CHAPTER 8

8086 Interrupts and Interrupt Applications

Most microprocessors allow normal program execution to be interrupted by some external signal or by a special instruction in the program. In response to an interrupt, the microprocessor stops executing its current program and calls a procedure which "services" the interrupt. An IRET instruction at the end of the interrupt-service procedure returns execution to the interrupted program. This chapter introduces you to the 8086 interrupt types, shows you how the microprocessors in the 8086 family respond to interrupts, teaches you how to write interrupt-service procedures, and describes how interrupts are used in a variety of applications.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. Describe the interrupt response of an 8086 family processor.
2. Initialize an 8086 interrupt vector (pointer) table.
3. Write interrupt-service procedures.
4. Describe the operation of an 8254 programmable counter/timer and write the instructions necessary to initialize an 8254 for a specified application.
5. Describe the operation of an 8259A priority interrupt controller and write the instructions needed to initialize an 8259A for a specified application.
6. Call a BIOS procedure using a software interrupt.

8086 INTERRUPTS AND INTERRUPT RESPONSES

Overview

An 8086 interrupt can come from any one of three sources. One source is an external signal applied to the *nonmaskable interrupt* (NMI) input pin or to the *interrupt* (INTR) input pin. An interrupt caused by a signal applied to one of these inputs is referred to as a *hardware interrupt*.

A second source of an interrupt is execution of the interrupt instruction, INT. This is referred to as a *software interrupt*.

The third source of an interrupt is some error condition produced in the 8086 by the execution of an instruction. An example of this is the divide-by-zero interrupt. If you attempt to divide an operand by zero, the 8086 will automatically interrupt the currently executing program.

At the end of each instruction cycle, the 8086 checks to see if any interrupts have been requested. If an interrupt has been requested, the 8086 responds to the interrupt by stepping through the following series of major actions.

1. It decrements the stack pointer by 2 and pushes the flag register on the stack.
2. It disables the 8086 INTR interrupt input by clearing the interrupt flag (IF) in the flag register.
3. It resets the trap flag (TF) in the flag register.
4. It decrements the stack pointer by 2 and pushes the current code segment register contents on the stack.
5. It decrements the stack pointer again by 2 and pushes the current instruction pointer contents on the stack.
6. It does an indirect far jump to the start of the procedure you wrote to respond to the interrupt.

Figure 8-1, p. 208, summarizes these steps in diagram form. As you can see, the 8086 pushes the flag register on the stack, disables the INTR input and the single-step function, and does essentially an indirect far call to the interrupt service procedure. An IRET instruction at the end of the interrupt service procedure returns execution to the main program. Now let's see how the 8086 actually gets to the interrupt procedure.

Remember from Chapter 5 that when the 8086 does a far call to a procedure, it puts a new value in the code segment register and a new value in the instruction pointer. For an indirect far call, the 8086 gets the new values for CS and IP from four memory addresses. Likewise, when the 8086 responds to an interrupt, it goes to four memory locations to get the CS and IP values for the start of the interrupt-service procedure. In an 8086 system, the first 1 Kbyte of memory, from 00000H to 003FFH, is set aside as a table for storing the starting addresses of interrupt service procedures. Since 4 bytes are required to store the CS and IP values

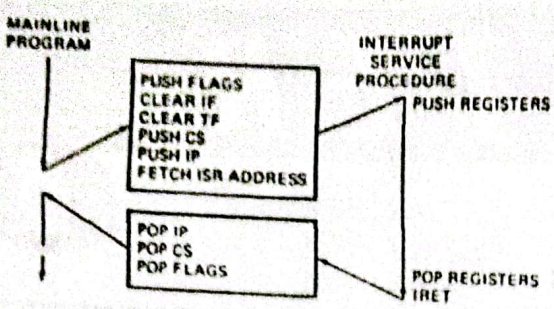


FIGURE 8-1 8086 interrupt response.

for each interrupt service procedure, the table can hold the starting addresses for up to 256 interrupt procedures. The starting address of an interrupt service procedure is often called the *interrupt vector* or the *interrupt pointer*, so the table is referred to as the *interrupt-vector table* or the *interrupt-pointer table*.

