained for each channel. Control and status registers 2 are linked with the overall operation of the $MPSC^2$ and have different meanings when addressed through different channels.

When initializing the $MPSC^2$, control register 2A (and 2B if desired) should be programmed first to establish the $MPSC^2$ processor/bus interface mode. You may then program each channel to be used separately, beginning with control register 4 to set the protocol mode for that channel. The remaining registers may then be programmed in any order.

G-5.1.1 Control Register 0

Figure G- 5.1 Control Register 0

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
CRC CONTROL COMMAND		COMMAND			REGISTER POINTER		

Register Pointer (D_0-D_2)

The register pointer specifies which register number is accessed at the next Control Register Write or Status Register Read. After a hardware or software reset, the register pointer is set to 0. Therefore, the first control byte goes to control register 0. When the register pointer is set to a value other than 0, the next control or status ($C/D = 1$) access is to the specified register, after which the pointer is reset to 0. You can freely combine other commands in control register 0 with setting the register pointer.

Command (D_3-D_5)

Commands commonly used during the operation of the MPSC^2 are grouped in control register 0. They are:

Null (000)

This command has no effect and is used when you wish to set only the register pointer or issue a CRC command.

Send Abort (001)

When operating in SDLC mode, this command causes the MPSC^2 to transmit the SDLC abort code, issuing 8 to 13 consecutive ones. Any data currently in the transmitter or the transmitter buffer is destroyed. After sending the abort, the transmitter reverts to the Idle Phase (flags).

Reset External/Status Interrupts (010)

When the External/Status Change flag is set, the condition bits D_0-D_2 of status register 0 are latched to allow you to capture short pulses that may occur. The Reset External/Status Interrupts Command clears a pending interrupt and re-enables the latches so that new interrupts may be sensed.

Channel Reset (011)

This command has the same effect on a single channel as an external reset at pin 2. A channel reset command to channel A resets the internal interrupt prioritization logic. This does not occur when you issue a Channel Reset command to channel B. You must reinitialize all control registers associated with the channel that you reset. After a channel reset, you must wait at least four system clock cycles before writing new commands or controls to that channel.

Enable Interrupt on Next Character (100)

When operating the MPSC^2 in Interrupt on First Received Character mode, you may issue this command at any time (generally at the end of a message), to re-enable the interrupt logic for the next received character.

Reset Pending Transmitter Interrupt/DMA Request (101)

You can reset a pending Transmitter Buffer Becoming Empty interrupt or DMA request without sending another character by issuing this command (typically at the end of a message). A new Transmitter Buffer Becoming Empty interrupt or DMA request is not made until another character has been loaded and transferred to the transmitter shift register or when, if operating in synchronous or SDLC mode, the CRC character has been completely sent and the first sync or flag character loaded into the transmitter shift register.

Error Reset (110)

This command resets a Special Receive Condition interrupt. It also re-enables the Parity and Overrun Error latches that allow you to check for these errors at the end of a message.

End of Interrupt (111)(Channel A only)

Once an interrupt request has been issued by the MPSC[2], all lower priority internal and external interrupts in the daisy chain are held off to permit the current interrupt to be serviced while allowing higher priority interrupts to occur. At some point in your interrupt service routine (generally at the end), you must issue the End of Interrupt command to channel A to re-enable the daisy chain and allow any pending lower priority internal interrupt requests to occur.

CRC Control Commands (D_6-D_7)

These commands control the operation of the CRC generator/ checker logic.

Null (00)

This command has no effect and is used when issuing other commands or setting the register pointer.

Reset Receiver CRC Checker (01)

This command resets the CRC checker to 0 when the channel is in a synchronous mode and resets to all ones when in SDLC mode.

Reset Transmitter CRC Generator (10)

This command resets the CRC generator to 0 when the channel is in a synchronous mode and resets to all ones when in SDLC mode.

Reset Idle/CRC Latch (11)

This command resets the Idle/CRC latch so that when a transmitter underrun condition occurs (that is, the transmitter has no more characters to send), the transmitter enters the CRC Phase of operation and begins to send the 16-bit CRC character calculated up to that point. The latch is then set so that if the underrun condition persists, idle characters are sent following the CRC. After a hardware or software reset, the latch is in the set state.

G-5.1.2 Control Register 1

Figure G-5.2 Control Register 1

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
WAIT FUNCTION ENABLE	∅	WAIT ON RECEIVER TRANSMITTER	RECEIVER INTERRUPT MODE		CONDITION AFFECTS VECTOR	TRANSMITTER INTERRUPT ENABLE	EXT/STATUS INT ENABLE

External/Status Interrupt Enable (D_0)

When this bit is set to one, the $MPSC^2$ issues an interrupt whenever any of the following occur:

transition of DCD input

transition of CTS input

transition of SYNC input

entering or leaving synchronous Hunt Phase break detection or termination

SDLC abort detection or termination

Idle/CRC latch becoming set (CRC being sent)

Transmitter Interrupt Enable (D_1)

When this bit is set to one, the $MPSC^2$ issues an interrupt when:

the character currently in the transmitter buffer is transferred to the shift register (Transmitter Buffer Becoming Empty) or,

the transmitter enters Idle Phase and begins transmitting sync or flag characters.

Status Affects Vector (D_2)

When this bit is set to 0, the fixed vector programmed in CR2B during $MPSC^2$ initialization is returned in an interrupt acknowledge sequence. When this bit is set to 1, the vector is modified to reflect the condition that caused the interrupt. See Section G-5.1.12 for a detailed explanation of the $MPSC^2$'s vectored interrupt feature.

Receiver Interrupt Mode (D_3-D_4)

This field controls how the $MPSC^2$'s interrupt/DMA logic handles the character received condition.

Receiver Interrupts/DMA Request Disabled (00)

The $MPSC^2$ does not issue an interrupt or a DMA request when a character has been received.

Interrupt on First Received Character Only (01)

(and issue a DMA Request)

In this mode, the $MPSC^2$ issues an interrupt only for the first character received after an Enable Interrupt on First Character Command (CRO) has been given. If the channel is in DMA mode, a DMA request is issued for each character received including the first. This mode is generally used when using the $MPSC^2$ in DMA or Block Transfer mode to signal the processor that the beginning of an incoming message has been received.

Interrupt (and issue a DMA Request) (10)

On All Received Characters
Parity Error is a Special Receive Condition

In this mode, an interrupt (and DMA request if DMA mode is selected) is issued whenever there is a character present in the receiver buffer. A parity error is considered a special receive condition.

Interrupt (and issue a DMA request) (11)

On All Received Characters
Parity Error is not a Special Receive Condition

This mode is the same as above except that a parity error is not considered a special receive condition. The following are considered special receive conditions and, when status affects vector is enabled, cause an interrupt vector different from that caused by a received character available condition:

Receiver Overrun Error

Parity Error (if specified)

SDLC End of Message (final flag received)

Wait on Receiver/Transmitter (D_5)

If the Wait function is enabled for block mode transfers, setting this bit to 0 causes the MPSC² to issue a wait (WAIT output goes low) when the processor attempts to write a character to the transmitter while the transmitter buffer is full. Setting this bit to 1 causes the MPSC² to issue a wait when the processor attempts to read a character from the receiver while the receiver buffer is empty.

Wait Function Enable (D_7)

Setting this bit to 1 enables the wait function as described above and in Section 4.3.3.

G-5.1.3 Control Register 2 (Channel A)

Figure G-5.3 Control Register 2 (Channel A)

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
PIN 10 $\overline{\text{SYNCB}}$/RTSB	∅	INTERRUPT VECTOR MODE			PRIORITY	DMA MODE SELECT	

DMA Mode Select (D_0-D_1)

Setting this field establishes whether channels A and B are used in DMA mode (i.e. data transfers are performed by a DMA controller) or in non-DMA mode where transfers are performed by the processor in either Polled, Interrupt, or Block Transfer modes. The functions of some MPSC² pins are also controlled by this field.

Table G-5.3 DMA Mode Selection

		Channel		Pin Function					
D_1	D_0	A	B	11	25	29	30	31	32
0	0	Non-DMA	Non-DMA	$\overline{\text{WAITB}}$	$\overline{\text{DTRB}}$	$\overline{\text{PRI}}$	$\overline{\text{PRO}}$	$\overline{\text{DTRA}}$	$\overline{\text{WAITA}}$
0	1	DMA	Non-DMA	DRQTxA	$\overline{\text{HAI}}$	$\overline{\text{PRI}}$	$\overline{\text{PRO}}$	$\overline{\text{HAO}}$	DRQRxA
1	0	DMA	DMA	DRQTxA	$\overline{\text{HAI}}$	DRQRxB	DRQTxB	$\overline{\text{HAO}}$	DRQRxA
1	1	Illegal	–	–	–	–	–	–	–

Priority (D_2)

This bit allows you to select the relative priorities of the various interrupt and DMA conditions according to your application.

Table G-5.4 DMA/Interrupt Priorities

	Mode		DMA Priority Relation	Interrupt Priority Relation
D_2	CHA	CHB		
0	INT	INT	———	RxA > TxA > RxB > TxB > ExTA > ExTB
1			———	RxA > RxB > TxA > TxB > ExTA > ExTB
0	DMA	INT	RxA TxA	RxA > RxB > TxB > ExTA > ExTB
1			RxA TxA	RxA > RxB > TxB > ExTA > ExTB
0	DMA	DMA	RxA TxA RxB TxB	RxA > RxB > ExTA > ExTB
1			RxA RxB TxA TxB	RxA > RxB > ExTA > ExTB

Interrupt Vector Mode (D_3-D_5)

This field determines how the MPSC² responds to an interrupt acknowledge sequence from the processor. See Section 4.3.2 for a detailed description of the MPSC² response in these modes.

Table G-5.5 Interrupt Acknowledge Sequence Response

D_5	D_4	D_3	Mode	Status Register 2B and Interrupt Vector bits affected when Condition Affects Vector is enabled
0	0	0	Non-Vectored	D_4 D_3 D_2
0	0	1	Non-Vectored	D_4 D_3 D_2
0	1	0	Non-Vectored	D_2 D_1 D_0
0	1	1	Illegal	–
1	0	0	8085 Master	D_4 D_3 D_2
1	0	1	8085 Slave	D_4 D_3 D_2
1	1	0	8086	D_2 D_1 D_0
1	1	1	Illegal	–

Pin 10 SYNCB/RTSB Select (D_7)

Programming a 0 into this bit selects RTSB as the function of pin 10. A one selects SYNCB as the function.

G-5.1.4 Control Register 2 (Channel B)

Figure G-5.4 Control Register 2 (Channel B)

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
INTERRUPT VECTOR							

G-Interrupt Vector (D_0-D_7)

When the MPSC² is used in Vectored Interrupt mode, the contents of this register are placed on the bus during the appropriate portion of the interrupt acknowledge sequence. Its value is modified if status affects vector is enabled. You can read the value of CR2B at any time. This feature is particularly useful in determining the cause of an interrupt when using the MPSC² in Non-vectored Interrupt mode.

G-5.1.5 Control Register 3

Figure G-5.5 Control Register 3

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
NUMBER OF RECEIVED BITS/CHARACTER		AUTO ENABLES	ENTER HUNT PHASE	RECEIVER CRC ENABLE	ADDRESS SEARCH MODE	SYNC CHARACTER LOAD INHIBIT	RECEIVER ENABLE

Receiver Enable (D_0)

After the channel has been completely initialized, setting this bit to 1 allows the receiver to begin operation. You may set this bit to 0 at any time to disable the receiver.

Sync Character Load Inhibit (D_1)

In a synchronous mode, this bit inhibits the transfer of sync characters to the receiver buffer, thus performing a "sync stripping" operation. When using the MPSC²'s CRC checking ability, you should use this feature only to strip leading sync characters preceding a message since the load inhibit does not exclude sync characters embedded in the message from the CRC calculation. Synchronous protocols using other types of block checking such as checksum or LRC are free to strip embedded sync characters with this bit.

Address Search Mode (D_2)

In SDLC Mode, setting this bit places the MPSC² in Address Search mode where character assembly does not begin until the 8-bit character (secondary address field) following the starting flag of a message matches either the address programmed into CR6 or the global address 11111111.

Receiver CRC Enable (D_3)

This bit enables and disables (1 = enable) the CRC checker in COP mode to allow you to selectively include or exclude characters from the CRC calculation. The MPSC² features a one-character delay between the receiver shift register and the CRC checker so that the enabling or disabling takes effect with the last charcter transferred from the shift register to the receiver buffer. Therefore, you have one full character time in which to read the character and decide whether it should be included in the CRC calculation.

Enter Hunt Phase (D_4)

Although the MPSC² receiver automatically enters Sync Hunt Phase after a reset, there are times when you may wish to reenter it, such as when you have determined that synchronization has been lost or, in SDLC mode, to ignore the current incoming message. Writing a 1 into this bit at any time after initialization causes the MPSC² to reenter Hunt Phase.

Auto Enables (D_5)

Setting this bit to 1 causes the DCD and CTS inputs to act as enable inputs to the receiver and transmitter, respectively.

Number of Received Bits/Character (D_6-D_7)

This field specifies the number of data bits assembled to make each character.

You may change the value on the fly while a character is being assembled and if the change is made before the new number of bits has been reached, it affects that character. Otherwise the new specifications take effect on the next character received.

Table G-5.6 Received Bits/Character

D_7	D_6	BITS/CHARACTER
0	0	5
0	1	7
1	0	6
1	1	8

G-5.1.6 Control Register 4

Figure G-5.6 Control Register 4

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
CLOCK RATE		SYNC MODE		NUMBER OF STOP BITS SYNC MODE		PARITY EVEN/ODD	PARITY ENABLE

Parity Enable (D_0)

Setting this bit to 1 adds an extra data bit containing parity information to each transmitted character. Each received character is expected to contain this extra bit and the receiver parity checker is enabled.

Parity Even/Odd (D_1)

Programming a 0 into this bit when parity is enabled causes the transmitted parity bit to take on the value required for odd parity. The received character is checked for odd parity. Conversely, a 1 in this bit signifies even parity generation and checking.

Number of Stop Bits/Sync Mode (D_2-D_3)

This field specifies whether the channel is used in synchronous (or SDLC) mode or in asynchronous mode. In asynchronous mode, this field also specifies the number of bit times used as the stop bit length by the transmitter. The receiver always checks for one stop bit.

Table G-5.7 Stop Bits

D_3	D_2	MODE
0	0	SYNCHRONOUS MODES
0	1	ASYNCHRONOUS 1 BIT TIME (1 STOP BIT)
1	0	ASYNCHRONOUS 1½ BIT TIMES (1½ STOP BITS)
1	1	ASYNCHRONOUS 2 BIT TIMES (2 STOP BITS)

Sync Mode (D_4-D_5)

When the Stop Bits/Sync Mode field is programmed for synchronous modes D_2 D_3 = 00), this field specifies the particular synchronous format to be used. This field is ignored in asynchronous mode.

Table G-5.8 Synchronous Formats

D_5	D_4	MODE
0	0	8-BIT INTERNAL SYNCHRONIZATION CHARACTER (MONOSYNC)
0	1	16-BIT INTERNAL SYNCHRONIZATION CHARACTER (BISYNC)
1	0	SDLC
1	1	EXTERNAL SYNCHRONIZATION ($\overline{SYNC}$ PIN BECOMES AN INPUT)

Clock Rate (D_6-D_7)

This field specifies the relationship between the transmitter and receiver clock inputs (TxC, RxC) and the actual data rate at TxD and RxD. When operating in a synchronous mode you must specify a 1x clock rate. In asynchronous modes, any of the rates may be specified, however, with a 1x clock rate the receiver cannot determine the center of the start bit. In this mode, you must externally synchronize the sampling (rising) edge of RxC with the data.

Table G-5.9 Clock Rates

CLOCK RATE 1 D_7	CLOCK RATE 2 D_6	CLOCK RATE
0	0	CLOCK RATE = 1x DATA RATE
0	1	CLOCK RATE = 16x DATA RATE
1	0	CLOCK RATE = 32x DATA RATE
1	1	CLOCK RATE = 64x DATA RATE

G-5.1.7 Control Register 5

Figure G-5.7 Control Register 5

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
DTR	NUMBER OF TRANSMITTED BITS/CHARACTER		SEND BREAK	TRANSMITTER ENABLE	CRC POLYNOMIAL SELECT	$\overline{RTS}$	TRANSMITTER CRC ENABLE

Transmitter CRC Enable (D_0)

A 1 or a 0 enables or disables, respectively, CRC generator calculation. The enable or disable does not take effect until the next character is transferred from the transmitter buffer to the shift register, thus allowing you to include or exclude specific characters from the

CRC calculation. By setting or resetting this bit just before loading the next character, it and subsequent characters are included or excluded from the calculation. If this bit is 0 when the transmitter becomes empty, the MPSC² goes to the Idle Phase, regardless of the state of the Idle/CRC latch.

RTS (D_1)

In synchronous and SDLC modes, setting this bit to 1 causes the RTS pin to go low while a 0 causes it to go high. In asynchronous mode, setting this bit to 0 does not cause RTS to go high until the transmitter is completely empty. This feature facilitates programming the MPSC² for use with asynchronous modems.

CRC Polynomial Select (D_2)

This bit selects the polynomial used by the transmitter and receiver for CRC generation and checking. A 1 selects the CRC-16 polynomial ($x^{16} + x^{15} + x^2 + 1$). A 0 selects the CRC-CCITT Polynomial ($x^{16} + x^{12} + x^5 + 1$). In SDLC mode, you must select CRC-CCITT. You may use either polynomial in other synchronous modes.

Transmitter Enable (D_3)

After a reset, the transmitted data output (TxD) is held high (marking) and the transmitter is disabled until this bit is set.

In asynchronous mode, TxD remains high until data is loaded for transmission.

In synchronous and SDLC modes, the MPSC² automatically enters Idle Phase and sends the programmed sync or flag characters.

When the transmitter is disabled in asynchronous mode, any character currently being sent is completed before TxD returns to the marking state.

If you disable the transmitter during the Data Phase in synchronous mode, the current character is sent, then TxD goes high (marking).

In SDLC mode, the current character is sent, but the marking line following is zero-inserted. That is, the lines goes low for one bit time out of every five.

You should never disable the transmitter during the SDLC Data Phase unless a reset is to follow immediately. In either case, any character in the buffer register is held.

Disabling the transmitter during the CRC Phase causes the remainder of the CRC character to be bit-substituted with sync (or flag). The total number of bits transmitted is correct and TxD goes high after they are sent.

If you disable the transmitter during the Idle Phase, the remainder of sync (flag) character is sent, then TxD goes high.

Send Break (D_4)

Setting this bit to 1 immediately forces the transmitter output (TxD) low (spacing). This function overrides the normal transmitter output and destroys any data being transmitted although the transmitter is still in operation. Resetting this bit releases the transmiter output.

Transmitted Bits/Character (D_5-D_6)

This field controls the number of data bits transmitted in each character. You may change the number of bits/character by rewriting this field just before you load the first character to use the new specification.

Table G-5.10 Transmitted Bits/Character

TRANSMIT BITS PER CHARACTER 1	TRANSMIT BITS PER CHARACTER	
D_6	D_5	BITS/CHARACTER
0	0	5 OR LESS (SEE BELOW)
0	1	7
1	0	6
1	1	8

Normally each character is sent to the MPSC[2] right-justified and the unused bits are ignored. However, when sending five bits or less the data should be formatted as shown below to inform the MPSC[2] of the precise number of bits to be sent.

Table G-5.11 Transmitted Bits/Character for 5 Characters and Less

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	NUMBER OF BITS/CHARACTER
1	1	1	1	0	0	0	D_0	1
1	1	1	0	0	0	D_1	D_0	2
1	1	0	0	0	D_2	D_1	D_0	3
1	0	0	0	D_3	D_2	D_1	D_0	4
0	0	0	D_4	D_3	D_2	D_1	D_0	5

DTR (Data Terminal Ready) (D_7)

When this bit is 1, the DTR output is low (active). Conversely, when this bit is 0, DTR is high.

G-5.1.8 Control Register 6

Figure G-5.8 Control Register 6

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SYNC BYTE 1							

Sync Byte 1 (D_0-D_7)

Sync byte 1 is used in the following modes:

Monosync:	8-bit sync character transmitted during the Idle Phase
Bisync:	Least significant (first) 8 bits of the 16-bit transmit and receive sync character
External Sync:	Sync character transmitted during the Idle Phase
SDLC:	Secondary address value matched to Secondary Address field of the SDLC frame when the MPSC² is in Address Search Mode

G-5.1.9 Control Register 7

Figure G-5.9 Control Register 7

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SYNC BYTE 2							

Sync Byte 2 (D_0-D_7)

Sync Byte 2 is used in the following modes:

Monosync:	8-bit sync character matched by the Receiver
Bisync:	Most significant (second) 8 bits of the 16-bit transmit and receive sync characters
SDLC:	You must program the flag character, 01111110, into control register 7 for flag matching by the MPSC² receiver

G-5.1.10 Status Register 0

Figure G-5.10 Status Register 0

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Break/ Abort	Idle/CRC	$\overline{CTS}$	Sync Status	$\overline{DCD}$	Transmitter Buffer Empty	Interrupt Pending	Received Character Available

Received Character Available (D_0)

When this bit is set, it indicates that one or more characters are available in the receiver buffer for the processor to read. Once all of the available characters have been read, the MPSC² resets this bit until a new character is received.

Interrupt Pending (D_1-Channel A Only)

The interrupt pending bit is used with the interrupt vector register (status register 2) to make it easier to determine the MPSC²'s interrupt status, particularly in Non-vectored Interrupt mode where the processor must poll each device to determine the interrupt source. In this mode, interrupt pending is set when you read status register 2B, the PRI input is active (low) and the MPSC² is requesting interrupt service.

You need not analyze the status registers of both channels to determine if an interrupt is pending. If status affects vector is enabled and interrupt pending is set, the vector you read from SR2 contains valid condition information.

In Vectored Interrupt mode, interrupt pending is set during the interrupt acknowledge cycle (on the leading edge of the 2nd INTA pulse) when the MPSC² is the highest priority device requesting interrupt service (PRI is active). In either mode, if there are no other pending interrupt requests, interrupt pending is reset when the End of Interrupt command is issued.

Transmitter Buffer Empty (D_2)

This bit is set whenever the transmitter buffer is empty, except during the transmission of CRC (the MPSC² uses the buffer to facilitate this function). After a reset, the buffer is considered empty and transmit buffer empty is set.

External/Status Flags

The following status bits reflect the state of the various conditions that cause an external/status interrupt. The MPSC² latches all external/status bits whenever a change occurs that would cause an external/status interrupt (regardless of whether this interrupt is enabled). This allows you to capture transient status changes on these lines with relaxed software timing requirements (see Appendix A for detailed timing specifications).

When you operate the $MPSC^2$ in interrupt-driven mode for external/status interrupts, you should read status register 0 when this interrupt occurs and issue a Reset External/Status Interrupt command to reenable the interrupt and the latches. To poll these bits without interrupts, you can issue the Reset External/Status Interrupt command to first update the status to reflect the current values.

DCD (D_3)

This bit reflects the inverted state of the DCD input. When DCD is low, the DCD status bit is high. Any transition on this bit causes an External/Status Interrupt request.

Sync Status (D^4)

The meaning of this bit depends on the operating mode of the $MPSC_2$.

Asynchronous mode: Sync status reflects the inverted state of the SYNC input. When SYNC is low, sync status is high. Any transition on this bit causes an External/Status Interrupt request.

External Synchronization mode: sync status operates in the same manner as asynchronous mode. The $MPSC^2$s receiver synchronization logic is also tied to the sync status bit in external synchronization mode and a low-to-high transition (SYNC input going low) informs the receiver that synchronization has been achieved and character assembly begins (see Appendix A for detailed timing information).

A low-to-high transition on the SYNC input indicates that synchronization has been lost and is reflected both in sync status becoming zero and the generation of an External/Status interrupt. The receiver remains in Receive Data Phase until you set the Enter Hunt Phase bit in Control Register 3.

Monosync, Bisync, SDLC modes: In these modes, sync status indicates whether the $MPSC^2$ receiver is in the Sync Hunt or Receive Data Phase of operation. A 0 indicates that the $MPSC^2$ is in the Receive Data Phase and a one indicates that the $MPSC^2$ is in the Sync Hunt Phase, as after a reset or setting the Enter Sync Hunt Phase bit. As in the other modes, a transition on this bit causes an External/Status interrupt to be issued. You should note that entering Sync Hunt Phase after either a reset or when programmed causes an External/Status Interrupt request which you may clear immediately with a Reset External/Status Interrupt command.

CTS (D_5)

This bit reflects the inverted state of the CTS input. When CTS is low, the CTS status bit is high. Any transition on this bit causes an External/Status Interrupt request.

Idle/CRC (D_6)

This bit indicates the state of the Idle/CRC latch used in synchronous and SDLC modes. After reset this bit is 1, indicating that when the transmitter is completely empty, the MPSC² enters Idle Phase and automatically transmits sync or flag characters.

A zero indicates that the latch has been reset by the Reset Idle/CRC Latch command. When the transmitter is completely empty, the MPSC² sends the 16-bit CRC character and sets the latch again. An External/Status interrupt is issued when the latch is set, indicating that CRC is being sent. No interrupt is issued when the latch is reset.

Break/Abort (D_7)

In asynchronous mode, this bit indicates the detection of a break sequence (a null character plus framing error, that occurs when the RxD input is held low (spacing) for more than 1 character time). Break/Abort is reset when RxD returns high (marking).

In SDLC mode, Break/Abort indicates the detection of an abort sequence when 7 or more ones are received in sequence. It is reset when a zero is received.

Any transition of the Break/Abort bit causes an External/Status Interrupt.

G-5.1.11 Status Register 1

Figure G-5.11 Status Register 1

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
End of SDLC Frame	CRC Framing Error	Overrun Error	Parity Error	SDLC Residue Code			All Sent

All Sent (D_0)

In asynchronous mode, this bit is set when the transmitter is empty and reset when a character is present in the transmitter buffer or shift register. This feature simplifies your modem control software routines. In synchronous and SDLC modes, this bit is always set to 1.

SDLC Residue Code (D_1-D_3)

Since the data portion of an SDLC message can consist of any number of bits and not necessarily an integral number of characters, the MPSC² features special logic to determine and report when the End of Frame flag has been received, the boundary between the data field, and the CRC character in the last few data characters that were just read.

When the end of frame condition is indicated, that is, status register 1 D_7 = 1 and Special Receive Condition interrupt (if enabled), the last bits of the CRC character are in the receiver buffer. The residue code for the frame is valid in the status register 1 byte associated with that data character (remember SR1 tracks the received data in its own buffer).

The meaning of the residue code depends upon the number of bits/characters specified for the receiver. The previous character refers to the last character read before the End of Frame, etc.

Table G-5.12 Residue Codes

8 Bits/Character

D_3	D_2	D_1	Previous Character	2nd Previous Character
1	0	0	C C C C C C C C	C C C C C D D D
0	1	0	C C C C C C C C	C C C C D D D D
1	1	0	C C C C C C C C	C C C D D D D D
0	0	1	C C C C C C C C	C C D D D D D D
1	0	1	C C C C C C C C	C D D D D D D D
0	1	1	C C C C C C C C	D D D D D D D D (no residue)
1	1	1	C C C C C C C D	D D D D D D D D
0	0	0	C C C C C C D D	D D D D D D D D

7 Bits/Character

D_3	D_2	D_1	Previous Character	2nd Previous Character
1	0	0	C C C C C C C	C C C C C D D
0	1	0	C C C C C C C	C C C C D D D
1	1	0	C C C C C C C	C C C D D D D
0	0	1	C C C C C C C	C C D D D D D
1	0	1	C C C C C C C	C D D D D D D
0	1	1	C C C C C C C	D D D D D D D (no residue)
0	0	0	C C C C C C D	D D D D D D D

6 Bits/Character

D_3	D_2	D_1	Previous Character	2nd Previous Character
1	0	0	C C C C C C	C C C C C D
0	1	0	C C C C C C	C C C C D D
1	1	0	C C C C C C	C C C D D D
0	0	1	C C C C C C	C C D D D D
1	0	1	C C C C C C	C D D D D D
0	0	0	C C C C C C	D D D D D D (no residue)

5 Bits/Character

D_3	D_2	D_1	2nd Previous Character	3rd Previous Character
1	0	0	C C C C C	D D D D D (no residue)
0	1	0	C C C C D	D D D D D
1	1	0	C C C D D	D D D D D
0	0	1	C C D D D	D D D D D
0	0	0	C D D D D	D D D D D

Special Receive Condition Flags

The status bits described below (Parity error [if Parity is a Special Receive condition is enabled], Receiver Overrun Error, CRC/Framing Error, and End of SDLC Frame), all represent Special Receive conditions.

When any of these conditions occurs and interrupts are enabled, the MPSC^2 issues an interrupt request. In addition, if you enabled Condition Affects Vector mode, the vector generated (and the contents of SR2B for non-vectored interrupts) is different from that of a Received Character Available condition. Thus, you need not analyze SR1 with each character to determine that an error has occurred.

As a further convenience, the Parity Error and Receiver Overrun Error flags are latched, that is, once one of these errors occurs, the flag remains set for all subsequent characters until reset by the Error Reset command. With this facility, you need only read SR1 at the end of a message to determine if either of these errors occurred anywhere in the message. The other flags are not latched and follow each character available in the receiver buffer.

Parity Error (D_4)

This bit is set and latched when parity is enabled and the received parity bit does not match the sense (odd or even) calculated from the data bits.

Receiver Overrun Error (D_5)

This error occurs and is latched when the receiver buffer already contains three characters and a fourth character is completely received, overwriting the last character in the buffer.

CRC/Framing Error (D_6)

In asynchronous mode, a framing error is flagged (but not latched) when no stop bit is detected at the end of a character (i.e. RxD is low 1 bit time after the center of the last data or parity bit). When this condition occurs, the MPSC² waits an additional ½ bit time before sampling again so that the framing error is not interpreted as a new start bit.

In synchronous and SDLC modes, this bit indicates the result of the comparison between the current CRC result and the appropriate check value and is usually set to 1 since a message rarely indicates a correct CRC result until correctly completed with the CRC check character. Note that a CRC error does not result in a Special Receive Condition interrupt.

End of SDLC Frame (D_7)

This flag is used only in SDLC mode to indicate that the End of Frame flag has been received and that the CRC error flag and residue code is valid. You can reset this flag at any time by issuing an Error Reset command. The MPSC² also automatically resets this bit for you on the first character of the next message frame.

G-5.1.12 Status Register 2

Figure G-5.12 Status Register 2

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Interrupt Vector							

Interrupt Vector (D_0-D_7 - Channel B Only)

Reading status register 2B returns the interrupt vector that you programmed into control register 2B. If Condition Affects Vector mode is enabled, the value of the vector is modified as follows:

Table G-5.13 Condition Affects Vector Modifications

8085 Modes 8086 Modes	D_4 D_2	D_3 D_1	D_2 D_0	CONDITION
	1	1	1	No Interrupt Pending
	0	0	0	Channel B Transmitter Buffer Empty
	0	0	1	Channel B External/Status Change
	0	1	0	Channel B Received Character Available
	0	1	1	Channel B Special Receive Condition
	1	0	0	Channel A Transmitter Buffer Empty
	1	0	1	Channel A External/Status Change
	1	1	0	Channel A Received Character Available
	1	1	1	Channel A Special Receive Condition

As you can see, code 111 can mean either channel A Special Receive condition or no interrupt pending. You can easily distinguish between the two by examining the Interrupt Pending bit (D_1) of status register 0, channel A. Remember, in Non-vectored Interrupt mode you must read the vector register first for Interrupt Pending to be valid.

G-5.2 MPSC^2 PROGRAMMING EXAMPLES

```
ASYNC.01
            ********** Asynchronous Mode ***************
Init:
    ISSUE Channel Reset Command (CR0)
    SET Bus Interface Options (CR2A)
    SET Interrupt Vector (CR2B)-if used
    SET Operating Mode (CR4):
      Asynchronous Mode, Parity Select, # of Stop Bits, Clock
      Rate
    SET Receive Enable, Auto Enables, Receive Character Length
    (CR2)
    SET Transmit Enable, Modem Controls, Transmit Char,
    Length (CR5)
    ISSUE Reset External/Status Interrupt Command
    SET Transmit Interrupt Enable, Receive Interrupt on Every
      Character, External Interrupt Enable, Wait Mode Disable.
                  **** End Of Initialization ******

Send:
    ISSUE First Byte To MPSC
    RETURN To Main Program OR Halt

Interrupt:
  CASE Interrupt Type DO:
```

```
Character Received:
    READ Character from MPSC
    PROCESS Character
    ISSUE End Of Interrupt Command
    RETURN From Interrupt

Special Receive Condition:
    READ SR1
    ISSUE Error Reset Command
    CALL Special Error Routine
    ISSUE End Of Interrupt Command
    RETURN From Interrupt

Transmitter Buffer Empty:
    IF Last Character Transferred was End of Message
      THEN ISSUE Reset Transmit Interrupt/DMA Pending
      Command
    ELSE
      Transfer Next Character to MPSC
      ISSUE End Of Interrupt Command
      RETURN From Interrupt

External/Status Change:
    READ SR1
    CALL Special Condition Routine
    ISSUE End Of Interrupt Command
    RETURN From Interrupt

**** END CASE ****

Terminate Transmit:
    RESET Transmit Enable, RTS (CR5)
    RETURN

Terminate Receive:
    RESET Receive Enable (CR1)
    RESET DTR (CR5)

ASYNC.01

    RETURN

END
```

Figure G-5.13 Asynchronous Initialization for Polled Transmit and Receive Receive

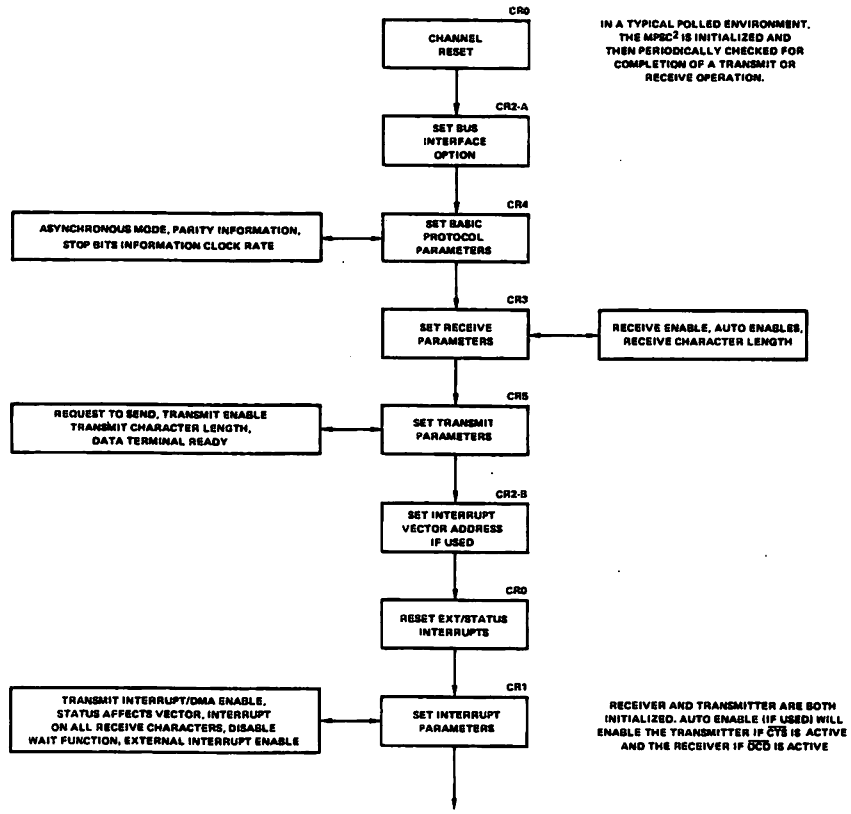


Figure G-5.14 Asynchronous Receive

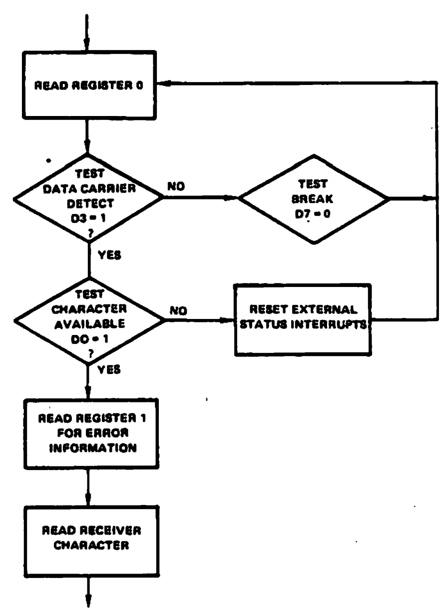
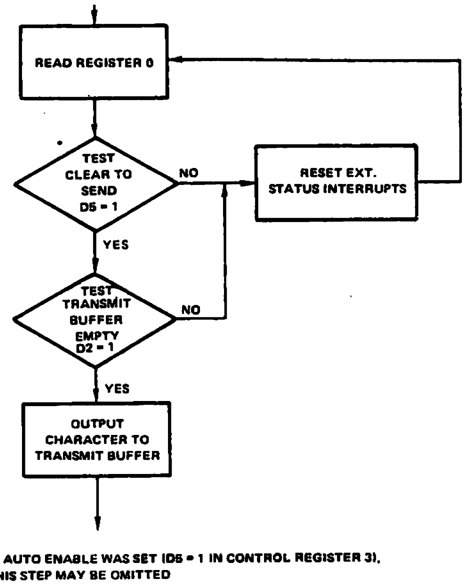


Figure G-5.15 Asynchronous Transmit



SYNC. PRG
***********SYNCHRONOUS OPERATION EXAMPLE***********
****This example uses the Block Transfer Mode***

Init:
    ISSUE Channel Reset Command
    SET Interface Option (CR2A)
    SET Interrupt Vector (CR2B)
    SET Parity Mode, Sync Mode, 1x Clock (CR4)
    SET Sync Character 1 (CR6)
    SET Sync Character 2 (CR7)
    RETURN

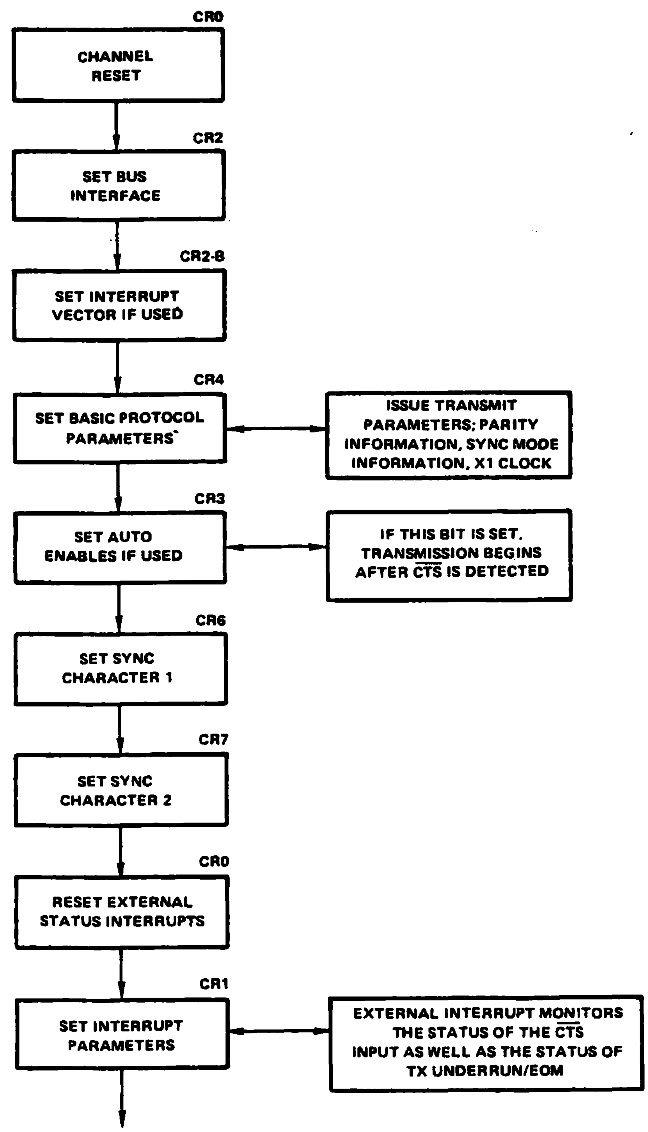
Initiate Transmit:
    ISSUE Reset External/Status Interrupt Command
    SET External Interrupt Enable, Transmit Interrupt Enable
      Wait Enable, Wait on Transmit (CR1)
    SET Transmit Enable, # of Bits/Character, RTS,
      CRC Polynomial Select.

****Transmitter is now enabled and will automatically begin sending Sync characters****

WAIT Several Character Times (a good idea to help system gain synchronization)
Next Message:
    ISSUE Reset Transmit CRC Command

```
Send Character:
    GET Character
    If Character Is To Be Included In CRC
    THEN
      SET CRC Generator On (CR5)
    ELSE
      SET CRC Generator Off (CR5)
    ENDIF
    WRITE Character To MPSC (Processor will "Wait" until
      Transmitter buffer is empty)
    IF Character Was Not The Last
    THEN
      GOTO Send Character (do next character)
    ELSE
      SET CRC Generator On (CR5)
      ISSUE Reset Idle/CRC Latch Command
      WAIT For External/Status Interrupt Indicating CRC Being
      Sent
      IF Next Message Is Ready To Be Transmitted
      THEN
        GOTO Next Message (Next message will be sent
        immediately following CRC)
    ELSE
      WAIT For Transmit Buffer Interrupt indicating Trailing
        Sync Being Sent
      SET Transmitter Enable Off, RTS Off (CR5)
    ENDIF
  ENDIF
  ****End of Transmit Routine****

SYNC.PRG

****Receive Routine****

Receive Message:
    SET External/Status Interrupt Enable, Receive Interrupt
      On First Character Mode, Wait Enabled, Wait on
      Receive (CR1)
    SET Receiver Enable On, Sync Character Load Inhibit,
      # of Bits/Character (CR1)
    SET DTR On (CR5)
    ISSUE Reset External Status Interrupt Command
    ISSUE Enable Interrupt On Next Received Character
      Command
    ISSUE Error Reset Command

    ****Receiver is now enabled and in the Hunt Phase****
    WAIT For External/Status Interrupt (indicating
      synchronization has been achieved)
    Issue Error Reset Command
    WAIT For Received Character Available Interrupt (first
      non-sync character is now available)
    ISSUE Reset CRC Checker Command
    SET Sync Character Load Inhibit Off
```

```
Get Character:
    GET Character from MPSC (processor will "Wait" until at
      least 1 character is available)

    IF Character Is To Be Included In CRC Calculation
    THEN
      Turn CRC Checker On (CR3)
    ELSE
      SET CRC Checker Off (CR3)
    ENDIF
    IF Character Is Part of Message Data
    THEN
      SAVE Character In Memory
    ENDIF
    IF Character Was NOT End Of Message
    THEN
      GOTO READ Character
    ENDIF

*** End Of Message***
    SET CRC Checker On
    READ 2 CRC Characters
    READ 2 Character (these characters may be part of the
      next message but must be read before CRC will be valid)
    READ SR1 (this must be done immediately so that next
      character status will not overwrite)
    IF Parity OR Overrun OR CRC = Error
    THEN
      GOTO Error Processor
    ENDIF

    IF More Messages Are To Be Received
    THEN
      GOTO Get Next Message

SYNC.PRG

    ELSE
      SET DTR Off
      SET Receive Enable Off
    SET External/Status Interrupts Off, Receiver Interrupt
    Mode Disabled (CR1)
    RETURN

    END
      RETURN
```



CR0
CHANNEL RESET
CR2
SET BUS INTERFACE
CR2-B
SET INTERRUPT VECTOR IF USED
CR4
SET BASIC PROTOCOL PARAMETERS
ISSUE TRANSMIT PARAMETERS; PARITY INFORMATION, SYNC MODE INFORMATION, X1 CLOCK
CR3
SET AUTO ENABLES IF USED
IF THIS BIT IS SET, TRANSMISSION BEGINS AFTER $\overline{CTS}$ IS DETECTED
CR6
SET SYNC CHARACTER 1
CR7
SET SYNC CHARACTER 2
CR0
RESET EXTERNAL STATUS INTERRUPTS
CR1
SET INTERRUPT PARAMETERS
EXTERNAL INTERRUPT MONITORS THE STATUS OF THE $\overline{CTS}$ INPUT AS WELL AS THE STATUS OF TX UNDERRUN/EOM

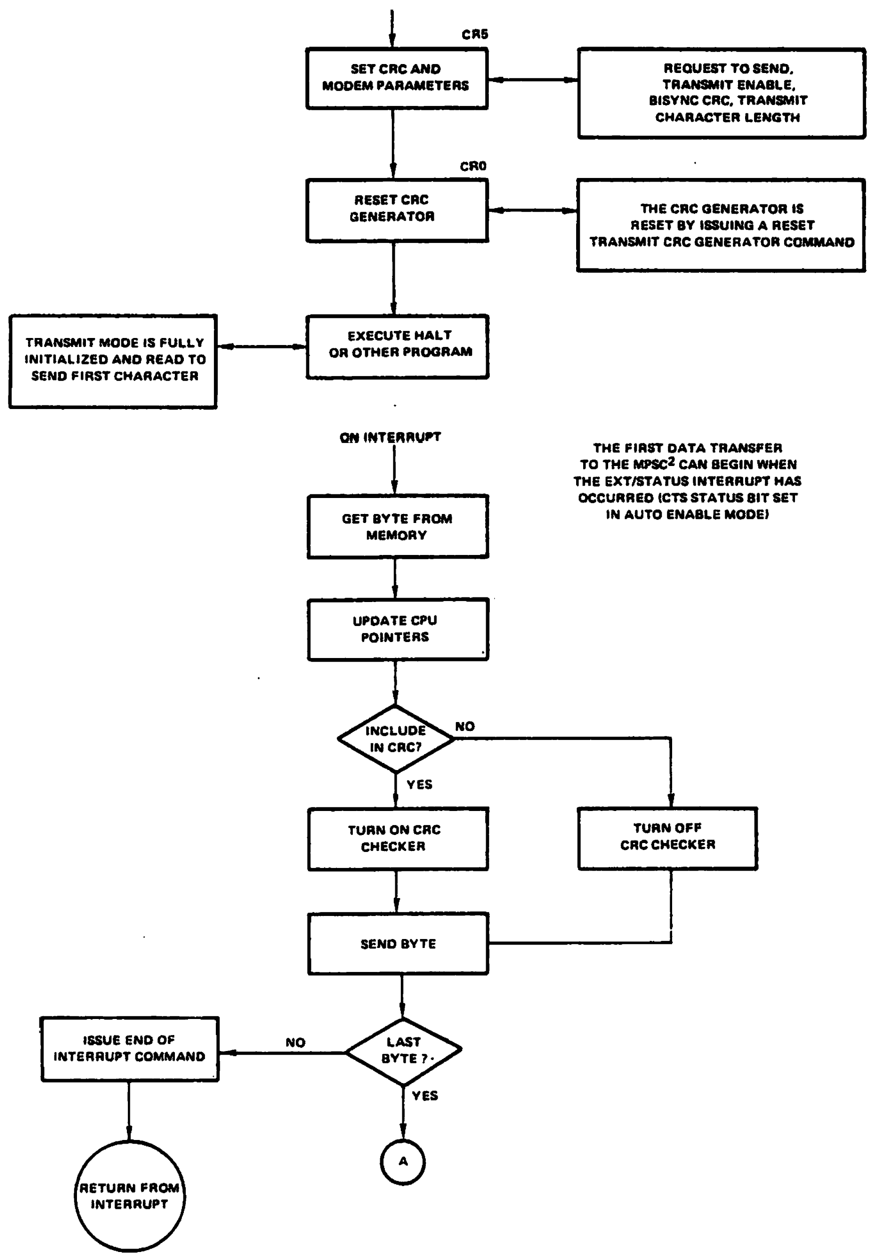
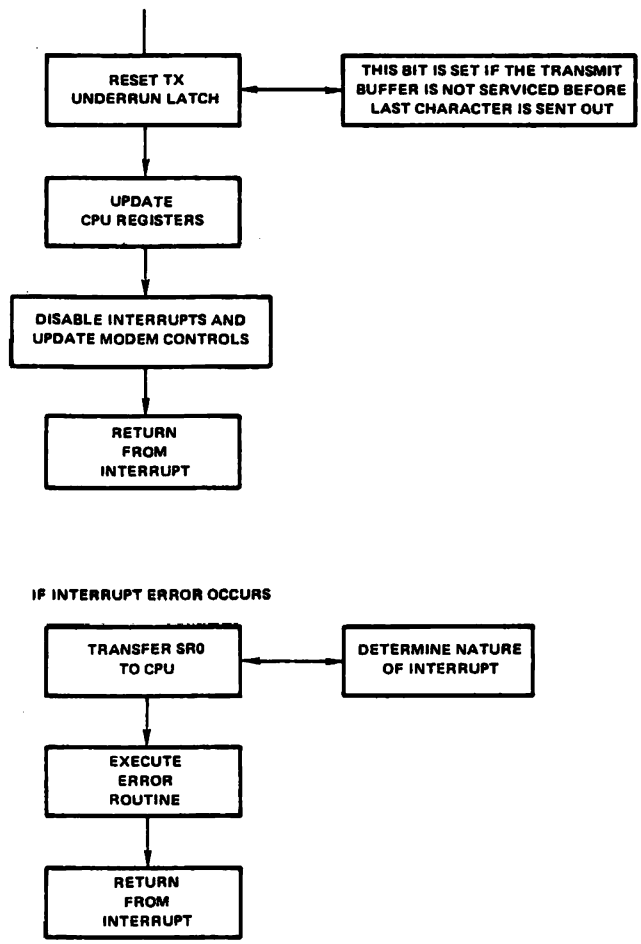
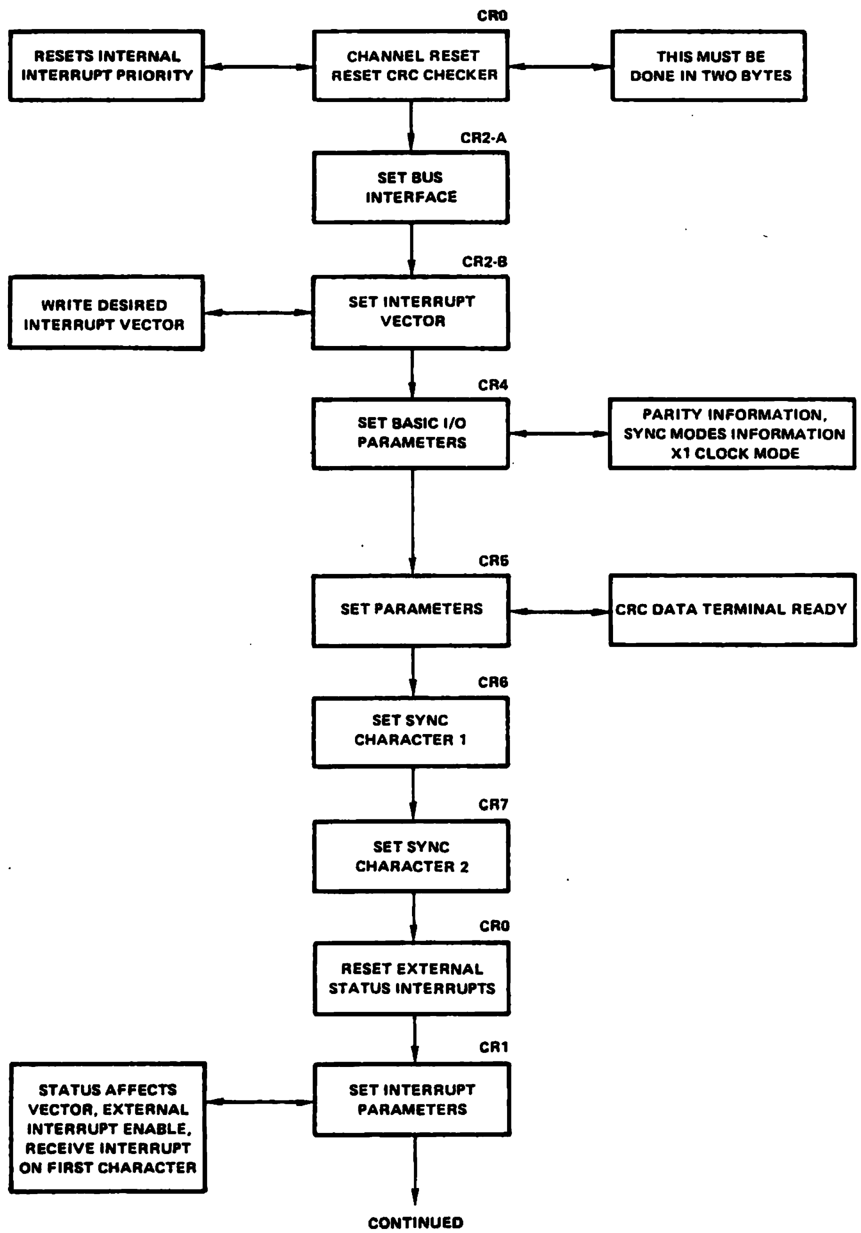

CR5
SET CRC AND MODEM PARAMETERS
REQUEST TO SEND, TRANSMIT ENABLE, BISYNC CRC, TRANSMIT CHARACTER LENGTH
CR0
RESET CRC GENERATOR
THE CRC GENERATOR IS RESET BY ISSUING A RESET TRANSMIT CRC GENERATOR COMMAND
TRANSMIT MODE IS FULLY INITIALIZED AND READ TO SEND FIRST CHARACTER
EXECUTE HALT OR OTHER PROGRAM
ON INTERRUPT
THE FIRST DATA TRANSFER TO THE MPSC2 CAN BEGIN WHEN THE EXT/STATUS INTERRUPT HAS OCCURRED (CTS STATUS BIT SET IN AUTO ENABLE MODE)
GET BYTE FROM MEMORY
UPDATE CPU POINTERS
INCLUDE IN CRC?
NO
YES
TURN ON CRC CHECKER
TURN OFF CRC CHECKER
SEND BYTE
ISSUE END OF INTERRUPT COMMAND
NO
LAST BYTE ?
YES
A
RETURN FROM INTERRUPT

Figure G-5.16 Bisync Initialization Transmit




CR0
RESETS INTERNAL INTERRUPT PRIORITY
CHANNEL RESET RESET CRC CHECKER
THIS MUST BE DONE IN TWO BYTES
CR2-A
SET BUS INTERFACE
CR2-B
WRITE DESIRED INTERRUPT VECTOR
SET INTERRUPT VECTOR
CR4
SET BASIC I/O PARAMETERS
PARITY INFORMATION, SYNC MODES INFORMATION X1 CLOCK MODE
CR5
SET PARAMETERS
CRC DATA TERMINAL READY
CR6
SET SYNC CHARACTER 1
CR7
SET SYNC CHARACTER 2
CR0
RESET EXTERNAL STATUS INTERRUPTS
CR1
STATUS AFFECTS VECTOR, EXTERNAL INTERRUPT ENABLE, RECEIVE INTERRUPT ON FIRST CHARACTER
SET INTERRUPT PARAMETERS
CONTINUED

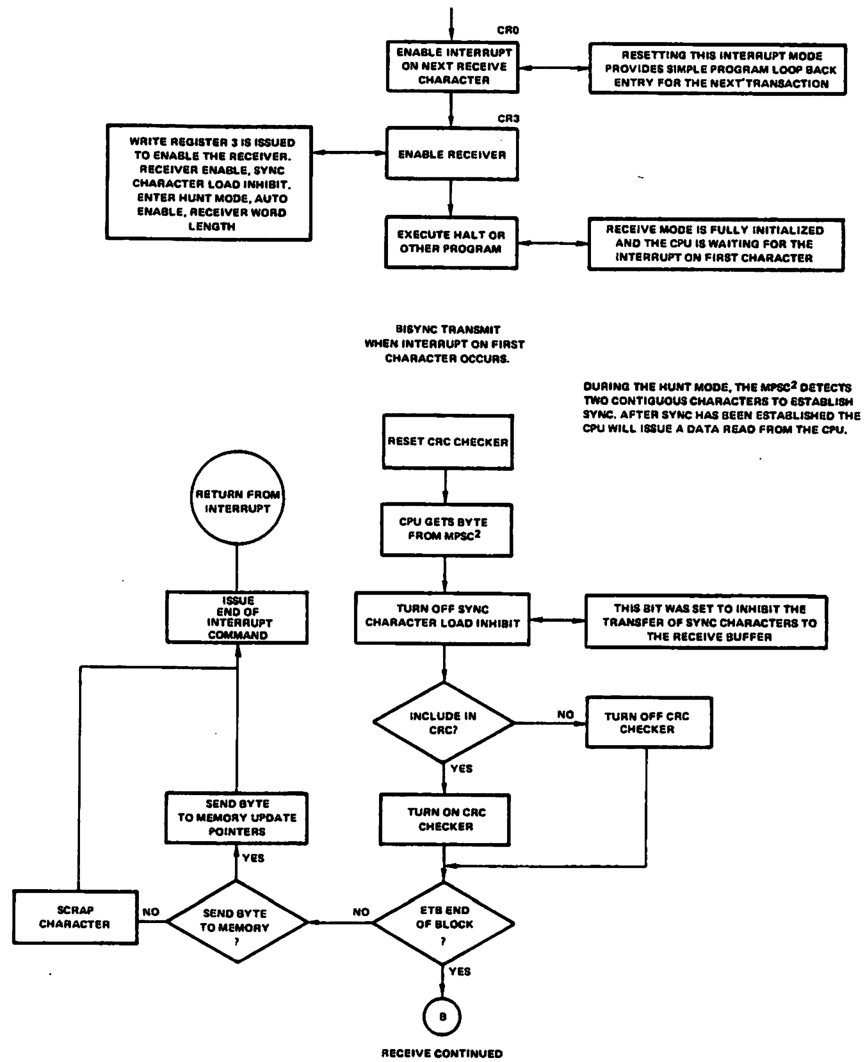
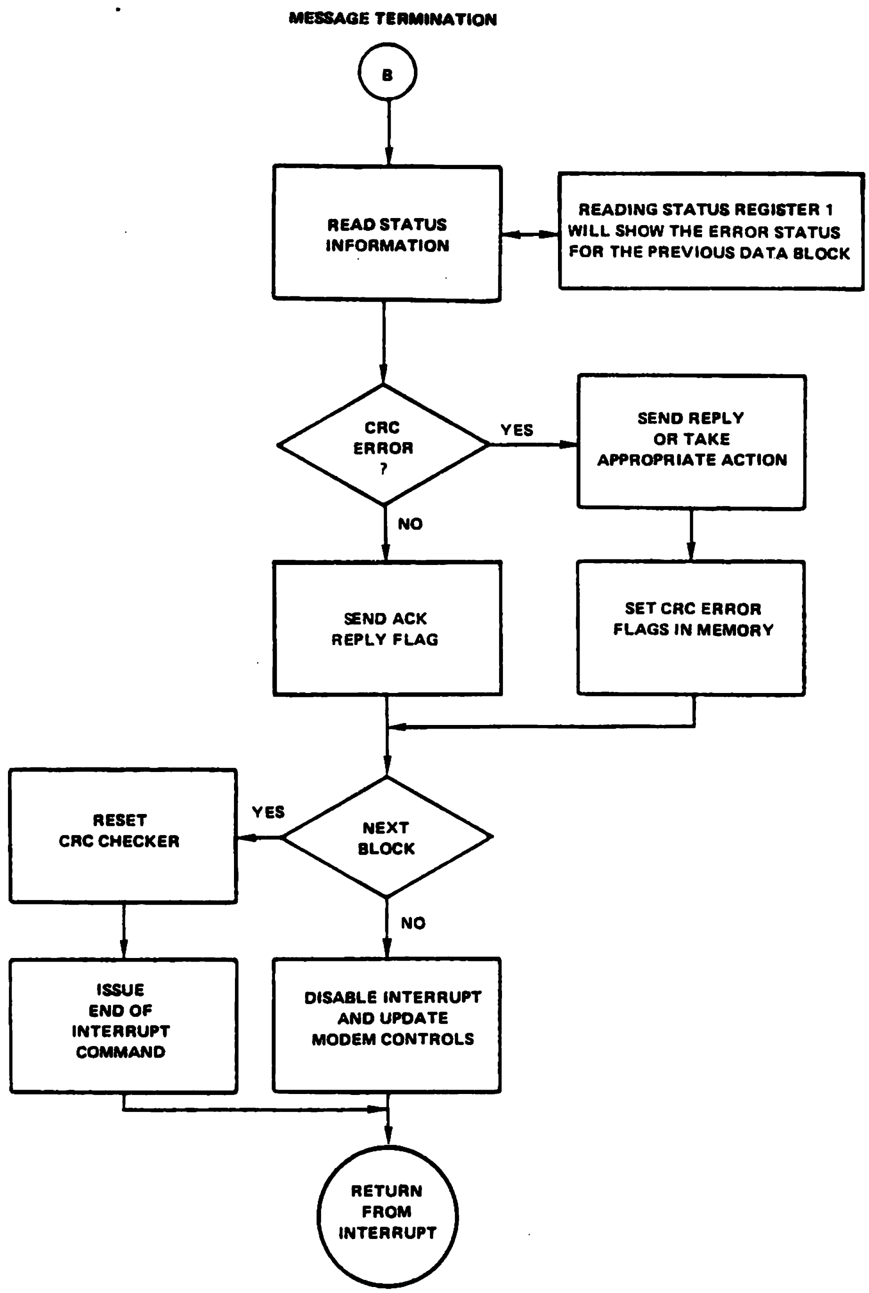

CR0
ENABLE INTERRUPT ON NEXT RECEIVE CHARACTER
RESETTING THIS INTERRUPT MODE PROVIDES SIMPLE PROGRAM LOOP BACK ENTRY FOR THE NEXT TRANSACTION
CR3
WRITE REGISTER 3 IS ISSUED TO ENABLE THE RECEIVER. RECEIVER ENABLE, SYNC CHARACTER LOAD INHIBIT. ENTER HUNT MODE, AUTO ENABLE, RECEIVER WORD LENGTH
ENABLE RECEIVER
EXECUTE HALT OR OTHER PROGRAM
RECEIVE MODE IS FULLY INITIALIZED AND THE CPU IS WAITING FOR THE INTERRUPT ON FIRST CHARACTER
BISYNC TRANSMIT WHEN INTERRUPT ON FIRST CHARACTER OCCURS.
DURING THE HUNT MODE, THE MPSC² DETECTS TWO CONTIGUOUS CHARACTERS TO ESTABLISH SYNC. AFTER SYNC HAS BEEN ESTABLISHED THE CPU WILL ISSUE A DATA READ FROM THE CPU.
RESET CRC CHECKER
RETURN FROM INTERRUPT
CPU GETS BYTE FROM MPSC²
ISSUE END OF INTERRUPT COMMAND
TURN OFF SYNC CHARACTER LOAD INHIBIT
THIS BIT WAS SET TO INHIBIT THE TRANSFER OF SYNC CHARACTERS TO THE RECEIVE BUFFER
INCLUDE IN CRC?
NO
TURN OFF CRC CHECKER
YES
SEND BYTE TO MEMORY UPDATE POINTERS
TURN ON CRC CHECKER
YES
SCRAP CHARACTER
NO
SEND BYTE TO MEMORY ?
NO
ETB END OF BLOCK ?
YES
B
RECEIVE CONTINUED

Figure G-5.17 Bisync Initialization Receive



*******************SDLC OPERATION EXAMPLE*******************

****This example uses DMA Transfer Mode****

Initialize:
ISSUE Channel Reset Command
SET Interface Option (CR2A)
SET Interrupt Vector (CR2B)
SET SDLC Mode, 1x Clock (CR4)
SET SDLC Flag (CR7)= 01111110
SET SDLC Secondary Address (CR6)
RETURN

Initiate Transmit:
ISSUE Reset External Status Interrupt Command
SET External Interrupt Enable, Transmit Interrupt/DMA Enable (CR1)
SET Transmit Enable, RTS, CRC-CCITT Polynomial (CR5)

****The Transmitter is now enabled and will automatically begin sending Flag characters****

Send Message:
SET DMA Controller to Beginning of Message, # of Characters in Message.
ISSUE Reset Transmit CRC Generator Command
SET 8 Bits/Character (CR5)
WRITE Address byte to MPSC
SET # of Bits/Character (CR5)
ISSUE Reset EOM/CRC Latch Command

****The MPSC will now transmit the message until the DMA Controller completes the required number of transfers****

WAIT for External/Status Change Interrupt (signifies CRC being sent)
IF Next Message Ready to be Transmitted
THEN
  GOTO Send Message (since MPSC will automatically issue a DMA request when ready, set DMA controller to address byte preceding message and skip the write)
ELSE
  ISSUE RESET External/Status Interrupt Command
  ISSUE RESET Transmit Interrupt/DMA Pending Command
  RETURN

****End of Transmit Routine****

Receive Message:
SET External/Status Interrupt Enable, Receive Interrupt on First Character (CR1)
SET Receiver Enable On, 8 Bits/Character, Receive CRC On, Address Search Mode On (CR3)
SET DTR On, CRC-CCITT (CR5)
ISSUE Reset External/Status Interrupt Command
ISSUE Enable Interrupt On Next Character Command

****Receiver is now enabled and in the Hunt Phase****

WAIT for External/Status Interrupt (indicating that a Flag character has been received)
ISSUE Reset External/Status Interrupt Command
RETURN From Interrupt

****Receiver is now in the Address Search Phase****

```
Next Message:
WAIT for Character Received Interrupt (indicating that an address
match or global address has occurred)
GET Address Character (for later processing)
SET DMA Controller
SET # of Bits/Character (CR3)
```

****Receiver is now in the Data Phase and will transfer all succeeding characters until the End of Frame Flag****

```
WAIT for Special Receive Condition Interrupt (indicating flag
received)
READ SR1 to Obtain CRC Status and Residue Code
SET DMA Controller Off
IF More Messages Are To Be Received
THEN
  GOTO Next Message
ELSE
  SET DTR Off
  SET Receive Enable Off
  RETURN
ENDIF
```

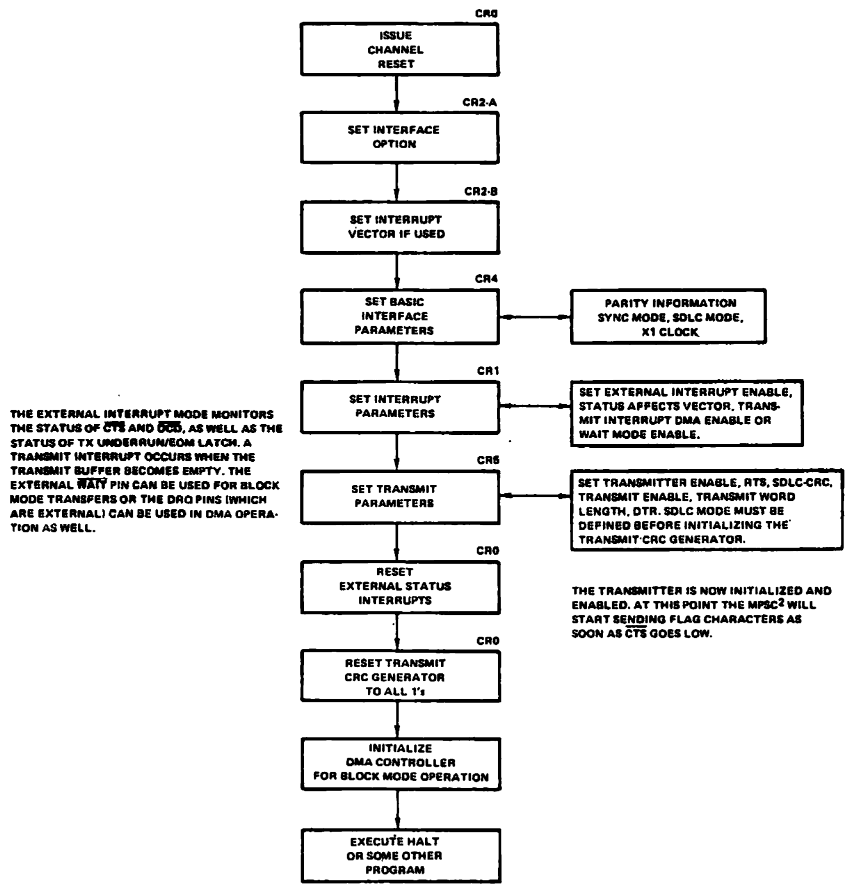
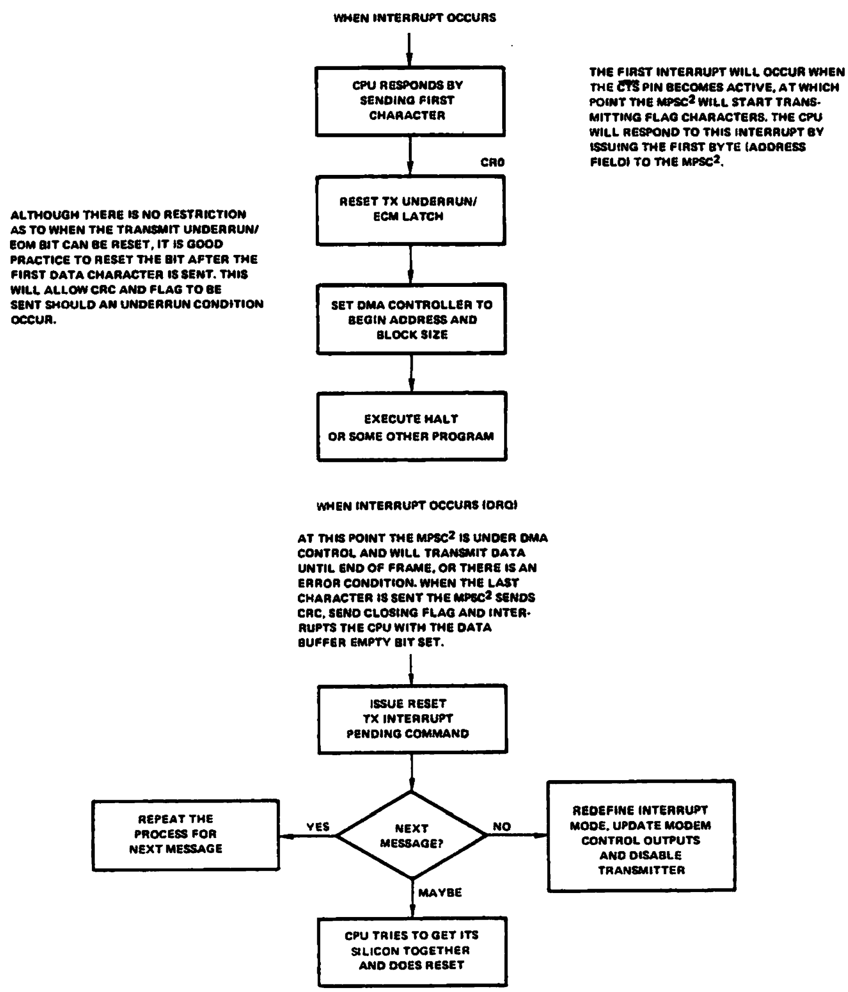
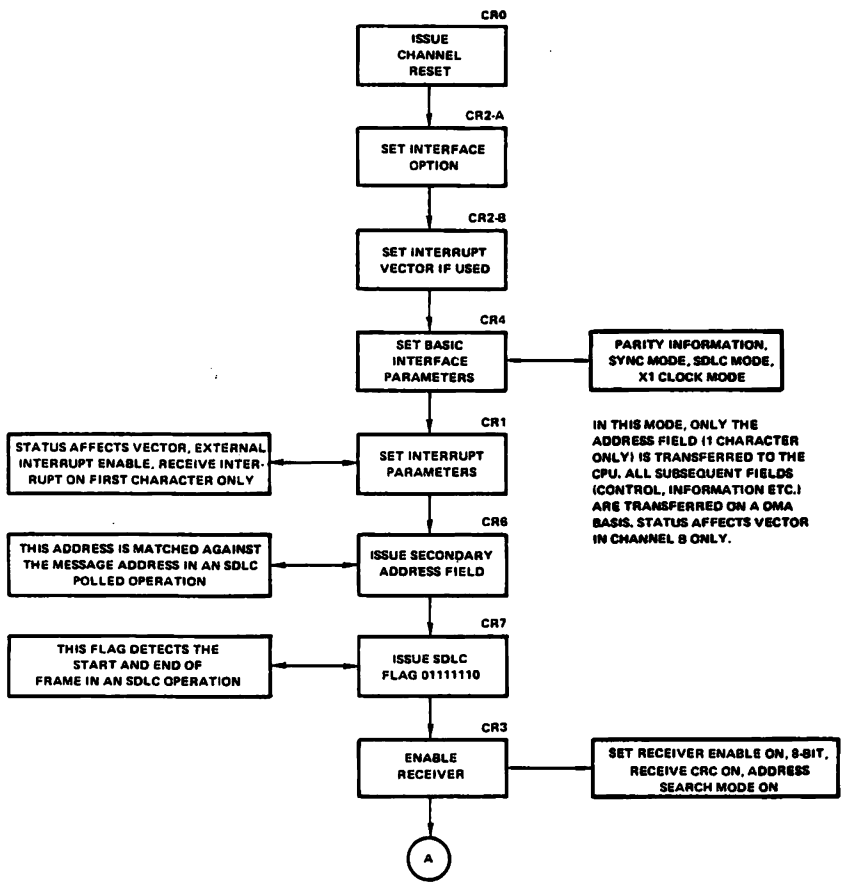
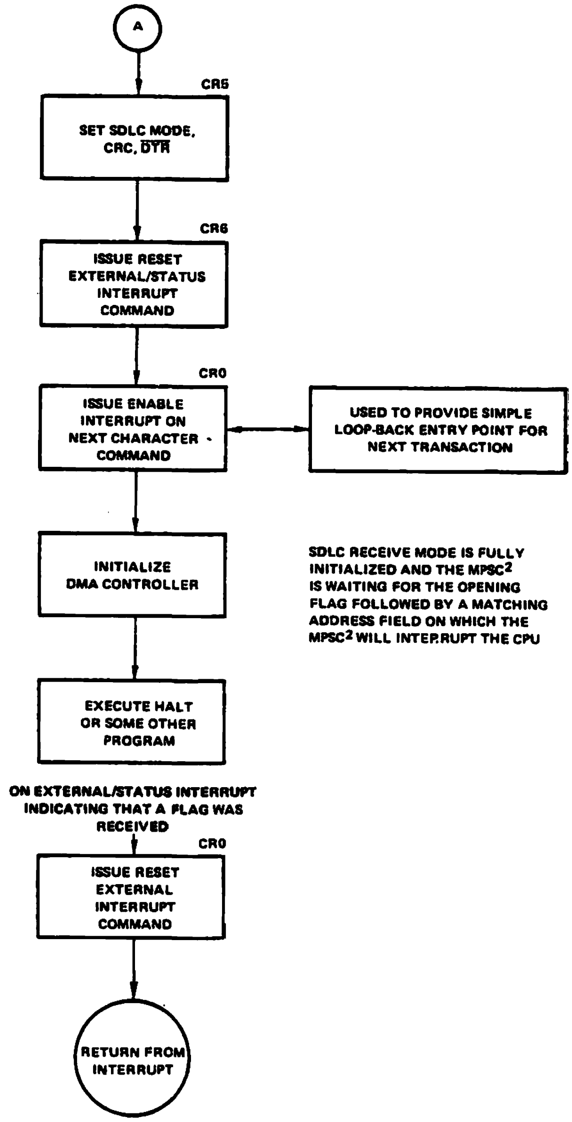

CR0
ISSUE CHANNEL RESET
CR2-A
SET INTERFACE OPTION
CR2-B
SET INTERRUPT VECTOR IF USED
CR4
SET BASIC INTERFACE PARAMETERS
PARITY INFORMATION SYNC MODE, SDLC MODE, X1 CLOCK
CR1
SET INTERRUPT PARAMETERS
SET EXTERNAL INTERRUPT ENABLE, STATUS AFFECTS VECTOR, TRANSMIT INTERRUPT DMA ENABLE OR WAIT MODE ENABLE.
THE EXTERNAL INTERRUPT MODE MONITORS THE STATUS OF CTS AND DCD, AS WELL AS THE STATUS OF TX UNDERRUN/EOM LATCH. A TRANSMIT INTERRUPT OCCURS WHEN THE TRANSMIT BUFFER BECOMES EMPTY. THE EXTERNAL WAIT PIN CAN BE USED FOR BLOCK MODE TRANSFERS OR THE DRQ PINS (WHICH ARE EXTERNAL) CAN BE USED IN DMA OPERATION AS WELL.
CR5
SET TRANSMIT PARAMETERS
SET TRANSMITTER ENABLE, RTS, SDLC-CRC, TRANSMIT ENABLE, TRANSMIT WORD LENGTH, DTR. SDLC MODE MUST BE DEFINED BEFORE INITIALIZING THE TRANSMIT-CRC GENERATOR.
CR0
RESET EXTERNAL STATUS INTERRUPTS
THE TRANSMITTER IS NOW INITIALIZED AND ENABLED. AT THIS POINT THE MPSC2 WILL START SENDING FLAG CHARACTERS AS SOON AS CTS GOES LOW.
CR0
RESET TRANSMIT CRC GENERATOR TO ALL 1's
INITIALIZE DMA CONTROLLER FOR BLOCK MODE OPERATION
EXECUTE HALT OR SOME OTHER PROGRAM

Figure G-5.18 SDLC Initialization Transmit




CR0
ISSUE CHANNEL RESET
CR2-A
SET INTERFACE OPTION
CR2-B
SET INTERRUPT VECTOR IF USED
CR4
SET BASIC INTERFACE PARAMETERS
PARITY INFORMATION, SYNC MODE, SDLC MODE, X1 CLOCK MODE
CR1
STATUS AFFECTS VECTOR, EXTERNAL INTERRUPT ENABLE, RECEIVE INTER-RUPT ON FIRST CHARACTER ONLY
SET INTERRUPT PARAMETERS
IN THIS MODE, ONLY THE ADDRESS FIELD (1 CHARACTER ONLY) IS TRANSFERRED TO THE CPU. ALL SUBSEQUENT FIELDS (CONTROL, INFORMATION ETC.) ARE TRANSFERRED ON A DMA BASIS. STATUS AFFECTS VECTOR IN CHANNEL B ONLY.
CR6
THIS ADDRESS IS MATCHED AGAINST THE MESSAGE ADDRESS IN AN SDLC POLLED OPERATION
ISSUE SECONDARY ADDRESS FIELD
CR7
THIS FLAG DETECTS THE START AND END OF FRAME IN AN SDLC OPERATION
ISSUE SDLC FLAG 01111110
CR3
ENABLE RECEIVER
SET RECEIVER ENABLE ON, 8-BIT, RECEIVE CRC ON, ADDRESS SEARCH MODE ON
A


A
CR5
SET SDLC MODE, CRC, $\overline{DTR}$
CR6
ISSUE RESET EXTERNAL/STATUS INTERRUPT COMMAND
CR0
ISSUE ENABLE INTERRUPT ON NEXT CHARACTER COMMAND
USED TO PROVIDE SIMPLE LOOP-BACK ENTRY POINT FOR NEXT TRANSACTION
INITIALIZE DMA CONTROLLER
SDLC RECEIVE MODE IS FULLY INITIALIZED AND THE MPSC2 IS WAITING FOR THE OPENING FLAG FOLLOWED BY A MATCHING ADDRESS FIELD ON WHICH THE MPSC2 WILL INTERRUPT THE CPU
EXECUTE HALT OR SOME OTHER PROGRAM
ON EXTERNAL/STATUS INTERRUPT INDICATING THAT A FLAG WAS RECEIVED
CR0
ISSUE RESET EXTERNAL INTERRUPT COMMAND
RETURN FROM INTERRUPT

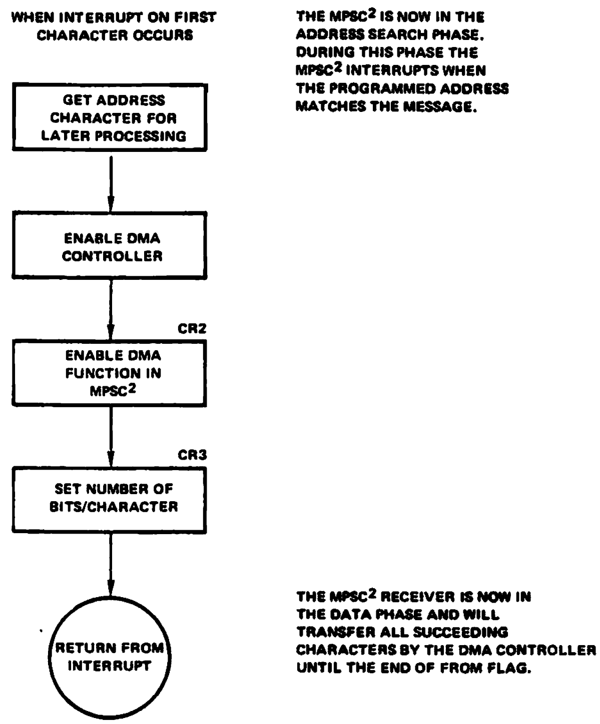
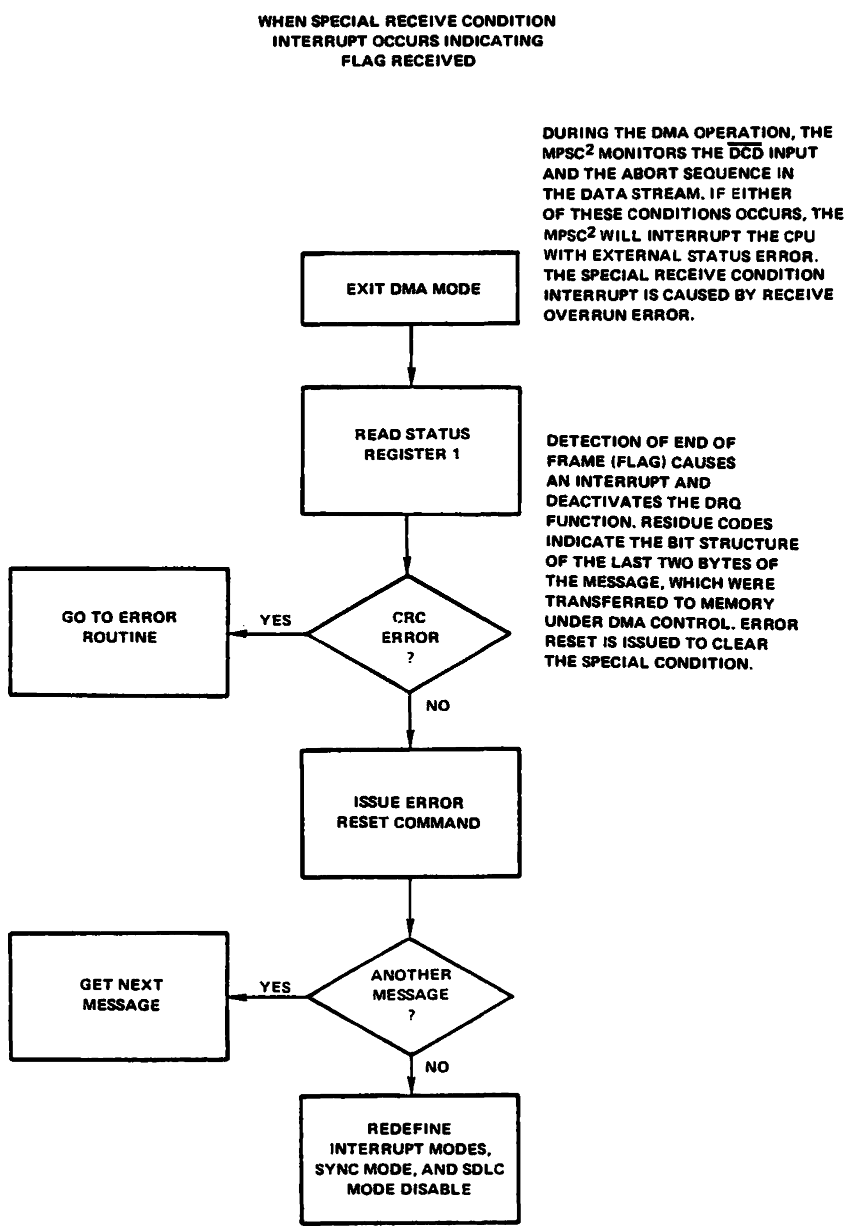

WHEN INTERRUPT ON FIRST CHARACTER OCCURS
THE MPSC2 IS NOW IN THE ADDRESS SEARCH PHASE. DURING THIS PHASE THE MPSC2 INTERRUPTS WHEN THE PROGRAMMED ADDRESS MATCHES THE MESSAGE.
GET ADDRESS CHARACTER FOR LATER PROCESSING
ENABLE DMA CONTROLLER
CR2
ENABLE DMA FUNCTION IN MPSC2
CR3
SET NUMBER OF BITS/CHARACTER
RETURN FROM INTERRUPT
THE MPSC2 RECEIVER IS NOW IN THE DATA PHASE AND WILL TRANSFER ALL SUCCEEDING CHARACTERS BY THE DMA CONTROLLER UNTIL THE END OF FROM FLAG.