Figure 8-2 shows how the 256 interrupt vectors are arranged in the table in memory. Note that the instruction pointer value is put in as the low word of the vector, and the code segment register is put in as the high word of the vector. Each doubleword interrupt vector is identified by a number from 0 to 255. Intel calls this number the *type* of the interrupt.

The lowest five types are dedicated to specific interrupts, such as the divide-by-zero interrupt, the single-step interrupt, and the nonmaskable interrupt. Later in this chapter we explain the operation of these interrupts in detail. Interrupt types 5 to 31 are reserved by Intel

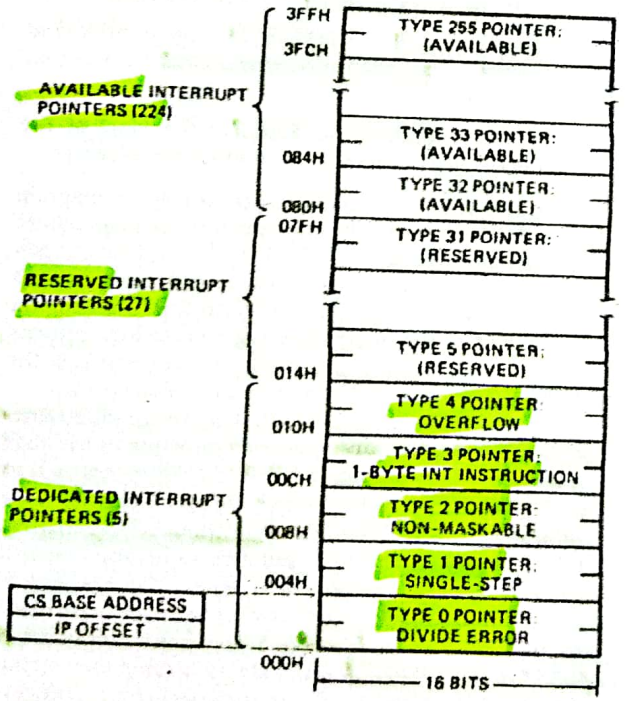


FIGURE 8-2 8086 interrupt-pointer table.

for use in more complex microprocessors, such as the 80286, 80386, and 80486. In a later chapter we discuss some of these interrupt types. The upper 224 interrupt types, from 32 to 255, are available for you to use for hardware or software interrupts.

As you can see in Figure 8-2, the vector for each interrupt type requires four memory locations. Therefore, when the 8086 responds to a particular type interrupt, it automatically multiplies the type by 4 to produce the desired address in the vector table. It then goes to that address in the table to get the starting address of the interrupt service procedure. We will show you later how you use instructions at the start of your program to load the starting address of a procedure into the vector table.

Now that you have an overview of how the 8086 responds to interrupts, we will discuss one type of interrupt in detail and show you how to write a procedure to service that interrupt.

An 8086 Interrupt Response Example—Type 0

Probably the easiest 8086 interrupt to understand is the **divide-by-zero interrupt**, identified as *type 0* in Figure 8-2. Before we get into the details of the type 0 interrupt response, let's refresh your memory about how the 8086 DIV and IDIV instructions work.

The 8086 DIV instruction allows you to divide a 16-bit unsigned binary number in AX by an 8-bit unsigned number from a specified register or memory location. The 8-bit result (quotient) from this division will be left in the AL register. The 8-bit remainder will be left in the AH register. The DIV instruction also allows you to divide a 32-bit unsigned binary number in DX and AX by a 16-bit number in a specified register or memory location. The 16-bit quotient from this division is left in the AX register, and the 16-bit remainder is left in the DX register. In the same manner, the 8086 IDIV instruction allows you to divide a 16-bit signed number in AX by an 8-bit signed number in a specified register or a 32-bit signed number in DX and AX by a 16-bit signed number from a specified register or memory location.