Figure G-5.19 SDLC Initialization Receive



G-6 APPLICATION HINTS

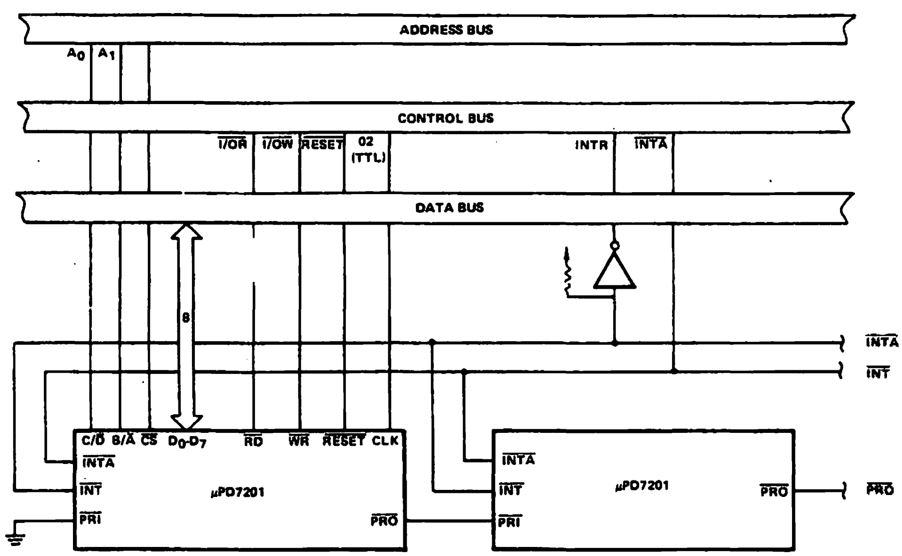
G-6.1 DESIGNING WITH THE MPSC²

Designing the MPSC² into your system is generally straightforward and requires a minimal number of external devices.

G-6.1.1 8080/86-Type Processors

The bus interface used by the MPSC² is directly compatible with 8080/86-type buses. Figure G-6.1 illustrates the basic interconnection scheme for these processors. This configuration supports polled, interrupt driven, and block mode operation.

Figure G-6.1 uPD7201 Interface to 8080 Standard System Bus (Non-DMA)



G-6.1.2 Other Processor Types

You may also connect the MPSC² to uPD780 (Z-80) and 6800/6502-type processors with a few additional gates. Figures G-6.2 and G-6.3, respectively, illustrate the circuits necessary to derive the correct signals. In both cases the MPSC² can be used in Non-vectored mode with minimal software overhead.

Figure G-6.2 uPD780 (Z-80) to MPSC² Adapter

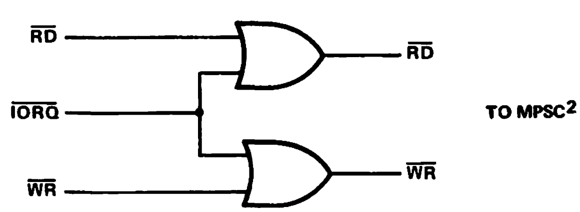
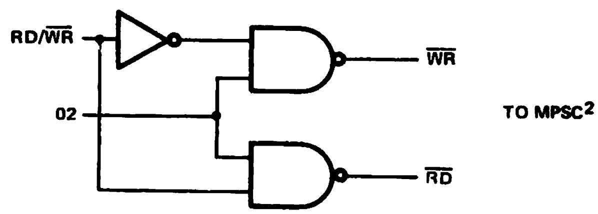
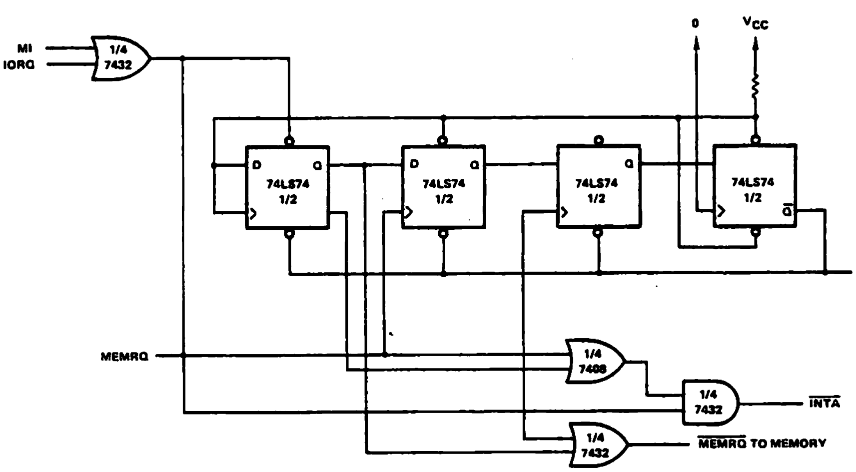


Figure G-6.3 6800/6502 to MPSC² Adapter



The MPSC² can also be used in Vectored Interrupt mode with the uPD780 operated in Interrupt Mode 0. In this mode, the uPD780 handles interrupt requests in much the same manner as an 8080 processor, that is, an interrupt acknowledge sequence is executed during which the processor expects the next instruction to come from the interrupting device. The 8080 INTA signal is generated by combining M1 and IORQ from the uPD780. There is one key difference that must be noted. In accepting a multibyte instruction such as the CALL generated by the MPSC², the 8080 issues a separate INTA pulse for each byte. The uPD780, however, issues an INTA on the first byte only. Succeeding bytes are accessed with memory read cycles. In order for the MPSC² to operate properly, a circuit such as the one shown in Figure G-6.4 should be used to derive the proper INTA sequence.

Figure G-6.4 INTA Generator for Z-80



Most other types of processors may be readily accommodated. The bus control inputs RD, WR, CS, C/D, B/A, and INTA have no timing requirements with respect to the system clock (CLK) and there is no hold time requirement for data after the trailing edge of WR. The only timing constraint you must observe is that the address lines C/D, B/A, and CS must be stable by the leading edge of RD or WR.

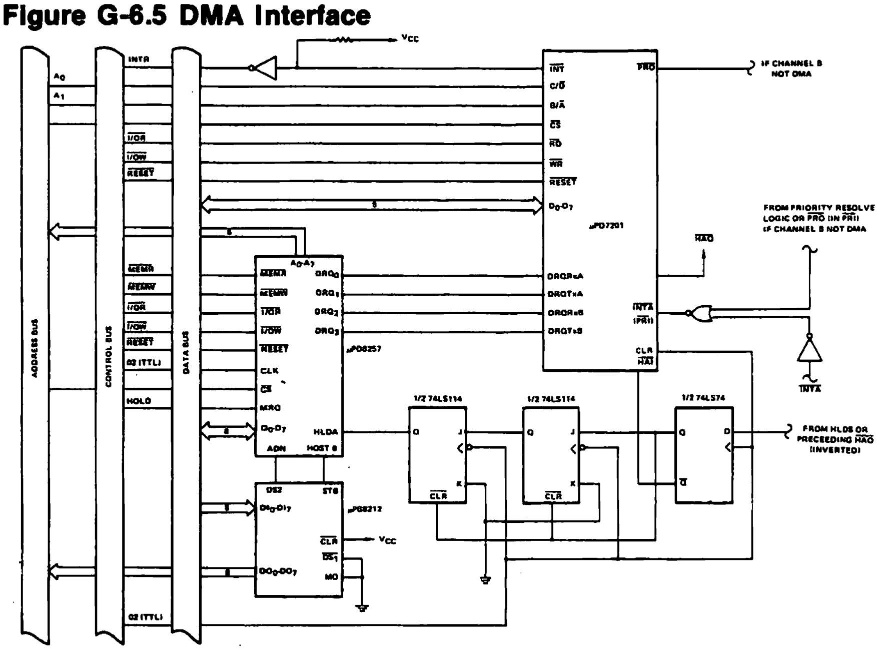
G-6.2 USING THE MPSC² WITH DMA CONTROLLERS

You can greatly increase the data handling capacity of your serial I/O subsystem by using the MPSC² with a DMA controller such as the uPD8257 or uPD8237, to permit direct transfer of data between the MPSC² and memory. Figure 6.5 illustrates a typical MPSC²/DMA configuration. In using the MPSC² in this manner, you should be aware of a few special considerations:

To minimize the number of pins required to implement four DMA channels, the MPSC² does not use the usual DRQ/DACK pins for each channel but rather only DRQ with a single Hold Acknowledge input, HAI. This arrangement eliminates three pins and in addition permits daisychained MPSC²s operating in DMA mode. However, it does require that the MPSC² and the DMA controller reach independent agreement on which DMA request is to be serviced in the case of multiple requests to the same controller.

To ensure that this agreement does occur, you should program the DMA controller for a fixed priority arrangement that agrees with the DMA priority you programmed into the MPSC² (see Secton G-5.1). You must also allow sufficient time for the MPSC² to determine its internal request priority before the DMA controller begins the data transfer. Activating the DMA controller's Hold Acknowledge input through the delay circuit shown in Figure G-6.5 provides this time delay.

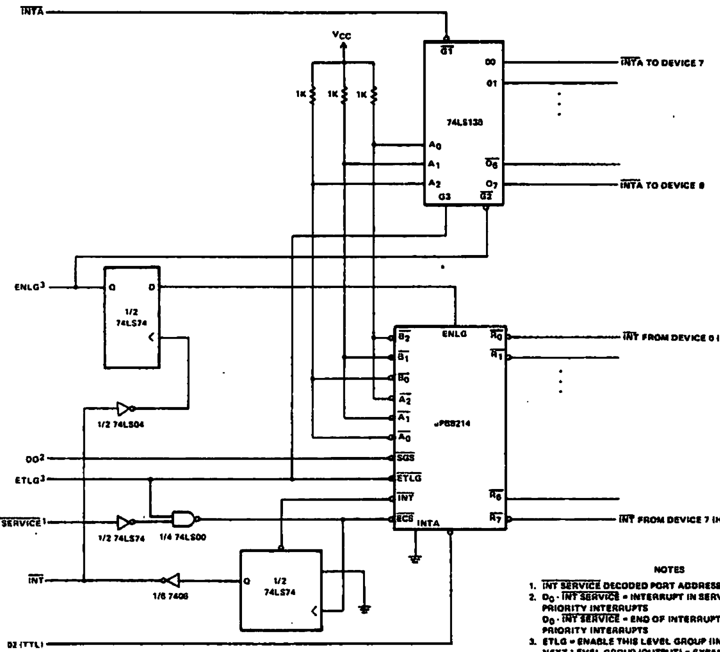
Figure G-6.5 DMA Interface



G-6.3 VECTORED INTERRUPTS WITHOUT USING PRI

There are circumstances when you may wish to use the MPSC²'s Vectored Interrupt feature and you cannot use PRI to inform the MPSC² whether it is the highest priority device requesting service. These situations can occur when both channels are being used in DMA mode (the PRI pin becomes DRQRxB) or when using other peripherals that are incompatible with daisychaining. To retain the Vectored Interrupt feature, you can pull PRI low if available (this is done automatically when both channels are DMA). Program the MPSC² for either 8080 Master or 8086 Vector mode, and gate INTA to the highest priority device with a circuit similar to Figure G-6.6.

Figure G-6.6 Priority Resolution Circuit for Non-daisychained Devices



NOTES
1. INT SERVICE DECODED PORT ADDRESS FROM PROCESSOR
2. D0 · INT SERVICE = INTERRUPT IN SERVICE; ENABLE HIGHER PRIORITY INTERRUPTS
D0 · INT SERVICE = END OF INTERRUPT; ENABLE LOWER PRIORITY INTERRUPTS
3. ETLG = ENABLE THIS LEVEL GROUP (INPUT); ENLG = ENABLE NEXT LEVEL GROUP (OUTPUT) = EXPANSION PINS
4. DEVICES SHOULD BE PROGRAMMED FOR MASTER MODE OF INTA RESPONSE, i.e., THEY SHOULD ISSUE THE COMPLETE CALL INSTRUCTION FOR 8085 SYSTEMS

You should note that an 8259-type interrupt controller programmed for Master Mode does not set its Slave Enable outputs until the second INTA pulse and so is incompatible with the MPSC²'s interrupt acknowledge timing.

G-6.4 TO DMA OR NOT TO DMA...

When operating an MPSC² channel in DMA mode, there are normally some interrupts in parallel with DMA requests. Here are the rules:

Interrupt on Each Character Mode: Both an interrupt and DMA request are made when a character is received.

Interrupt on First Character: The first character received (after issuing an Enable Interrupt On Next Character) generates both an interrupt and a DMA request. Subsequent characters cause only a DMA request to be issued. As an exception, a Special Receive condition always causes both an interrupt and a DMA request.

Transmitter Buffer Becoming Empty: Only DMA requests are issued when the MPSC² is transmitting under DMA control.

G-6.5 HANDLING AN SDLC UNDERRUN FAULT

Since SDLC-type protocols do not allow flags to be imbedded within a message as filler, a fault condition can sometimes occur where the transmitter runs out of data to send. This situation is particularly common in interrupt-driven systems that are heavily task-loaded. You can use the MPSC²s Idle/CRC latch feature to detect these underrun faults and abort the message before an erroneous End of Frame flag is sent. This is accomplished by issuing a Reset Idle/CRC Latch command to the MPSC² immediately after loading it with the first character of the message. If an underrun condition occurs, the MPSC² automatically begins to send the CRC character calculated up to that point and issues an External/Status Change interrupt to indicate that the CRC is being sent. Since your software routine knows that the end of the message has not been reached, an underrun is indicated and your routine can immediately abort the message with a Send Abort command.

G-6.6 SENDING SYNCHRONOUS PAD CHARACTERS

If you want to send one or more pad characters between synchronous messages, you can do it two ways with the MPSC²:

When the MPSC² issues an External/Status interrupt to indicate that CRC is being sent, you can begin loading your pad characters into the transmitter.

Instead of loading pad characters in response to the above interrupt, you can simply change the programmed sync character on the fly, and the MPSC² will transmit pads when it enters Idle Phase after sending CRC.

G-6.7 TRANSMITTING BISYNC TRANSPARENT MODE

Because of the ability to change the sync registers (CR6, CR7) on the fly, the MPSC² is truly compatible with bisync protocol's Transparent mode. On entering this mode, program CR6 with the DLE character and, if an underrun condition occurs, the correct DLE-SYN sequence is transmitted. On leaving Transparent mode you should reset CR6 back to SYN.

G-6.8 VECTORING THE MPSC² IN NON-VECTORED MODE

If you're using the MPSC² in Non-vectored Interrupt mode, you can still use the Condition Affects Vector feature to direct your software to the correct routine. The following example, written in 8080 assembler, assumes that the MPSC² has been programmed for either 8085 master or slave mode (D_3-D_5 modified) and that CR2B was programmed with a zero.

```
MPSCINT:
          PUSH B          ;Save state so registers are free for
          PUSH D          ;your service routine
          PUSH H
          PUSH PSW
```

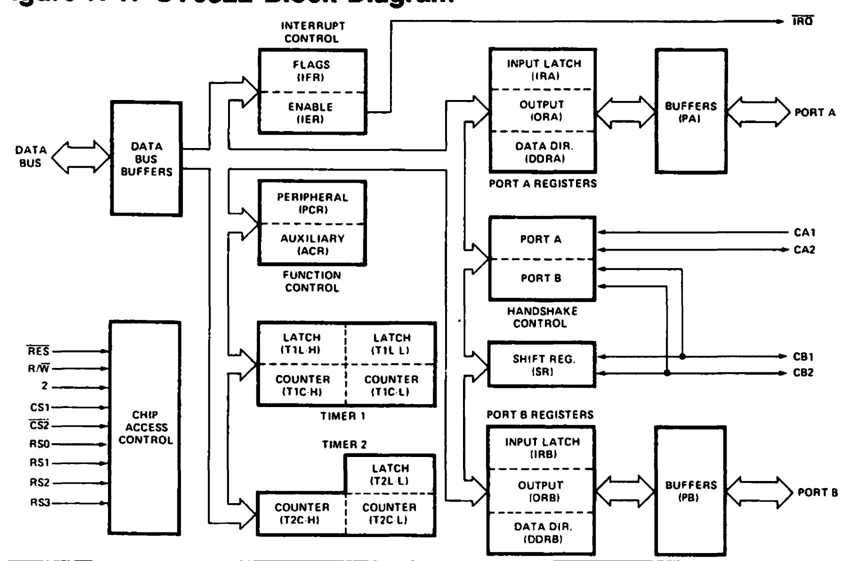
```
        MVI A,2             ;Set channel B register pointer to 2
        OUT MPSCBC
        IN MPSCBC           ;Register A = modified vector
        LXI H, JMPTBL       ;HL→ vector jump table
        MVI D,O             ;DE = offset into table
        MOV E,A
        DAD D               ;HL→ jump table + offset
        PCHL                ;Jump to jump table entry

JMPTBL  JMP TBEB            ;Channel B transmitter buffer empty
        NOP
        JMP EXTB            ;External/Status change
        NOP
        JMP RCVB            ;Received character available

        NOP
        JMP SPRB            ;Special receive condition
        NOP
        •
        •                   ;Repeat for channel A interrupts
        •
        END
```

- Two 8-Bit Bi-directional I/O Ports
- Two 16-Bit Programmable Timer/Counters
- Serial Data Port
- Single +5V Power Supply
- TTL Compatible
- CMOS Compatible Peripheral Control Lines
- Expanded "Handshake" Capability Allows Positive Control of Data Transfers Between Processor and Peripheral Devices
- Latched Output and Input Registers
- 1 MHz and 2 MHz Operation

The SY6522 Versatile Interface Adapter (VIA) is a very flexible I/O control device. In addition, this device contains a pair of very powerful 16-bit interval timers, a serial-to-parallel/parallel-to-serial shift register and input data latching on the peripheral ports. Expanded handshaking capability allows control of bi-directional data transfers between VIA's in multiple processor systems.

Control of peripheral devices is handled primarily through two 8-bit bi-directional ports. Each line can be programmed as either an input or an output. Several peripheral I/O lines can be controlled directly from the interval timers for generating programmable frequency square waves or for counting externally generated pulses. To facilitate control of the many powerful features of this chip, an interrupt flag register, an interrupt enable register and a pair of function control registers are provided.

Figure H-1: SY6522 Block Diagram



ABSOLUTE MAXIMUM RATINGS

This device contains circuitry to protect the inputs against damage due to high static voltages. However, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages.

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	-0.3 to +7.0	V
Input Voltage	V_{IN}	-0.3 to +7.0	V
Operating Temperature Range	T_A	0 to +70	°C
Storage Temperature Range	T_{stg}	-55 to +150	°C

ELECTRICAL CHARACTERISTICS

(V_{CC} 5.0V ±5%, T_A = 0-70° C unless otherwise noted)

Symbol	Characteristic	Min.	Max.	Unit
V_{IH}	Input High Voltage (all except ϕ2)	2.4	V_{CC}	V
V_{CH}	Clock High Voltage	2.4	V_{CC}	V
V_{IL}	Input Low Voltage	-0.3	0.4	V
I_{IN}	Input Leakage Current – V_{IN} = 0 to 5 Vdc R/$\overline{W}$, $\overline{RES}$, RS0, RS1, RS2, RS3, CS1, $\overline{CS2}$, CA1, Φ2	–	±2.5	µA
I_{TSI}	Off-state Input Current – V_{IN} = .4 to 2.4V V_{CC} = Max, D0 to D7	–	±10	µA
I_{IH}	Input High Current – V_{IH} = 2.4V PA0-PA7, CA2, PB0-PB7, CB1, CB2	-100	–	µA
I_{IL}	Input Low Current – V_{IL} = 0.4 Vdc PA0-PA7, CA2, PB0-PB7, CB1, CB2	–	-1.6	mA
V_{OH}	Output High Voltage V_{CC} = min, I_{load} = -100 µAdc PA0-PA7, CA2, PB0-PB7, CB1, CB2	2.4	–	V
V_{OL}	Output Low Voltage V_{CC} = min, I_{load} = 1.6 mAdc	–	0.4	V
I_{OH}	Output High Current (Sourcing) V_{OH} = 2.4V V_{OH} = 1.5V (PB0-PB7)	 -100 -1.0	 – –	 µA mA
I_{OL}	Output Low Current (Sinking) V_{OL} = 0.4 Vdc	1.6	–	mA
I_{OFF}	Output Leakage Current (Off state) $\overline{IRQ}$	–	10	µA
C_{IN}	Input Capacitance – T_A = 25°C, f = 1 MHz (R/$\overline{W}$, $\overline{RES}$, RS0, RS1, RS2, RS3, CS1, $\overline{CS2}$, D0-D7, PA0-PA7, CA1, CA2, PB0-PB7) (CB1, CB2) (Φ2 Input)	– – –	7.0 10 20	pF pF pF
C_{OUT}	Output Capacitance – T_A = 25°C, f = 1 MHz	–	10	pF
P_D	Power Dissipation	–	700	mW

Figure H-2: Test Load (for all Dynamic Parameters)

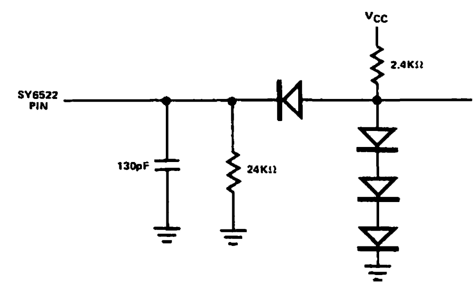
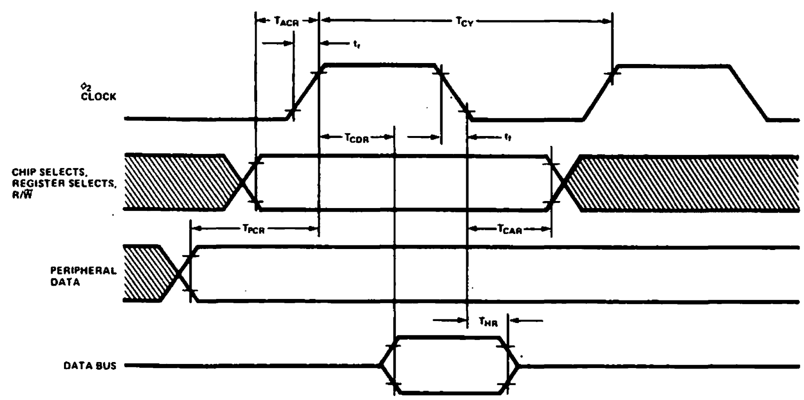


Figure H-3: Read Timing Characteristics

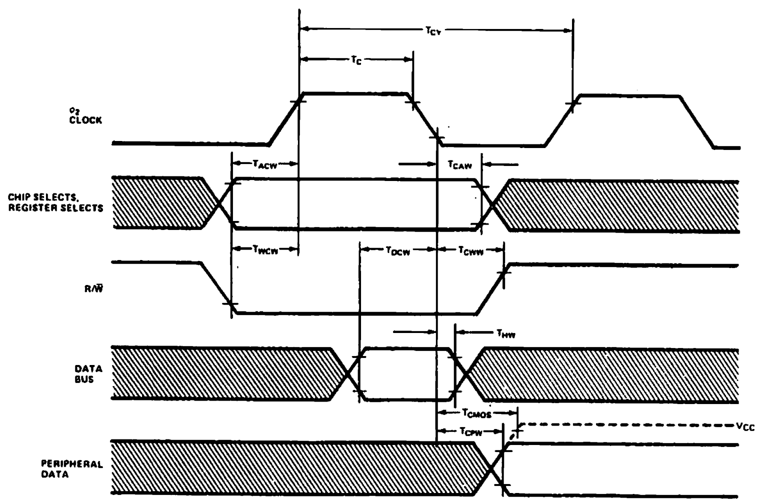


READ TIMING CHARACTERISTICS (FIGURE H-3)

Symbol	Parameter	SY6522 Min.	SY6522 Max.	SY6522A Min.	SY6522A Max.	Unit
T_{CY}	Cycle Time	1	50	0.5	50	μs
T_{ACR}	Address Set-Up Time	180	–	90	–	ns
T_{CAR}	Address Hold Time	0	–	0	–	ns
T_{PCR}	Peripheral Data Set-Up Time	300	–	300	–	ns
T_{CDR}	Data Bus Delay Time	–	340	–	200	ns
T_{HR}	Data Bus Hold Time	10	–	10	–	ns

NOTE: tr, tf = 10 to 30ns.

Figure H-4: Write Timing Characteristics



WRITE TIMING CHARACTERISTICS (FIGURE 4)

Symbol	Parameter	SY6522		SY6522A		Unit
		Min.	Max.	Min.	Max.	
T_{CY}	Cycle Time	1	50	0.50	50	μs
T_C	ϕ2 Pulse Width	0.44	25	0.22	25	μs
T_{ACW}	Address Set-Up Time	180	–	90	–	ns
T_{CAW}	Address Hold Time	0	–	0	–	ns
T_{WCW}	R/$\overline{W}$ Set-Up Time	180	–	90	–	ns
T_{CWW}	R/$\overline{W}$ Hold Time	0	–	0	–	ns
T_{DCW}	Data Bus Set-Up Time	300	–	200	–	ns
T_{HW}	Data Bus Hold Time	10	–	10	–	ns
T_{CPW}	Peripheral Data Delay Time	–	1.0	–	1.0	μs
T_{CMOS}	Peripheral Data Delay Time to CMOS Levels	–	2.0	–	2.0	μs

NOTE: tr, tf = 10 to 30ns.

PERIPHERAL INTERFACE CHARACTERISTICS

Symbol	Characteristic	Min.	Max.	Unit	Figure
t_r, t_f	Rise and Fall Time for CA1, CB1, CA2, and CB2 Input Signals	–	1.0	μs	–
T_{CA2}	Delay Time, Clock Negative Transition to CA2 Negative Transition (read handshake or pulse mode)	–	1.0	μs	5a, 5b
T_{RS1}	Delay Time, Clock Negative Transition to CA2 Positive Transition (pulse mode)	–	1.0	μs	5a
T_{RS2}	Delay Time, CA1 Active Transition to CA2 Positive Transition (handshake mode)	–	2.0	μs	5b
T_{WHS}	Delay Time, Clock Positive Transition to CA2 or CB2 Negative Transition (write handshake)	0.05	1.0	μs	5c, 5d
T_{DS}	Delay Time, Peripheral Data Valid to CB2 Negative Transition	0.20	1.5	μs	5c, 5d
T_{RS3}	Delay Time, Clock Positive Transition to CA2 or CB2 Positive Transition (pulse mode)	–	1.0	μs	5c
T_{RS4}	Delay Time, CA1 or CB1 Active Transition to CA2 or CB2 Positive Transition (handshake mode)	–	2.0	μs	5d
T_{21}	Delay Time Required from CA2 Output to CA1 Active Transition (handshake mode)	400	–	ns	5d
T_{IL}	Set-up Time, Peripheral Data Valid to CA1 or CB1 Active Transition (input latching)	300	–	ns	5e
T_{SR1}	Shift-Out Delay Time – Time from ϕ_2 Falling Edge to CB2 Data Out	–	300	ns	5f
T_{SR2}	Shift-In Setup Time – Time from CB2 Data In to ϕ_2 Rising Edge	300	–	ns	5g
T_{SR3}	External Shift Clock (CB1) Setup Time Relative To ϕ_2 Trailing Edge	100	T_{CY}	ns	5g
T_{IPW}	Pulse Width – PB6 Input Pulse	2	–	μs	5i
T_{ICW}	Pulse Width – CB1 Input Clock	2	–	μs	5h
t_{IPS}	Pulse Spacing – PB6 Input Pulse	2	–	μs	5i
t_{ICS}	Pulse Spacing – CB1 Input Pulse	2	–	μs	5h

Figure H-5a: CA2 Timing for Read Handshake, Pulse Mode

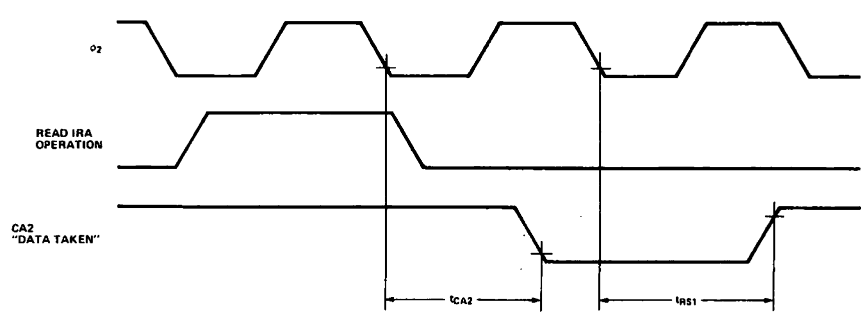


Figure H-5b: CA2 Timing for Read Handshake, Handshake Mode

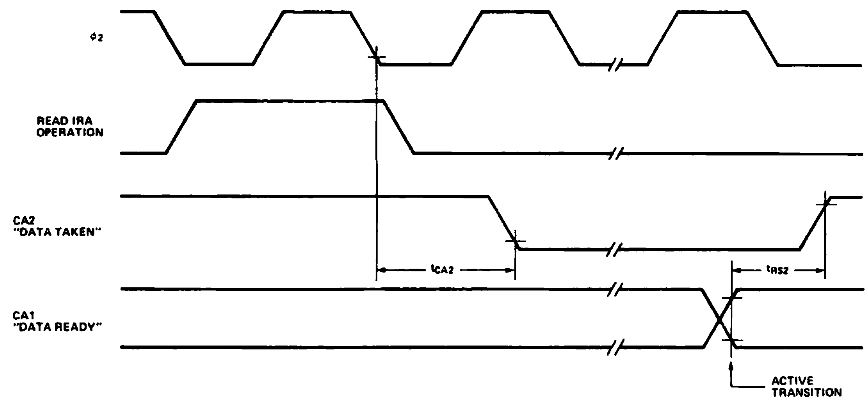


Figure H-5c: CA2, CB2 Timing for Write Handshake, Pulse Mode

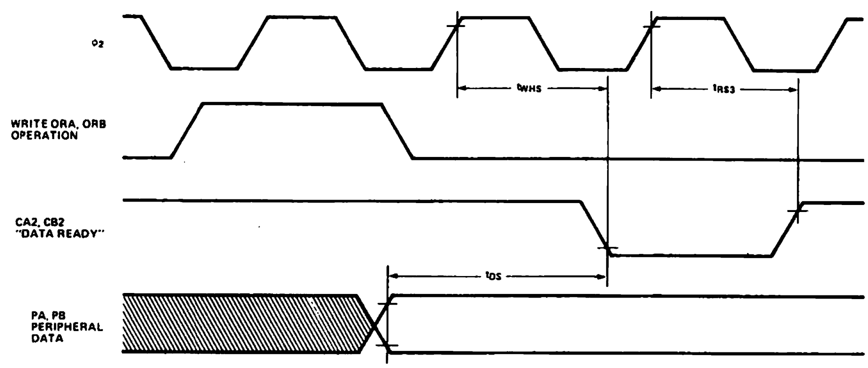


Figure H-5d: CA2, CB2 Timing for Write Handshake, Handshake Mode

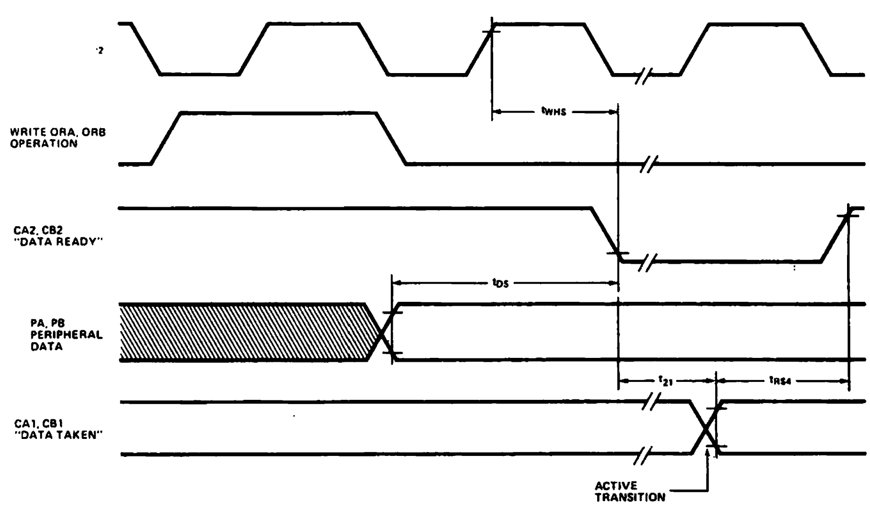


Figure H-5e: Peripheral Data Input Latching Timing

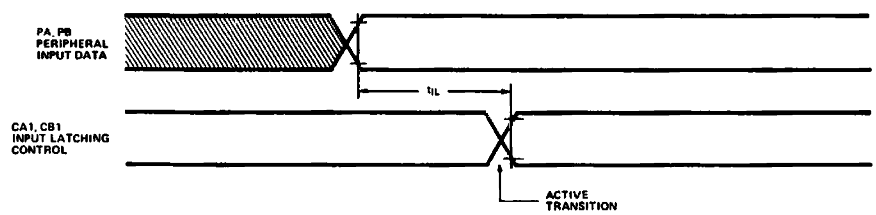


Figure H-5f: Timing for Shift Out with Internal or External Shift Clocking

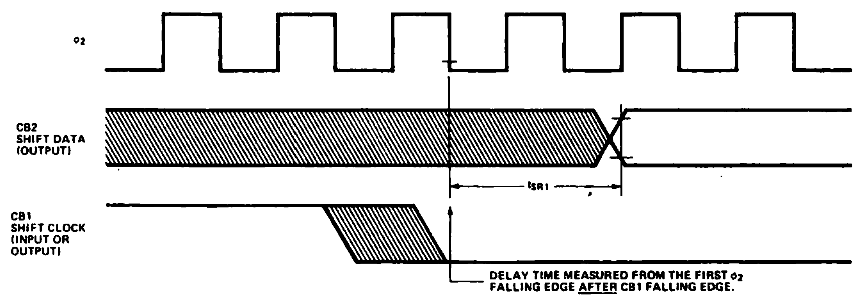


Figure H-5g: Timing for Shift In with Internal or External Shift Clocking

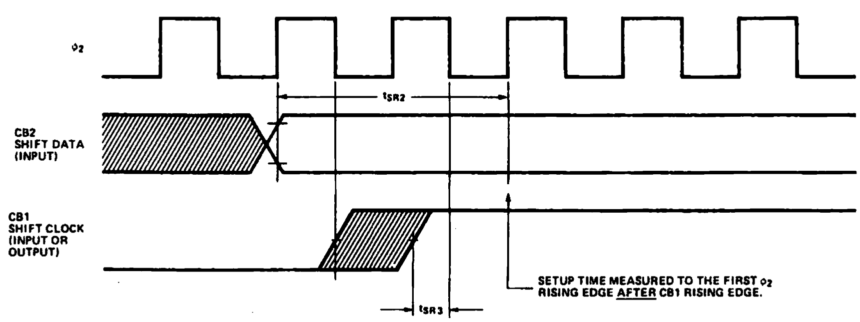


Figure H-5h: External Shift Clock Timing

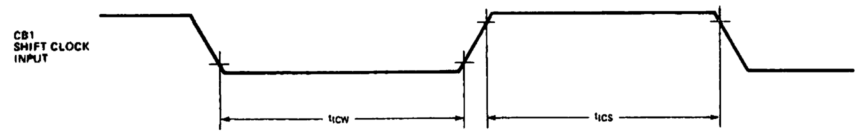
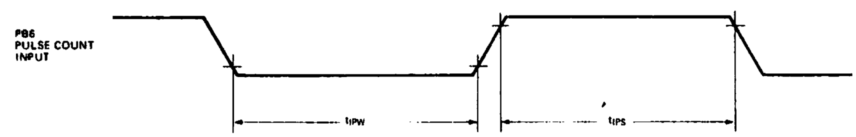


Figure H-5i: Pulse Count Input Timing



PIN DESCRIPTIONS

RES (RESET)

The reset input clears all internal registers to logic 0 (except T1 and T2 latches and counters and the Shift Register). This places all peripheral interface lines in the input state, disables the timers, shift register, etc. and disables interrupting from the chip.

ϕ2 (INPUT CLOCK)

The input clock is the system ϕ2 clock and is used to trigger all data transfers between the system processor and the SY6522.

R/W (READ/WRITE)

The direction of the data transfers between the SY6522 and the system processor is controlled by the R/W line. If R/W is low, data will be transferred out of the processor into the selected SY6522 register (write operation). If R/W is high and the chip is selected, data will be transferred out of the SY6522 (read operation).

DB0-DB7 (DATA BUS)

The eight bi-directional data bus lines are used to transfer data between the SY6522 and the system processor. During read cycles, the contents of the selected SY6522 register are placed on the data bus lines and transferred into the processor. During write cycles, these lines are high-impedance inputs and data is transferred from the processor into the selected register. When the SY6522 is unselected, the data bus lines are high-impedance.

CS1, CS2 (CHIP SELECTS)

The two chip select inputs are normally connected to processor address lines either directly or through decoding. The selected SY6522 register will be accessed when CS1 is high and CS2 is low.

RS0-RS3 (REGISTER SELECTS)

The four Register Select inputs permit the system processor to select one of the 16 internal registers of the SY6522, as shown in Figure H-6.

Figure H-6: SY6522 Internal Register Summary

Register Number	RS Coding				Register Desig.	Description	
	RS3	RS2	RS1	RS0		Write	Read
0	0	0	0	0	ORB/IRB	Output Register "B"	Input Register "B"
1	0	0	0	1	ORA/IRA	Output Register "A"	Input Register "A"
2	0	0	1	0	DDRB	Data Direction Register "B"	
3	0	0	1	1	DDRA	Data Direction Register "A"	
4	0	1	0	0	T1C-L	T1 Low-Order Latches	T1 Low-Order Counter
5	0	1	0	1	T1C-H	T1 High-Order Counter	
6	0	1	1	0	T1L-L	T1 Low-Order Latches	
7	0	1	1	1	T1L-H	T1 High-Order Latches	
8	1	0	0	0	T2C-L	T2 Low-Order Latches	T2 Low-Order Counter
9	1	0	0	1	T2C-H	T2 High-Order Counter	
10	1	0	1	0	SR	Shift Register	
11	1	0	1	1	ACR	Auxiliary Control Register	
12	1	1	0	0	PCR	Peripheral Control Register	
13	1	1	0	1	IFR	Interrupt Flag Register	
14	1	1	1	0	IER	Interrupt Enable Register	
15	1	1	1	1	ORA/IRA	Same as Reg 1 Except No "Handshake"	

IRQ (INTERRUPT REQUEST)

The Interrupt Request output goes low whenever an internal Interrupt Flag is set and the corresponding interrupt enable bit is a logic 1. This output is "open-drain" to allow the interrupt request signal to be "wire-or'ed" with other equivalent signals in the system.

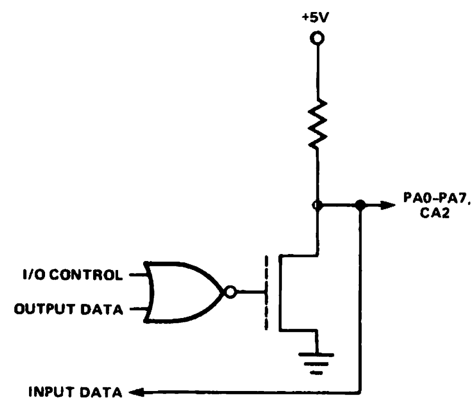
PA0-PA7 (PERIPHERAL A PORT)

The Peripheral A port consists of 8 lines which can be individually programmed to act as inputs or outputs under control of a Data Direction Register. The polarity of output pins is controlled by an Output Register and input data may be latched into an internal register under control of the CA1 line. All of these modes of operation are controlled by the system processor through the internal control registers. These lines represent one standard TTL load in the input mode and will drive one standard TTL load in the output mode. Figure H-7 illustrates the output circuit.

CA1, CA2 (PERIPHERAL A CONTROL LINES)

The two Peripheral A control lines act as interrupt inputs or as handshake outputs. Each line controls an internal Interrupt Flag with a corresponding interrupt enable bit. In addition, CA1 controls the latching of data on Peripheral A port input lines. CA1 is a high-impedance input only while CA2 represents one standard TTL load in the input mode. CA2 will drive one standard TTL load in the output mode.

Figure H-7: Peripheral A Port Output Circuit



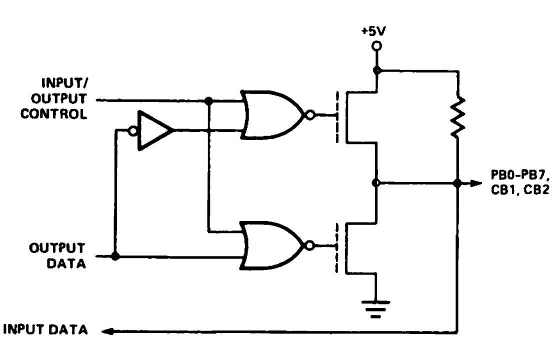
PB0-PB7 (PERIPHERAL B PORT)

The Peripheral B port consists of eight bi-directional lines which are controlled by an output register and a data direction register in much the same manner as the PA port. In addition, the polarity of the PB7 output signal can be controlled by one of the interval timers while the second timer can be programmed to count pulses on the PB6 pin. Peripheral B lines represent one standard TTL load in the input mode and will drive one standard TTL load in the output mode. In addition, they are capable of sourcing 1.0mA at 1.5VDC in the output mode to allow the outputs to directly drive Darlington transistor circuits. Figure H-8 is the circuit schematic.

CB1, CB2 (PERIPHERAL B CONTROL LINES)

The Peripheral B control lines act as interrupt inputs or as handshake outputs. As with CA1 and CA2, each line controls an Interrupt Flag with a corresponding interrupt enable bit. In addition, these lines act as a serial port under control of the Shift Register. These lines represent one standard TTL load in the input mode and will drive one standard TTL load in the output mode. Unlike PB0-PB7, CB1 and CB2 *cannot* drive Darlington transistor circuits.

Figure H-8: Peripheral B Port Output Circuit



FUNCTIONAL DESCRIPTION

PORT A AND PORT B OPERATION

Each 8-bit peripheral port has a Data Direction Register (DDRA, DDRB) for specifying whether the peripheral pins are to act as inputs or outputs. A "0" in a bit of the Data Direction Register causes the corresponding peripheral pin to act as an input. A "1" causes the pin to act as an output.

Each peripheral pin is also controlled by a bit in the Output Register (ORA, ORB) and an Input Register (IRA, IRB). When the pin is programmed as an output, the voltage on the pin is controlled by the corresponding bit of the Output Register. A "1" in the Output Register causes the output to go high, and a "0" causes the output to go low. Data may be written into Output Register bits corresponding to pins which are programmed as inputs. In this case, however, the output signal is unaffected.

Reading a peripheral port causes the contents of the Input Register (IRA, IRB) to be transferred onto the data bus. With input latching disabled, IRA will always reflect the levels on the PA pins. With input latching enabled, IRA will reflect the levels on the PA pins at the time latching occurred (via CA1).

The IRB register operates similar to the IRA register. However, for pins programmed as outputs there is a difference. When reading IRA, the *level on the pin* determines whether a "0" or a "1" is sensed. When reading IRB, however, the bit stored in the *output register*, ORB, is the bit sensed. Thus, for outputs which have large loading effects and which pull an output "1" down or which pull an output "0" up, reading IRA may result in reading a "0" when a "1" was actually programmed, and reading a "1" when a "0" was programmed. Reading IRB, on the other hand, will read the "1" or "0" level actually programmed, no matter what the loading on the pin.

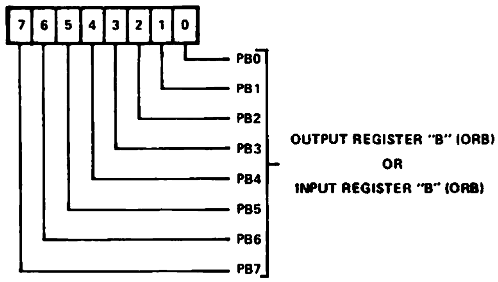
Figures H-9, H-10, and H-11 illustrate the formats of the port registers. In addition, the input latching modes are selected by the Auxiliary Control Register (Figure H-16.)

HANDSHAKE CONTROL OF DATA TRANSFERS

The SY6522 allows positive control of data transfers between the system processor and peripheral devices through the operation of "handshake" lines. Port A lines (CA1, CA2) handshake data on both a read and a write operation while the Port B lines (CB1, CB2) handshake on a write operation only.

Figure H-9: Output Register B (ORB), Input Register B (IRB)

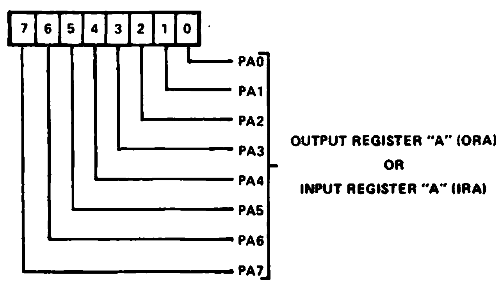
REG 0 – ORB/IRB



Pin Data Direction Selection	WRITE	READ
DDRB = "1" (OUTPUT)	MPU writes Output Level (ORB)	MPU reads output register bit in ORB. Pin level has no affect.
DDRB = "0" (INPUT) (Input latching disabled)	MPU writes into ORB, but no effect on pin level, until DDRB changed.	MPU reads input level on PB pin.
DDRB = "0" (INPUT) (Input latching enabled)		MPU reads IRB bit, which is the level of the PB pin at the time of the last CB1 active transition.

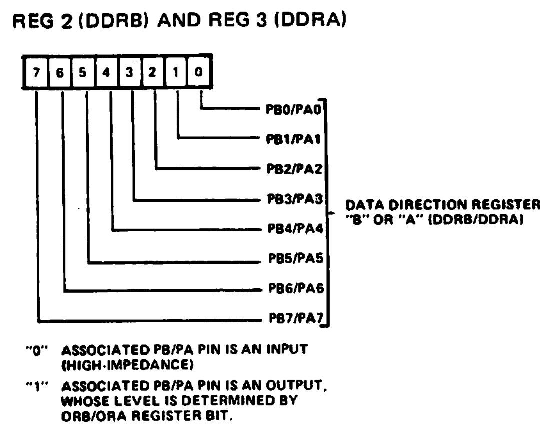
Figure H-10: Output Register A (ORA), Input Register A (IRA)

REG 1 – ORA/IRA



Pin Data Direction Selection	WRITE	READ
DDRA = "1" (OUTPUT) (Input latching disabled)	MPU writes Output Level (ORA).	MPU reads level on PA pin.
DDRA = "1" (OUTPUT) (Input latching enabled)		MPU reads IRA bit which is the level of the PA pin at the time of the last CA1 active transition.
DDRA = "0" (INPUT) (Input latching disabled)	MPU writes into ORA, but no effect on pin level, until DDRA changed.	MPU reads level on PA pin.
DDRA = "0" (INPUT) (Input latching enabled)		MPU reads IRA bit which is the level of the PA pin at the time of the last CA1 active transition.

Figure H-11: Data Direction Registers (DDRB, DDRA)

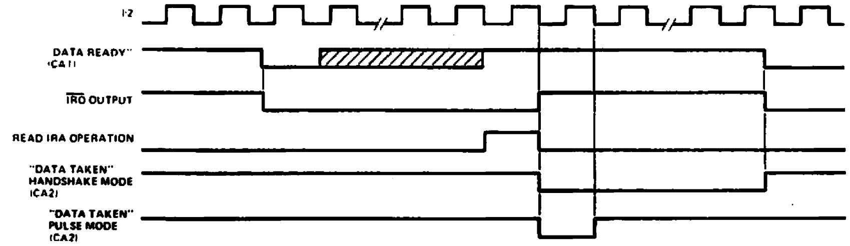


READ HANDSHAKE

Positive control of data transfers from peripheral devices into the system processor can be accomplished very effectively using Read Handshaking. In this case, the peripheral device must generate the equivalent of a "Data Ready" signal to the processor signifying that valid data is present on the peripheral port. This signal normally interrupts the processor, which then reads the data, causing generation of a "Data Taken" signal. The peripheral device responds by making new data available. This process continues until the data transfer is complete.

In the SY6522, automatic "Read" Handshaking is possible on the Peripheral A port only. The CA1 interrupt input pin accepts the "Data Ready" signal and CA2 generates the "Data Taken" signal. The "Data Ready" signal will set an internal flag which may interrupt the processor or which may be polled under program control. The "Data Taken" signal can either be a pulse or a level which is set low by the system processor and is cleared by the "Data Ready" signal. These options are shown in Figure H-12, which illustrates the normal Read Handshaking sequence.

Figure H-12: Read Handshake Timing (Port A, Only)

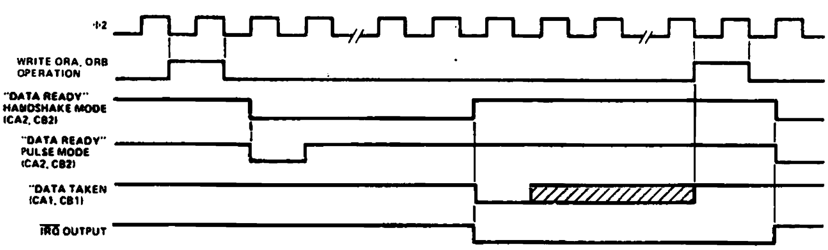


WRITE HANDSHAKE

The sequence of operations which allows handshaking data from the system processor to a peripheral device is very similar to that described for Read Handshaking. However, for Write Handshaking, the SY6522 generates the "Data Ready" signal and the peripheral device must respond with the "Data Taken" signal. This can be accomplished on both the PA port and the PB port on the SY6522. CA2 or CB2 act as a "Data Ready" output in either the handshake mode or pulse mode and CA1 or CB1 accept the "Data Taken" signal from the peripheral device, setting the Interrupt Flag and cleaning the "Data Ready" output. This sequence is shown in Figure H-13.

Selection of operating modes for CA1, CA2, CB1, and CB2 is accomplished by the Peripheral Control Register (Figure H-14).

Figure H-13: Write Handshake Timing



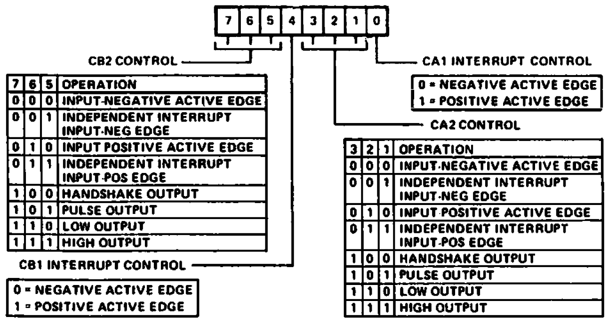
TIMER OPERATION

Interval Timer T1 consists of two 8-bit latches and a 16-bit counter. The latches are used to store data which is to be loaded into the counter. After loading, the counter decrements at $\phi 2$ clock rate. Upon reaching zero, an Interrupt Flag will be set, and IRQ will go low if the interrupt is enabled. The timer will then disable any further interrupts, or will automatically transfer the contents of the latches into the counter and will continue to decrement. In addition, the timer may be programmed to invert the output signal on a peripheral pin each time it "times-out". Each of these modes is discussed separately below.

The T1 counter is depicted in Figure H-15 and the latches in Figure H-16.

Figure H-14: CA1, CA2, CB1, CB2 Control

REG 12 – PERIPHERAL CONTROL REGISTER

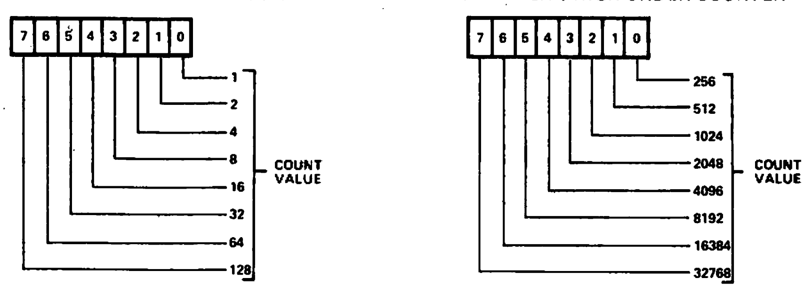


Two bits are provided in the Auxiliary Control Register (bits 6 and 7) to allow selection of the T1 operating modes. The four possible modes are depicted in Figure H-17.

Figure H-15: T1 Counter Registers

REG 4 – TIMER 1 LOW-ORDER COUNTER

REG 5 – TIMER 1 HIGH-ORDER COUNTER



WRITE – 8 BITS LOADED INTO T1 LOW-ORDER LATCHES. LATCH CONTENTS ARE TRANSFERRED INTO LOW-ORDER COUNTER AT THE TIME THE HIGH-ORDER COUNTER IS LOADED (REG 5).

READ – 8 BITS FROM T1 LOW-ORDER COUNTER TRANSFERRED TO MPU. IN ADDITION, T1 INTERRUPT FLAG IS RESET (BIT 6 IN INTERRUPT FLAG REGISTER).

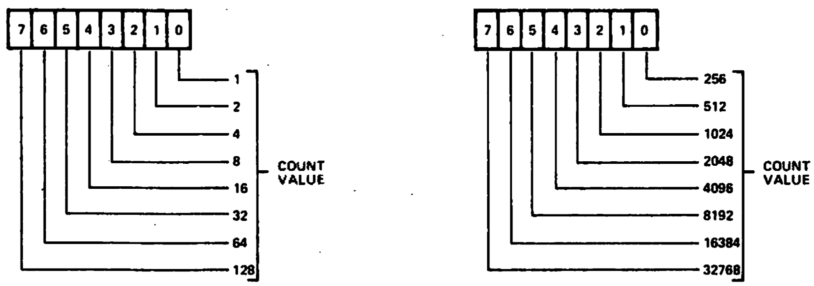
WRITE – 8 BITS LOADED INTO T1 HIGH-ORDER LATCHES. ALSO, AT THIS TIME BOTH HIGH AND LOW-ORDER LATCHES TRANSFERRED INTO T1 COUNTER. T1 INTERRUPT FLAG ALSO IS RESET.

READ – 8 BITS FROM T1 HIGH-ORDER COUNTER TRANSFERRED TO MPU.

Figure H-16: T1 Latch Registers

REG 6 – TIMER 1 LOW-ORDER LATCHES

REG 7 – TIMER 1 HIGH-ORDER LATCHES



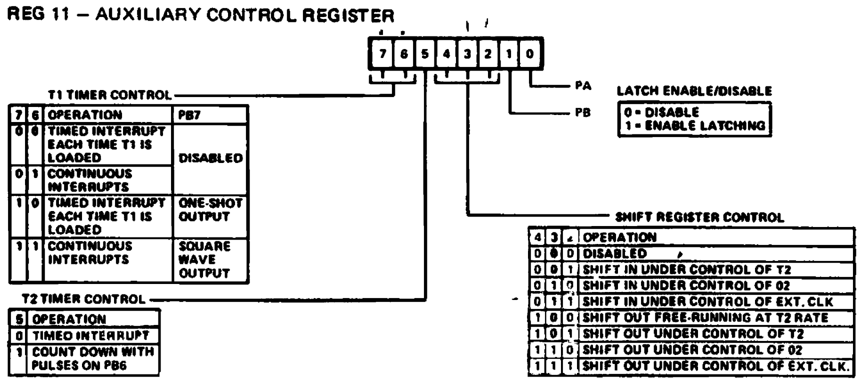
WRITE – 8 BITS LOADED INTO T1 LOW-ORDER LATCHES. THIS OPERATION IS NO DIFFERENT THAN A WRITE INTO REG 4.

READ – 8 BITS FROM T1 LOW-ORDER LATCHES TRANSFERRED TO MPU. UNLIKE REG 4 OPERATION, THIS DOES NOT CAUSE RESET OF T1 INTERRUPT FLAG.

WRITE – 8 BITS LOADED INTO T1 HIGH-ORDER LATCHES. UNLIKE REG 4 OPERATION NO LATCH-TO-COUNTER TRANSFERS TAKE PLACE.

READ – 8 BITS FROM T1 HIGH-ORDER LATCHES TRANSFERRED TO MPU.

Figure H-17: Auxiliary Control Register

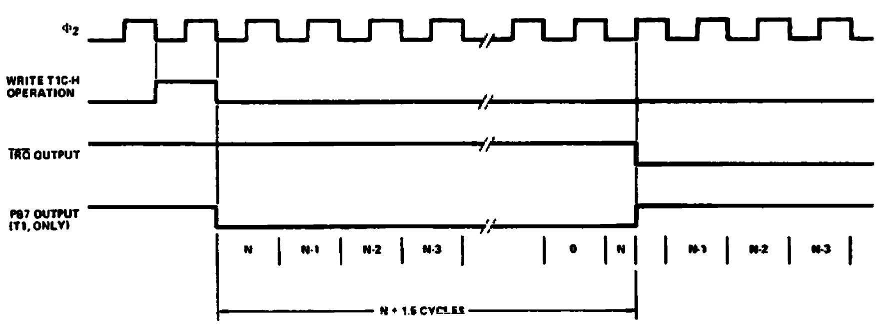


7	6	OPERATION	PB7
0	0	TIMED INTERRUPT EACH TIME T1 IS LOADED	DISABLED
0	1	CONTINUOUS INTERRUPTS	
1	0	TIMED INTERRUPT EACH TIME T1 IS LOADED	ONE-SHOT OUTPUT
1	1	CONTINUOUS INTERRUPTS	SQUARE WAVE OUTPUT

5	OPERATION
0	TIMED INTERRUPT
1	COUNT DOWN WITH PULSES ON PB6

4	3	2	OPERATION
0	0	0	DISABLED
0	0	1	SHIFT IN UNDER CONTROL OF T2
0	1	0	SHIFT IN UNDER CONTROL OF 02
0	1	1	SHIFT IN UNDER CONTROL OF EXT. CLK
1	0	0	SHIFT OUT FREE-RUNNING AT T2 RATE
1	0	1	SHIFT OUT UNDER CONTROL OF T2
1	1	0	SHIFT OUT UNDER CONTROL OF 02
1	1	1	SHIFT OUT UNDER CONTROL OF EXT. CLK.

Figure H-18: Timer 1 and Timer 2 One-Shot Mode Timing



TIMER 1 ONE-SHOT MODE

The interval timer one-shot mode allows generation of a single interrupt for each timer load operation. As with any interval timer, the delay between the "write T1C-H" operation and generation of the processor interrupt is a direct function of the data loaded into the timing counter. In addition to generating a single interrupt, Timer 1 can be programmed to produce a single negative pulse on the PB7 peripheral pin. With the output enabled (ACR7 = 1) a "write T1C-H" operation will cause PB7 to go low. PB7 will return high when Timer 1 times out. The result is a single programmable width pulse.

In the one-shot mode, writing into the high order latch has no effect on the operation of Timer 1. However, it will be necessary to assure that the low order latch contains the proper data before initiating the count-down with a "write T1C-H" operation. When the processor writes into the high order counter, the T1 Interrupt Flag will be cleared, the contents of the low order latch will be transferred into the low order counter, and the timer will begin to decrement at system clock rate. If the PB7 output is enabled, this signal will go low on the phase two following the write operation. When the counter reaches zero, the T1 Interrupt Flag will be set, the IRQ pin will go low (interrupt enabled), and the signal on PB7 will go high. At this time the counter will continue to decrement at system clock rate. This allows the system processor to read the contents of the counter to determine the time since interrupt. However, the T1 Interrupt Flag cannot be set again unless it has been cleared as described in this specification.

Timing for the SY6522 interval timer one-shot modes is shown in Figure H-18.

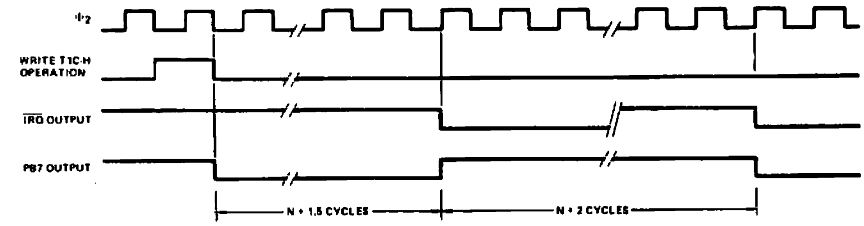
TIMER 1 FREE-RUN MODE

The most important advantage associated with the latches in T1 is the ability to produce a continuous series of evenly spaced interrupts and the ability to produce a square wave on PB7 whose frequency is not affected by variations in the processor interrupt response time. This is accomplished in the "free-running" mode.

In the free-running mode, the Interrupt Flag is set and the signal on PB7 is inverted each time the counter reaches zero. However, instead of continuing to decrement from zero after a time-out, the timer automatically transfers the contents of the latch into the counter (16 bits) and continues to decrement from there. The Interrupt Flag can be cleared by writing T1C-H, by reading T1C-L, or by writing directly into the flag as described later. However, it is not necessary to rewrite the timer to enable setting the Interrupt Flag on the next time-out.

All interval timers in the SY6522 are "re-triggerable". Rewriting the counter will always re-initialize the time-out period. In fact, the time-out can be prevented completely if the processor continues to rewrite the timer before it reaches zero. Timer 1 will operate in this manner if the processor writes into the high order counter (T1C-H). However, by loading the latches only, the processor can access the timer during each down-counting operation without affecting the time-out in process. Instead, the data loaded into the latches will determine the length of the next time-out period. This capability is particularly valuable in the free-running mode with the output enabled. In this mode, the signal on PB7 is inverted and the interrupt flag is set with each time-out. By responding to the interrupts with new data for the latches, the processor can determine the period of the next half cycle during each half cycle of the output signal on PB7. In this manner, very complex waveforms can be generated. Timing for the free-running mode is shown in Figure H-19.

Figure H-19: Timer 1 Free-Run Mode Timing



Note: A precaution to take in the use of PB7 as the timer output concerns the Data Direction Register contents for PB7. Both DDRB bit 7 and ACR bit 7 must be 1 for PB7 to function as the timer output. If one is 1 and the other is 0, then PB7 functions as a normal output pin, controlled by ORB bit 7.

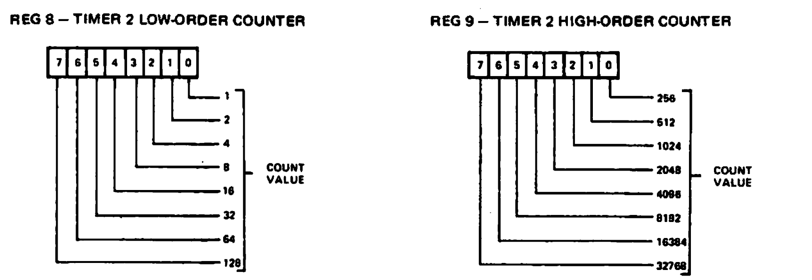
TIMER 2 OPERATION

Timer 2 operates as an interval timer (in the "one-slot" mode only), or as a counter for counting negative pulses on the PB6 peripheral pin. A single control bit is provided in the Auxiliary Control Register to select between these two modes. This timer is comprised of a "write-only" low-order latch (T2L-L), a "read-only" low-order counter and a read/write high order counter. The counter registers act as a 16-bit counter which decrements at $\phi 2$ rate. Figure H-20 illustrates the T2 Counter Registers.

TIMER 2 ONE-SHOT MODE

As an interval timer, T2 operates in the "one-shot" mode similar to Timer 1. In this mode, T2 provides a single interrupt for each "write T2C-H" operation. After timing out, the counter will continue to decrement. However, setting of the Interrupt Flag will be disabled after initial time-out so that it will not be set by the counter continuing to decrement through zero. The processor must rewrite T2C-H to enable setting of the Interrupt Flag. The Interrupt Flag is cleared by reading T2C-L or by writing T2C-H. Timing for this operation is shown in Figure H-18.

Figure H-20: T2 Counter Registers



WRITE – 8 BITS LOADED INTO T2 LOW-ORDER LATCHES.
READ – 8 BITS FROM T2 LOW-ORDER COUNTER TRANSFERRED TO MPU. T2 INTERRUPT FLAG IS RESET.

WRITE – 8 BITS LOADED INTO T2 HIGH-ORDER COUNTER. ALSO, LOW-ORDER LATCHES TRANSFERRED TO LOW-ORDER COUNTER. IN ADDITION, T2 INTERRUPT FLAG IS RESET.
READ – 8 BITS FROM T2 HIGH-ORDER COUNTER TRANSFERRED TO MPU.

TIMER 2 PULSE COUNTING MODE

In the pulse counting mode, T2 serves primarily to count a predetermined number of negative-going pulses on PB6. This is accomplished by first loading a number into T2. Writing into T2C-H clears the Interrupt Flag and allows the counter to decrement each time a pulse is applied to PB6. The Interrupr Flag will be set when T2 reaches zero. At this time the counter will continue to decrement with each pulse on PB6. However, it is necessary to rewrite T2C-H to allow the Interrupt Flag to set on subsequent down-counting operations. Timing for this mode is shown in Figure H-21. The pulse must be low on the leading edge of $\phi 2$.

SHIFT REGISTER OPERATION

The Shift Register (SR) performs serial data transfers into and out of the CB2 pin under control of an internal modulo-8 counter. Shift pulses can be applied to the CB1 pin from an external source or, with the proper mode selection, shift pulses generated internally will appear on the CB1 pin for controlling external devices.

The control bits which select the various shift register operating modes are located in the Auxiliary Control Register. Figure H-22 illustrates the configuration of the SR data bits and the SR control bits of the ACR.

Figures H-23 and H-24 illustrate the operation of the various shift register modes.

INTERRUPT OPERATION

Controlling interrupts within the SY6522 involves three principle operations. These are flagging the interrupts, enabling interrupts and signaling to the processor that an active interrupt exists within the chip. Interrupt flags are set by interrupting conditions which exist within the chip or on inputs to the chip. These flags normally remain set until the interrupt has been serviced. To determine the source of an interrupt, the microprocessor must examine these flags in order from highest to lowest priority. This is accomplished by reading the flag register into the processor accumulator, shifting this register either right or left and then using conditional branch instructions to detect an active interrupt.

Associated with each Interrupt Flag is an interrupt enable bit. This can be set or cleared by the processor to enable interrupting the processor from the corresponding Interrupt Flag. If an interrupt flag is set to a logic 1 by an interrupting condition, and the corresponding interrupt enable bit is set to a 1, the Interrupt Request Output (IRQ) will go low. IRQ is an "open-collector" output which can be "wire-or'ed" with other devices in the system to interrupt the processor.

In the SY6522, all the Interrupt Flags are contained in one register. In addition, bit 7 of this register will be read as a logic 1 when an interrupt exists within the chip. This allows very convenient polling of several devices within a system to locate the source of an interrupt.

Figure H-21: Timer 2 Pulse Counting Mode

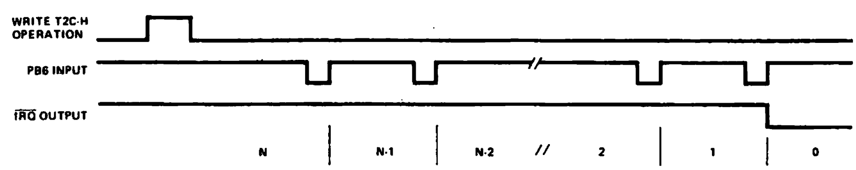
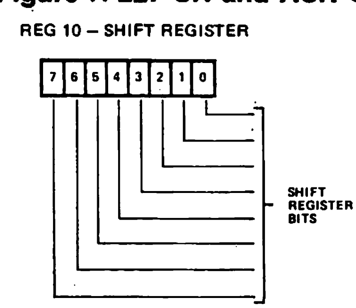
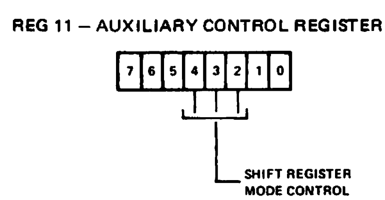


Figure H-22: SR and ACR Control Bits



NOTES:
1. WHEN SHIFTING OUT, BIT 7 IS THE FIRST BIT OUT AND SIMULTANEOUSLY IS ROTATED BACK INTO BIT 0.
2. WHEN SHIFTING IN, BITS INITIALLY ENTER BIT 0 AND ARE SHIFTED TOWARDS BIT 7.



4	3	2	OPERATION
0	0	0	DISABLED
0	0	1	SHIFT IN UNDER CONTROL OF T2
0	1	0	SHIFT IN UNDER CONTROL OF ϕ_2
0	1	1	SHIFT IN UNDER CONTROL OF EXT CLK
1	0	0	SHIFT OUT FREE-RUNNING AT T2 RATE
1	0	1	SHIFT OUT UNDER CONTROL OF T2
1	1	0	SHIFT OUT UNDER CONTROL OF ϕ_2
1	1	1	SHIFT OUT UNDER CONTROL OF EXT CLK

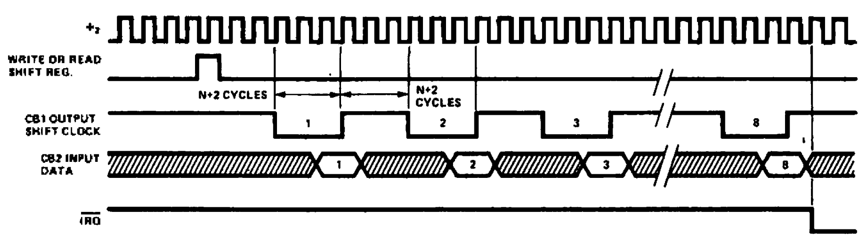
SR Disabled (000)

The 000 mode is used to disable the Shift Register. In this mode the microprocessor can write or read the SR, but the shifting operation is disabled and operation of CB1 and CB2 is controlled by the appropriate bits in the Peripheral Control Register (PCR). In this mode the SR Interrupt Flag is disabled (held to a logic 0).

Shift In Under Control of T2 (001)

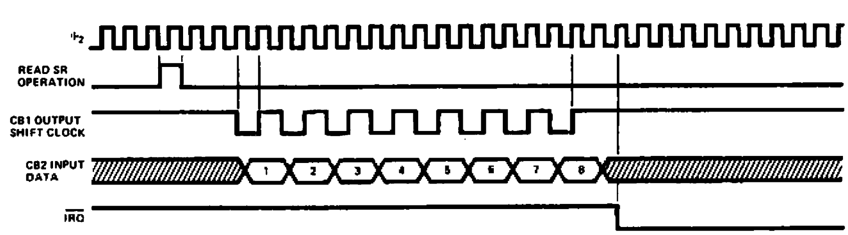
In the 001 mode the shifting rate is controlled by the low order 8 bits of T2. Shift pulses are generated on the CB1 pin to control shifting in external devices. The time between transitions of this output clock is a function of the system clock period and the contents of the low order T2 latch (N).

The shifting operation is triggered by writing or reading the shift register. Data is shifted first into the low order bit of SR and is then shifted into the next higher order bit of the shift register on the negative-going edge of each clock pulse. The input data should change before the positive-going edge of the CB1 clock pulse. This data is shifted into the shift register during the ϕ_2 clock cycle following the positive-going edge of the CB1 clock pulse. After 8 CB1 clock pulses, the shift register Interrupt Flag will be set and IRQ will go low.



Shift In Under Control of ϕ_2 (010)

In mode 010 the shift rate is a direct function of the system clock frequency. CB1 becomes an output which generates shift pulses for controlling external devices. Timer 2 operates as an independent interval timer and has no effect on SR. The shifting operation is triggered by reading or writing the Shift Register. Data is shifted first bit 0 and is then shifted into the next higher order bit of the shift register on the trailing edge of each ϕ_2 clock pulse. After 8 clock pulses, the shift register Interrupt Flag will be set, and the output clock pulses on CB1 will stop.

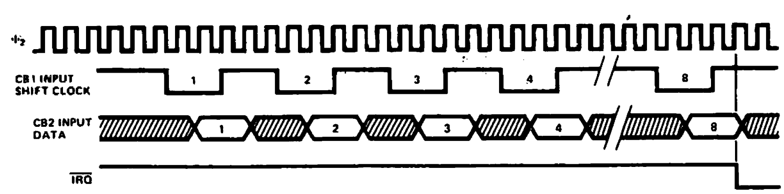


Shift In Under Control of External CB1 Clock (011)

In mode 011 CB1 becomes an input. This allows an external device to load the shift register at its own pace. The shift register counter will interrupt the processor each time 8 bits have been shifted in. However, the shift register counter does not stop the shifting operation; it acts simply as a pulse counter. Reading or writing the Shift Register resets the Interrupt flag and initializes the SR counter to count another 8 pulses.

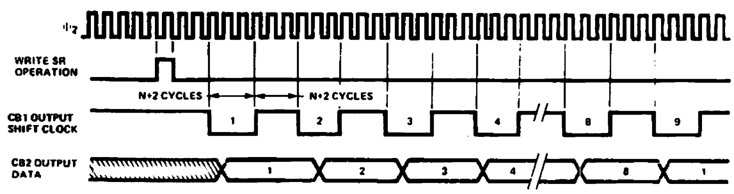
Note that the data is shifted during the first system clock cycle following the positive-going edge of the CB1 shift pulse. For this reason, data must be held stable during the first full cycle following CB1 going high.

Figure H-23: Shift Register Input Modes



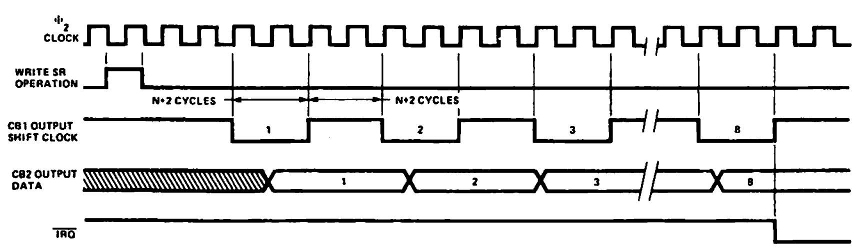
Shift Out Free-Running at T2 Rate (100)

Mode 100 is very similar to mode 101 in which the shifting rate is set by T2. However, in mode 100 the SR counter does not stop the shifting operation. Since the Shift Register bit 7 (SR7) is recirculated back into bit 0, the 8 bits loaded into the shift register will be clocked onto CB2 repetitively. In this mode the shift register counter is disabled.



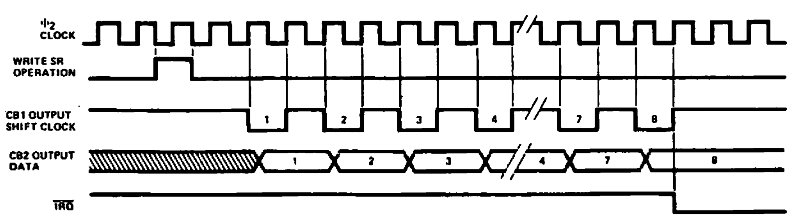
Shift Out Under Control of T2 (101)

In mode 101 the shift rate is controlled by T2 (as in the previous mode). However, with each read or write of the shift register the SR Counter is reset and 8 bits are shifted onto CB2. At the same time, 8 shift pulses are generated on CB1 to control shifting in external devices. After the 8 shift pulses, the shifting is disabled, the SR Interrupt Flag is set and CB2 remains at the last data level.



Shift Out Under Control of ϕ_2 (110)

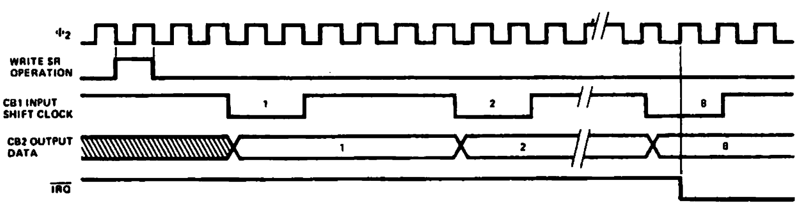
In mode 110, the shift rate is controlled by the ϕ_2 system clock.



Shift Out Under Control of External CB1 Clock (111)

In mode 111 shifting is controlled by pulses applied to the CB1 pin by an external device. The SR counter sets the SR Interrupt flag each time it counts 8 pulses but it does not disable the shifting function. Each time the microprocessor writes or reads the shift register, the SR Interrupt flag is reset and the SR counter is initialized to begin counting the next 8 shift pulses on pin CB1. After 8 shift pulses, the Interrupt flag is set. The microprocessor can then load the shift register with the next byte of data.

Figure H-24: Shift Register Output Modes



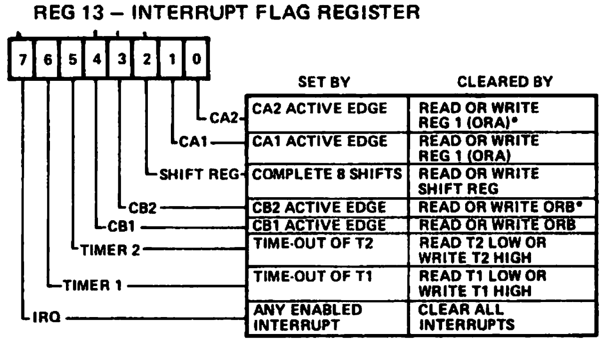
The Interrupt Flag Register (IFR) and Interrupt Enable Register (IER) are depicted in Figures H-25 and H-26, respectively.

The IFT may be read directly by the processor. In addition, individual flag bits may be cleared by writing a "1" into the appropriate bit of the IFR. When the proper chip select and register signals are applied to the chip, the contents of this register are placed on the data bus. Bit 7 indicates the status of the IRQ output. This bit corresponds to the logic function: IRQ = IFR6xIER6 + IFR5x IER5 + IFR4xIER4 + IFR3xIER3 + IFR2xIER2 + IFR1xIER1 + IFR0xIER0. Note: X = logic AND, + = Logic OR.

The IFR bit 7 is not a flag. Therefore, this bit is not directly cleared by writing a logic 1 into it. It can only be cleared by clearing all the flags in the register or by disabling all the active interrupts as discussed in the next section.

Figure H-25: Interrupt Flag Register (IFR)

REG 13 – INTERRUPT FLAG REGISTER



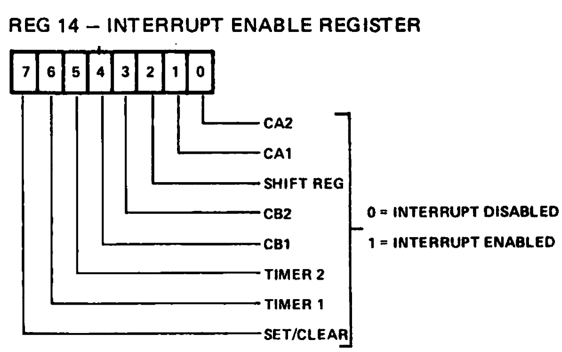
* IF THE CA2/CB2 CONTROL IN THE PCR IS SELECTED AS "INDEPENDENT" INTERRUPT INPUT, THEN READING OR WRITING THE OUTPUT REGISTER ORA/ORB WILL NOT CLEAR THE FLAG BIT. INSTEAD, THE BIT MUST BE CLEARED BY WRITING INTO THE IFR, AS DESCRIBED PREVIOUSLY.

For each Interrupt Flag in IFR, there is a corresponding bit in the Interrupt Enable Register. The system processor can be set or clear selected bits in this register to facilitate controlling individual interrupts without affecting others. This is accomplished by writing to address 1110 (IER address). If bit 7 of the data placed on the system data bus during this write operation is a 0, each 1 in bits 6 through 0 clears the corresponding bit in the Interrupt Enable Register. For each zero in bits 6 through 0, the corresponding bit is unaffected.

Setting selected bits in the Interrupt Enable Register is accomplished by writing to the same address with bit 7 in the data word set to a logic 1. In this case, each 1 in bits 6 through 0 will set the corresponding bit. For each zero, the corresponding bit will be unaffected. This individual control of the setting and clearing operations allows very convenient control of the interrupts during system operation.

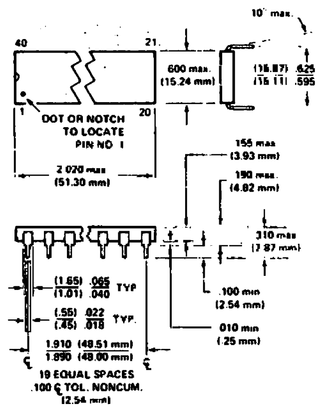
In addition to setting and clearing IER bits, the processor can read the contents of this register by placing the proper address on the register select and chip select inputs with the R/W line high. Bit 7 will be read as a logic 0.

Figure H-26: Interrupt Enable Register (IER)



NOTES:
1. IF BIT 7 IS A "0", THEN EACH "1" IN BITS 0 - 6 DISABLES THE CORRESPONDING INTERRUPT.
2. IF BIT 7 IS A "1", THEN EACH "1" IN BITS 0 - 6 ENABLES THE CORRESPONDING INTERRUPT.
3. IF A READ OF THIS REGISTER IS DONE, BIT 7 WILL BE "0" AND ALL OTHER BITS WILL REFLECT THEIR ENABLE/DISABLE STATE.

PACKAGE OUTLINE

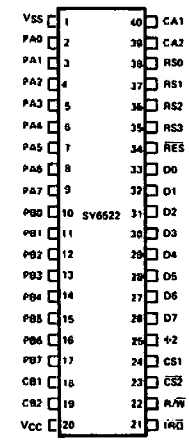


NOTE: Pin No. 1 is in lower left corner when symbolization is in normal orientation

ORDERING INFORMATION

Order Number	Package Type	Frequency Option
SYP 6522	Plastic	1 MHz
SYP 6522A	Plastic	2 MHz
SYC 6522	Ceramic	1 MHz
SYC 6522A	Ceramic	2 MHz

PIN CONFIGURATION



I.1 8086 REGISTER MODEL

AX:	AH	AL	ACCUMULATOR	GENERAL REGISTER FILE
BX:	BH	BL	BASE	
CX:	CH	CL	COUNT	
DX:	DH	DL	DATA	
	SP		STACK POINTER	
	BP		BASE POINTER	
	SI		SOURCE INDEX	
	DI		DESTINATION INDEX	

IP	INSTRUCTION POINTER
$FLAGS_H$ $FLAGS_L$	STATUS FLAGS

CS	CODE SEGMENT	SEGMENT REGISTER FILE
DS	DATA SEGMENT	
SS	STACK SEGMENT	
ES	EXTRA SEGMENT	

Instructions which reference the flag register file as a 16-bit object use the symbol FLAGS to represent the file:

15								7							0
X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF

X = Don't care

AF:	AUXILIARY CARRY—BCD	8080 FLAGS
CF:	CARRY FLAG	
PF:	PARITY FLAG 8080 FLAGS	
SF:	SIGN FLAG	
ZF:	ZERO FLAG	

DF:	DIRECTION FLAG (STRINGS)	8086 FLAGS
IF:	INTERRUPT ENABLE FLAG	
OF:	OVERFLOW FLAG (CF ⊕ SF)	
TF:	TRAP—SINGLE STEP FLAG	

I.2 OPERAND SUMMARY

"REG" FIELD BIT ASSIGNMENTS

16-BIT(W=1)	8-BIT(W=0)	SEGMENT
000 AX	000 AL	00 ES
001 CX	001 CL	01 CS
010 DX	010 DL	10 SS
011 BX	011 BL	11 DS
100 SP	100 AH	
101 BP	101 CH	
110 SI	110 DH	
111 DI	111 BH	

I.3 SECOND INSTRUCTION BYTE SUMMARY

| mod xxx r/m |

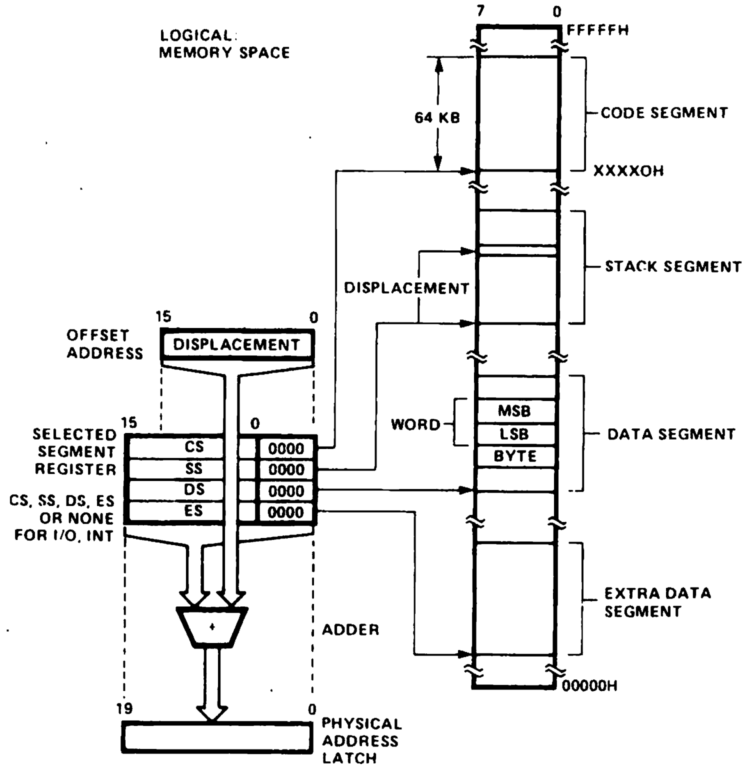
MOD	DISPLACEMENT
00	DISP=0*; disp-low and disp-high are absent
01	DISP=disp-low sign-extended to 16-bits, disp-high is absent
10	DISP=disp-high:disp-low
11	r/m is treated as a "reg" field

R/M	OPERAND ADDRESS	DEFAULT SEGMENT
000	(BX) + (SI) + DISP	DS
001	(BX) + (DI) + DISP	DS
010	(BP) + (SI) + DISP	SS
011	(BP) + (DI) + DISP	SS
100	(SI) + DISP	DS
101	(DI) + DISP	DS
110	(BP) + DISP*	SS
111	(BX) + DISP	DS

DISP follows 2nd byte of instruction (before data if required).
*except if mod=00 and r/m=110; then EA=disp-high: disp-low.