If the quotient from dividing a 16-bit number is too large to fit in AL or the quotient from dividing a 32-bit number is too large to fit in AX, the result of the division will be meaningless. A special case of this is where an attempt is made to divide a 32-bit number or a 16-bit number by zero. The result of dividing by zero is infinity (actually undefined), which is somewhat too large to fit in AX or AL. Whenever the quotient from a DIV or IDIV operation is too large to fit in the result register, the 8086 will automatically do a type 0 interrupt. In response to this interrupt the 8086 proceeds as follows.

The 8086 first decrements the stack pointer by 2 and copies the flag register to the stack. It then clears IF and TF. Next, it saves the return address on the stack. To do this, the 8086 decrements the stack pointer by 2, pushes the CS value of the return address on the stack, decrements the stack pointer by 2 again, and pushes the IP value of the return address on the stack. The 8086 then gets the starting address of the interrupt-service procedure from the type 0 locations in the

interrupt-vector table. As you can see in Figure 8-2, it gets the new value for CS from addresses 00002H and 00003H and the new value for IP from addresses 00000H and 00001H. After the starting address of the procedure is loaded into CS and IP, the 8086 then fetches and executes the first instruction of the procedure.

At the end of the interrupt-service procedure, an IRET instruction is used to return execution to the interrupted program.

The IRET instruction pops the stored value of IP off the stack and increments the stack pointer by 2. It then pops the stored value of CS off the stack and increments the stack pointer again by 2. Finally, it restores the flags by popping off the stack the values stored during the interrupt response and increments the stack pointer by 2. Remember from the previous paragraph that during its interrupt response, the 8086 disables the INTR and single-step interrupts by clearing IF and TF. If the INTR input and/or the trap interrupt were enabled before the interrupt, they will be enabled upon return to the interrupted program. The reason for this is that flags from the interrupted program were pushed on the stack before IF and TF were cleared by the 8086 in its interrupt response. To summarize, then, IRET returns execution to the interrupted program and restores IF and TF to the state they were in before the interrupt. Now that we have described the type 0 response, we can show you how to write a program to handle this interrupt.

An 8086 Interrupt Program Example

DEFINING THE PROBLEM AND WRITING THE ALGORITHM

In the last chapter we were working mostly with hardware, so instead of jumping directly into the program, let's use this example to review how you go about writing any program.

For the example program here, assume we have four word-sized hexadecimal values stored in memory. We want to divide each of these values by a byte-type scale factor to give a byte-type scaled value. If the result of the division is valid, we want to put the scaled value in an array in memory. If the result of the division is invalid (too large to fit in the 8-bit result register), we want to put 0 in the array for that scaled value. Figure 8-3 shows the algorithm for this program in pseudocode.

As shown in Figure 8-3*a*, the mainline part of this program gets each 16-bit value from memory in turn and divides that value by the 8-bit scale factor. If the result of the division is valid, it is stored in the appropriate memory location; else a 0 is stored in the memory location. Not indicated in the algorithm is how we determine whether the quotient is valid or not. With 8086 family microprocessors, a type 0 interrupt procedure is a handy way to do this.

Remember from the preceding discussion that if the result of the division is too large to fit in the quotient register, AL, then the 8086 will do a type 0 interrupt immediately after the divide instruction finishes. Figure 8-3*b* shows the algorithm for a procedure to service this type 0 interrupt. The main function of this procedure is to set a *flag* which will be checked by the mainline

```
INITIALIZATION LIST

REPEAT
  Get INPUT_VALUE
  Divide by scale factor
  If result valid THEN
    store result as scaled value
  ELSE store zero
UNTIL all values scaled
```

(*a*)

```
Save registers
Set error flag
Restore registers
Return to mainline
```

(*b*)

FIGURE 8-3 Algorithm for divide-by-zero program example. (*a*) Mainline program. (*b*) Interrupt-service procedure.

program. The flag in this case is not one of the flags in the 8086 flag register. The flag here is a bit in a memory location we set aside for this purpose. In the actual program, we give this memory location the name BAD_DIV_FLAG. At the end of the interrupt-service procedure, execution is returned to the interrupted mainline program.

After the divide operation in the mainline program, we check the value of the BAD_DIV_FLAG to determine if the result of the division is valid. If the result of the division was too large, then the 8086 will have done a type 0 interrupt, and the interrupt-service procedure will have set the BAD_DIV_FLAG to a 1. If the result of the division is valid, then the 8086 will not have done the type 0 interrupt, and the BAD_DIV_FLAG will be 0.