OPERAND ADDRESS (EA) TIMING (CLOCKS):
Add 4 clocks for word operands at ODD ADDRESSES.
Immed offset=6
Base (BX, BP, SI, DI)=5
Base + DISP=9
Base + index (BP + DI, BX + SI)=7
Base + index (BP + SI, BX + DI)=8
Base + index (BP + DI, BX + SI) + DISP=11
Base + index (BP + SI, BX + DI) + DISP=12

I.4 MEMORY SEGMENTATION MODEL



SEGMENT OVERRIDE PREFIX

0	0	1	REG	1	1	0

Timing: 2 clocks

USE OF SEGMENT OVERRIDE

OPERAND REGISTER	DEFAULT	WITH OVERRIDE PREFIX
IP (code address)	CS	Never
SP (stack address)	SS	Never
BP (stack address or stack marker)	SS	BP + DS, or ES, or CS
SI or DI (not incl. strings)	DS	ES, SS, or CS
SI (implicit source addr. for strings)	DS	ES, SS, or CS
DI (implicit dest. addr. for strings)	ES	Never

I.5 INSTRUCTION SET DATA

Section I.5.2 presents instuction set data, grouped by function. Section I.9 provides an alphabetic index to the data.

I.5.1 KEY TO FLAG EFFECTS

The following key refers to the flag sections in the instruction set data in Section I.5.2.

FLAG EFFECT KEY

IDENTIFIER	EXPLANATION
(blank)	Not altered
0	Cleared to 0
1	Set to 1
X	Set or cleared according to result
U	Undefined—contains no reliable value
R	Restored from previously-saved value

I.5.2 DATA TRANSFER

MOV=Move

Flags: O D I T S Z A P C

Register/memory to/from register

1 0 0 0 1 0 d w	mod reg r/m

Timing (clocks): register to register 2
memory to register 8+EA
register to memory 9+EA

Immediate to register/memory

1 1 0 0 0 1 1 w	mod 0 0 0 r/m	data	data if w=1

Timing: 10+EA clocks

Immediate to register

1 0 1 1 w reg	data	data if w=1

Timing: 4 clocks

Memory to accumulator

1 0 1 0 0 0 0 w	addr-low	addr-high

Timing: 10 clocks

Accumulator to memory

1 0 1 0 0 0 1 w	addr-low	addr-high

Timing: 10 clocks

Register/memory to segment register

1 0 0 0 1 1 1 0	mod 0 reg r/m

Timing (clocks): register to register 2
memory to register 8+EA

Segment register to register/memory

1 0 0 0 1 1 0 0	mod 0 reg r/m

Timing (clocks): register to register 2
register to memory 9+EA

PUSH=Push

Flags: O D I T S Z A P C

Register/memory

1 1 1 1 1 1 1 1	mod 1 1 0 r/m

Timing (clocks): register 10
memory 16+EA

0 1 0 1 0 reg

Timing: 10 clocks

Segment register

0 0 0 reg 1 1 0

Timing: 10 clocks

POP=Pop

Flags: O D I T S Z A P C

Register/memory

1 0 0 0 1 1 1 1	mod 0 0 0 r/m

Timing (clocks): register 8
memory 17+EA

Register

0 1 0 1 1 reg

Timing: 8 clocks

Segment register

0 0 0 reg 1 1 1

Timing: 8 clocks

XCHG=Exchange

Flags: O D I T S Z A P C

Register/memory with register

1 0 0 0 0 1 1 w	mod reg r/m

Timing (clocks): register with register 4
memory with register 17+EA

Register with accumulator

1 0 0 1 0 reg

Timing: 3 clocks

IN=Input to AL/AX from

Flags: O D I T S Z A P C

Fixed Port

1 1 1 0 0 1 0 w	port

Timing: 10 clocks

Variable port (DX)

1 1 1 0 1 1 0 w

Timing: 8 clocks

OUT=Output from AL/AX to

Flags: O D I T S Z A P C

Fixed Port

1 1 1 0 0 1 1 w	port

Timing: 10 clocks

Variable port (DX)

1 1 1 0 1 1 1 w

Timing: 8 clocks

XLAT=Translate Byte to AL

Flags: O D I T S Z A P C

1 1 0 1 0 1 1 1

Timing: 11 clocks

LEA=Load EA to Register

Flags: O D I T S Z A P C

1 0 0 0 1 1 0 1	mod reg r/m

Timing: 2+EA clocks

LDS=Load Pointer to DS

Flags: O D I T S Z A P C

1 1 0 0 0 1 0 1	mod reg r/m

Timing: 16+EA clocks

LES=Load Pointer to ES

Flags: O D I T S Z A P C

1 1 0 0 0 1 0 0	mod reg r/m

Timing: 16+EA clocks

LAHF=Load AH with Flags

Flags: O D I T S Z A P C

1 0 0 1 1 1 1 1

Timing: 4 clocks

SAHF=Store AH into Flags

Flags:	O	D	I	T	S	Z	A	P	C
					R	R	R	R	R

1 0 0 1 1 1 1 0

Timing: 4 clocks

PUSHF=Push Flags

Flags: O D I T S Z A P C

1 0 0 1 1 1 0 0

Timing: 10 clocks

POPF=Pop Flags

Flags:	O	D	I	T	S	Z	A	P	C
	R	R	R	R	R	R	R	R	R

1 0 0 1 1 1 0 1

Timing: 8 clocks

I.5.3 ARITHMETIC

ADD=Add

Flags:	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	X

Reg./memory with register to either

0 0 0 0 0 0 d w	mod reg r/m

Timing (clocks): register to register 3
memory to register 9+EA
register to memory 16+EA

Immediate to register/memory

1 0 0 0 0 0 s w	mod 0 0 0 r/m	data	data if s:w=01

Timing (clocks): immediate to register 4
immediate to memory 17+EA

Immediate to accumulator

0 0 0 0 0 1 0 w	data	data if w=1

Timing: 4 clocks

ADC=Add with Carry

Flags:	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	X

Reg./memory with register to either

0 0 0 1 0 0 d w	mod reg r/m

Timing (clocks): register to register 3
memory to register 9+EA
register to memory 16+EA

Immediate to register/memory

1 0 0 0 0 0 s w	mod 0 1 0 r/m	data	data if s:w=01

Timing (clocks): immediate to register 4
immediate to memory 17+EA

Immediate to accumulator

0 0 0 1 0 1 0 w	data	data if w=1

Timing: 4 clocks

INC=Increment

Flags:	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	X

Register/memory

1 1 1 1 1 1 1 w	mod 0 0 0 r/m

Timing (clocks):		
	register	2
	memory	15+EA

Register

0 1 0 0 0 reg

Timing: 2 clocks

AAA=ASCII Adjust for Add

Flags:	O	D	I	T	S	Z	A	P	C
	U				U	U	X	U	X

0 0 1 1 0 1 1 1

Timing: 4 clocks

DAA=Decimal Adjust for Add

Flags:	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	X

0 0 1 0 0 1 1 1

Timing: 4 clocks

SUB=Subtract

Flags:	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	X

0 0 1 0 1 0 d w	mod reg r/m

Timing (clocks):		
	register from register	3
	memory from register	9+EA
	register from memory	16+EZ

Immediate from register/memory

1 0 0 0 0 0 s w	mod 1 0 1 r/m	data	data if s:w=01

Timing (clocks):		
	immediate from register	4
	immediate from memory	17+EA

Immediate from accumulator

0 0 1 0 1 1 0 w	data	data if w=1

Timing: 4 clocks

SBB=Subtract with Borrow

Flags: O D I T S Z A P C
X X X X X X

0 0 0 1 1 0 d w	mod reg r/m

Timing (clocks): register from register 3
memory from register 9+EA
register from memory 16+EA

Immediate from register/memory

1 0 0 0 0 0 s w	mod 0 1 1 r/m	data	data if s:w=01

Timing (clocks): immediate from register 4
immediate from memory 17+EA

Immediate from accumulator

0 0 0 1 1 1 0 w	data	data if w=1

Timing: 4 clocks

DEC=Decrement

Flags: O D I T S Z A P C
X X X X X

Register/memory

1 1 1 1 1 1 1 w	mod 0 0 1 r/m

Timing (clocks): register 2
memory 15+EA

Register

0 1 0 0 1 reg

Timing: 2 clocks

NEG=Change Sign

Flags: O D I T S Z A P C
X X X X X 1*

*0 if destination=0

1 1 1 1 0 1 1 w	mod 0 1 1 r/m

Timing (clocks): register 3
memory 16+EA

CMP=Compare

Flags: O D I T S Z A P C
X X X X X X

Register/memory and register

0 0 1 1 1 0 d w	mod reg r/m

Timing (clocks): register with register 3
memory with register 9+EA
register with memory 9+EA

Immediate with register/memory

1 0 0 0 0 0 s w	mod 1 1 1 r/m	data	data if s:w=01

Timing (clocks): immediate with register 4
immediate with memory 17+EA

Immediate with accumulator

0 0 1 1 1 1 0 w	data	data if w=1

Timing: 4 clocks

AAS=ASCII Adjust for Subtract

Flags:	O	D	I	T	S	Z	A	P	C
	U				U	U	X	U	X

0 0 1 1 1 1 1 1

Timing: 4 clocks

DAS=Decimal Adjust for Subtract

Flags:	O	D	I	T	S	Z	A	P	C
	U				X	X	X	X	X

0 0 1 0 1 1 1 1

Timing: 4 clocks

MUL=Multiply (Unsigned)

Flags:	O	D	I	T	S	Z	A	P	C
	X				U	U	U	U	X

1 1 1 1 0 1 1 w	mod 1 0 0 r/m

Timing (clocks): 8-bit 71+EA
16-bit 124+EA

IMUL=Integer Multiply (Signed)

Flags:	O	D	I	T	S	Z	A	P	C
	X				U	U	U	U	X

1 1 1 1 0 1 1 w	mod 1 0 1 r/m

Timing (clocks): 8-bit 90+EA
16-bit 144+EA

AAM=ASCII Adjust for Multiply

Flags:	O	D	I	T	S	Z	A	P	C
	U				X	X	U	X	U

1 1 0 1 0 1 0 0	0 0 0 0 1 0 1 0

Timing: 83 clocks

DIV=Divide (Unsigned)

Flags:	O	D	I	T	S	Z	A	P	C
	U				U	U	U	U	U

1 1 1 1 0 1 1 w	mod 1 1 0 r/m

Timing (clocks): 8-bit 90+EA
16-bit 155+EA

IDIV=Integer Divide (Signed)

Flags:	O	D	I	T	S	Z	A	P	C
	U				U	U	U	U	U

1 1 1 1 0 1 1 w	mod 1 1 1 r/m

Timing (clocks): 8-bit 112+EA
16-bit 177+EA

AAD=ASCII Adjust for Divide

Flags:	O	D	I	T	S	Z	A	P	C
	U				X	X	U	X	U

1 1 0 1 0 1 0 1	0 0 0 0 1 0 1 0

Timing: 60 clocks

CBW=Convert Byte to Word

Flags: O D I T S Z A P C

1 0 0 1 1 0 0 0

Timing: 2 clocks

CWD=Convert Word to Double Word

Flags: O D I T S Z A P C

1 0 0 1 1 0 0 1

Timing: 5 clocks

I.5.4 LOGIC

NOT=Invert

Flags: O D I T S Z A P C

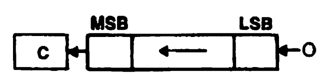
1 1 1 1 0 1 1 w	mod 0 1 0 r/m

Timing (clocks): register 3
memory 16+EA

SHL/SAL=Shift Logical/Arithmetic Left

Flags:	O	D	I	T	S	Z	A	P	C
	X								X

1 1 0 1 0 0 v w	mod 1 0 0 r/m

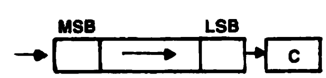


Timing (clocks): single-bit register 2
single-bit memory 15+EA
variable-bit register 8+4/bit
variable-bit memory 20+EA+4/bit

SHR=Shift Logical Right

Flags:	O	D	I	T	S	Z	A	P	C
	X								X

1 1 0 1 0 0 v w	mod 1 0 1 r/m

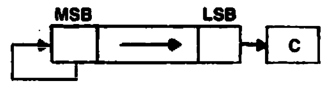


Timing (clocks): single-bit register 2
single-bit memory 15+EA
variable-bit register 8+4/bit
variable-bit memory 20+EA+4/bit

SAR=Shift Arithmetic Right

Flags:	O	D	I	T	S	Z	A	P	C
	X				X	X	U	X	X

1 1 0 1 0 0 v w	mod 1 1 1 r/m

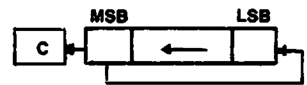


Timing (clocks):		
	single-bit register	2
	single-bit memory	15+EA
	variable-bit register	8+4/bit
	variable-bit memory	20+EA+4/bit

ROL=Rotate Left

Flags:	O	D	I	T	S	Z	A	P	C
	X								X

1 1 0 1 0 0 v w	mod 0 0 0 r/m

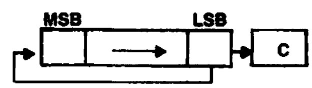


Timing (clocks):		
	single-bit register	2
	single-bit memory	15+EA
	variable-bit register	8+4/bit
	variable-bit memory	20+EA+4/bit

ROR=Rotate Right

Flags:	O	D	I	T	S	Z	A	P	C
	X								X

1 1 0 1 0 0 v w	mod 0 0 1 r/m

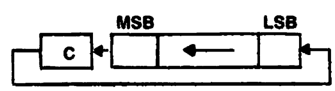


Timing (clocks):		
	single-bit register	2
	single-bit memory	15+EA
	variable-bit register	8+4/bit
	variable-bit memory	20+EA+4/bit

RCL=Rotate Through Carry Left

Flags:	O	D	I	T	S	Z	A	P	C
	X								X

1 1 0 1 0 0 v w	mod 0 1 0 r/m

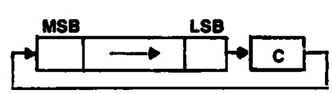


Timing (clocks):		
	single-bit register	2
	single-bit memory	15+EA
	variable-bit register	8+4/bit
	variable-bit memory	20+EA+4/bit

RCR=Rotate Through Carry Right

Flags:	O	D	I	T	S	Z	A	P	C
	X								X

1 1 0 1 0 0 v w	mod 0 1 1 r/m



Timing (clocks):		
	single-bit register	2
	single-bit memory	15+EA
	variable-bit register	8+4/bit
	variable-bit memory	20+EA+4/bit

AND=And

Flags:

O	D	I	T	S	Z	A	P	C
O				X	X	U	X	O

Reg./memory and register to either

0 0 1 0 0 0 d w	mod reg r/m

Timing (clocks): register to register 3
memory to register 9+EA
register to memory 16+EA

Immediate to register/memory

1 0 0 0 0 0 0 w	mod 1 0 0 r/m	data	data if w=1

Timing (clocks): immediate to register 4
immediate to memory 17+EA

Immediate to accumulator

0 0 1 0 0 1 0 w	data	data if w=1

Timing: 4 clocks

TEST=And Function to Flags, No Result

Flags:

O	D	I	T	S	Z	A	P	C
O				X	X	U	X	O

Register/memory and register

1 0 0 0 0 1 0 w	mod reg r/m

Timing (clocks): register to register 3
register with memory 9+EA

Immediate data and register/memory

1 1 1 1 0 1 1 w	mod 0 0 0 r/m	data	data if w=1

Timing (clocks): immediate with register 4
immediate with memory 10+EA

Immediate data and accumulator

1 0 1 0 1 0 0 w	data	data if w=1

Timing: 4 clocks

OR=Or

Flags:

O	D	I	T	S	Z	A	P	C
O				X	X	U	X	O

Reg./memory and register to either

0 0 0 0 1 0 d w	mod reg r/m

Timing (clocks): register to register 3
memory to register 9+EA
register to memory 16+EA

Immediate to register/memory

1 0 0 0 0 0 0 w	mod 0 0 1 r/m	data	data if w=1

Timing (clocks): immediate to register 4
immediate to memory 17+EA

Immediate to accumulator

0 0 0 0 1 1 0 w	data	data if w=1

Timing: 4 clocks

XOR=Exclusive Or

Flags:	O	D	I	T	S	Z	A	P	C
	O				X	X	U	X	O

Reg./memory and register to either

0 0 1 1 0 0 d w	mod reg r/m

Timing (clocks): register to register 3
memory to register 9+EA
register to memory 16+EA

Immediate to register/memory

1 0 0 0 0 0 0 w	mod 1 1 0 r/m	data	data if w=1

Timing (clocks): immediate to register 4
immediate to memory 17+EA

Immediate to accumulator

0 0 1 1 0 1 0 w	data	data if w=1

Timing: 4 clocks

I.5.5 STRING MANIPULATION

REP=Repeat

Flags: O D I T S Z A P C

1 1 1 1 0 0 1 z

Timing: 6 clocks/loop

MOVS=Move String

Flags: O D I T S Z A P C

1 0 1 0 0 1 0 w

Timing: 17 clocks

CMPS=Compare String

Flags:	O	D	I	T	S	Z	A	P	C
	X				X	X	X	X	X

1 0 1 0 0 1 1 w

Timing: 22 clocks

SCAS=Scan String

Flags: O D I T S Z A P C
X X X X X X

1 0 1 0 1 1 1 w

Timing: 15 clocks

LODS=Load String

Flags: O D I T S Z A P C

1 0 1 0 1 1 0 w

Timing: 12 clocks

STOS=Store String

Flags: O D I T S Z A P C

1 0 1 0 1 0 1 w

Timing: 10 clocks

I.5.6 CONTROL TRANSFER

NOTE: Queue reintialization is not included in the timing information for transfer operations. To account for instruction loading, add 8 clocks to timing numbers.

CALL=Call

Flags: O D I T S Z A P C

Direct within segment

1 1 1 0 1 0 0 0	disp-low	disp-high

Timing: 11 clocks

Indirect within segment

1 1 1 1 1 1 1 1	mod 0 1 0 r/m

Timing: 13+EA clocks

Direct intersegment

1 0 0 1 1 0 1 0	offset-low	offset-high
	seg-low	seg-high

Timing: 20 clocks

Indirect intersegment

1 1 1 1 1 1 1 1	mod 0 1 1 r/m

Timing: 29+EA clocks

JMP=Unconditional Jump

Flags: O D I T S Z A P C

Direct within segment

1 1 1 0 1 0 0 1	disp-low	disp-high

Timing: 7 clocks

Direct within segment-short

1 1 1 0 1 0 1 1	disp

Timing: 7 clocks

Indirect within segment

1 1 1 1 1 1 1 1	mod 1 0 0 r/m

Timing: 7+EA clocks

Direct intersegment

1 1 1 0 1 0 1 0	offset-low	offset-high
	seg-low	seg-high

Timing: 7 clocks

Indirect intersegment

1 1 1 1 1 1 1 1	mod 1 0 1 r/m

Timing: 16+EA clocks

RET=Return from CALL

Flags: O D I T S Z A P C

Within segment

1 1 0 0 0 0 1 1

Timing: 8 clocks

Within seg. adding immediate to SP

1 1 0 0 0 0 1 0	data-low	data-high

Timing: 12 clocks

Intersegment

1 1 0 0 1 0 1 1

Timing: 18 clocks

Intersegment, adding immediate to SP

1 1 0 0 1 0 1 0	data-low	data-high

Timing: 17 clocks

JE/JZ=Jump on Equal/Zero

Flags: O D I T S Z A P C

0 1 1 1 0 1 0 0	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JL/JNGE=Jump on Less/Not Greater or Equal

Flags: O D I T S Z A P C

0 1 1 1 1 1 0 0	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JLE/JNG=Jump on Less or Equal/Not Greater

Flags: O D I T S Z A P C

0 1 1 1 1 1 1 0	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JB/JNAE=Jump on Below/Not Above or Equal

Flags: O D I T S Z A P C

0 1 1 1 0 0 1 0	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JBE/JNA=Jump on Below or Equal/Not Above

Flags: O D I T S Z A P C

0 1 1 1 0 1 1 0	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JP/JPE=Jump on Parity/Parity Even

Flags: O D I T S Z A P C

0 1 1 1 1 0 1 0	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JO=Jump on Overflow

Flags: O D I T S Z A P C

0 1 1 1 0 0 0 0	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JS=Jump on Sign

Flags: O D I T S Z A P C

0 1 1 1 1 0 0 0	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JNE/JNZ=Jump on Not Equal/Not Zero

Flags: O D I T S Z A P C

0 1 1 1 0 1 0 1	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JNL/JGE=Jump on Not Less/Greater or Equal

Flags: O D I T S Z A P C

0 1 1 1 1 1 0 1	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JNLE/JG=Jump on Not Less or Equal/Greater

Flags: O D I T S Z A P C

0 1 1 1 1 1 1 1	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JNB/JAE=Jump on Not Below/Above or Equal

Flags: O D I T S Z A P C

0 1 1 1 0 0 1 1	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JNBE/JA=Jump on Not Below or Equal/Above

Flags: O D I T S Z A P C

0 1 1 1 0 1 1 1	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JNP/JPO=Jump on Not Parity/Parity Odd

Flags: O D I T S Z A P C

0 1 1 1 1 0 1 1	disp

Timing (clocks): jump is taken 8
jump is not taken 4

JNO=Jump on Not Overflow

Flags: O D I T S Z A P C

0 1 1 1 0 0 0 1	disp

Timing (clocks): jump is taken 8
jump is not take 4

JNS=Jump on Not Sign

Flags: O D I T S Z A P C

0 1 1 1 1 0 0 1	disp

Timing (clocks): jump is taken 8
jump is not taken 4

LOOP=Loop CX Times

Flags: O D I T S Z A P C

1 1 1 0 0 0 1 0	disp

Timing (clocks): jump is taken 9
jump is not taken 5

LOOPZ/LOOPE=Loop While Zero/Equal

Flags: O D I T S Z A P C

1 1 1 0 0 0 0 1	disp

Timing (clocks): jump is taken 11
jump is not taken 5

LOOPNZ/LOOPNE=Loop While Not Zero/Not Equal

Flags: O D I T S Z A P C

1 1 1 0 0 0 0 0	disp

Timing (clocks): jump is taken 11
jump is not taken 5

JCXZ=Jump on CX Zero

Flags: O D I T S Z A P C

1 1 1 0 0 0 1 1	disp

Timing (clocks): jump is taken 9
jump is not taken 5

8086 Conditional Transfer Operations

INSTRUCTION	CONDITION	INTERPRETATION
JE or JZ	ZF=1	"equal" or "zero"
JL or JNGE	(SR xor OF)=1	"less" or "not greater or equal"
JLE or JNG	((SP xor OF) or ZF)=1	"less or equal" or "not greater"
JB or JNAE	CF=1	"below" or "not above or equal"
JBE or JNA	(CF or ZF)=1	"below or equal" or "not above"
JP or JPE	PF=1	"parity" or "parity even"
JO	OF=1	"overflow"
JS	SF=1	"sign"
JNE or JNZ	ZF=0	"not equal" or "not zero"
JNL or JGE	(SF xor OF)=0	"not less" or "greater or equal"
JNLE or JG	((SF xor OF) or ZF)=0	"not less or equal" or "greater"
JNB or JAE	CF=0	"not below" or "above or equal"
JNBE or JA	(CF or ZF)=0	"not below or equal" or "above"
JNP or JPO	PF=0	"not parity" or "parity odd"
JNO	OF=0	"not overflow"
JNS	OF=0	"not sign"

NOTE: "Above and below" refer to the relation between two unsigned values, while "greater" and "less" refer to the relation between two signed values.

INT=Interrupt

Flags: O D I T S Z A P C
O O

Type specified

1 1 0 0 1 1 0 1	type

Timing: 50 clocks

Type 3

`11001100`

Timing: 51 clocks

INTO=Interrupt on Overflow

Flags:	O	D	I	T	S	Z	A	P	C
			O	O					

`11001110`

Timing: 52 clocks if pass   4 clocks if fail

IRET=Interrupt Return

Flags:	O	D	I	T	S	Z	A	P	C
	R	R	R	R	R	R	R	R	R

`11001111`

Timing: 24 clocks

I.5.7 PROCESSOR CONTROL

CLC=Clear Carry

Flags:	O	D	I	T	S	Z	A	P	C
									O

`11111000`

Timing: 2 clocks

STC=Set Carry

Flags:	O	D	I	T	S	Z	A	P	C
									1

`11111001`

Timing: 2 clocks

CMC=Complement Carry

Flags:	O	D	I	T	S	Z	A	P	C
									X

`11110101`

Timing: 2 clocks

NOP=No Operation

Flags:	O	D	I	T	S	Z	A	P	C

`10010000`

Timing: 3 clocks

CLD=Clear Direction

Flags:	O	D	I	T	S	Z	A	P	C
		O							

`11111100`

Timing: 2 clocks

STD=Set Direction

Flags: O D I T S Z A P C
1

1 1 1 1 1 1 0 1

Timing: 2 clocks

CLI=Clear Interrupt

Flags: O D I T S Z A P C
O

1 1 1 1 1 0 1 0

Timing: 2 clocks

STI=Set Interrupt

Flags: O D I T S Z A P C
1

1 1 1 1 1 0 1 1

Timing: 2 clocks

HLT=Halt

Flags: O D I T S Z A P C

1 1 1 1 0 1 0 0

Timing: 2 clocks

WAIT=Wait

Flags: O D I T S Z A P C

1 0 0 1 1 0 1 1

Timing: 3 clocks

LOCK=Bus Lock Prefix

Flags: O D I T S Z A P C

1 1 1 1 0 0 0 0

Timing: 2 clocks

ESC=Escape (To External Device)

Flags: O D I T S Z A P C

1 1 0 1 1 x x x	mod x x x r/m

Timing: 7+EA clocks

NOTES:
If d=1 then "to"; if d=0 then "from."
If w=1 then word instruction; if w=0 then byte instruction.
If s:w=01 then 16 bits of immediate data form the operand.
If s:w=11 then an immediate data byte is sign extended to form the 16-bit operand.
If v=0 then "count"=1; if v=1 then "count" in (CL).
X=don't care.
Z is used for some string primitives to compare with ZF FLAG.
AL=8-bit accumulator.
AX=16-bit accumulator.
CX=Count register.
DS=Data segment.
DX=Variable port register.
ES=Extra segment.
Above/below refers to unsigned value.
Greater=more positive signed values.
Less=less positive (more negative) signed values.
See section I.2 for Operand summary.
See section I.4 for Segment Override summary.

I.6 PROCESSOR RESET REGISTER INITIALIZATION

Flags=0000H (to disable interrupts and single-stepping)

CS=FFFFH (to begin execution at FFFF0H)
IP=0000H

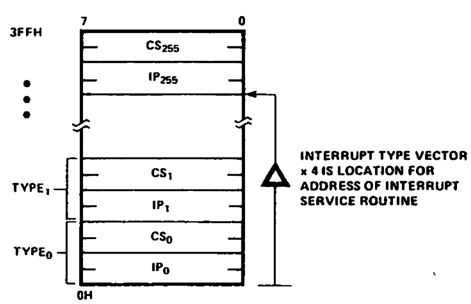
DS=0000H
SS=0000H
ES=0000H

No other registers are acted upon during reset.

I.7 8088 RESERVED LOCATIONS

INTERRUPT	LOCATION	FUNCTION
0	00H-03H	Divide by zero
1	04H-07H	Single step
2	08H-0BH	Non-maskable interrupt
3	0CH-0FH	One-byte interrupt instruction
4	10H-13H	Interrupt on overflow

Interrupt Pointer Table



All mnemonics © Intel Corporation 1981.

I.8 8088 INSTRUCTION SET MATRIX

NOTES:

b=byte operation	m=memory
d=direct	r/m=EA is second byte
f=from CPU reg	si=short intrasegment
i=immediate	sr=segment register
ia=immed. to accum.	t=to CPU reg
id=indirect	v=variable
is=immed. byte, sign ext.	w=word option
l=long ie. intersegment	z=zero

I.9 MNEMONIC INDEX

```
PL/M-86 COMPILER     SIRIUS Systems Technology, Inc. (c) 1982  S-1 Hardware       04/01/82          PAGE   1
                     Example software drivers for S-1 Hardware


SERIES-III PL/M-86 V1.0 COMPILATION OF MODULE HARDWARE
NO OBJECT MODULE REQUESTED
COMPILER INVOKED BY:  P.86 TEMP.SRC OPTIMIZE(3) PAGELENGTH(42) PAGEWIDTH(109) PRINT(:F4:HW.LS) NOOBJECT


            $TITLE                                    (c) 1982
            $SUBTITLE ('Example software drivers for S-1 Hardware')
             /***********************************************************************
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              *                                                                      *
              ************************************************************************/


PL/M-86 COMPILER                                                                  04/01/82          PAGE   2
                     Example software drivers for S-1 Hardware


            $eject
            $SMALL ROM

  1         Hardware: do;

  2   1       Declare dcl       literally 'declare';
  3   1      Dcl     lit       literally 'literally';
  4   1       Dcl        addr      lit       'address',
                         ext       lit       'external',
                         init      lit       'initial',
                         intg      lit       'integer',
                         proc      lit       'procedure',
                         ptr       lit       'pointer',
                         pub       lit       'public',
                         rent      lit       'reentrant',
                         ret       lit       'return',
                         struc     lit       'structure',
                         boolean   lit       'byte',
                         true      lit       '0FFH',
                         false     lit       '0000H';


PL/M-86 COMPILER                                                                  04/01/82          PAGE   3
                     KB: Hardware bit defs


            $subtitle('KB: Hardware bit defs')


  5   1     dcl SR$intbit               lit '4';   /* KB shift register interrupt mask in6522 IER/IFR */
  6   1     dcl SR$enable               lit '0ch';           /* KB shift register enable in 6522 ACR */
  7   1     dcl CB1$intbit              lit '10h';      /* KB RDY edge-sense interrupt mask 6522 PCR */
  8   1     dcl CB1$pos_edge            lit '10h';          /* KB RDY edge-sense control in 6522 PCR */

  9   1     dcl kb$databit              lit '40h';           /* KB DATA level                          */
 10   1     dcl kb$ackctl               lit '2';             /* KB ACK control for 6522 output         */
 11   1     dcl kb$TIMEOUT              lit '300';           /* error timeout in milliseconds          */
 12   1     dcl timer1_ena              lit '0c0h';        /* timer 1 interrupt mask in 6522 IER/IFR */
```

PL/M-86 COMPILER 04/01/82 PAGE 4
KB: Hardware bit defs

```
            $eject

            /*  KYBRD PORT (e8040..e804f)  */

 13   1       dcl via(16)  struc(                        /* 6522 port organization           */

                                RB      byte,
                                RA      byte,
                                DDRB    byte,
                                DDRA    byte,
                                TIMER1  word,
                                TIMER1L word,
                                TIMER2  word,
                                SR      byte,
                                ACR     byte,
                                PCR     byte,
                                IFR     byte,
                                IER     byte,
                                RAX     byte) at(0e8000h);

 14   1     dcl kb$state            byte;               /* current state of keyboard stateware */
 15   1     dcl kb$data             byte;               /* constructed data from keyboard      */

                                               /* nybble convert table for inverted shift reg */
 16   1     dcl Ctable(*) byte data (0,8,4,0ch, 2,0ah,6,0eh, 1,9,5,0dh, 3,0bh,7,0fh);

 17   1     dcl tick                lit '50';           /* console clock rate in milliseconds  */
```

PL/M-86 COMPILER 04/01/82 PAGE 5
KB: external routines

```
            $subtitle('KB: external routines')

              /*
               *    signal user about keyboard error state -- ring bell
               */
 18   1       dcl signal$KB$error lit 'Ringbell';
                 /* Ringbell found in SOUND module */

              /*
               *    Process key board event -- in external module
               */
 19   1     Process$Event: proc(event) byte ext;
 20   2       dcl event byte;
 21   2     end;

              /*
               *   Software clock resource -- set timeout for interrupt to KB$reset
               */

 22   1     set$KB$clock: proc(Period) ext;
 23   2       dcl Period intg;                          /* timeout delay in milliseconds       */
 24   2     end set$KB$clock;
```

PL/M-86 COMPILER SIRIUS Systems Technology, Inc. (c) 1982 S-1 Hardware 04/01/82 PAGE 6
KB: Keyboard Stateware

```
            $subtitle('KB: Keyboard Stateware')
              /*
               *    KB interrupt entry  (level 6)
               */
 25   1     kb$irq: proc pub rent;
 26   2        do case kb$state;
              /*
               *    state 0 to state 1:  shift register (full) interrupt
               */
 27   3       kbst0: do;
 28   4             via(4).ACR= via(4).ACR and not SR$enable; /* disable shift register              */
                                              /* prepare for interrupt on negative edge of KB RDY */
 29   4             via(4).PCR= via(4).PCR and not CB1$pos_edge;
 30   4             via(4).IER= 80h or CB1$intbit;
 31   4            disable;                                 /* time critical section               */
 32   4             kb$data = via(4).SR;                    /* get KB data from SR (clears SR IRQ) */
```

```
33   4           via(4).IER= SR$intbit;                  /* disable SR interrupt                  */
                                                          /* assert KB ACK control on interrupt    */
34   4           via(4).RB = via(4).RB or kb$ackctl;     /* (CB1 IRQ is reset)                    */
35   4          enable;                                  /* end of critical section               */
36   4           kb$state = 1;                           /* set to state 1                        */
37   4         end;
             /*
              *    state 1 to state 2:  interrupt from negative edge on KB$RDY
              */
38   3       kbst1: do;
39   4          disable;                                 /* time critical section                 */
40   4           if (via(4).RA and kb$databit) <> 0 then  /* if data bit is not low then          */
41   4              call kb$error;                       /* stop bit error has occurred           */
42   4            else do;                 /* prepare for interrupt on positive edge of KB RDY */
43   5              via(4).PCR= via(4).PCR or CB1$pos_edge;
                                                          /* release KB ACK control on interrupt  */
44   5              via(4).RB = via(4).RB and not kb$ackctl; /* (CB1 IRQ is reset)                */
45   5              kb$state = 2;                        /* set to state 2                        */
46   5            end;
47   4          enable;                                  /* end of critical section               */
48   4         end;
```

PL/M-86 COMPILER   KB: Keyboard Stateware   04/01/82   PAGE 7

```
           $eject

             /*
              * state 2 to state 0:  interrupt from positive edge on KB$RDY
              */
49   3       kbst2: do;
50   4           if (via(4).RA and kb$databit) = 0 then  /* if data bit is low then               */
51   4              call kb$error;                       /* stop bit error has occurred           */
52   4            else do;
53   5              call kb$reset;                     /*reset hardware/software for next event */
                               /* call event processing routine with order of bits reversed to */
                               /*   reflect physical key number and event type (open or close) */
54   5              if not Process$Event( shl(Ctable(kb$data and 0fh),4)
                          or Ctable(shr(kb$data,4)) ) then
55   5                    call signal$KB$error;         /* signal error in event process        */
56   5            end;
57   4         end;

58   3       end;

59   2     end kb$irq;
```

PL/M-86 COMPILER   KB: Keyboard support routines   04/01/82   PAGE 8

```
           $subtitle('KB: Keyboard support routines')

60   1     kb$reset:  proc rent;                      /* puts KB hardware/software into state 0 */
61   2       dcl dummy byte;

62   2         via(4).IER = CB1$intbit;                  /* clear CB1 interrupts                  */
63   2         via(4).RB  = via(4).RB and not kb$ackctl; /* release kb$ack                        */
64   2         via(4).ACR = via(4).ACR or SR$enable;     /* enable shft reg                       */
65   2         dummy      = via(4).SR;                   /* clr any pending irq                   */
66   2         via(4).IER = 80h or SR$intbit;            /* enable sr interrupts                  */
67   2         kb$state  = 0;                            /* init keybrd state                     */
68   2         call set$KB$clock(0);                     /* clear timeout counter                 */
69   2     end kb$reset;

70   1     kb$error:  proc rent;
71   2         via(4).RB  = via(4).RB or kb$ackctl;      /* force kb$ack high                     */
72   2         via(4).IER = 7fh;                         /* allow no interrupts                   */
73   2         call set$KB$clock(kb$TIMEOUT);            /* time out keyboard                     */
74   2     end kb$error;

75   1     kb$init:  proc  pub rent;

76   2         via(4).RB   = via(4).RB and (0FFh-3);
77   2         via(4).DDRA = via(4).DDRA and not kb$databit;
78   2         via(4).DDRB = via(4).DDRB or kb$ackctl;
79   2         via(4).IER  = 7fh;
80   2         via(4).PCR  = 0;
81   2         via(4).ACR  = 0;
```

```
 82   2           via(2).ACR= (via(2).ACR and 0c0h) or 40h;
 83   2           via(2).timer1L= tick*1000;
 84   2           via(2).IER  = timer1_ena and 7fh;
 85   2           call kb$reset;
 86   2         end kb$init;
```

```
PL/M-86 COMPILER    SIRIUS Systems Technology, Inc. (c) 1982  S-1 Hardware       04/01/82          PAGE   9
                    CRTreg: controller chip registers


           $SUBTITLE ('CRTreg: controller chip registers')

 87   1    DCL   CRT$0     byte   AT (0E8000H);            /* CRT-chip address register          */
 88   1    DCL   CRT$1     BYTE   AT (0E8001H);            /* CRT-chip internal register port    */


           /*
            *      Set CRT register
            */
 89   1    set$CRT$reg: proc (reg,value) rent;
 90   2      dcl reg byte;
 91   2      dcl value byte;
 92   2        CRT$0= reg;                                 /* select register                    */
 93   2        CRT$1= value;                               /* set data                           */
 94   2      end set$CRT$reg;
```

```
PL/M-86 COMPILER                                                                 04/01/82          PAGE  10
                    CRTreg: cursor-display mode control


           $SUBTITLE ('CRTreg: cursor-display mode control')

 95   1    dcl   rast$start  lit    '10';            /* CRT reg: cursor-start & cursor-display mode */


 96   1     DCL   Cursor$PAR    BYTE; /* VAR: contents for CRT cursor-start raster & cursor display mode */
 97   1     dcl   blink$on      boolean;                     /* FLAG:  =0  Blinking cursor on (fast) */
 98   1     dcl   curs$off      boolean;                     /* FLAG: [|0  Cursor off                */


           /*
            *      Set cursor to current Cursor parameter byte.
            */
 99   1    set$cursor: proc rent;
100   2        call set$CRT$reg(rast$start,Cursor$PAR);     /* set raster start reg                */
101   2      end set$cursor;

           /*
            *      Set block cursor.
            */
102   1    BLOCK$CRS:PROC  RENT;
103   2        Cursor$PAR = Cursor$PAR AND 0E0h;            /* set block cursor                    */
104   2        call set$cursor;                             /* set cursor mode reg                 */
105   2      END BLOCK$CRS;

           /*
            *      Set underscore cursor.
            */
106   1    UNDERSCORE$CRS:PROC RENT;
107   2        Cursor$PAR = 00Fh OR (Cursor$PAR AND 0E0h);  /* set underscore cursor               */
108   2        call set$cursor;                             /* set cursor mode reg                 */
109   2      END UNDERSCORE$CRS;
```

```
PL/M-86 COMPILER                                                                 04/01/82          PAGE  11
                    CRTreg: cursor-display mode control


           $eject
           /*
            *    Return cursor to previous modes: block or underline, steady or flashing
            */
110   1    CURSOR$ON:PROC  RENT;
111   2        curs$off= false;                             /* reset cursor off flag               */
112   2        if blink$on then Cursor$par= Cursor$par or 060h; /* set to flashing mode            */
114   2         else Cursor$par= Cursor$par and 01Fh;       /* set to steady mode                  */
115   2        call set$cursor;                             /* set cursor mode reg                 */
116   2      END CURSOR$ON;
```

```
              /*
               *      Turn cursor off.
               */
117   1       CURSOR$OFF:PROC  RENT;
118   2           curs$off= true;                          /* set cursor off flag                   */
119   2           Cursor$PAR = 020h OR (Cursor$PAR AND 01Fh);  /* set to off mode                  */
120   2           call set$cursor;                         /* set cursor mode reg                   */
121   2         END CURSOR$OFF;

              /*
               *      Set cursor blinking.
               */
122   1       CRS$BLINK$ON:PROC  RENT;
123   2           blink$on= true;                          /* set blinking on flag                  */
124   2           if not curs$off then Cursor$PAR= 060h OR Cursor$PAR; /* set flashing,if not off   */
126   2           call set$cursor;                         /* set cursor mode reg                   */
127   2        END CRS$BLINK$ON;
              /*
               *      Set cursor steady.
               */
128   1       CRS$BLINK$OFF:PROC  RENT;
129   2           blink$on= false;                         /* reset blinking on flag                */
130   2           if not curs$off then Cursor$PAR= 01Fh and Cursor$PAR; /* set steady,if not off    */
132   2           call set$cursor;                         /* set cursor mode reg                   */
133   2         END CRS$BLINK$OFF;


PL/M-86 COMPILER                                                   04/01/82           PAGE  12
                   CRTreg: Cursor positioning

              $SUBTITLE ('CRTreg: Cursor positioning')


134   1       dcl    cursaddrH    lit    '14'; /* CRT reg: MSByte of cursor location word, bits: xx54$3210 */
135   1       dcl    cursaddrL    lit    '15';                 /* CRT reg: LSByte of cursor location word */


                /*
                 *   Position Cursor to Absolute Font Cell number
                 *     and display bank
                 */

136   1       POS$Cursor: proc (Cell$number) pub rent;
137   2         dcl Cell$Number word;                        /* Absolute Font Cell Number & diplay bank */
138   2           call set$CRT$reg (cursaddrL, low(Cell$number));
139   2           call set$CRT$reg (cursaddrH, high(Cell$number));
140   2         end POS$Cursor;


PL/M-86 COMPILER                                                   04/01/82           PAGE  13
                   CRT: video contrast & brightness

              $SUBTITLE ('CRT: video contrast & brightness')

141   1       DCL   CBctrl    BYTE    AT (0E8040H);          /* Contrast & Brightness control register */
                                                             /* bits: CCCB$BB--                        */

              /*
               *      Raise video contrast one level.
               */
142   1       contrast$up: proc rent;
143   2         dcl a byte;
144   2         if (a:= (CBctrl + 20h) and 0E0h) [| 0 then    /* add & check upper limit               */
145   2           CBctrl= (CBctrl and 01FH) or a;           /* set contrast, bits: 765               */
146   2       end contrast$up;
              /*
               *      Lower video contrast one level.
               */
147   1       contrast$down: proc rent;
148   2         dcl a byte;
149   2         if (a:= (CBctrl - 20h) and 0E0h) [| 0E0h then  /* sub & check lower limit              */
150   2           CBctrl= (CBctrl and 01FH) or a;           /* set contrast, bits: 765               */
151   2       end contrast$down;

              /*
               *      Raise video brightness one level.
               */
152   1       bright$up: proc rent;
153   2         dcl a byte;
154   2         if (a:= (CBctrl + 4) and 01CH) [| 0 then      /* add & check upper limit               */
155   2           CBctrl= (CBctrl and 0E3H) or a;           /* set brightness, bits: 432             */
156   2       end bright$up;
```

```
              /*
               *       Lower video brightness one level.
               */
157   1       bright$down: proc rent;
158   2         dcl a byte;
159   2         if (a:= (CBctrl - 4) and 01Ch) [| 01Ch then      /* sub & check lower limit          */
160   2           CBctrl= (CBctrl and 0E3H) or a;                 /* set brightness, bits: 432        */
161   2       end bright$down;
```

```
PL/M-86 COMPILER                                                        04/01/82        PAGE  14
                    CRT: display RAM/Font Cells

              $SUBTITLE ('CRT: display RAM/Font Cells')

162   1       dcl   screen$ram word at (0F0000h);             /* memory address of display RAM      */
163   1       dcl   screen$addr  ptr;                        /* display ram pointer, base of word ARRAY */
164   1       DCL   SCREEN based screen$addr (2000) word;     /* ARRAY of Font Cell Pointers        */

                /*
                 *    Screen Buffer Word variables
                 */

165   1       dcl   char$mode     word      pub;              /* CRT attribute bits: 7654$3---      */

166   1       dcl   char$base     word      pub;              /* CRT Font Cell Pointer base for     */
                                                              /*     ASCII symbol index             */

167   1        DCL   REVBIT       LIT    '8000H';
168   1        DCL   BGBIT        LIT    '4000H';
169   1        DCL   UNDBIT       LIT    '2000H';
170   1        dcl   INVBIT       lit    '1000h';
171   1        dcl   extraBIT     lit    '0800h';


                 /*
                  *  Display symbol from character set (typically ASCII)
                  *     at absolute Font Cell number
                  *      (typically: [line| * [display width| + [column| )
                  *     with current Cursor & Display modes.
                  */

172   1       Display$symbol: proc (Symbol$code,Cell$number) pub rent;
173   2         dcl Symbol$code byte;                          /* Symbol print code                  */
174   2         dcl Cell$Number word;                          /* Absolute Font Cell Number          */
175   2           screen(Cell$Number)= (Symbol$code + char$base) OR char$mode;
176   2         end Display$symbol;
```

```
PL/M-86 COMPILER                                                        04/01/82        PAGE  15
                    CRT hardware initialization

              $SUBTITLE ('CRT hardware initialization')
                                                              /* COMMENT THIS !!!!                  */
177   1       DCL CRT$config (*) BYTE DATA (92,80, 81,0CFh, 25,6, 25,25, 3,14, 0,15, 0,0, 0,0);


178   1       CRT$Init:  PROC;
179   2         DCL  I  BYTE;

180   2         screen$addr= @screen$ram;

181   2         char$mode= BGBIT;
182   2         char$base= 20;

183   2         curs$off= false;
184   2         blink$on= false;
185   2         Cursor$PAR= 0;

186   2         DO I=0 TO 0FH;
187   3           CALL SET$CRT$REG (I,(CRTconfig(I)));
188   3           END;

189   2         END CRT$Init;
```

```
PL/M-86 COMPILER                                                          04/01/82          PAGE  16
                    SOUND variables & hardware defs

              $SUBTITLE ('SOUND variables & hardware defs')


 190   1      dcl bell$freq LIT    '76';                       /* period of bell tone: frequency= 14.9KHz */

 191   1      dcl     codec$clk         word at (0E8084h);          /* TIMER1: codec clock frequency         */
 192   1      dcl     codec$ctl         byte at (0E808Bh);          /* ACR: codec clock control register     */
 193   1      dcl     codec$sda         word at (0E8060h);          /*                                        */
 194   1      dcl     volume            byte at (0E802Ah);          /* SR: volume shift-register             */
 195   1      dcl     vol$ctl           byte at (0E802Bh);          /* ACR: SR control register              */
 196   1      dcl     vol$clk           word at (0E8028h);          /* TIMER2: volume SR clock               */

 197   1      dcl bell$on   byte;                                   /* FLAG: bell sound presently active     */
 198   1      dcl vol$level byte;     /*current volume level (nine levels: 0 --| 8)
                                                                  /* volume shift pattern lookup table     */
 199   1      dcl vol$table (*) byte data (0FFh,7FH,3FH,1FH,0FH,7,3,1,0);



PL/M-86 COMPILER                                                          04/01/82          PAGE  17
                    SOUND: Bell control

              $SUBTITLE ('SOUND: Bell control')

                /*
                 *   Software clock resource -- set timeout for interrupt to Bell$clock
                 */

 200   1      set$BELL$clock: proc (Period) ext;
 201   2        dcl Period intg;                                  /* timeout delay in milliseconds         */
 202   2        end set$BELL$clock;

                /*
                 *    CODEC Hardware reset
                 */

 203   1      Bell$init: proc pub rent;
 204   2         vol$level= length(vol$table)-2;                  /* set initial volume level near max     */
 205   2         call Bell$clock;                                /* set hardware to a known & quiet state */
 206   2        end Bell$init;



PL/M-86 COMPILER                                                          04/01/82          PAGE  18
                    SOUND: Bell control

              $eject

 207   1      Bell$clock:      proc  pub rent;

 208   2         codec$ctl = codec$ctl and not 0C0h;               /* disable codec clock                   */

 209   2         codec$sda = 5E00h;                               /* initialize codec SDA to input mode... */
 210   2         codec$sda = 0D40h;                                /* ... to reduce extraneous noise        */
 211   2         codec$sda = 0AA80h;
 212   2         codec$sda = 00C0h;

 213   2         vol$ctl = (vol$ctl and not 3Ch) or 10h;          /* set SR & T2 volume register modes     */
 214   2         vol$clk = 1;                            /* volume clock frequency set beyond perception */
 215   2         volume = vol$table(vol$level);                    /* set volume to current level           */
 216   2         bell$on = false;                                  /* set bell state to off                 */

 217   2         end bell$clock;

 218   1      Ring$bell: proc pub rent;
 219   2          if not bell$on then do;                          /* start bell if sound is off            */
 221   3          call bell$clock;                                  /* init codec hardware on every bell    */
 222   3          codec$sda = 0f80h;                               /* set output waveform to 4 up & 4 down, */
                                                                  /*    a low amplitude triangle wave.    */
 223   3          codec$ctl = codec$ctl or 0c0h;                     /* set codec clock to free run          */
 224   3          codec$clk = bell$freq;                             /* set audio pitch frequency            */
 225   3          bell$on = true;                                    /* set bell state on                    */
 226   3            end;
 227   2          call set$bell$clock(100);                        /* turn off bell in 100 milliseconds     */
 228   2       end;
```

```
PL/M-86 COMPILER                                                    04/01/82        PAGE  19
                  SOUND: volume control

              $SUBTITLE ('SOUND: volume control')
              /*
               *     Raise CODEC volume one level.
               */
 229   1      volume$up: proc rent;
 230   2          if vol$level |= length(vol$table)-1 then      /* check upper limit          */
 231   2              vol$level= length(vol$table)-1;           /* set to max volume          */
 232   2           else vol$level= vol$level+1;                 /* bump level up by one       */
 233   2              volume= vol$table(vol$level);             /* set volume register        */
 234   2       end volume$up;

              /*
               *     Lower CODEC volume one level.
               */
 235   1      volume$down: proc rent;
 236   2          if vol$level |= length(vol$table)-1 then      /* check upper limit          */
 237   2              vol$level= length(vol$table)-2;           /* set to max volume-1        */
           else
 238   2              if vol$level[|0 then vol$level= vol$level-1; /* drop level by one       */
          volume= vol$table(vol$level);                   /* set volume register        */
 241   2      end volume$down;


PL/M-86 COMPILER                                                    04/01/82        PAGE  20
                  SIO: Serial I/O dvrs for TTY: and UL1:

              $subtitle('SIO: Serial I/O dvrs for TTY: and UL1:')

              /*ctr device dcls*/
 242   1      dcl sioctr struc
                  (adata byte,
                   bdata byte,
                   xxx   byte,
                   ctrctl byte) at (0E0020h);
              /*sio device dcls*/
 243   1      dcl siodev struc
                  (adata byte,
                   bdata byte,
                   actl  byte,
                   bctl  byte) at(0E0040h);
 244   1      dcl rx$avail literally '1',
                  tx$empty literally '4';

 245   1      dcl serial_params struc
                  (actrlsb byte,                          /*LSByte of chan a.'s baud rate      */
                   actrmsb byte,                          /*MSByte ...                         */
                   bctrlsb byte,                          /*LSByte of chan b.'s baud rate      */
                   bctrmsb byte,                          /*MSByte ...                         */
                     /* if [baud| then lsb = ??h  msb = ??h 1.25Mhz/([baud|*16)
                           50 ===|        1Ah        06h     50.00  -0-    (min.tol.dist.43.75%)
                           75 ===|        11h        04h     75.00  -0-    (    "        43.75%)
                          110 ===|        C6h        02h    110.00  -0-    (    "        43.75%)
                          134.5 ===|      44h        02h    134.00  -0.37% (    "        40.23%)
                          150 ===|        08h        02h    150.00  -0-    (    "        43.75%)
                          200 ===|        86h        01h    200.00  -0-    (    "        43.75%)

                          300 ===|        04h        01h    300.00  -0-    (    "        43.75%)

                          600 ===|        82h        00h    600.00  -0-    (    "        43.75%)
                         1.2k ===|        41h        00h   1201.00  +0.08% (    "        42.99%)


PL/M-86 COMPILER                                                    04/01/82        PAGE  21
                  SIO: Serial I/O dvrs for TTY: and UL1:

              $eject

              ------------------------------------------------
                                          2Ch        00h   1775.00  -1.39% (    "        30.54%)
                         1.8k ===|        2Bh        00h   1816.00  +0.09% (    "        42.88%)
              ------------------------------------------------
                                          28h        00h   1953.00  -2.36% (    "        21.33)
                         2.0k ===|        27h        00h   2003.00  +0.15% (    "        42.32)
              ------------------------------------------------
                         2.4k ===|        21h        00h   2367.00  -1.38% (    "        30.64%)
                                          20h        00h   2441.00  +1.71% (    "        27.51%)
              ------------------------------------------------
```

```
                3.6k ===|          16h          00h   3551.00  -1.36% (        "        30.83%)
                                   15h          00h   3720.00  +3.33% (        "        12.4%)
-------------------------------------------------
                                   11h          00h   4595.00  -4.27% (        "        3.185%)
                4.8k ===|          10h          00h   4882.00  +1.02% (        "        34.06%)
-------------------------------------------------
                                   09h          00h   8680.55  -9.58% (DISTORTED)
                9.6k ===|          08h          00h   9765.56  +1.73% (min.tol.dist.27.32%)
-------------------------------------------------
                                   06h          00h  13020.83  -9.58% (DISTORTED)
                                   05h          00h  15625.00  +8.51% (DISTORTED)
-------------------------------------------------
                                   05h          00h  15625.00 -18.62% of 19.2k (DISTORTED)
               19.2k ===|          04h          00h  19531.25  +1.02% (min.tol.dist.34.06%)

min.tol.dist. figure assumes no channel noise effects.
  NOTE: possible noise DOES NOT includes bias distorition
        caused by various cable capacitance effects*/
```



```
$eject

     cr2a    byte,      /*bus interface option: 10h if baud a |= baud b
                                                14h if baud a | baud b*/

     cr4a    byte,
     cr4b    byte,
       /*cr4x (16x)$54$(stops)$(even)$(parenb) = 4?h
              01    00 ss      e      p
                           ss = 01 1 stop
                              = 10 1.5 stop
                              = 11 2 stop
                                   e = 1 even
                                   e = 0 odd, byte transparent
                                          p = 1 even or odd
                                          p = 0 byte transparent*/

     cr3a    byte,
     cr3b    byte,
       /*cr3x (rbits)$(autoenb)$4$3$2$1$(renb) = ?1h
              bb      1       0 0 0 0  1
              bb = 11 byte transparent cr3x = E1h
                 = 01 even,odd         cr3x = 61h*/

     cr5a    byte,
     cr5b    byte) EXT;
       /*cr5x (dtr)$(tbits)$(br)$(tenb)$2$(rts)$0 = ?Ah
              1     bb     0    1     0  1    0
                    bb = 11 space,mark cr5x = EAh
                    bb = 01 even,odd,no cr5x = AAh*/
```



```
            $subtitle('SIO: Serial I/O dvrs for port A -- TTY$INSTAT & TTY$STAT')

 246   1    TTY$in$stat:proc boolean PUB;

 247   2    if ( (siodev.actl AND rx$avail) [| 0)
            then return(true);
 249   2    return(false);

 250   2    end TTY$in$stat;


 251   1    TTY$stat:proc boolean PUB;

 252   2    if ( (siodev.actl AND tx$empty) = 0)
            then return(true);
 254   2    return(false);

 255   2    end TTY$stat;
```

```
PL/M-86 COMPILER                                                          04/01/82          PAGE  24
                     SIO: Serial I/O dvrs for port A -- TTY$GET & TTY$PUT

               $subtitle('SIO: Serial I/O dvrs for port A -- TTY$GET & TTY$PUT')


 256    1      TTY$get:proc byte PUB;

               /*user must not activate this procedure if siodev chan. a reg. ptr
                 is not set to 0 (only [| 0 if user has been mucking with hardware*/
 257    2      do while( (siodev.actl AND rx$avail) = 0);        /*wait forever till empty              */
 258    3      end;
 259    2      return(siodev.adata);                             /*input form 7201                      */

 260    2      end TTY$get;


 261    1      TTY$put:proc(char) PUB;
 262    2      dcl char byte;

               /*user must not activate this procedure if siodev chan. a reg. ptr
                 is not set to 0 (only [| 0 if user has been mucking with hardware*/
 263    2      do while( (siodev.actl AND tx$empty) = 0);        /*wait forever till empty              */
 264    3      end;
 265    2      siodev.adata = char;                              /*output a char                        */
 266    2      return;

 267    2      end TTY$put;



PL/M-86 COMPILER                                                          04/01/82          PAGE  25
                     SIO: Serial I/O dvrs for port B -- UL1$STAT & UL1PUT

               $subtitle('SIO: Serial I/O dvrs for port B -- UL1$STAT & UL1PUT')


 268    1      UL1$stat:proc boolean PUB;

 269    2      if ( (siodev.bctl AND tx$empty) = 0)
               then return(true);
 271    2      return(false);

 272    2      end UL1$stat;


 273    1      UL1$put:proc(char) PUB;
 274    2      dcl char byte;

               /*user must not activate this procedure if siodev chan. b reg. ptr
                 is not set to 0 (only [| 0 if user has been mucking with hardware*/
 275    2      do while( (siodev.bctl AND tx$empty) = 0);        /*wait forever till empty              */
 276    3      end;
 277    2      siodev.bdata = char;                              /*output a char                        */
 278    2      return;

 279    2      end UL1$put;



PL/M-86 COMPILER                                                          04/01/82          PAGE  26
                     SIO: Serial I/O dvrs for ports A & B -- SIO$INIT

               $subtitle('SIO: Serial I/O dvrs for ports A & B -- SIO$INIT')
 280    1      SIO$init:proc PUB;
 281    2      siodev.actl = 00$011$000b;                        /*chan. a reset                        */
 282    2      siodev.bctl = 00$011$000b;                        /*chan. b reset                        */

               /*load timer now; cant touch 7201 chip for 4 2.5Mhz clocks*/
 283    2      sioctr.ctrctl = 36h;                              /*7$(ctra)$(rl)$(mode)$(bin)           */
 284    2      sioctr.adata = serial_params.actrlsb;
 285    2      sioctr.adata = serial_params.actrmsb;
 286    2      sioctr.ctrctl = 76h;                              /*7$(ctrb)$(rl)$(mode)$(bin)           */
 287    2      sioctr.bdata = serial_params.bctrlsb;
 288    2      sioctr.bdata = serial_params.bctrmsb;

               /*cr2a bus interface option*/
 289    2      siodev.actl = 2;                                  /*--|cr4a                              */
 290    2      siodev.actl = serial_params.cr2a;

               /*cr4x*/
```

```
291   2       siodev.actl = 4;                             /*--|cr4a                  */
292   2       siodev.actl = serial_params.cr4a;
293   2       siodev.bctl = 4;                             /*--|cr4b                  */
294   2       siodev.bctl = serial_params.cr4b;

              /*cr3x*/
295   2       siodev.actl = 3;                             /*--|cr3a                  */
296   2       siodev.actl = serial_params.cr3a;
297   2       siodev.bctl = 3;                             /*--|cr3b                  */
298   2       siodev.bctl = serial_params.cr3b;
```



```
              $eject

              /*cr5x*/
299   2       siodev.actl = 5;                             /*--|cr5a                  */
300   2       siodev.actl = serial_params.cr5a;
301   2       siodev.bctl = 5;                             /*--|cr5b                  */
302   2       siodev.bctl = serial_params.cr5b;

              /*cr0x reset ext/st intrs to enable modem control sense--| autoenb chans.
                also --| cr1x, set intr params*/
303   2       siodev.actl = 00$010$001b;
304   2       siodev.actl = 0;                             /*no intrs                 */
305   2       siodev.bctl = 00$010$001b;
306   2       siodev.bctl = 0;                             /*no intrs                 */

307   2       end sio$init;
```



```
              $subtitle ('PPORT -- centronics interface routines ')

              /*
               * This module implements the initialization, LISTST, and LIST functions
               * for a Centronics-compatible parallel printer interface, using the
               * 6522 VIA chip.
               *
               * Our entry points are named pp$init, LPT$stat, and LPT$put respectively,
               * it's up to our caller to decode the I/O byte and call the approp-
               * riate routines.
               */
```



```
              $eject
308   1       declare pp$base pointer;                     /* baseaddr for a 6522          */
309   1       declare pp based pp$base structure (         /* 6522 template                */
                      rb byte,                             /* out-in reg 'b'               */
                      ra byte,                             /* out-in reg 'a'               */
                      ddrb byte,                           /* data-direction, reg 'b'      */
                      ddra byte,                           /* data-direction, reg 'a'      */
                      t1cl byte,                           /* t1 ctr(r)/lat(w) lo          */
                      t1ch byte,                           /* t1 ctr hi                    */
                      t1ll byte,                           /* t1 latch lo                  */
                      t1lh byte,                           /* t1 latch hi                  */
                      t2cl byte,                           /* t2 ctr(r)/lat(w) lo          */
                      t2ch byte,                           /* t2 ctr hi                    */
                      sr byte,                             /* shift register               */
                      acr byte,                            /* auxiliary ctrl reg           */
                      pcr byte,                            /* peripheral ctrl reg          */
                      ifr byte,                            /* interrupt flg register       */
                      ier byte,                            /* interrupt enbl register      */
                      rax byte                             /* out-in reg 'a' NO HANDSHAKE  */
              );
              /*
               * Bit definitions for Centronics-style parallel interface, 'via1'.
               */
310   1       declare via1$base literally '0e8020h';       /* baseaddr for this chip       */
311   1       declare ds$l literally '01h';                /* data strobe (pb0)            */
312   1       declare pi$h literally '02h';                /* this datum for vfu (pb1)     */
313   1       declare bz$h literally '20h';                /* printer busy (pb5)           */
```

```
314   1      declare ak$l literally '40h';                  /* printer ack (pb6)                  */
315   1      declare sl$h literally '80h';                  /* on-line and no error (pb7)         */
             /*
              * Bit definitions for multi-use pio, 'via2'.
              */
316   1      declare via2$base literally '0e8040h';         /* baseaddr for this chip             */
317   1      declare te$h literally '01h';                  /* talk-enable line                   */
```



```
             $eject
             /*
              * initial setup for parallel printer port
              * Note we use via2 during this setup to get talk-enable turned on, and
              * thus someone MUST ALREADY HAVE VIA2 INITIALIZED.
              */

318   1      pp$init: procedure public;
319   2          pp$base = via2$base;                       /* point to secondary chip for te     */
320   2          pp.rb = pp.rb or te$h;                     /* set 'talk enbl'                    */
321   2          pp$base = vial$base;                       /* point struc at primary chip        */
322   2          pp.ra = 0;                                 /* ra is dataport, init with 0's      */
323   2          pp.ddra = 0ffh;                            /* set all ra bits as outgoing        */
324   2          pp.rb = ds$l;                              /* rb is ctrlport, init no ds/pi      */
325   2          pp.ddrb = ds$l or pi$h;                    /* these 2 only are outgoing          */
                                                            /* cal/ca2 cbl/cb2 not used           */
                                                            /* timers/shiftreg not used           */
326   2      end pp$init;
```



```
             $eject

             /*
              * Test status of printer, return true if on-line and not busy, else
              * false.  For some reason, the Altos code explicitly deasserted data
              * strobe before testing; we'll assume that this represents an Altos
              * fubar and is not required here.
              */
327   1      LPT$stat: procedure byte public;
328   2          if (pp.rb and (sl$h or bz$h)) = sl$h then return 0ffh;
330   2          return 0;
331   2      end LPT$stat;

             /*
              * Put one character to the printer interface.
              */
332   1      LPT$put: procedure(ch) public;
333   2         declare ch byte;
334   2          do while LPT$stat = 0; end;                /* wait for printer ready             */
336   2          pp.ra = ch;                                /* put outgoing char on the port      */
337   2       disable;
338   2          pp.rb = pp.rb and not ds$l;                /* assert data strobe                 */
339   2          pp.rb = pp.rb or ds$l;                     /* deassert data strobe               */
340   2       enable;
341   2          return;
342   2      end LPT$put;
```

```
            $SUBTITLE ('Example software drivers for S-1 Hardware')

 343   1    end Hardware;



MODULE INFORMATION:

     CODE AREA SIZE     = 073EH   1854D
     CONSTANT AREA SIZE = 0000H      0D
     VARIABLE AREA SIZE = 0014H     20D
     MAXIMUM STACK SIZE = 000EH     14D
     807 LINES READ
     0 PROGRAM WARNINGS
     0 PROGRAM ERRORS

END OF PL/M-86 COMPILATION
```

INDEX

SUPPLEMENTAL TECHNICAL REFERENCE MATERIAL

APPLICATION NOTE: ØØ2

Revision Ø

Supplemental Technical Reference Material

For information contact:

VICTOR Publications
380 El Pueblo Road
Scotts Valley, CA 95066
(408) 438-6680

First VICTOR printing March 1983.

CONTENTS

CONTENTS continued

CONTENTS continued

CHAPTER 1

Victor 9000 System Overview

1.1 Computer

The Victor 9000 computer is based upon the Intel 8088 16-bit microprocessor. This processor chip is directly related to the Intel 8086 16-bit microprocessor, but with two subtle differences:

8088	8086
8-bit data bus	16-bit data bus
4 instruction look-ahead	6 instruction look-ahead

The major difference, the 8-bit data bus, has some effect on the relative abilities of the two chips; the main difference is that while the 8086 can load an entire 16-bit word of data directly, the 8088 has to load two 8-bit bytes to achieve the same result - the outcome of which being that the 8088 processor is a little slower than the 8086. The loss of speed, however, is balanced by the fact that the cost of the main circuit board and add-on boards are lower than for the wider 8086 requirement. This means that the end-user will have the best cost/performance ratio for a 16-bit computer.

1.2 Memory

The Victor 9000 has a maximum memory capacity of 896 kilobytes of **R**andom **A**ccess **M**emory or "RAM" (a measure of a computer's internal storage capacity; a "kilobyte" is 1,024 bytes). A byte is able to store one character of data - thus the Victor 9000, with full 896k memory capacity is able to hold, internally, nearly 1 million characters - compare this figure with the older Z80 or 6502 computers that have a maximum memory capacity of less than 70,000 characters or 64k bytes of RAM.

1.3 Disk System

The Victor 9000 has several integral disk configurations available; these are:

- Twin single-sided 600k bytes per drive 5 1/4-inch minifloppies, giving a total capacity of 1.2Mbytes (1,200kbytes) available on-line.
- Twin double-sided 1.2M bytes per drive 5 1/4-inch minifloppies, giving a total capacity of 2.4Mbytes (2,400kbytes) available on-line.
- Single 10M byte hard disk (Winchester) plus a single double-sided 1.2M byte 5 1/4-inch minifloppy, giving a total capacity of 11.2Mbytes (11,200kbytes) available on-line.

Future disk systems will include an external 10Mbyte hard disk (Winchester) that will allow expansion of any of the above systems by a further 10,000k bytes.

Although the Victor 9000 uses 5 1/4-inch minifloppies of a similar type to those used in other computers, the floppy disks themselves are not readable on other machines, nor can the Victor 9000 read a disk from another manufacturers machine. The Victor 9000 uses a unique recording method to allow the data to be packed as densely as 600kbytes on a single-sided single-density minifloppy; this recording method involves the regulation of the speed at which the floppy rotates, explaining the fact that the noise from the drive sometimes changes frequency.

1.4 Display System

The display unit swivels and tilts to permit optimum adjustment of the viewing angle, and the unit incorporates a 12-inch antiglare screen to prevent eye strain. The display, in normal mode, is 25 lines, each line having 80 columns. Characters are formed, in normal mode, in a 10-x-16 font cell, providing a highly-readable display. The screen may be used in high-resolution mode, providing a bit-mapped screen with 800-x-400 dot matrix resolution. The high-resolution mode is available only under software control, there is no means of simply "switching" in to high-resolution. Victor Technologies has provided software to allow full use of the screen in high-resolution mode in the Graphics Tool Kit.

Character sets are "soft" - that is they may be substituted for alternative character sets of the users choice, or creation. Only one 256-character character set may be displayed on the screen at one time - multiple character sets cannot, currently, be displayed simultaneously - but this feature may well become available in the future. Character set manipulation software is available in both the Graphics and Programmers Tool Kits.

1.5 Keyboard

Several different types of keyboards are offered. Each keyboard is a separate, low-profile module with an optional palm rest for ease of use. Every key is programmable, permitting the offering of a National keyboard in each country in which it is marketed. As a result, the keyboard can be customized to satisfy the requirements of foreign languages and so that striking a key enters a character or predetermined set of commands.

Keyboards are as soft as the character sets - this allows a keyboard to be generated to match a newly created or special character set. Each key on the keyboard has three potential states; the unshifted, shifted and alternate. The unshifted mode is accessed when the shift key is not depressed along with the desired key; the shifted mode is accessed when the shift key is depressed along with the desired key; and the alternate mode is accessed when the ALT key is depressed along with the desired key. Keyboard manipulation software is available in both the Graphics and Programmers Tool Kits.

1.6 Memory Map

The Victor 9000 is currently supplied with two major disk operating systems; CP/M-86 from Digital Research, and MS-DOS from Microsoft. Athough these two operating systems appear superficially similar, they are quite different in their operation, program interfacing techniques, and their memory structure. The following diagrams are the memory maps for CP/M-86 and MS-DOS; you will notice that some aspects of the machine never change, such as the screen RAM and interrupt vector locations, these areas are hardware defined, and as such never alter. The memory maps for MS-DOS and CP/M-86 are not fixed in the Victor 9000, thus some of the elements of the map will not be specific; this is not to be deliberately vague, but improvements to the performance aspects of the software do take place forcing the diagrams to be unspecific to some degree.

1.6.1 Memory Map -- MS-DOS Operating System

```
     FFFFF
                      Boot Proms
     FC000
                      Reserved for Future Expansion
     F4000
                      Screen High-Speed Static RAM
     F0000
                      Memory-Mapped I/O Space

     E0000

etc.                                 BIOS
256k=3FFF0            Operating System -----------
128k=1FFF0                           MS-DOS
                      Command - Resident Portion
                      Command - Transient Portion


                      Transient Program Area (TPA)


                      Alternate Character Set          4k bytes

                      128 Character Set                4k bytes

                      Logo                             2k bytes
     00480
                      "Stub" - Jump Vectors            128 bytes
     00400
                      Interrupt Vector Table           1k bytes
     00000
```

1.6.2 Memory Map -- CP/M-86 Operating System

FFFFF		
	Boot Proms	
FCØØØ		
	Reserved for Future Expansion	
F4ØØØ		
	Screen High-Speed Static RAM	
FØØØØ		
	Memory-Mapped I/O Space	
EØØØØ		
	Operating System: BIOS / BDOS	
	Transient Program Area (TPA)	
	Alternate Character Set	4k bytes
	128 Character Set	4k bytes
	Logo	2k bytes
ØØ48Ø		
	"Stub" - Jump Vectors	128 bytes
ØØ4ØØ		
	Interrupt Vector Table	1k bytes
ØØØØØ		

CHAPTER 2

Display Driver Specifications

2.1 Overview

The display system in the Victor 9000 is, like so much of the machine, soft. The operating system BIOS contains the Zenith H-19 video terminal emulator, which is an enhanced control set of the DEC VT52 crt. The BIOS takes all ASCII characters received and either displays them or uses their control characteristics. The control characters 00hex (00decimal) thru 1Fhex (31decimal) and 7Fhex (127decimal) are not displayed under normal circumstances. The non-display characters previously discussed, plus those characters having the high-bit set, being 80hex (128decimal) through FFhex (255decimal), may be displayed on the screen under program control, but extensive use of these characters is easier with the Victor Technologies character graphics utilities.

Most of the control characters act by themselves; for example, the TAB key (Control I, 09hex, 09decimal) will cause the cursor to move to the right to the next tab position. For more complex cursor/screen control the multiple character escape sequences should be used. The control characters, and the escape sequences are fully described below.

2.2 Screen Control Sequences

Single Control Characters

Bell (Control G, Ø7hex, Ø7decimal - ASCII BEL)
This ASCII character is not truly a displaying character, but causes the loudspeaker to make a beep.

Backspace (Control H, Ø8hex, Ø8decimal - ASCII BS)
Causes the cursor to be positioned one column to the left of its current position. If at column 1, it causes the cursor to be placed at column 8Ø of the previous line; if the cursor is at column 1, line 1, then the cursor moves to column 8Ø of line 1.

Horizontal Tab (Control I, Ø9hex, Ø9decimal - ASCII HT)
Positions the cursor at the next tab stop to the right. Tab stops are fixed, and are at columns 9, 17, 25, 33, 41, 49, 57, 65, and 72 through 8Ø. If the cursor is at column 8Ø, it remains there.

Line Feed (Control J, ØAhex, 1Ødecimal - ASCII LF)
Positions the cursor down one line. If at line 24, then the display scrolls up one line. This key may be treated as a carriage return -- see ESC x9.

Carriage Return (Control M, ØDhex, 13decimal - ASCII CR)
Positions the cursor at column 1 of the current line. This key may be treated as a line feed -- see ESC x8.

Shift Out (Control N, ØEhex, 14decimal - ASCII SO)
Shift out of the standard system character set, and shift into the alternative system character set (Character set 1, G1). This gives the ability to access and display those characters having the high-bit set - being those characters from 8Øhex (128decimal) through FFhex (255decimal).

Shift In (Control O, ØFhex, 15decimal - ASCII SI)
Shift into the standard system character set (Character set Ø, GØ). This gives the ability to access and display the standard ASCII character set - being those characters from ØØhex (ØØdecimal) through 7Fhex (127decimal).

2.3 Multi-Character Escape Sequences

2.3.1 Cursor Functions

Escape Sequence/Function	ASCII Code	Performed Function
ESC A	1B, 41hex 27, 65dec	Move cursor up one line without changing column.
ESC B	1B, 42hex 27, 66dec	Move cursor down one line without changing column.
ESC C	1B, 43hex 27, 67dec	Move cursor forward one character position.
ESC D	1B, 44hex 27, 68dec	Move cursor backward one character position.
ESC H	1B, 48hex 27, 72dec	Move cursor to the home position. Cursor moves to line 1, column 1.
ESC I	1B, 49hex 27, 73dec	Reverse index. Move cursor up to previous line at current column position.
ESC Y l c	1B, 59hex 27, 89dec	Moves the cursor via direct (absolute) addressing to the line and column location described by 'l' and 'c'. The line ('l') and column ('c') coordinates are binary values offset from 20hex (32decimal). (For further information on the use of direct addressing see section 2.4).
ESC j	1B, 6Ahex 27, 106dec	Store the current cursor position. The cursor location is saved for later restoration (see ESC k).
ESC k	1B, 6Bhex 27, 107dec	Returns cursor to the previously saved location (see ESC j).
ESC n	1B, 6Ehex 27, 110dec	Return the current cursor position. The current cursor location is returned as line and column, offset from 20hex (32decimal), in the next character input request.

2.3.2 Editing Functions

Escape Sequence/Function	ASCII Code	Performed Function
ESC @	1B, 40hex 27, 64dec	Enter the character insert mode. Characters may be added at the current cursor position, as each new character is added, the character at the end of the line is lost.
ESC E	1B, 45hex 27, 69dec	Erase the entire screen.
ESC J	1B, 4Ahex 27, 74dec	Erase from the current cursor position to the to the end of the screen.
ESC K	1B, 4Bhex 27, 75dec	Erase the screen from the current cursor position to the end of the line.
ESC L	1B, 4Chex 27, 76dec	Insert a blank line on the current cursor line. The current line, and all following lines are moved down one, and the cursor is placed at the beginning of the blank line.
ESC M	1B, 4Dhex 27, 77dec	Delete the line containing the cursor, place the cursor at the start of the line, and move all following lines up one - a blank line is inserted at line 24.
ESC N	1B, 4Ehex 27, 78dec	Delete the character at the cursor position, and move all other characters on the line after the cursor to the left one character position.
ESC O	1B, 4Fhex 27, 79dec	Exit from the character insert mode (see ESC @).
ESC b	1B, 62hex 27, 98dec	Erase the screen from the start of the screen up to, and including, the current cursor position.

2.3.2 Editing Functions -- continued

Escape Sequence/Function	ASCII Code	Performed Function
ESC l	1B, 6Chex 27, 1Ø8dec	Erase entire current cursor line.
ESC o	1B, 6Fhex 27, 111dec	Erase the beginning of the line up to, and including, the current cursor position.

2.3.3 Configuration Functions

Escape Sequence/Function	ASCII Code	Ps	Performed Function
ESC x Ps	1B, 78hex 27, 120dec		Sets mode(s) as follows:
		Ps	**Mode**
	31hex, 49dec	1	Enable 25th line
	33hex, 51dec	3	Hold screen mode on
	34hex, 52dec	4	Block cursor
	35hex, 53dec	5	Cursor off
	38hex, 56dec	8	Auto line feed on receipt of a carriage return.
	39hex, 57dec	9	Auto carriage return on receipt of line feed
	41hex, 65dec	A	Increase audio volume
	42hex, 66dec	B	Increase CRT brightness
	43hex, 67dec	C	Increase CRT contrast
ESC y Ps	1B, 79hex 27, 120dec		Resets mode(s) as follows:
		Ps	**Mode**
	31hex, 49dec	1	Disable 25th line
	33hex, 51dec	3	Hold screen mode off
	34hex, 52dec	4	Underscore cursor
	35hex, 53dec	5	Cursor on
	38hex, 56dec	8	No auto line feed on rec-eipt of a carriage return.
	39hex, 57dec	9	No auto carriage return on receipt of line feed
	41hex, 65dec	A	Decrease audio volume
	42hex, 66dec	B	Decrease CRT brightness
	43hex, 67dec	C	Decrease CRT contrast
ESC [	1B, 5Bhex 27, 91dec		Set hold mode
ESC \	1B, 5Chex 27, 92dec		Clear hold mode
ESC ^	1B, 5Ehex 27, 94dec		Toggle hold mode on/off.

2.3.4 Operation Mode Functions

Escape Sequence/Function	ASCII Code	Performed Function
ESC (	1B, 28hex 27, 4Ødec	Enter high intensity mode. All characters displayed after this point will be displayed in high-intensity.
ESC)	1B, 29hex 1B, 41dec	Exit high intensity mode.
ESC Ø	1B, 3Øhex 27, 48dec	Enter underline mode. All characters displayed after this point will be underlined.
ESC 1	1B, 31hex 27, 49dec	Exit underline mode.
ESC p	1B, 7Øhex 27, 112dec	Enter reverse video mode. All characters displayed after this point will be displayed in reverse video.
ESC q	1B, 71hex 27, 113dec	Exit reverse video mode.

2.3.5 Special Functions

Escape Sequence/Function	ASCII Code	Performed Function
ESC #	1B, 23hex 27, 35dec	Return the current contents of the page. The entire contents of the screen are made available at the next character input request(s). (For further information on the use of this function, see section 2.5).
ESC $	1B, 24hex 27, 36dec	Return the value of the character at the current cursor position. The character is returned in the next character input request.
ESC +	1B, 2Bhex 27, 43dec	Clear the foreground. Clear all high-intensity displayed characters.
ESC 2	1B, 32hex 27, 5Ødec	Make cursor blink.
ESC 3	1B, 33hex 27, 51dec	Stop cursor blink.
ESC 8	1B, 38hex 27, 56dec	Set the text (literally) mode for the next single character. This allows the display of characters from Ø1hex (Ø1dec) thru 1Fhex (31dec) on the screen. Thus the BELL character (Ø7hex, Ø7dec) will not cause the bleep, but a character will appear on the screen.
ESC Z	1B, 5Ahex 27, 9Ødec	Identify terminal type. The VT52 emulator will return ESC\Z in the next character input request.
ESC]	1B, 5Dhex 27, 93dec	Return the value of the 25th line. The next series of character input requests will receive the current contents of the 25th line.

2.3.5 Special Functions -- continued

Escape Sequence/Function	ASCII Code	Performed Function
ESC v	1B, 76hex 27, 118dec	Enable wrap-around at the end of each screen line. A character placed after column 80 of a line will be placed on the next line at column 1.
ESC w	1B, 77hex 27, 119dec	Disable wrap-around at the end of each line.
ESC z	1B, 7Ahex 27, 122dec	Reset terminal emulator to the power-on state. This clears all user selected modes, clears the screen, and homes the cursor.
ESC {	1B, 7Bhex 27, 123dec	Enable keyboard input. (see ESC }).
ESC }	1B, 7Dhex 27, 125dec	Disable keyboard input. This locks the keyboard. Any character(s) typed are ignored until an ESC { is issued.
ESC i Ps	1B, 69hex 27, 105dec	Displays banner as follows:

ASCII Code	Ps	Mode
30hex, 48dec	0	Display entire banner
31hex, 49dec	1	Display company logo
32hex, 50dec	2	Display operating system
33hex, 51dec	3	Display configuration

2.4 Direct Cursor Addressing -- Examples of Use

The direct cursor addressing function is accessed by sending the ESC Y l c sequence to the screen (see section 2.3.1). "l" is the line number required, whose valid coordinates are between 1 and 24. An offset of 1Fhex (31decimal) must be added to the location required in order to correctly locate the cursor. "c" is the column number required, whose valid coordinates are between 1 and 80. An offset of 1Fhex (31decimal) must be added to the location required in order to correctly locate the cursor.

Note that the true offset requirement of 20hex (32decimal) for line and column may only be used accurately when the line number is viewed 0 to 23, and the column number 0 to 79.