This sequence of operations is repeated until all the values have been scaled.

WRITING THE INITIALIZATION LIST

After you have worked out the data structure and the algorithm for a program, the next step is to make an initialization list such as the one shown in Chapter 3. Here is a list for this program.

1. Initialize the interrupt-vector table. In other words, the starting address of our type 0 interrupt service routine must be put in locations 00000H and 00002H.
2. Set up the data segment where the values to be scaled, the scale factor, the scaled values, and the BAD_DIV_FLAG will be put.
3. Initialize the data segment register to point to the base address of the data segment containing the values to be scaled.
4. Set up a stack to store the flags and return address.
5. Initialize the stack segment and stack pointer registers.
6. Initialize a pointer to the start of the data to be scaled, a counter to keep track of how many values have been scaled, and a pointer to the start of the array where the scaled values are to be written.

at 00000H and 00001H. The statement MOV WORD PTR ES:0000 SEG BAD_DIV is used to load the segment base address of BAD_DIV into memory at 00002H and 00003H. It is necessary to load the interrupt procedure addresses in this way if you are using an SDK-86 board or using the MASM and Link programs on an IBM PC-type machine.

Next, we initialize SI as a pointer to the first input value and initialize BX as a pointer to the first of the locations we set aside for the 8-bit scaled results. CX is initialized as a counter to keep track of how many values have been scaled.

Finally, after everything is initialized, we get to the operations we set out to do. The statement MOV AX,[SI] copies an input value from memory to the AX register, where it has to be for the divide operation. The DIV SCALE_FACTOR instruction divides the number in AX by 09H, the value we assigned to SCALE_FACTOR previously with a DB directive. The 8-bit quotient from this division will be put in AL, and the 8-bit remainder will be put in AH. If the quotient is too large to fit in AL, then the 8086 will automatically do a type 0 interrupt. For our program here, the 8086 will push the flags on the stack, reset IF and TF, and push the return address on the stack. It will then go to addresses 0000H and 0002H to get the IP and CS values for the start of BAD_DIV, the procedure we wrote to service a type 0 interrupt. It will then execute the BAD_DIV procedure. Now let's look at the procedure in Figure 8-4*b* and see how it works.

The BAD_DIV procedure starts by letting the assembler know that the name BAD_DIV_FLAG represents a variable of type byte and that this variable is defined in a segment called DATA in some other (EXTRN) assembly module. We also tell the assembler that the label BAD_DIV should be made available to other assembly modules (PUBLIC).

Next, we declare a logical segment called INT_PROC. We could have put this procedure in the segment CODE with the mainline program. However, in system programs where there are many interrupt-service procedures, a separate segment is usually set aside for them. The statement BAD_DIV PROC FAR identifies the actual start of the procedure and tells the assembler that both the CS and IP values for this procedure must be saved.

Now, an important operation to do at the start of any interrupt-service procedure is to push on the stack any registers that are used in the procedure. You can then restore these registers by popping them off the stack just before returning to the interrupted program. The interrupted program will then resume with its registers as they were before the interrupt. In the procedure in Figure 8-4*b*, we saved AX and DS. Since we use the same data segment, DATA, in the mainline and in the procedure, you may wonder why we saved DS. The point is that an interrupt-service procedure should be written so that it can be used at any point in a program. By saving the DS value for the interrupted program, this interrupt-service procedure can be used in a program section that does not use DATA as its data segment.

The ASSUME statement tells the assembler the name of the segment to use as a data segment, but remember that it does not load the DS register with a value for the start of that segment. The instructions MOV AX,DATA and MOV DS,AX do this in our procedure.

Finally, we get to the whole point of this procedure with the MOV BAD_DIV_FLAG,01 instruction. This instruction simply sets the least significant bit of the memory location we set aside with a DB directive at the start of the mainline program. Note that in order to access this variable by name, you have to let the assembler know that it is external, and you have to make sure that the DS register contains the segment base for the segment in which BAD_DIV_FLAG is located.

To complete the procedure, we pop the saved registers off the stack and return to the interrupted program. The IRET instruction, remember, is different from the regular RET instruction in that it pops the flag register and the return address off the stack. Note in the program in Figure 8-4*b* that the procedure must be "closed" with an ENDP directive, and the segment must as usual be closed with an ENDS directive.

Now let's look back in the mainline to see what it does with this BAD_DIV_FLAG. Immediately after the DIV instruction, the mainline checks to see if the BAD_DIV_FLAG is set by comparing it with 01. If the BAD_DIV_FLAG was not set by the type 0 interrupt-service procedure, then a jump is made to the MOV [BX],AL instruction. This instruction copies the result of the division in AL to the memory location in SCALED_VALUES pointed to by BX. If BAD_DIV_FLAG was set by a type 0 interrupt, then 0 is put in the memory location in SCALED_VALUES and a jump will be made to the MOV BAD_DIV_FLAG,00 instruction, which resets the BAD_DIV_FLAG. Since this jump passes over the MOV [BX],AL instruction, the invalid result of the division will not be copied into one of the locations in SCALED_VALUES.

After putting the scaled value or 0 in the array and resetting the flag, we get ready to operate on the next input value. The ADD SI,02 instruction increments SI by 2 so that it points to the next 16-bit value in INPUT_VALUES. The INC BX instruction points BX at the next 8-bit location in SCALED_VALUES. The LOOP instruction after these automatically decrements the CX register by 1 and, if CX is not then 0, causes the 8086 to jump to the specified label, NEXT.

The preceding section has shown you how to set up an interrupt-pointer table, how to write an interrupt-service procedure, and how the 8086 responds to a type 0 interrupt. Now we can discuss some of the other types of 8086 interrupts.

8086 Interrupt Types

The preceding sections used the type 0 interrupt as an example of how the 8086 interrupts function. In this section we discuss in detail the different ways an 8086 can be interrupted and how the 8086 responds to each of these interrupts. We discuss these in order, starting with type 0, so that you can easily find a particular discussion when you need to refer back to it. However, as you read through this section, you should not attempt to learn all the details of all the interrupt types at once.

Read through all the types to get an overview, and then focus on the details of the hardware-caused NMI interrupt, the software interrupts produced by the INT instruction, and the hardware interrupt produced by applying a signal to the INTR input pin.

DIVIDE-BY-ZERO INTERRUPT—TYPE 0

As we described in the preceding section, the 8086 will automatically do a type 0 interrupt if the result of a DIV operation or an IDIV operation is too large to fit in the destination register. For a type 0 interrupt, the 8086 pushes the flag register on the stack, resets IF and TF, and pushes the return address (CS and IP) on the stack. It then gets the CS value for the start of the interrupt-service procedure from address 00002H in the interrupt-pointer table and the IP value for the start of the procedure from address 00000H in the interrupt pointer-table.

Since the 8086 type 0 response is automatic and cannot be disabled in any way, you have to account for it in any program where you use the DIV or IDIV instruction. One way is to in some way make sure the result will never be too large for the result register. We showed one way to do this in the example program in Figure 5-27*b*. In that example, you may remember, we first make sure the divisor is not zero, and then we do the division in several steps so that the result of the division will never be too large.

Another way to account for the 8086 type 0 response is to simply write an interrupt-service procedure which takes the desired action when an invalid division occurs. The advantage of this approach is that you don't have the overhead of a more complex division routine in your mainline program. The 8086 automatically does the checking and does the interrupt procedure only if there is a problem.

SINGLE-STEP INTERRUPT—TYPE 1

In a section of Chapter 3 on debugging assembly language programs, we discussed the use of the single-step feature found in some monitor programs and debugger programs. When you tell a system to single-step, it will execute one instruction and stop. You can then examine the contents of registers and memory locations. If they are correct, you can tell the system to go on and execute the next instruction. In other words, when in single-step mode, a system will stop after it executes each instruction and wait for further direction from you. The 8086 trap flag and type 1 interrupt response make it quite easy to implement a single-step feature in an 8086-based system.