The line/column number requested must be handled as a binary digit, examples of this follow:

2.4.1 Microsoft MS-BASIC -- Direct Cursor Positioning

The following method uses offsets from line 1, column 1:

```
10 PRINT CHR$(27)+"E"    :REM CLEAR THE SCREEN
20 DEF FNM$(LIN,COL)=CHR$(27)+"Y"+CHR$(31+LIN)+CHR$(31+COL)
30 PRINT "Enter line (1-24) and column (1-80), as LINE,COL ";
40 INPUT LIN, COL
50 PRINT FNM$(LIN,COL);
60 FOR I = 1 TO 1000    :REM PAUSE BEFORE OK MESSAGE DISPLAYED
70 NEXT I
```

The alternative method, using offsets from zero is shown below:

```
10 PRINT CHR$(27)+"E"    :REM CLEAR THE SCREEN
20 DEF FNM$(LIN,COL)=CHR$(27)+"Y"+CHR$(32+LIN)+CHR$(32+COL)
30 PRINT "Enter line (0-23) and column (0-79), as LINE,COL ";
40 INPUT LIN, COL
50 PRINT FNM$(LIN,COL);
60 FOR I = 1 TO 1000    :REM PAUSE BEFORE OK MESSAGE DISPLAYED
70 NEXT I
```

2.4.2 Microsoft MACRO-86 Assembler -- Direct Cursor Positioning

```
line_off   equ  20h                     ;line position offset from 0
col_off    equ  20h                     ;column position offset from 0
esc        equ  1bh                     ;escape character
msdos      equ  21h                     ;interrupt to MS-DOS

clear_screen          db   esc,'E$'     ;clear screen request
dir_cur_pos_lead      db   esc,'Y$'     ;cursor positioning lead-in

; the cursor position required is handed down in BX
;    where BH = line (0-23 binary), BL = column (0-79 binary)

clear_and_locate:
     mov  ah,9h                         ;string output up to $
     mov  dx,offset clear_screen        ;get the clear screen string
     int  msdos                         ;and output it up to the $
;
; the cursor position required is in BX
;
     add  bh,line_off                   ;normalize line for output
     add  bl,col_off                    ;normalize column for output
;
; send the direct cursor positioning lead-in
;
     mov  ah,9h                         ;select screen output up to $
     mov  dx,offset dir_cur_pos_lead ;select the lead in ESC Y
     int  msdos                         ;and output it up to $
;
; now the contents of BX must be sent to the terminal emulator
;
     mov  dl,bh                         ;ready the line number
     mov  ah,6h                         ;direct console output of DL
     int  msdos                         ;output the line coordinate
;
     mov  dl,bl                         ;ready the column number
     mov  ah,6h                         ;direct console output of DL
     int  msdos                         ;send the column coordinate
;
; the cursor is now at the location selected in BX
```

2.4.3 Microsoft Pascal Compiler -- Direct Cursor Positioning

```
program position (input,output);
{This method uses offsets from line Ø, column Ø.}

const
   clear_screen = chr(27) * chr(69);

var
   result : array[1..4] of char;
   i, line, column : integer
   row, col : char;

begin
   result[1] := chr(27);          {RESULT = ESC}
   result[2] := chr(89);          {RESULT = "Y"}
   write (clear_screen);
   write ('  Enter line (Ø-23) and column (Ø-79), as LINE COLUMN: ');
   readln (line, column);
   writeln (clear_screen);
   row := chr(32 + line);
   col := chr(32 + column);
   result[3] := row;              {RESULT = ROW}
   result[4] := col;              {RESULT = COL}
   for i := 1 to 4 do
       write (result[i]);         {PRINT CURSOR TO SCREEN}
   for i := 1 to 32ØØØ do         {PAUSE}
end.
```

2.5 Transmit Page -- Examples of Use

The transmit page function is accessed by sending the ESC # sequence to the screen (see section 2.3.5). The result of this sequence is that all characters on the screen, as well as the cursor positioning sequences required to re-create the screen, are sent to the keyboard buffer. Reading the keyboard via a normal keyboard input request will return the entire screen of data to the program. The screen buffer within the program should be at least 1920decimal bytes long to accomodate the entire screen - the program will need to perform 1920 single character inputs to empty the keyboard buffer. Note that the character input requests must be done rapidly to prevent the keyboard buffer overflowing and causing loss of data - note, too, that on a keyboard buffer overflow, the bell sounds.
The following sample programs demonstrate the use for this function request:

2.5.1 Microsoft MS-BASIC -- Transmit Page

```
10 DIM A$(1920)
20 PRINT CHR$(27)+"#";
30 FOR I = 1 TO 1920
40 A$(I)=INKEY$
50 NEXT I
60 PRINT CHR$(27)+"E";
70 FOR I = 1 TO 1920
80 PRINT A$(I);
90 NEXT I
```

2.5.2 Microsoft MACRO-86 Assembler -- Transmit Page

```
coniof        equ    6h                  ;direct console i/o function
conin         equ    Øffh                ;console input request
printf        equ    9h                  ;screen o/p up to $
msdos         equ    21h                 ;interrupt operating system
buffer_length    equ    192Ø             ;entire screen count

read_screen  db    1bh,'#$'              ;read entire screen
clear_screen db    1bh,'E$'              ;clear screen/home cursor
buffer       db    buffer_length dup (?) ;main buffer region

     mov     ax,DS                       ;get buffer data segment
     mov     ES,ax                       ;ready for store
     mov     di,offset buffer            ;get storage buffer
     mov     si,di                       ;init for later use
     mov     dx,offset read_screen       ;read entire screen string
     mov     ah,printf                   ;o/p it up to $
     int     msdos                       ;call the OS
;
; now read entire screen in to BUFFER
;
     mov     ah,coniof                   ;read from keyboard buffer
     mov     dl,conin                    ;ready to read
     mov     cx,buffer_length            ;count of chars to read
;
in_loop:
     int     msdos                       ;get a char in AL
     stosb                               ;save the char in BUFFER
     loop    in_loop                     ; and loop til buffer full
;
     mov     ah,printf                   ;ready to clear the screen
     mov     dx,offset clear_screen      ;get the string
     int     msdos                       ; and o/p it up to $
;
; now replace the screen data
;
     mov     cx,buffer_length            ;get the count
     mov     ah,coniof                   ;get the o/p char function
;
out_loop:
     lodsb                               ;get a char
     mov     dl,al                       ;  ready to go
     int     msdos                       ;o/p it
     loop    out_loop                    ;loop til buffer empty
     ret                                 ;
```

2.5.3 Microsoft Pascal Compiler -- Transmit Page

```
PROGRAM Scrnbuf;

  CONST
    clear_screen   = CHR(27)*CHR(69)*CHR(36);
    transmit_page  = CHR(27)*CHR(35)*CHR(36);
    err_msg        = 'ERROR$';
    direct_conio   = #6;
    conin          = #0FF;
    print_string   = #9;

  VAR
    screen_dump : ARRAY [1..1920] OF CHAR;
    ch : CHAR;
    i : INTEGER;
    param : WORD;
    status : BYTE;

FUNCTION DOSXQQ( command, parameter : WORD ) : BYTE; EXTERNAL;

BEGIN
  EVAL(DOSXQQ(print_string,WRD(ADR(transmit_page) ) ) );
  param:= BYWORD( 0, conin );
  status:= DOSXQQ( direct_conio, param );
  IF status <> 0 THEN
    BEGIN
      i:= 1;
      WHILE status <> 0 DO
        BEGIN
          ch:= CHR(status);
          screen_dump[i]:= ch;
          i:= i + 1;
          status:= DOSXQQ( direct_conio, param );
        END;
      i:= i - 1;
      EVAL(DOSXQQ(print_string,WRD(ADR(clear_screen) ) ) );
      FOR VAR J:= 1 TO i DO
        EVAL(DOSXQQ( direct_conio, WRD(screen_dump[J]) ) );
    END
  ELSE
    EVAL(DOSXQQ(print_string,WRD(ADR(err_msg) ) ) );
END.
```

CHAPTER 3

Victor 9000 Input/Output Port Specification

3.1 Device Connection

There are 5 ports available on the Victor 9000 - they are as follows:

2 x Serial (RS232C) - Ports A and B
1 x Parallel (Centronics)
2 x Parallel (control - located on CPU board)

The ports are located on the rear of the Victor 9000 as shown in the following diagram:

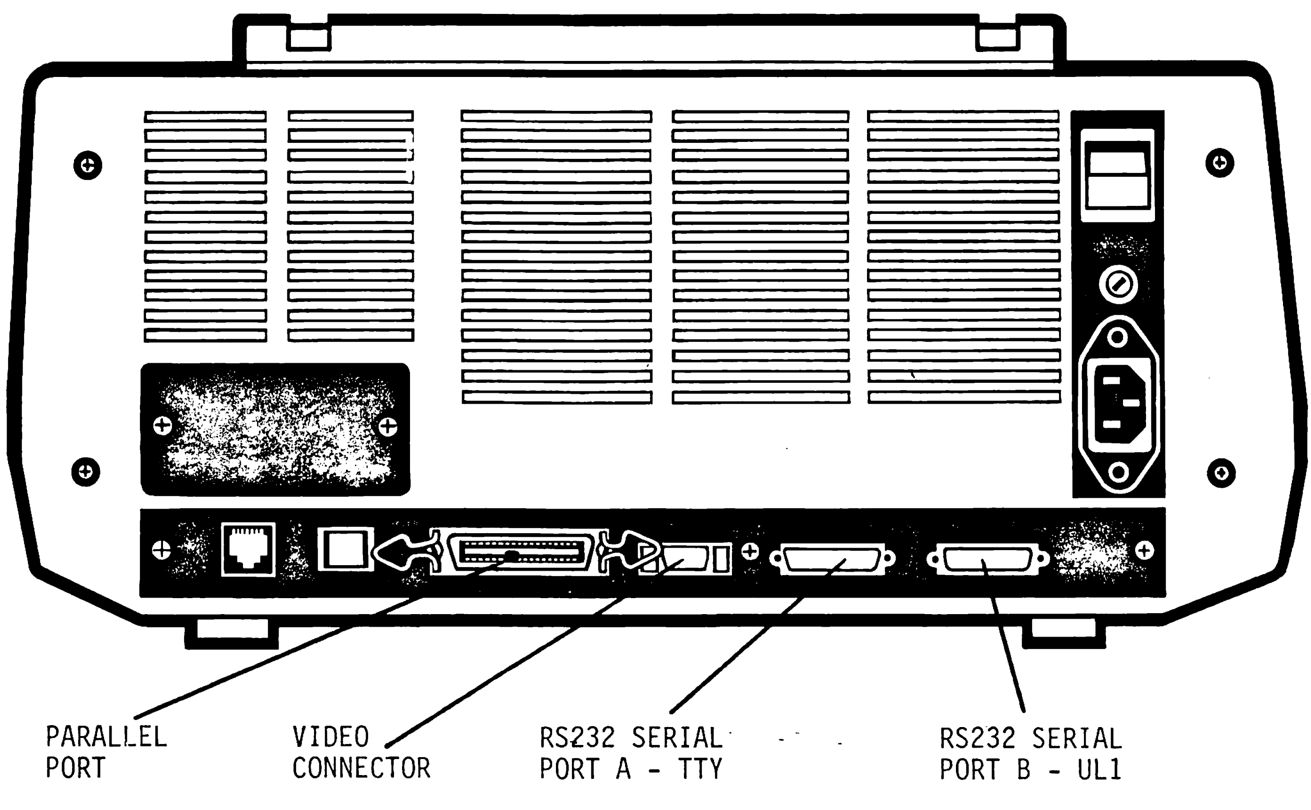


Figure 1
Victor 9000 Parallel and Serial Ports

3.2 Parallel Printer Connection

To connect a parallel printer to the Victor 9000, a suitable cable is required - if the printer is supplied by Victor Technologies, then it will be a matter of plugging the cable into both machines; cables should be attached as follows:

1) Disconnect power from both the computer and printer.
2) Disconnect the Victor video connector (see 3.1)
3) Attach interface cable to Victor and printer
4) Re-attach the video connector
5) Set the printer dip-switches as required

3.3 Parallel Cable Requirements

If a suitable parallel cable is not available, you will need to make one - use the guidelines that follow to create your own cable:

> You will need a male centronics-compatible Amphenol 57-30360 type connector for the Victor 9000 end of the cable; use the type of connector suggested by the printer manufacturer for the printer end, in general, another male centronics-compatible Amphenol 57-30360 type connector will be required. You will also require a length of 12-core cable (10 feet maximum length).

Refer to the port layout in your printer handbook - compare this with the Victor 9000 parallel port layout (see C.1). If the pin numbers and signal requirements are the same, then construct the cable as follows:

```
 1 ------------------- 1
 2 ------------------- 2
 3 ------------------- 3
 4 ------------------- 4
 5 ------------------- 5
 6 ------------------- 6
 7 ------------------- 7
 8 ------------------- 8
 9 ------------------- 9
10 ------------------- 10
11 ------------------- 11
16 ------------------- 16
```

It does not matter which end of the cable is connected to the printer or the computer.

If your printer has the same signals as the Victor 9000, but on differing pins, then use the following guidelines:

1) Label one connector "Computer" and the other "Printer".
2) Connect pin 1 at the computer connector to the Data strobe pin at the printer connector.
3) Connect pins 2 thru 9 at the computer connector to the Data1 (may be labelled Data0) thru Data8 (may be labelled Data7) at the printer connector.
4) Connect pin 10 at the computer connector to the ACK pin at the printer connector.
5) Connect pin 11 at the computer connector to the BUSY pin at the printer connector.
6) Connect pin 16 at the computer connector to the GROUND (may be labelled GND) pin at the printer connector.

The printer cable is now complete - it must always be attached to the devices as marked on the connectors - if it is not, then the printer will not work.

3.4 Serial Printer Connection

To connect a serial printer to the Victor 9000, a suitable cable is required - if the printer is supplied by Victor Technologies, then it will be a matter of plugging the cable into both machines; cables should be attached as follows:

1) Attach the cable between the Victor 9000 serial port B (see 3.1) and the printer connector.
2) Set the printer switches for 7-data bits, 1 stop bit, 1200 baud and no parity. Set DTR protocol (refer to printer manual).

You may set the baud rate at a rate different from that mentioned in (2) - but you will then be required to set the baud rate using the baud rate selection utility, PORTSET or PORTCONF (see 3.6), or alternatively you will need to build a new operating system.

3.5 Serial Cable Requirements

If a suitable serial cable is not available, you will need to make one - use the guidelines that follow to create your own cable:

You will require 1 x D25 male, 1 x D25 female connectors, and a length of 6-12 core cable, with a maximum length of fourty feet. Refer to the port layout in your printer manual, if pin 3 is received data (labelled RXD or RD), and pin 2Ø is data terminal ready (labelled DTR), then construct your cable as follows:

```
Computer                    Printer

    1 ---------------------- 1
    2 ---------------------- 3
    3 ---------------------- 2
    7 ---------------------- 7
    5 ---------------------- 2Ø
```

This cable, often called a Modem Eliminator Cable, must be attached as shown - mark the Computer/Printer connectors as a reference.

If pin 3 is receive data (RXD or RD) and pin 2Ø is not data terminal ready (DTR) then construct your cable as follows:

```
Computer                    Printer

    1 ---------------------- 1
    3 ---------------------- 2
    2 ---------------------- 3
    7 ---------------------- 7
    5 ---------------------- 4
```

This cable must be attached as shown - mark the Computer/Printer connectors as a reference.

3.6 Operating System Port Utilities

Victor Technologies supplies a selection of programs under both CP/M-86 and MS-DOS to allow the temporary selection of both baud rate and list device port. If you attach a printer to your system you may be required to perform some of the following steps in order to utilize the printer. Before you use any of the utilities discussed you need to be aware of the port the printer is attached to; Port A, B or Parallel. You will also need to know, except in the case of a parallel printer, what the baud rate, stop-bits and parity your printer is set up at. Note that many printers will start to lose data at baud rates above 4800, you must, therefore, select a baud rate that your printer can handle.

3.6.1 SETIO - MS-DOS List Device Selection Utility

To select the correct port for the list device you have attached, the SETIO program has been provided. This program is used as follows:

```
SETIO LST = TTY    - printer is attached to port A
SETIO LST = UL1    - printer is attached to port B
SETIO LST = LPT    - printer is attached to parallel port
```

It is recommended that your printer be attached to either port B or the parallel port.

Once SETIO has executed, it displays a map of the ports, with the ones you selected highlighted on the screen - if this is not corrcet, repeat the process.

3.6.2 STAT - CP/M-86 List Device Selection Utility

To select the correct port for the list device you have attached, the STAT program has been provided. This program is used as follows:

```
STAT LST:=TTY:  - printer is attached to port A
STAT LST:=UL1:  - printer is attached to port B
STAT LST:=LPT:  - printer is attached to parallel port
```

It is recommended that your printer be attached to either port B or the parallel port.

3.6.3 PORTSET - MS-DOS Baud Rate Selection Utility

To select the correct baud rate for ports A or B (but this is not applicable to the parallel port), the PORTSET program is provided. This program is menu driven, and is used as follows:

To the prompt type PORTSET, the screen will display a choice of three ports:

1) Port A (RS232C)
2) Centronics/Parallel Port
3) Port B (RS232C)

Type either 1,2 or 3. If you type 1 or 3, the next menu screen is displayed - this screen has baud-rate choices labelled A through N - select one of the baud-rates.

3.6.4 PORTCONF - CP/M-86 Baud Rate Selection Utility

This program is used in exactly the same manner as PORTSET (see 3.6.3).

3.7 Serial Input/Ouput Ports

The two serial input/output ports are memory mapped ports located in the memory segment EØØØhex; and they are mapped as follows:

EØØØ:4Ø	-	port A data (input/output)
EØØØ:41	-	port B data (input/output)
EØØØ:42	-	port A control (read/write)
EØØØ:43	-	port B control (read/write)

The following information is available in each port's control register:

bit Ø	-	rx character available
bit 1	-	not used
bit 2	-	tx buffer empty
bit 3	-	DCD
bit 4	-	not used
bit 5	-	CTS
bit 6	-	not used
bit 7	-	not used

See Appendix C.2 for information on each port's pinouts.

Note that writing a 1Øhex to the relevent control register allows the resensing of the modem leads (i.e. DCD and CTS) with their current values being updated in the port's control register.
Since the Victor 9ØØØ configures the NEC 72Ø1 chip to operate in auto-enable mode, DCD (pin 8 on the port connector) must be ON, and CTS (pin 5 on the port connector) must be ON to enable the 72Ø1's receiver and trasmitter respectively. RTS and DTR are always ON as a convenient source for an RS-232C control ON (+11 volts).

3.8 Baud Rate and Data Input/Output - Sample Programs

The means of establishing the baud rates, receiving and transmitting data are discussed in the following programs. The serial port's control register are discussed in 3.7 - the means of accessing them is better described with the programming examples that follow.

The following programs provide information on how to set up the baud rates on the serial ports (A and B) - they also demonstrate how to send and receive data from these ports.

3.8.1 Microsoft MS-BASIC -- Baud Rate and Data Input/Output

The following program may be used in place of PORTSET or PORTCONF if you omit the lines 500 through 740 inclusive.

```
10 DIM RATE(14)
20 REM Select the data port
30 PRINT CHR$(27)+"E"; : REM Clear the screen
40 PRINT : PRINT : PRINT : PRINT
50 PRINT "The serial ports are:" : PRINT
60 PRINT ,"           A - Serial Port TTY - left hand on back"
70 PRINT ,"           B - Serial Port UL1 - right hand on back"
80 PRINT : PRINT
90 PRINT ,"Select the port you want to use, A or B ";
100 PORT$ = INPUT$(1)
110 PRINT PORT$
120 IF PORT$ = "a" THEN STATIO=2 : DATIO=0 : GOTO 210
130 IF PORT$ = "A" THEN STATIO=2 : DATIO=0 : GOTO 210
140 IF PORT$ = "b" THEN STATIO=3 : DATIO=1 : GOTO 210
150 IF PORT$ = "B" THEN STATIO=3 : DATIO=1 : GOTO 210
160 GOTO 30
200 REM Set the baud rate
210 PRINT CHR$(27)+"E"; : REM Clear the screen
220 PRINT : PRINT : PRINT : PRINT
230 PRINT "The available baud rates are as follows:" : PRINT
240 PRINT ," 1 =     300 baud"
250 PRINT ," 2 =     600 baud"
260 PRINT ," 3 =    1200 baud"
270 PRINT ," 4 =    2400 baud"
280 PRINT ," 5 =    4800 baud"
290 PRINT ," 6 =    9600 baud"
300 PRINT ," 7 =   19200 baud"
310 PRINT : PRINT : PRINT
320 PRINT "Select one of the above baud rates: ";
330 RATE$ = INPUT$(1)
340 IF RATE$ > "7" THEN 210
350 IF RATE$ < "1" THEN 210
360 PRINT RATE$
400 REM Now set the baud rate in the port selected
410 DEF SEG = &HE002
420 IF DATIO = 0 THEN POKE 3,54 : IF DATIO = 1 THEN POKE 3,118
430 FOR I = 1 TO 14
440 READ RATE(I) : REM Set the baud rate matrix
450 NEXT I
460 NODE = (VAL(RATE$)-1)*2+1
470 POKE DATIO,RATE(NODE)
480 POKE DATIO,RATE(NODE+1)
```

-- Listing Continued on Next Page --

```
500 REM Now data may be entered and sent down line
510 PRINT CHR$(27)+"E"; : REM Clear the screen
520 PRINT : PRINT ,"Baud rate established"
530 PRINT : PRINT : PRINT
540 DEF SEG = &HE004
550 PRINT ,"Enter data to be sent down line with return to end"
560 PRINT ,"or just press return to receive data -"
570 PRINT
580 TEXT$=INKEY$
590 IF TEXT$="" THEN 630
600 IF TEXT$=CHR$(13) THEN PRINT TEXT$ :TEXT$=CHR$(126) :GOTO 620
610 PRINT TEXT$;
620 GOSUB 650
630 GOSUB 690
640 GOTO 580
650 STATUS=PEEK (STATIO) : STATUS=STATUS AND 4
660 IF STATUS = 0 THEN 650 :REM Waiting to send char
670 POKE DATIO, ASC(TEXT$)
680 RETURN
690 STATUS = PEEK(STATIO) :STATUS = STATUS AND 1
700 IF STATUS = 0 THEN RETURN : REM No char available
710 DATUM = PEEK (DATIO) : DATUM = DATUM AND 127
720 IF DATUM = 126 THEN PRINT CHR$(13) : RETURN
730 PRINT CHR$(DATUM); :REM Show char from line
740 RETURN
1000 DATA 04,1,&H82,0,&H41,0,&H20,0,&H10,0,8,0,4,0
```

3.8.2 MACRO-86 Assembler -- Baud Rate and Data Input/Output

The following assembler modules may be included in a program and called with the stated parameters. The character input and output modules will need re-coding if your program requires status return rather than looping for good status.

```
rates   db      Ø4h,1h,82h,Øh       ;baud rate conversion table
        db      41h,Øh,2Øh,Øh
        db      1Øh,Øh,8h,Øh
        db      4h,Øh

;**********************************************************
;
; Routine:      BAUD_SET
;
; Function:     To set Port A or B baud rate
;
; Entries:      AL = Ø=PortA, 1=PortB
;               DX = Ø=3ØØ baud,  1=6ØØ baud,  2=12ØØ baud
;                    3=24ØØ baud, 4=48ØØ baud, 5=96ØØ baud
;                    6=192ØØ baud
;
; Returns:      None
;
; Corruptions: ES, AX, BX, CX, DX
;
;**********************************************************

baud_set:
        mov     cx,Øe ØØ2h                  ;get the segment
        mov     ES,cx                       ;init the segment register
        mov     bx,3                        ;point to counter control
        or      al,al                       ;see if Port A or B to be set
        jnz     set_B                       ;AL > Ø, so set Port B counter
;
        mov     byte ptr ES:[bx],36h        ;set it for port A
        jmp     short set_rate              ;  and input the Baud rate
;
set_B:
        mov     byte ptr ES:[bx],76h        ;set port B counter
;
set_rate:
        mov     bx,offset rates             ;get the baud rate table
        shl     dx,1                        ;DX = DX * 2 for words
        add     bx,dx                       ;point to baud rate entry
        mov     dx,[bx]                     ;get the baud rate
        xor     bh,bh                       ;BH=Ø
        mov     bl,al                       ;get the required port
        mov     byte ptr ES:[bx],dl         ;send first byte
        mov     byte ptr ES:[bx],dh         ; and last byte of rate
        ret                                 ;baud rate established
```

3.8.2 Baud Rate and Data Input/Output -- continued

```
;***********************************************************
;
; Routine:        SEND_CHAR
;
; Function:       To output a character to a serial port
;
; Entries:        AL = Ø=PortA, 1=PortB
;                 AH = Character to send
;
; Returns:        None
;
; Corruptions: ES, AX, BX
;
;***********************************************************

send_char:
        mov     bx,Øe0Ø4h               ;get the port segment
        mov     ES,bx                   ;set the segment
        xor     bh,bh                   ;BH=Ø
        mov     bl,al                   ;get the required port
        add     bl,2                    ;required port status
;
in_status_loop:
        mov     al,ES:[bx]              ;get the status
        and     al,4h                   ;mask for TX empty
        jz      in_status_loop          ;not ready - loop
;
        sub     bl,2                    ;point to data
        mov     ES:[bx],ah              ;character gone
        ret
```

3.8.2 Baud Rate and Data Input/Output -- continued

```
;*********************************************************
;
; Routine:        GET_CHAR
;
; Function:       To input a character from a serial port
;
; Entries:        AL = Ø=PortA, 1=PortB
;
; Returns:        AL = character
;
; Corruptions: ES, AX, BX
;
;*********************************************************

get_char:
        mov     bx,Øe ØØ4h              ;get the port segment
        mov     ES,bx                   ;set the segment
        xor     bh,bh                   ;BH=Ø
        mov     bl,al                   ;get the required port
        add     bl,2                    ;required port status
;
out_status_loop:
        mov     al,ES:[bx]              ;get the status
        and     al,1h                   ;mask for RX character avail
        jz      out_status_loop         ;not ready - loop
;
        sub     bl,2                    ;point to data
        mov     al,ES:[bx]              ;character received
        ret
```

APPENDIX A

A.1 ASCII Codes Used in the Victor 9000 Computer

The **A**merican **S**tandard **C**odes for **I**nformation **I**nterchange (ASCII) has been defined to allow data communication between computers, their peripherals, and other computers. The other major code standard is the **E**xtended **B**inary **C**oded-**D**ecimal **I**nterchange **C**ode (EBCDIC) used on some mainframe computers. The Victor 9000 computer is designed to function in ASCII, but communication software is available that allows the Victor 9000 to receive EBCDIC data and have it translated into ASCII, and vice versa.

The following table contains the 7-ASCII codes and their meanings. It is called 7-ASCII as only 7-bits of the potential 8-bits are used to carry data; the "spare" bit is utilized in the Victor 9000 computer to support characters not otherwise available in the 7-ASCII set.

An Eight Bit Byte is pictured as follows:

[7][6][5][4][3][2][1][0]

the bits are numbered 0 through 7 (which adds up to eight bits), and it is the 8th bit (bit 7 in computer jargon) which is not used in 7-ASCII.

A.2 ASCII / HEXADECIMAL / DECIMAL Character Set

ASCII	Hex	Dec	ASCII	Hex	Dec	ASCII	Hex	Dec	ASCII	Hex	Dec
NUL	00	00	space	20	32	@	40	64	`	60	96
SOH	01	01	!	21	33	A	41	65	a	61	97
STX	02	02	"	22	34	B	42	66	b	62	98
ETX	03	03	#	23	35	C	43	67	c	63	99
EOT	04	04	$	24	36	D	44	68	d	64	100
ENQ	05	05	%	25	37	E	45	69	e	65	101
ACK	06	06	&	26	38	F	46	70	f	66	102
BEL	07	07	'	27	39	G	47	71	g	67	103
BS	08	08	(	28	40	H	48	72	h	68	104
HT	09	09	)	29	41	I	49	73	i	69	105
LF	0A	10	*	2A	42	J	4A	74	j	6A	106
VT	0B	11	+	2B	43	K	4B	75	k	6B	107
FF	0C	12	,	2C	44	L	4C	76	l	6C	108
CR	0D	13	-	2D	45	M	4D	77	m	6D	109
SO	0E	14	.	2E	46	N	4E	78	n	6E	110
SI	0F	15	/	2F	47	O	4F	79	o	6F	111
DLE	10	16	0	30	48	P	50	80	p	70	112
DC1	11	17	1	31	49	Q	51	81	q	71	113
DC2	12	18	2	32	50	R	52	82	r	72	114
DC3	13	19	3	33	51	S	53	83	s	73	115
DC4	14	20	4	34	52	T	54	84	t	74	116
NAK	15	21	5	35	53	U	55	85	u	75	117
SYN	16	22	6	36	54	V	56	86	v	76	118
ETB	17	23	7	37	55	W	57	87	w	77	119
CAN	18	24	8	38	56	X	58	88	x	78	120
EM	19	25	9	39	57	Y	59	89	y	79	121
SUB	1A	26	:	3A	58	Z	5A	90	z	7A	122
ESC	1B	27	;	3B	59	[	5B	91	{	7B	123
FS	1C	28	<	3C	60	\	5C	92	|	7C	124
GS	1D	29	=	3D	61	]	5D	93	}	7D	125
RS	1E	30	>	3E	62	^	5E	94	~	7E	126
US	1F	31	?	3F	63	_	5F	95	DEL	7F	127

APPENDIX B

B.1 Victor 9000 Keyboard Layout

Legend:

Shaded region indicates unused key switch

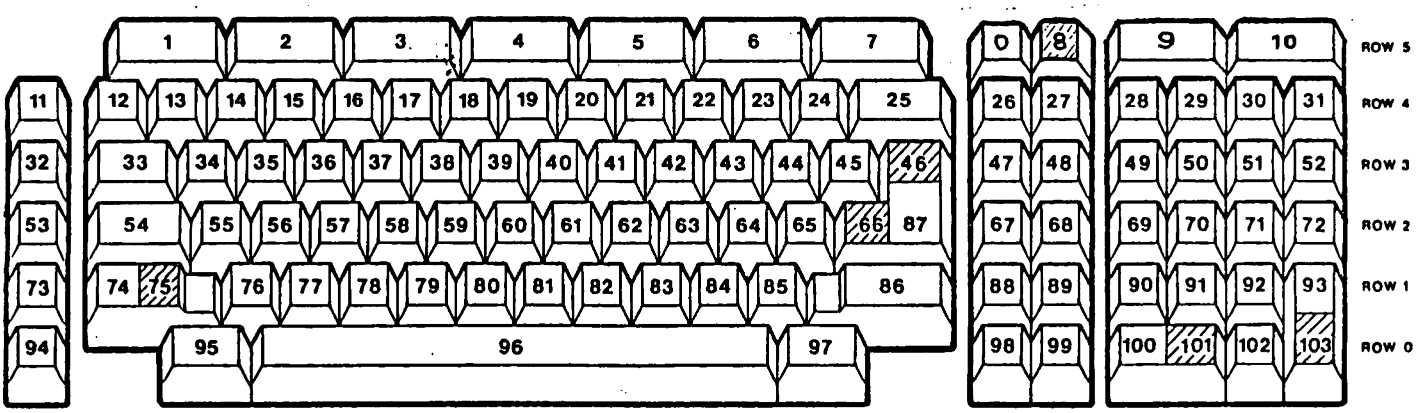


Figure 2
Victor 9000 Keyboard Configuration
with Key Switch Positions and Logical Key Numbers

APPENDIX C

C.1 Victor 9000 Parallel (Centronics) Port

Pin Number	Signal
1	Data Strobe
2	Data 1
3	Data 2
4	Data 3
5	Data 4
6	Data 5
7	Data 6
8	Data 7
9	Data 8
10	ACK
11	Busy
17	Pshield
12,18,30,31	Not connected
Remaining	GND

C.2 Victor 9ØØØ Serial (RS-232C) Port

Pin Number	Signal	
1	FG	Frame Ground
2	TD	Transmitted Data
3	RD	Received Data
4	RTS	Request to Send
5	CTS	Clear to Send
6	DSR	Data Set Ready
7	SG	Signal Ground
8	DCD	Data Carrier Detect
15	TC	Transmitter Clock
17	RC	Receiver Clock
2Ø	DTR	Data Terminal Ready
22	RI	Ring Indicator

C.3 Victor 9000 IEEE-488 Port

The Victor 9000 IEEE-488 cable attaches to the parallel port - the pin number refers to the actual computer port connector; the IEEE-488 pin number refers to the standard IEEE-488 pin-out as they must attach to the parallel port.

The IEEE pin numbers referred to with the (**z) are wires that are to be bound together as twisted pairs.

Pin Number	IEEE Signal	IEEE Pin Number	
1	DAV	6	(**a)
19	GND	18	(**a)
2	DIO1	1	
3	DIO2	2	
4	DIO3	3	
5	DIO4	4	
6	DIO5	13	
7	DIO6	14	
8	DIO7	15	
9	DIO8	16	
10	NRFD	7	(**b)
28	GND	19	(**b)
11	SRQ	10	(**c)
29	GND	22	(**c)
13	NDAC	8	(**d)
33	GND	20	(**d)
15	EOI	5	
17	shield	12	
34	REN	17	
35	ATN	11	(**e)
16	GND	23	(**e)
36	IFC	9	(**f)
27	GND	21	(**f)
20	GND	24	

C.4 Victor 9ØØØ Control Port

Pin Number	Signal
1	-12V
2	-12V
3	Not connected
4	Not connected
5	+12V
6	+12V
7	+5V
8	+5V
9	Not connected
1Ø	Light Pen
11	GND
12	CA1
13	GND
14	CA2
15	GND
16	PAØ
17	GND
18	PA1
19	GND
2Ø	PA2
21	GND
22	PA3
23	GND
24	PA4
25	GND
26	PA5
27	GND
28	PA6
29	GND
3Ø	PA7
31	GND
32	PBØ
33	GND
34	PB1
35	GND
36	PB2
37	GND
38	PB3
39	GND
4Ø	PB4
41	GND
42	PB5
43	GND
44	PB6
45	GND
46	PB7 / CODEC Clock Output
47	GND
48	CB1
49	GND
5Ø	CB2

APPENDIX D

D.1 Example Assembler Shell Program for MS-DOS Interfacing

The Microsoft MACRO-86 assembler follows closely the Intel ASM-86 specifications. The operating system interfacing technique is via a straightforward interrupt (INT 21Hex), with the required operational parameter in the AH register. MS-DOS does not corrupt any registers other than the ones used for the sending or receiving of data. An example of the running and exiting program technique, plus the required assembler directives, follows. The program example is for the small memory model; but it will apply equally well to the compact or large memory model. The 8080 memory model is not recommended as it results in poor usage of the potential of the 8086/8088 processor. At link time, this programming example will generate an .EXE file - the header information on this file type will be found in E.1.

```
title   Example of MS-DOS/MACRO-86 Assembly Programming

dgroup  group   data
cgroup  group   code

msdos   equ     00021h                  ;interrupt to operating system

data    segment public   'data'
;######  insert your data here ######
data    ends

code    segment public   'code'
        assume  CS: cgroup, DS: dgroup

example proc    near                    ;origin of code

begin:
        push    ES                      ;save return segment address
        call    run_module              ;run the program
;
; run ends - select close down
;
exit    proc    far                     ;close down code
        xor     ax,ax                   ;zero for PSP:0
        push    ax                      ;save for far return
        ret                             ;and close down
exit    endp                            ;close down code ends

run_module:
        mov     ax,DATA                 ;get the data segment origin
        mov     DS,ax                   ; and initialize the segment
;##### insert your code at this point ######
        ret                             ;return to exit module

        example endp
        code    ends
        end
```

D.2 Example Assembler Shell Program for CP/M-86 Interfacing

The Digital Research ASM-86 assembler does not follow the standard Intel ASM-86 structure - this makes for a more complex task when transferring assembler programs between the CP/M-86 and the MS-DOS operating systems. The operating system interfacing technique is via a straightforward interrupt (INT EØHex), with the required operational parameter in the CL register. CP/M-86 corrupts all registers, excepting the CS and IP - it is, therefore, recommended that all registers be pushed prior to the INT EØHex being issued. An example of the running and exiting program technique, plus the required assembly directives, follows. The program example follows that of the MS-DOS MACRO-86 example. At GENCMD time, this programming example will generate a .CMD file - the header information on this file type is shown in the System Guide for CP/M-86.

```
title   'Example of CP/M-86/ASM-86 Programming'


reset   equu    ØØØØØh          ;system reset function
cpm     equ     ØØØeØh          ;interrupt to operating system

        cseg

begin:
        call    run_module      ;run the program
;
; run ends - select close down
;
        mov     cl,reset        ;select system reset
        mov     dl,ØØh          ;select memory recovery
        int     cpm             ;return to operating system
;
run_module:
;##### insert your code at this point ######
        ret                     ;return to exit module

        dseg
;##### insert your data here #####
        end
```

E.1 MS-DOS -- EXE File Header Structure

The Microsoft linker outputs .EXE files in a relocatable format, suitable for quick loading into memory and relocation. EXE files consist of the following parts:

- o Fixed length header
- o Relocation table
- o Memory image of resident program

A run file is loaded in the following manner:

- o Read into RAM at any paragraph (16 byte) boundary
- o Relocation is then applied to all words described by the relocation table.

The resulting relocated program is then executable. Typically, programs using the PL/M small memory model have little or no relocation; programs using larger memory models have relocation for long calls, jumps, static long pointers, etc.

The following is a detailed description of the format of an EXE file:

Microsoft .EXE File Main Header

Byte	Name	Function
0+1	wSignature	Must contain 4D5Ahex.
2+3	cbLastp	Number of bytes in the memory image modulo 512. If this is 0 then the last page is full, else it is the number of bytes in the last page. This is useful in reading overlays.
4+5	cpnRes	Number of 512 byte pages of memory needed to load the resident and the end of the EXE file header.
6+7	irleMax	Number of relocation entries in the table.
8+9	cparDirectory	Number of paragraphs in EXE file header.
A+B	cparMinAlloc	Minimum number of 16-byte paragraphs required above the end of the loaded program.
C+D	cparMaxAlloc	Maximum number of 16-byte paragraphs required above the end of the loaded program. 0FFFFh means that the program is located as low as possible into memory.
E+F	saStack	Initial value to be loaded into SS before starting program execution.
10+11	raStackInit	Initial value to be loaded into SP before starting program execution.
12+13	wchksum	Negative of the sum of all the words in the run file.
14+15	raStart	Initial value to be loaded into IP before starting program execution.
16+17	saStart	Initial value to be loaded into CS before starting program execution.
18+19	rbrgrle	Relative byte offset from beginning of run file to the relocation table.
1A+1B	iov	Number of the overlay as generated by LINK-86. The resident part of a program will have iov = 0.

The relocation table follows the fixed portion of the run file header and contains irleMax entries of type rleType, defined by:

```
rleType     bytes 0+1 ra
            bytes 2+3 sa
```

Taken together, the ra and sa fields are an 8086/8088 long pointer to a word in the EXE file to which the relocation factor is to be added. The relocation factor is expressed as the physical address of the first byte of the resident divided by 16. Note that the sa portion of an rle must first be relocated by the relocation factor before it in turn points to the actual word requiring relocation. For overlays, the rle is a long pointer from the beginning of the resident into the overlay area.

The resident begins at the first 512 byte boundary following the end of the relocation table.

The layout of the EXE file is:

28-byte Header

Relocation Table

padding (<200hex bytes)

memory image

F.1 Victor 9000 Technical Specification

Processor

- o Intel 8088 16-bit microprocessor
- o 128k bytes RAM internally upgradeable to 896k bytes
- o 4k bytes Auto-boot ROM (read only memory)
- o 4 internal expansion slots for plug-in card options
- o 2 x RS232C serial communications ports
- o 1 x Parallel (Centronics) or IEEE-488 port
- o 2 x Parallel user port (50-way KK Connector on CPU board)

Display System

- o 25 line x 80 column screen / 50 line x 132 column screen
- o 12" CRT, Green p39 phosphor
- o Adjustable horizontal viewing angle ($\pm$ 45 degree swivel)
- o Adjustable vertical viewing angle (0 deg to 11 deg tilt)

Floppy Drives

- o Standard 5 1/4-inch, single-sided 96 TPI dual disk drives, with a maximum capacity of 600k bytes per drive.
- o Optional 5 1/4-inch, double-sided 96 TPI dual disk drives, with a maximum capacity of 1200k bytes per drive.
- o Optional single 10,000k byte Hard Disk - non-removable; with single 5 1/4-inch, double sided 96 TPI disk drive with a maximum capacity of 1200k bytes.

Single-sided floppy drive offers 80 tracks at 96 TPI
Double-sided floppy drive offers 160 tracks at 96 TPI
Floppy drives have 512 byte sectors; utilising a GCR, 10-bit recording technique.

Floppy access times:

2 micro-second per bit data transfer rate, with an interleave factor of 3. Average seek time is approximately 90 milli-seconds.

Hard Disk access times:

0.2 micro-second per bit data transfer rate, with an interleave factor of 5. Average seek time is approximately 100 milli-seconds.

Keyboard

Separate Intel 8048 microprocessor
Fully software definable with 10 soft function keys
Full IBM Selectric III (56 key) keyboard layout
Type ahead buffering to 32 levels and full n-key rollover
Keyswitches rated for 100 million operations

Electrical

Input voltage 90-137 VAC or 190-270 VAC (internal jumper)
Input frequency 47-63 Hz

Environment

Operating temperature 0 deg C to 40 deg C
Operating humidity 20% to 80% (non-condensing)
Storage temperature -20 deg C to 70 deg C
Storage humidity 5% to 95% (non-condensing)

F.2 Victor 9000 Physical Specifications

Mainframe Assembly

Height	Width	Depth	Weight (approx)
178 mm	422 mm	356 mm	12.6 kg
7 in	16.6 in	14 in	281 lbs

Display Assembly

Height	Width	Depth	Weight (approx)
264 mm	326 mm	339 mm	8.1 kg
10.4 in	12.9 in	13.4 in	18 lbs

Keyboard Assembly

Height	Width	Depth	Weight (approx)
45 mm	483 mm	203 mm	1.5 kg
1.8 in	19 in	6.4 in	3 lbs

System Assembly

Height	Width	Depth	Weight (approx)
457 mm	483 mm	559 mm	22.2 kg
18 in	19 in	20.4 in	49 lbs

Width without the keyboard module is 396 mm / 15.6 in

G.1 Glossary of Terms

The following table is a glossary of terms found in this manual:

BAUD	The term baud rate means the number of bits sent down a line per second. A baud rate of 300 will, therefore, be capable of transmitting data at 300 bits per second. Since a textual character is composed of 8 bits, then 37.5 characters could be sent per second at this baud rate.
BIOS	This means the Basic Input Output System. The BIOS is a fundamental portion of an Operating System, allowing the operating system to communicate correctly with any peripheral devices; typical BIOS modules include the disk driver; the keyboard input driver; the screen driver; the printer driver.
BIT	A bit is a binary digit. The bit can, therefore, contain either One or Zero. A One is bit HIGH or ON. A zero is bit LOW or OFF. A bit may be likened to a light-switch - the switch can only be on or off. See BYTE.
BOOT	This term comes from the phrase "the computer pulls itself up by its boot-strap". The term boot-strap means the same, but is no longer in such common use. To boot a computer is to load an operating system - the computer does this by means of a boot-strap program. The computer, when switched on, is not aware of its environment - but it automatically runs its boot-strap program. The Victor 9000 boot-strap program is stored in the boot PROM; it first causes the display of the little disk picture - it then searches for a disk with an operating system - when it finds this disk, it loads the operating system and begins to execute it. The boot-strap program is not used again until the reset switch is pressed, or the power is switched off and on.
BUS	A bus in computer jargon is not unlike a bus to carry passengers. When data is moved around inside a computer it is moved along the bus wires. These bus wires connect the Victor 9000 microprocessor to its memory, disk(s) and screen.