If the 8086 trap flag is set, the 8086 will automatically do a type 1 interrupt after each instruction executes. When the 8086 does a type 1 interrupt, it pushes the flag register on the stack, resets TF and IF, and pushes the CS and IP values for the next instruction on the stack. It then gets the CS value for the start of the type 1 interrupt-service procedure from address 00006H and it gets the IP value for the start of the procedure from address 00004H.

The tasks involved in implementing single stepping are: Set the trap flag, write an interrupt-service procedure which saves all registers on the stack, where they can later be examined or perhaps displayed on the CRT, and load the starting address of the type 1 interrupt-service procedure into addresses 00004H and 00006H. The actual single-step procedure will depend very much on the system on which it is to be implemented. We do not have space here to show you the different ways to do this. We will, however, show you how the trap flag is set or reset, because this is somewhat unusual.

The 8086 has no instructions to directly set or reset the trap flag. These operations are done by pushing the flag register on the stack, changing the trap flag bit to what you want it to be, and then popping the flag register back off the stack. Here is the instruction sequence to set the trap flag.

```
PUSHF              ; Push flags on stack
MOV BP,SP          ; Copy SP to BP for use as index
OR WORD PTR[BP+0],0100H
                   ; Set TF bit
POPF               ; Restore flag register
```

To reset the trap flag, simply replace the OR instruction in the preceding sequence with the instruction AND WORD PTR [BP+0], OFEFFH.

NOTE: We have to use [BP+0] because BP cannot be used as a pointer without a displacement. See Figure 3-8.

The trap flag is reset when the 8086 does a type 1 interrupt, so the single-step mode will be disabled during the interrupt-service procedure.

NONMASKABLE INTERRUPT—TYPE 2

The 8086 will automatically do a *type 2* interrupt response when it receives a low-to-high transition on its NMI input pin. When it does a type 2 interrupt, the 8086 will push the flags on the stack, reset TF and IF, and push the CS value and the IP value for the next instruction on the stack. It will then get the CS value for the start of the type 2 interrupt-service procedure from address 0000AH and the IP value for the start of the procedure from address 00008H.

The name *nonmaskable* given to this input pin on the 8086 means that the type 2 interrupt response cannot be disabled (masked) by any program instructions. Because this input cannot be intentionally or accidentally disabled, we use it to signal the 8086 that some condition in an external system must be taken care of. We could, for example, have a pressure sensor on a large steam boiler connected to the NMI input. If the pressure goes above some preset limit, the sensor will send an interrupt signal to the 8086. The type 2 interrupt-service procedure for this case might turn off the fuel to the boiler, open a pressure-relief valve, and sound an alarm.

Another common use of the type 2 interrupt is to save program data in case of a system power failure. Some external circuitry detects when the ac power to the system fails and sends an interrupt signal to the NMI

input. Because of the large filter capacitors in most power supplies, the dc system power will remain for perhaps 50 ms after the ac power is gone. This is more than enough time for a type 2 interrupt-service procedure to copy program data to some RAM which has a battery backup power supply. When the ac power returns, program data can be restored from the battery-backed RAM, and the program can resume execution where it left off. A practice problem at the end of the chapter gives you a chance to write a simple procedure for this task.

BREAKPOINT INTERRUPT—TYPE 3

The *type 3* interrupt is produced by execution of the INT 3 instruction. The main use of the type 3 interrupt is to implement a breakpoint function in a system. In Chapter 4 we described the use of breakpoints in debugging assembly language programs. We hope that you have been using them in debugging your programs. When you insert a breakpoint, the system executes the instructions up to the breakpoint and then goes to the breakpoint procedure. Unlike the single-step feature, which stops execution after each instruction, the breakpoint feature executes all the instructions up to the inserted breakpoint and then stops execution.

When you tell most 8086 systems to insert a breakpoint at some point in your program, they actually do it by temporarily replacing the instruction byte at that address with CCH, the 8086 code for the INT 3 instruction. When the 8086 executes this INT 3 instruction, it pushes the flag register on the stack, resets TF and IF, and pushes the CS and IP values for the next mainline instruction on the stack. The 8086 then gets the CS value of the start of the type 3 interrupt-service procedure from address 0000EH and the IP value for the procedure from address 0000CH. A breakpoint interrupt-service procedure usually saves all the register contents on the stack. Depending on the system, it may then send the register contents to the CRT display and wait for the next command from the user, or in a simple system it may just return control to the user. In this case an *Examine Register* command can be used to check if the register contents are correct at that point in the program.

OVERFLOW INTERRUPT—TYPE 4

The 8086 overflow flag (OF) will be set if the signed result of an arithmetic operation on two signed numbers is too large to be represented in the destination register or memory location. For example, if you add the 8-bit signed number 01101100 (108 decimal) and the 8-bit signed number 01010001 (81 decimal), the result will be 10111101 (189 decimal). This would be the correct result if we were adding unsigned binary numbers, but it is not the correct signed result. For signed operations, the 1 in the most significant bit of the result indicates that the result is negative and in 2's complement form. The result, 10111101, then actually represents −67 decimal, which is obviously not the correct result for adding +108 and +89.

There are two major ways to detect and respond to an overflow error in a program. One way is to put the Jump if Overflow instruction, JO, immediately after the arithmetic instruction. If the overflow flag is set as a result of the arithmetic operation, execution will jump to the address specified in the JO instruction. At this address you can put an error routine which responds to the overflow in the way you want.

The second way of detecting and responding to an overflow error is to put the *Interrupt on Overflow* instruction, INTO, immediately after the arithmetic instruction in the program. If the overflow flag is not set when the 8086 executes the INTO instruction, the instruction will simply function as an NOP. However, if the overflow flag is set, indicating an overflow error, the 8086 will do a *type 4* interrupt after it executes the INTO instruction.

When the 8086 does a type 4 interrupt, it pushes the flag register on the stack, resets TF and IF, and pushes the CS and IP values for the next instruction on the stack. It then gets the CS value for the start of the interrupt-service procedure from address 00012H and the IP value for the procedure from address 00010H. Instructions in the interrupt-service procedure then perform the desired response to the error condition. The procedure might, for example, set a "flag" in a memory location as we did in the BAD_DIV procedure in Figure 8-4*b*. The advantage of using the INTO and type 4 interrupt approach is that the error routine is easily accessible from any program.

SOFTWARE INTERRUPTS—TYPES 0 THROUGH 255

The 8086 INT instruction can be used to cause the 8086 to do any one of the 256 possible interrupt types. The desired interrupt type is specified as part of the instruction. The instruction INT 32, for example, will cause the 8086 to do a *type 32* interrupt response. The 8086 will push the flag register on the stack, reset TF and IF, and push the CS and IP values of the next instruction on the stack. It will then get the CS and IP values for the start of the interrupt-service procedure from the interrupt-pointer table in memory. The IP value for any interrupt type is always at an address of 4 times the interrupt type, and the CS value is at a location two addresses higher. For a type 32 interrupt, then, the IP value will be put at 4 × 32 or 128 decimal (80H), and the CS value will be put at address 82H in the interrupt-vector table.

Software interrupts produced by the INT instruction have many uses. In a previous section we discussed the use of the INT 3 instruction to insert breakpoints in programs for debugging. Another use of software interrupts is to test various interrupt-service procedures. You could, for example, use an INT 0 instruction to send execution to a divide-by-zero interrupt-service procedure without having to run the actual division program. As another example, you could use an INT 2 instruction to send execution to an NMI interrupt-service procedure. This allows you to test the NMI procedure without needing to apply an external signal to the NMI input of the 8086. In a later section of the chapter, we show an example of another important application of software interrupts.

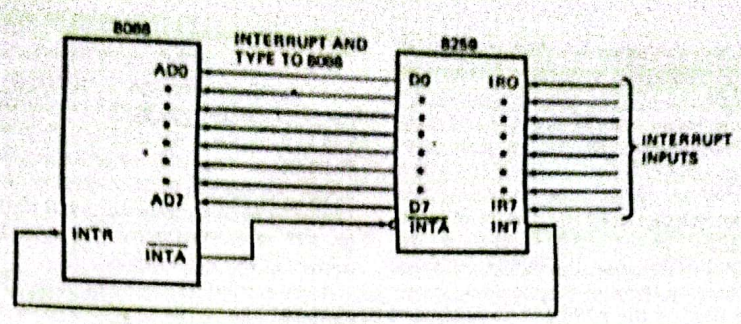


FIGURE 8-5 Block diagram showing an 8259 connected to an 8086.