BYTE	A byte is a collection of 8-bits or two nibbles. A byte may store one character of text, or a number from Ø to 255 in binary.
DOT MATRIX	A printed character on the screen or a dot-matrix printer may be viewed as a square containing dots. On the Victor 9ØØØ screen a character has a square cell (matrix) of 16 dots high by 1Ø dots wide - within this box, the dot on/off patterns create a viewable character.
FONT CELL	In reference to DOT MATRIX, the font cell is the collection of bytes of data that make up the character dots that are to be displayed on the screen. Each character on the screen is composed of pre-defined patterns of dots to make the viewed dot matrix. These patterns of dots are stored in the Victor 9ØØØ memory as data - the screen controller chip scans these data bytes and the resulting character image is displayed on the screen.
HEADER	A header on a file gives information to the operating system on where and how the file is to be loaded in to memory. Many files provided by Victor Technologies (such as keyboard and character set files) contain headers that are not used by the operating system, but are used by Victor Technologies utilities.
INTERRUPT	An interrupt is some event occuring in the computers environment that the computer will stop all other activities for. An example of an interrupt is a key-press. If you press a key on the Victor 9ØØØ, an interrupt is generated; at this point the processor stores all information on its current task and gets and saves the value of the key pressed; it then picks up all the information it stored on its last task and continues where it left off. This whole series of events takes only a few micro-seconds.
NIBBLE	Sometimes spelled NYBBLE; a nibble is half a byte or 4-bits. See BYTE and BIT.
OPERATING SYSTEM	An operating system allows the computer to be aware of its environment and gives the user the ability to enter and retrieve data from the computer.

PROM	Programmable Read Only Memory, PROM, is a chip or collection of chips that is used to store permanently a single computer program or collection of computer programs. The boot-prom, sometimes called boot-rom, contains all the information the Victor 9000 computer needs to read an operating system from disk. There are different types of prom; EPROM which is erasable prom, simply shine a high-powered ultra-violet lamp on the chip, and it can be re-programmed; etc.
RAM	Random Access Memory, RAM, is a chip or collection of chips that is used to store temporarily (until the power is removed) data, computer program(s), text, etc. This is the memory of a computer.
REGISTER	A computer register is a portion of the processor. The Victor 9000 uses the Intel 8088 micro-processor - there are several different types of registers within this chip; there are 8-bit registers, and 16-bit registers. Data is generally not manipulated in RAM, but is brought in to a register of the processor and manipulated there, then the result saved from the register back into RAM.
WORD	A word is a number of bits, generally greater than 8. The Victor 9000 has a 16-bit word - thus a word in the Victor 9000 is composed of two bytes. The DEC PDP-8 computer has a 12-bit word - on this machine, therefore, a word is composed of one byte and one nibble.

H.1 MS-DOS Base Page Structure

The MS-DOS Base Page (sometimes called the Program Segment Prefix or PSP), is created when you enter an external command. COMMAND.COM will allocate a memory region to the external program, and will insert the Base Page prior to the origin of this program.

In the memory segment that the program is to load, COMMAND.COM places a Base Page, COMMAND.COM then loads the program at an offset of 100hex, and hands over control to the external program. The external program, once its function is complete, hands control back to the operating system by a far JUMP or far RETURN to location zero within the Base Page; the instruction at this location is an INT 20, or return control to MS-DOS. This stage must be executed to allow MS-DOS to recover memory correctly (see Appendix D.1).

When an external program is loaded, the following conditions are true:

> The file control blocks at Base Page locations 5Chex and 6Chex are created from the first two parameters entered on the command line.
>
> The command line at Base Page location 80hex is created from the command line entered AFTER the program filename. The byte at location 80hex contains the command line character count, the following bytes contain the raw command line as entered at the keyboard.
>
> The word at offset 6 in the Base Page contains the number of bytes available in the segment.
>
> The contents of register AX are established to reflect the validity of the drive(s) on the command line. Thus the following may be found:
>
> > AL = FFhex when the first drive letter on the command line was not recognized by MS-DOS.
> > AH = FFhex when the second drive letter on the command line was not recognized by MS-DOS.

The above applies equally to both .EXE and .COM type files. The .EXE and .COM files do have differences when the they load, and these are described more fully below.

When .EXE files load:

The contents of register DS and register ES are pointing at the Base Page segment address.

The registers CS, IP, SS and SP are initialized to those values passed by the linker.

When .COM files load:

The contents of registers CS, DS, ES, and SS are pointing to the Base Page segment address.

The register IP is set at 100hex.

The register SP is set the high address in the program segment, or to the base of the transient portion of COMMAND.COM, whichever is the lower. The contents of the word at Base Page offset 6 are decremented by 100hex to allow for a stack of that size.

A word of zeros is placed at the top of the stack.

The Base Page

The Base Page is structured as follows - with offsets in Hex

Offset	Contents
0000	INT 20hex. Word.
0002	Total Memory size in paragraph form (i.e. 2000hex is equivalent to 256k bytes). Word.
0005	Far CALL to MS-DOS function dispatcher. 5 bytes.
000A	Program Terminate address as IP and CS. 2 words.
000E	Control Break address as CS and IP. 2 words.
005C	File Control Block #1, formatted as normal unopened FCB. 8 words.
006C	File Control Block #2, formatted as normal unopened FCB. 8 words.
0080	Count of characters on command line; followed by command line entered. This region may be used as disk transfer address.

Normal File Control Block

The normal file control block is structured as follows - with offsets in decimal:

Byte	Contents
Ø	The drive number. The drives are numbered as follows: Before opening file: Ø=default drive 1=drive A 2=drive B 3=drive C, etc After opening file: 1=drive A 2=drive B, etc MS-DOS replaces the default drive prefix of Ø with the correct drive number after the open is processed.
1-8	Filename, left justified with trailing ASCII space(s). If a device name is placed in this region, the trailing colon should be omitted.
9-11	Extent, left justified with trailing ASCII space(s).
12-13	Current block number relative to the beginning of the file, starting with zero (automatically set to zero by the open function request). A block consists of 128 records, each record being of the size specified in the logical record size field. The current block number is used with the current record field for sequential reads/writes.
14-15	Logical record size in bytes. Set to 8Øhex by the open function request.
16-19	File size in bytes. The first word represents the low-order part of the file size.

20-21 Date the file was created or last updated. The date is set by the open function request. The date is formatted as follows:

```
<                21          > <          20            >
15 14 13 12 11 10  9   8   7   6   5   4   3   2   1   0
y  y  y  y  y  y   y   m   m   m   m   d   d   d   d   d
```

where m	month	1thru12
d	day	1thru31
y	year	0thru119 (1980thru2099)

22-23 Time the file was created or last updated. The time is set by the open function request. The time is formatted as follows:

```
<                23          > <          22            >
15 14 13 12 11 10  9   8   7   6   5   4   3   2   1   0
h  h  h  h  h  m   m   m   m   m   m   s   s   s   s   s
```

where h	hours	0thru23
m	minutes	0thru59
s	seconds*2	0thru59

24-31 Reserved for system use.

32 Current relative record number (0-127) within the current block. This must be set before doing seqeuntial read/write operations on the file. The open function request does not set this field.

33-36 Relative record number, relative to the origin of the file, starting at zero. This field must be set prior to doing random read/write operations on the file. The open function request does not set this field.

If the record size is less than 64 bytes, both words are used. If the record size is greater than 64 bytes, then only the first three bytes are used.

Notes: The File Control Block at 5Chex in the Base Page overlaps both the File Control Block at 6Chex and the first byte of the command line area/disk transfer area at 80hex.

Bytes 0thru15 and 32thru36 must be set by the user program. Bytes 16thru31 are set by MS-DOS and may only be changed at the programmers own risk.

In the 8086/8088 all word fields are stored least significant byte first - this is true in setting the record length, etc.

Extended File Control Block

The extended FCB is used to create or search for files having special attributes. The extended FCB adds an additional 7 bytes preceeding the normal FCB. The extended FCB is structured as follows:

Byte	**Contents**
FCB-7	Set to FFhex indicates that an extended FCB is being used.

FCB-6 to FCB-2 are reserved.

FCB-1	Attribute byte to include hidden files (Ø2hex) or system files (Ø4hex) in directory searches.
FCB-Ø	Origin of normal FCB (drive byte).

APPENDIX I

I.1 Interrupt Driven Serial Input/Output

This appendix is designed to show the methodology involved in driving the Victor 9000 in interrupt mode when communicating via the serial port(s). Some pitfalls are described, and tested sample routines are included. There are, currently, no system level facilities that enable this task to be accomplished easily, and some chips, namely the PIC 8259, PIT 8253, SIO 7201 and the VIA 6522 will require re-programming. It is up to the programmer to reset the machine to the original state prior to exiting the interrupt driven application.

A typical interrupt driven application will normally follow the steps outlined below:

1) Save the original vector, set the new vector.
2) Set the direction bits.
3) Enable clocks (internal or external).
4) Reset SIO 7201 device, define your communication characteristics.
5) Set the baud rate.
6) Set the PIC 8259 to enable SIO interrupts.

These steps will be discussed in more detail throughout the text.

I.2 Interrupt Vectors

There are 256 software interrupts available to the Victor 9000. Most are reserved for system functions, and diagnostics. A block of vectors from 80Hex thru BFHex are set aside for applications.

I.2.1 Vectors available on the Victor 9000

00 - 1FHex	Intel reserved.
20 - 3FHex	Microsoft reserved.
40 - 7FHex	Victor reserved.
80 - BFHex	Applications reserved.
C0 - FFHex	Victor reserved.

Vectors 40Hex thru 47Hex are those belonging to devices controlled by the Programmable Interrupt Controller (PIC).

40Hex	Sync IRQ
41Hex	SIO 7201
42Hex	Timer 8253
43Hex	General Interrupt Handler (all 6522 IRQ's)
44Hex	IRQ4
45Hex	IRQ5
46Hex	Keyboard - keystroke
47Hex	8087 math processor

I.2.2 Location of Vectors

Vectors consist of a long pointer (double word) to an interrupt service routine. This pointer is a 4 byte entry consisting of the Segment and Offset of the Interrupt Service Routine. The vectors are stored in a table that has its origin at 0000:0000. The first entry in this table is, therefore, Interrupt 0; the vector for Interrupt 1 is the second, with its vector having an origin of 0000:0004. The interrupt vector for Interrupt 41Hex (the SIO 7201) will be found at location 0000:0104 (4*41Hex).

To set a vector into this table, the MS-DOS function 25Hex can be used, but since it is desirable to restore the old vector prior to the application program exiting, it is less cumbersome to simply set the new vector "by hand", and restore the old vector when the application terminates.

I.2.3 Set Vector - Assembler Example

```
;store old vector, and set new vector for SIO

     cli                                    ;clear interrupts
     xor  ax,ax                             ; AX = 0000
     mov  ES,ax                             ;access table via ES
     mov  ax,word ptr ES:[104h]             ;get old offset
     mov  word ptr old_offset,ax            ;save old offset in DS
     mov  ax,word ptr ES:[106h]             ;old segment
     mov  word ptr old_segment,ax           ;save old segment
     mov  ax,my_sio_isr                     ;get offset to my code
     mov  word ptr ES:[104h],ax             ;set vector offset
     mov  word ptr ES:[106h],CS             ; and the new segment
     sti                                    ;enable interrupts
     ret                                    ;all done, exit

;to replace the old vector prior to exit

     cli                                    ;clear interrupts
     xor  ax,ax                             ; AX = 0000
     mov  ES,ax                             ;access table via ES
     mov  ax,word ptr old_offset            ;get old offset
     mov  word ptr ES:[104h],ax             ;restore old offset
     mov  ax,word ptr old_segment           ;get old segment
     mov  word ptr ES:[106h],ax             ;restore old segment
     sti                                    ;enable interrupts
     ret                                    ;all done, exit
```

I.3 Enabling Internal and External Clocks

In an asynchronous environment the transmit clock is generated internally, as opposed to a synchronous environment where the transmit clock is typically provided by an external source.

Internal clocking is selected by masking off the appropriate bit in register 1 of the keyboard Versatile Interface Adaptor (VIA).

The keyboard VIA, resgister 1, is located at **E8Ø4:ØØØ1.**
The appropriate bits are:

Bit Ø (PAØ) for port A
Bit 1 (PA1) for port B

Thus, by setting PAØ to zero, the internal clock is enabled for port A; setting PA1 to zero will enable the internal clock for port B. Setting PAØ or PA1 to one will enable the external clock, disabling the internal clock. CAUTION: Care must be taken to leave the other bits in the pre-selected state.

To enable internal clocks for ports A and B mask off the two least significant bits in register 1:

```
mov  ax,Øe8Ø4h                  ;keyboard VIA segment
mov  ES,ax                      ;select the segment register
and  byte ptr ES:[ØØØ1],Øfch    ;A & B internal clocks done
```

To enable external clocks on either channel then set the relevent bit by OR'ing the bit in. The following sample sets the external clocks for both ports A and B:

```
mov  ax,Øe8Ø4h                  ;keyboard VIA segment
mov  ES,ax                      ;select the segment register
or   byte ptr ES:[ØØØ1],Ø3h     ;A & B external clocks done
```

I.3.1 Providing Clocks

In a synchronous environment it sometimes becomes necessary to provide transmit and receive clocks from the Victor 9ØØØ. This requires that the cable used has pins 15, 17 and 24 jumpered at the Victor 9ØØØ end. The Victor 9ØØØ always has a clock on pin 24, this being provided by the internal baud rate generator; thus by jumpering pin 24 to both pins 15 and 17, this clock becomes available for both the transmitter and the receiver, at both ends of the cable.

When providing clocks from the Victor 9ØØØ, the external clock must be set as well as a baud rate selected. In synchronous mode, the "divide by rate" of the PIT 8253 is 1, therefore the values used to set the required baud rate is 1/16 the values used in an asynchronous environment. (See section 3.8.2 for values).

I.4 Initializing the SIO

There is little magic used in this step, but it is recommended that the programmer read the entire Intel/NEC 7201 chip data sheet. The SIO segment is found in segment location E004Hex. The offsets for the data ports A and B and control ports A and B are at 0, 1, 2, 3 respectively.

The following example of initializing the SIO 7201 is for Port A:

```
     cli                              ;disable interrupts
     mov  ax,0e004h                   ;the SIO segment
     mov  ES,ax                       ;  using ES
     mov  byte ptr ES:[0002h],18h     ;channel reset

; now delay at least 4 system clock cycles

     nop
     nop                              ;delay for 7201

     mov  byte ptr ES:[0002h],12h     ;reset external/status
                                      ;  interrupts

;and select register 2

     mov  byte ptr ES:[0002h],14h     ;non-vectored
     mov  byte ptr ES:[0003h],02h     ;select CR2 B
     mov  byte ptr ES:[0003h],00h     ;set vector to 0

; set for clock rate of 16*; 1 stop bit; parity disabled

     mov  byte ptr ES:[0002h],04h     ;select CR4 A
     mov  byte ptr ES:[0002h],44h     ;

; this register defines the operation of the receiver:
; 7 data bits; auto enable and receive enable

     mov  byte ptr ES:[0002h],03h     ;select CR3 A
     mov  byte ptr ES:[0002h],61h     ;

; CR5 controls the operation of the transmitter
; 7 data bits, dtr; assumes half-duplex

     mov  byte ptr ES:[0002h],05h     ;select CR5 A
     mov  byte ptr ES:[0002h],0a0h    ;

; set status:  affects the vector, interrupt on every character,
; enable transmitter interrupt

     mov  byte ptr ES:[0002h],01h     ;select CR1 A
     mov  byte ptr ES:[0002h],17h     ;

     sti                              ; enable interrupts
```

I.4.1 Baud Rate for SIO

At this point, baud rate must be selected. In an asynchronous environment the PIT 8253 divides the supplied baud rate by 16; but in a synchronous environment the baud rate is divided by 1. Thus, to set the baud rate in an asynchronous environment, the value written to the PIT 8253 is 16 times the desired baud rate value. The common baud rate values, and the method of establishing the baud rates, are shown in Section 3.8.2 of this manual.

I.4.2 Set the PIC to Enable SIO Interrupts

In the Victor 9ØØØ the PIC is normally initialized to operate the SIO in a polled environment. The following lines of code sets the PIC to operate the SIO in an interrupt environment:

The PIC resides at segment EØØØHex and the register required here is at offset ØØØ1:

```
cli                            ;disable interrupts
mov  ax,Øe000h                 ;get the PIC segment
mov  ES,ax                     ;
and  byte ptr ES:[ØØØ1h],(not Ø2h) ;mask off bit 1
.
.
sti                            ;allow interrupts
```

Prior to exiting the interrupt drievn application, the PIC should be returned to operating the SIO in polled mode. This is done by setting bit 1:

```
cli                            ;disable interrupts
mov  ax,ØeØØØh                 ;get the PIC segment
mov  ES,ax                     ;
or   byte ptr ES:[ØØØ1h],Ø2h ;set polled
sti                            ;allow interrupts
```

I.5 Interrupt Service Routine - ISR

When an interrupt occurs in non-vectored mode, SIO register CR2 B contains the vector number of the interrupting device. Assuming the SIO was initialized as earlier described in this appendix, CR2 B contains a value in the range Ø-7, which serves as the index to the following interrupt vector table:

I.5.1 Sample Interrupt Service Routine

```
data segment   public    'data'
int_vectors    dw   tx_int_b           ;tx int for port B
               dw   ext_status_b       ;external status changed
               dw   recv_int_b         ;recv int port B
               dw   recv_err_b         ;recv error port B
               dw   tx_int_a           ;tx in for port A
               dw   ext_status_a       ;external status changed
               dw   recv_int_a         ;recv in port A
               dw   recv_err_a         ;recv error port A
data ends

code segment   public    'code'
     assume    CS:cgroup, DS:dgroup

sio_isr:
     mov  word ptr CS:current_ss,SS    ;save stack seg
     mov  word ptr CS:current_sp,SP    ; and stack pointer
     mov  SS,word ptr CS:ss_origin     ;internal stack
     mov  SP,offset dgroup:stack_top   ; defined in DS (dgroup)

     push ax                           ;save environment
     push bx
     push cx
     push dx
     push bp
     push DS
     push ES

     mov  DS,dgroup                    ;set to internal data
     mov  ax,0e004h                    ;set SIO segment
     mov  ES,ax                        ;
     mov  byte ptr ES:[0003h],02h      ;select CR2 B
     mov  al,ES:[0003h]                ;read int device
     add  al,al                        ;word align
     mov  ah,0                         ; hi = 0
     mov  bx,offset int_vectors        ;get vector table
     add  bx,ax                        ;point to entry
     call [bx]                         ;service routine
     cli                               ;keep disabled
```

--See next page for continuation--

I.5.1 continued

```
; now an "end of interrupt" (EOI) must be issued to the
; SIO (port A) and to the PIC.

     mov  ax,0e000h                          ;PIC segment
     mov  DS,ax                              ;
     mov  byte ptr [0042h],38h               ;EOI to ctrl A of SIO
     mov  byte ptr [0000h],61h               ;EOI to PIC ctrl port A

     pop  ES                                 ;restore environment
     pop  DS
     pop  bp
     pop  dx
     pop  cx
     pop  bx
     pop  ax

     mov  SS,word ptr CS:current_ss          ;get SS
     mov  SP,word ptr CS:current_sp          ;get SP
     iret                                    ;interupt return

; the SS origin is stored here during initialization
ss_origin      dw   0                        ;stack segment origin
current_sp     dw   0                        ;SP on ISR entry
current_ss     dw   0                        ;SS on ISR entry
```

NOTE: Some variables are stored within the code segment, as the CS register is the only register containing a known value at the time of interrupt.

I.6 Setting Direction Bits

This function need only be performed once, and is performed by the operating system BIOS following a hardware reset. This step need not be implemented, therefore, if a standard Victor or Sirius operating system is used. If a standard operating system is not used, then this step needs to be performed immediately prior to the enable clock code.

```
;The offset to the data direction register is ØØØ3Hex.

     cli                                ;disable interrupts
     mov  ax,Øe8Ø4h                     ;kbd VIA segment
     mov  ES,ax                         ;
     mov  al,byte ptr ES:[ØØØ3h]        ;get the old value
     or   al,Ø3h                        ;set for output
;
; set the PA2-5 to zero, to enable DSR and RI input
;
     and  al,Øc3h                       ;mask in
     mov  byte ptr ES:[ØØØ3h],al        ;rewrite new value
     .
     .
     sti                                ;enable interrupts
```

APPENDIX J

J.1 Character Set Header

All files with the extension .CHR are Character Set table files. These files contain data corresponding to the actual dot matrix displayed for each character on the console. These files also contain information regarding the character set name, version number, origin, date of creation, and display class. The Character Set table file header is a 128 byte field, structured as follows:

Byte No. Hex	Dec	Function
ØØ	ØØ	Character Set type, ASCII 'C' = character
Ø1	Ø1	Character Set Version Number (ASCII Ø thru 9)
Ø2-ØD	Ø2-13	Display Class
ØE-15	14-21	Character Set Name
16	22	Filler (ASCII Space)
17-19	23-25	Banner Class
1A	26	Filler (ASCII Space)
1B-3D	27-61	Comment
3E-4D	62-77	Originator
4E-55	78-85	Creation Date - arranged as YY/MM/DD
56-59	86-89	Number of records in the file in ASCII. A character set file of 128 characters has 32 records; a character set file of 256 characters has 64 records. The record count for a 32 record file is stored as 3Ø 3Ø 33 32 (ØØ32).
5A-5B	9Ø-91	Reserved.

Over...

Byte No. Hex	Dec	Function
5C	92	This byte is used to house three variables. Bit 7 is used to show the Horizontal/Vertical alignment of the character set - bit 7 ON infers a Vertical character set. Bits 6 thru 4 of the high nibble is used to store the binary Super/Subscript value (which may be 1 thru 7) offset from 1 - thus a Super/Subsript value of two would be stored as binary 2. The low nibble is used to store the binary Character Height offset from Ø - thus a Character Height value of 16 would be stored as binary F. The Character height is a function of the number of vertical pixels the character will occupy in the 16x1Ø pixel matrix available for each character on the screen. If the Horizontal/Vertical bit, the Super/Subscript value and the Character Height value was as stated above, then this byte would read AF. The byte appears:

Bit	[7]	[6 5 4]	[3 2 1 Ø]
Function	Horiz/Vert	Super/Sub	Character Height

5D	93	This byte contains two values; the User/System character set toggle, bit Ø stores this value; and the Stock/Special character set toggle, bit 1 stores this value. Bit Ø ON infers that the character set is a system character set. Bit 1 ON infers that the character set is a special character set.
5E	94	This byte contains information on the character set width. If the high nibble is Ø, then the low nibble contains the binary information, offset from Ø, of all the characters in the character set - thus a character set width value of 16 would be stored as F. If the high nibble is F, then the character set is a proportional one - the proportional character set has a trailing record containing information on the width of each individual character in the character set. A proportional character set is designed to be used in high-resolution mode as it requires a 16x16 screen cell.
5F-7F	95-127	Reserved.
8Ø-	128-	The character set font information.

Sample Character Set Table File Header

Following is an actual header taken from the Character Set Table file for the character set PROP.CHR. PROP contains 128 characters, and is a proportional character set:

Hex Offset	Value in Hex															
0:	43	30	49	6E	74	27	6C	20	20	20	20	20	20	20	50	52
10:	4F	50	20	20	20	20	20	43	48	52	20	54	68	69	6E	20
20:	70	72	6F	70	6F	72	74	69	6F	6E	61	6C	20	63	68	61
30:	72	61	63	74	65	72	20	73	65	74	20	20	20	20	53	69
40:	72	69	75	73	20	53	79	73	74	65	6D	73	20	20	38	32
50:	2F	30	37	2F	31	36	30	30	33	30	00	00	7F	00	FF	00
60:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
70:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

J.2 Proportional Character Set Trailer Information

In the case of a proportional character set, the trailing 128 bytes of the character set file contains information on the proportional width of each of the characters in the file. A proportional character set may not, therefore, contain more than 256 characters.
The following is a sample taken from the character set PROP.CHR; the hex figures represent the width for each proportional character starting with the space character. Note that each width value is offset from Ø, with a value range of 1 thru 16 decimal. Each byte is stored, and represented below, in low/high order; the two nibbles would be exchanged to give the value to the character(s) in high/low order. Each character is mapped from the proportional width as follows:

    29 95 98 49 77 88 84 93

The above figures are for the first 16 display characters including the space character - they correspond as follows:

```
space = 1Ø (corresponding to 9)
!     = 3  (corresponding to 2)
"     = 6  (corresponding to 5)
#     = 1Ø (corresponding to 9)
$     = 9  (corresponding to 8)
%     = 1Ø (corresponding to 9)
&     = 1Ø (corresponding to 9)
'     = 5  (corresponding to 4)
(     = 8  (corresponding to 7)
)     = 8  (corresponding to 7)
*     = 9  (corresponding to 8)
etc
```

J.3 Keyboard Table Header

All files with the extension .KB are Keyboard Table file. These files contain information regarding keyboard code generated when a key on the keyboard is pressed. These files also contain information regarding the Keyboard Table name, version number, origin, date of creation, and display class. The Keyboard Table table file header is a 128 byte field, structured as follows:

Byte No. Hex	Dec	Function
ØØ	ØØ	Keyboard table type, ASCII 'K' = character
Ø1	Ø1	Keyboard table Version Number (ASCII Ø-9)
Ø2-ØD	Ø2-13	Display Class
ØE-15	14-21	Keyboard Table Name
16	22	Filler (ASCII Space)
17-19	23-25	Banner Class
1A	26	Filler (ASCII Space)
1B-3D	27-61	Comment
3E-4D	62-77	Originator
4E-55	78-85	Creation Date - arranged as YY/MM/DD
56-59	86-89	Number of records in the file in ASCII. A character set file of 128 characters has 32 records; a character set file of 256 characters has 64 records. The record count for a 32 record file is stored as 3Ø 3Ø 33 32 (ØØ32).
5A-7F	9Ø-127	Reserved.
8Ø-	128-	Keyboard table information.

J.4 Banner Skeleton Files

Files with the extension .BAN are banner skeleton files. The banner is information printed on the screen during system boot. The banner also prints the Logo (if selected) along with other information regarding configuration. The banner is a set of ASCII strings containing the escape sequences and characters necessary to print the logo and configuration information on the console.

The first 128 bytes of the Banner Skeleton has the following format. The first byte is zero followed by ØDh, ØAh. This is followed by the length of the file in ASCII decimal with a leading and trailing space, and followed by ØDh, ØAh.

The location of the keyboard name and character set name follow in the same format as the file name length. If the file length is 639 characters, the keyboard name is at byte 5Ø2, and the character set name is at 541, then the first 24 bytes of the banner file would be as follows:

```
30 0D 0A 20 36 33 39 20 0D 0A 20 35 30 32 20 0D 0A 20
35 34 31 20 0D 0A
```

APPENDIX K

K.1 Victor 9000 Disk Structure

K.1.1 Victor 9000 Floppy Disk Structure

The Victor 9000 disk system requires that each track has a variable number of sectors, with each sector containing 512 bytes, with 4 sectors per Allocation Unit (AU), the track structure is as follows:

Track Format

Zone Number	Track Numbers Lower Head (*)	Upper Head	Sectors Per Track	Rotational Period (MS)
0	0-3	(unused)	19	237.9
1	4-15	0-7	18	224.5
2	16-26	8-18	17	212.2
3	27-37	19-29	16	199.9
4	38-48	30-40	15	187.6
5	49-59	41-51	14	175.3
6	60-70	52-62	13	163.0
7	71-79	63-74	12	149.6
8	(unused)	75-79	11	144.0

Notes:
(*) The upper head is not present on the single-sided floppy machine; only the double-sided floppy machine has the upper and lower heads as specified in the table.

MS-DOS allocates space on a Single Sided diskette (SS) and a Double Sided (DS) diskette as follows:

Track 0 Sector 0	Disk Label
Track 0 Sectors 1-2	Two copies of the File Allocation Table (FAT), one FAT in each sector. (SS).
Track 0 Sectors 1-4	Two copies of the FAT, two sectors per FAT. (DS).
Track 0 Sectors 3-10	Directory (SS)
Track 0 Sectors 5-12	Directory (DS)
Track 0 Sectors 11-	Data Region (SS)
Track 0 Sectors 13-	Data Region (DS)

Files, under MS-DOS, are not necessarily written sequentially on the diskette. Diskette space for a file in the data region is allocated on a sector by sector basis, skipping any currently allocated sectors. The first unused sector found in the data region will be the next sector used, regardless of where it appears on the diskette, This method allows for the most efficient use of the disk space available, as sectors made available once a file has been erased can be re-allocated to new files.

K.1.2 Victor 9000 Hard Disk Structure

The hard disk system, in the Victor 9000, is split into virtual volumes - thus what is in reality one physical disk may be broken into several virtual disks. This means that one large disk system is broken up into several smaller, and therefore, more managable smaller 'disks'.

The virtual volumes are described by a volume list placed in the drive label by the hard-disk configuration utility. This list could be of any length, but in practice will contain only a few entries. Partitioned into smaller 'disks', where each hard disk partition will appear as contiguous storage to the user; this is achieved by dividing the physical address space into Regions and translating logical addresses into these areas. Regions typically represent usable areas between unusable spots in the media. The initial Region list is created after the unit is formatted and configured, and it is ordered by physical address. If areas of the disk should become 'bad' during use, the list can be re-ordered to effectively replace the bad track with a spare track located elsewhere on the disk.

K.1.2.1 Victor 9000 Hard-Disk Label Format

The hard-disk has a label that is used both at boot and run time, this label informs the system of the size and structure of the hard-disk media. Located in sector 0, the label is as follows:

Field Name	Data Type	Contents	BOOT	BIOS	HDSETUP	TEST
Label_Type	WORD	0000 = unqualified 0001 = Current Rev.	R	R	R/W	O
Device_ID	WORD	0001 = Current Rev.	R	R	R	W
Serial_Number	BYTE(16)	ASCII	-	-	R	W
Sector_Size	WORD	512	R	R	R	W
IPL_Vector						
Disk_Address	DWORD	Logical Address	R	-	W	O
Load_Address	WORD	Paragraph Number				
Load_Length	WORD	Paragraph Count				
Cod_Entry	PTR	Memory Address				
Primary_Boot_Volume	WORD	Virtual Volume #	-	R	W	O
Control_Parms	BYTE(16)	(for Tandon TM603SE)	R	R	-	W
# Cylinders	BYTE(Hi)	00Hex				
	BYTE(Lo)	E6Hex (=230)				
# Heads	BYTE	06Hex (=6)				
1st reduced-	BYTE(Hi)	00Hex				
current cyl.	BYTE(Lo)	80Hex (=128)				
1st write-	BYTE(Hi)	00Hex				
precomp cyl.	BYTE(Lo)	80Hex (=128)				
ECC data burst	BYTE	0BHex (=11)				
Options	BYTE	02Hex (=2)				
Interleave	BYTE	05Hex (=5, note that 0 means 5)				
Spares	BYTE(6)	00Hex				
Available_Media_List			-	-	R	W
Region_Count	BYTE	Number of Regions				
Region_Descr	(var)	(Variable by Region_Count)				
Region_PA	DWORD	Physical Address				
Region_Size	DWORD	Block Count				
Working_Media_List			R	R	R/W	O
Region_Count	BYTE	Number of Regions				
Region_Descr	(var)	(Variable by Region_Count)				
Region_PA	DWORD	Physical Address				
Region_Size	DWORD	Block Count				
Virtual_Volume_List			-	R	R/W	O
Volume_Count	BYTE	Number of Virtual Vols.				
Volume_Address	DWORD	Virtual Volume label Logical Address				

The above table describes those elements found in the hard-disk label, following is a discussion of the meanings of the entries themselves:

- o Label Type - this defines the state of the drive layout and the revision level of the label.
- o Device ID - Classification identifying the arrangement, for example, the drive manufacturer, controller revision number. This allows for the identification of compatible controllers/drives.
- o Serial Number - the serial number of the unit is stored here.
- o Sector size - the physical atomical unit of storage on the media.
- o Initial Program Load Vector (IPL) - this is a descriptor identifying the boot program and its location on disk. This information is generated from the primary boot volume label via the utility HDSETUP.
    - o Disk Address - the logical disk address of the boot program image.
    - o Load Address - the paragraph address of the memory where the boot program is to load. A zero entry indicates a default load to the highest RAM location.
    - o Load Length - the length of the boot program in paragraphs.
    - o Code Entry - a long memory address of the starting entry of the boot program. Segment of zero defaults to the segment of the loaded program.
- o Primary Boot Volume - the logical address of the virtual volume label containing the IPL vector and configuration information.
- o Controller Parameters - a list of controller dependent information, for use in device reset and configuration.
- o Available Media List - a list of permanent usable areas of the disk. This is derived from the available media list and from the format funciton of the HDSETUP utility.
    - o Physical Address - disk address of the region.
    - o Region Size - the number of physical blocks in the region.

- Working Media List - a list of the working areas of the disk. This is derived from the Available Media List and from the format function of the HDSETUP utility.
    - Physical Address - disk address of the region.
    - Region Size - the number of physical blocks in the region.
- Virtual Volume List - a list of the logical disk addresses of all virtual volume labels.

K.1.2.2 Victor 9000 Hard-Disk Virtual Volume Label Format

The Virtual Volume Label provides information on the structure of the Virtual Volume. Generally the operating system references this label, while the HDSETUP utility will create and reference it. The Virtual Volume Label appears as follows:

Field Name	Data Type	Contents	BOOT	BIOS	HDSETUP	TEST
Label_Type	WORD	0000 = null	-	R	R/W	-
Volume_Name	BYTE(16)	ASCII	-	-	R/W	-
IPL_Vector			R	-	W	O
Disk_Address	DWORD	Virtual Address				
Load_Address	WORD	Paragraph Number				
Load_Length	WORD	Paragraph Count				
Code_Entry	PTR	Memory Address				
Volume_Capacity	DWORD	# of Physical Blocks	-	R	R/W	-
Data_Start	DWORD	Virtual Address	-	R	R/W	-
Host_Block_Size	WORD	MS-DOS = 512 bytes	-	R	R/W	-
Allocation_Unit	WORD	# of Physical Blocks	-	R	R/W	-
Number_of_Directory_Entries	WORD	Entry Count	-	R	R/W	-
Reserved	BYTE(16)	Future Expansion Set to Nulls	-	-	W	-
Configuration_Information			-	R	R/W	-
Assignment_Count	BYTE	# of assignment mappings				
Assignment	(var)	(Variable by Assignment_Count)				
Device_Unit	WORD	Physical Unit Number				
Volume_Index	WORD	Index into Virtual Volume List				

The above table describes those elements found in the hard-disk Virtual Volume label, following is a discussion of the meanings of the entries themselves:

- o Label Type - this defines the type of operating environment that the virtual volume is configured for. It is used for type checking when assigning volumes to drives.

- o Volume Name - the name of the virtual volume as defined by the user. It is used for identifying volumes.

- o Initial Program Load Vector (IPL) - this is a descriptor identifying the boot program and its location within the virtual volume. This field is used to generate the IPL vector on the drive label when configuring the primary boot volume.

    - o Disk Address - the virtual disk address of the boot program image.
    - o Load Address - the paragraph address of the memory where the boot program is to load. A zero entry indicates a default load to the highest RAM location.
    - o Load Length - the length of the boot program in paragraphs.
    - o Code Entry - a long memory address of the starting entry of the boot program. Segment of zero defaults to the segment of the loaded program.

- o Volume Capacity - the number of actual blocks that comprise the virtual volume.

- o Data Start - the offset (in blocks) into the virtual volume for the start of data space.

- o Host Block Size - the atomical unit used by the host in data trasnsfer operations.

- o Allocation Unit (AU) - this operating system dependent field means the storage allocation size used by the host in the virtual volume. It is used in determining disk parameter tables and disk definitions.

- o Number of Directory Entries - this operating system dependent field means the number of entries in the hosts directory. It is used in determining disk parameters tables and disk definitions.

- o Configuration Information - a list of the drive assignments for a system at boot time. It is used to map logical drives to virtual volumes. This field is referenced via the label of the booted drive.

K.2 MS-DOS Disk Directory Structure

The FORMAT/HDSETUP utilities structure the directory for 128 entries on a floppy diskette, and a user defined number on the hard-disk. The directory entries are structured as follows:

0-7 — Filename (0E5Hex in byte 0 indicates that this directory entry is unused).

8-10 — Filename extension.

11 — File attribute. In MS-DOS 1.25, the contents of this byte may be 02Hex indicating a hidden file and 04Hex indicating a system file. A directory search will not show files with the above attributes, unless the extended FCB is used. Files without attributes will contain 00Hex in this byte. A file may be made hidden/system only when created.

12-23 — Reserved.

24-25 — Date when file was created or last updated. The mm/dd/yy are mapped as follows:

```
<           25          > <       24       >
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
y  y  y  y  y  y   y  m  m  m  m  d  d  d  d  d
```

where:

yy is a value from 0-199 (1980-2099)
mm is a value from 1-12
dd is a value from 1-31

26-27 — Starting AU; the relative AU number of the first block in the file. For file allocation purposes only, relative AU's start at 000.

Note that relative AU's 000 and 001 are the last two AU's of the directory. Therefore the data region starts at relative AU 002. The relative AU number is stored in normal Intel fashion, Least Significant byte first.

28-31 — File size in bytes. The first word contains the low-order part of the size. Both words are stored Least Significant byte first.

K.3 MS-DOS File Allocation Tables

The file allocation table (FAT) is used by DOS to allocate disk space for a file, one sector at a time. The FAT is composed of a 12 bit entry for each Allocation Unit (AU), starting with Track Ø Sector 11 on a single sided disk; Track Ø Sector 13 on a double sided disk, and going through to Track 79 Sector 12 on a single sided disk; Track 158 Sector 11 on a double sided disk.

The third FAT entry (relative AU ØØ2) begins the mapping of the data region; each entry contains three hex digits:

ØØØ	If the AU is unused, and available.
FFF	The last AU in the file.
nnn	Any other hex digits that are the relative AU number of the NEXT AU in the file. The relative AU number of the first AU in the file is kept in the files directory entry.

A copy of the FAT for the last used disk in each drive is kept in RAM, and is written back to the disk whenever the status of the disk space used changes.

APPENDIX L

L.1 Generation of Frequencies with the CODEC

This appendix covers the use of the CODEC chip within the Victor 9000 to generate sound. It is beyond the scope of this text to cover actual human-voice generation, Victor does provide tools to achieve this, but generating a frequency will be discussed.

The CODEC chip generates sounds by producing a wave form; frequency generation is achieved by causing a sine wave to be produced by the CODEC, then varying the time base of the sine wave to create various frequencies. Two steps are involved with frequency generation; first the initialization step. The intialization of the CODEC is to produce the sine wave with no time base, the is achieved by the following code:

```
codec_seg       equ     0e800h  ;codec chip segment
codec_tab       dw      05E00H, 00D40H, 00F80H, 000C0H
codec_lngth     equ     4       ;4 words in the codec table
ssda            equ     00060H  ;SSDA chip port offset
clkctr          equ     0008BH  ;Codec clock port
cdclk           equ     00084H  ;Codec frequency clock

init_codec:
        push    ES
        mov     bx,codec_seg     ;codec chip segment address
        mov     ES,bx            ;ready the segment origin
        mov     bx,ssda          ;point to the serial chip
        mov     si,offset codec_tab ;get the init code
        mov     cx,codec_lngth   ;get the table length value
        cld
;
load_loop:
        lodsw
        mov     ES:[bx],ax       ;save the table value
        loop    load_loop        ;  and loop til CX = 0
        mov     bx,clkctr
        mov     ES:byte ptr [bx],0C0H ;enable the CODEC clock
        pop     ES
        ret
```

Once intialized, the CODEC is ready to respond to frequency generation requests. This is simple to achieve by supplying the following subroutine with the correct parameters as follows:

```
        ;    Routine: PLAY_NOTE
        ;    Function: To play a single voice note via the CODEC
        ;    Entries:  CX = Frequency in Hz to be played
        ;              DX = Duration of note in multiples of 2.5mS
        ;              AL=0=play note, AL=FF=halt note
play_note:
        or      cx,cx               ;is freq 0
        jz      dono_end            ;yes, exit
        or      dx,dx               ;is duration 0
        jz      dono_end            ;yes, exit
dono10:
        push    si
        push    bx
        push    ES
        mov     bx,codec_seg        ;codec chip segment address
        mov     ES,bx               ;ready the segment origin
        cmp     al,0ffH             ;if AL = FF then stop note
        jne     dono50
        mov     word ptr ES:cdclk,0 ;stop note
        jmp short dono_ret
dono50:
        push    dx                  ;save the duration
; Now the input to the SSDA must be calculated - note that the
; following calculation achieves a fairly linear tone generation
;  - any deviance from linearity should be fairly minor due to
;  lack of precision in the divide. The calculation itself is:
;       N=((500 000/F)/8)-1 ;where F=desired frequency in Hz
;                           ;N is the value for the CODEC clock
; This equation may be broken down to:
;       N=(62500/F)-1
        mov     ax,62500d           ;ready LSW of 62500 decimal
        xor     dx,dx               ;make MSW zeroes
        div     cx                  ;get CODEC input value
        sub     ax,1                ;normalise to desired value
        pop     dx                  ;get back duration of note
        mov     bx,cdclk
        mov     ES:[bx],ax          ;give the frequency to clock
time_loop:
        mov     ax,25d              ;ready 1 2.5 millisecond period
micro_loop:
        mov     cl,78h              ;ready the timing value
        shr     cl,cl
        dec     ax                  ;100 microseconds has passed - more?
        jnz     micro_loop          ;no, so loop til done
        dec     dx                  ;see if the note is finished with
        jnz     time_loop           ; not finished - round again
time_done:            ; note playing is over - flush speaker
        mov     ES:word ptr [bx],0          ;clear the speaker to silence
dono_ret:
        pop     ES
        pop     bx
        pop     si                          ;stack clear
dono_end:
        ret                                 ; and exit
```