INTR INTERRUPTS—TYPES 0 THROUGH 255

The 8086 INTR input allows some external signal to interrupt execution of a program. Unlike the NMI input, however, INTR can be masked (disabled) so that it cannot cause an interrupt. If the interrupt flag (IF) is cleared, then the INTR input is disabled. IF can be cleared at any time with the *Clear Interrupt* instruction, CLI. If the interrupt flag is set, the INTR input will be enabled. IF can be set at any time with the *Set Interrupt* instruction, STI.

When the 8086 is reset, the interrupt flag is automatically cleared. Before the 8086 can respond to an interrupt signal on its INTR input, you have to set IF with an STI instruction. The 8086 was designed this way so that ports, timers, registers, etc., can be initialized before the INTR input is enabled. In other words, this allows you to get the 8086 ready to handle interrupts before letting an interrupt in, just as you might want to get yourself ready in the morning with a cup of coffee before turning on the telephone and having to cope with the interruptions it produces.

Remember that the interrupt flag (IF) is also automatically cleared as part of the response of an 8086 to an interrupt. This is done for two reasons. First, it prevents a signal on the INTR input from interrupting a higher-priority interrupt-service procedure in progress. However, if you want another INTR input signal to be able to interrupt an interrupt procedure in progress, you can reenable the INTR input with an STI instruction at any time.

The second reason for automatically disabling the INTR input at the start of an INTR interrupt-service procedure is to make sure that a signal on the INTR input does not cause the 8086 to interrupt itself continuously. The INTR input is activated by a high level. In other words, whenever the INTR input is high and INTR is enabled, the 8086 will be interrupted. If INTR were not disabled during the first response, the 8086 would be continuously interrupted and would never get to the actual interrupt-service procedure.

The IRET instruction at the end of an interrupt-service procedure restores the flags to the condition they were in before the procedure by popping the flag register off the stack. This will reenable the INTR input. If a high-level signal is still present on the INTR input, it will cause the 8086 to be interrupted again. If you do not want the 8086 to be interrupted again by the same input signal, you have to use external hardware to make sure that the signal is made low again before you reenable INTR with the STI instruction or before the IRET from the INTR service procedure.

When the 8086 responds to an INTR interrupt signal, its response is somewhat different from its response to other interrupts. The main difference is that for an INTR interrupt, the interrupt type is sent to the 8086 from an external hardware device such as the 8259A *priority interrupt controller*, as shown in Figure 8-5. We discuss the 8259A in detail later in the chapter, but here's an introduction.

When an 8259A receives an interrupt signal on one of its IR inputs, it sends an interrupt request signal to the INTR input of the 8086. If the INTR input of the 8086 has been enabled with an STI instruction, the 8086 will respond as shown by the waveforms in Figure 8-6.

The 8086 first does two interrupt-acknowledge ma-

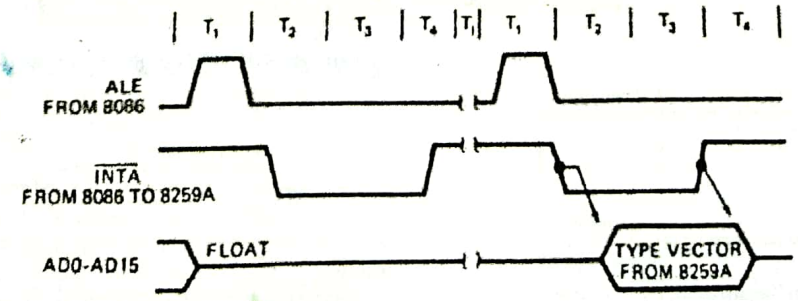


FIGURE 8-6 8086 interrupt-acknowledge machine cycles.

chine cycles, as shown in Figure 8-6. The purpose of these two machine cycles is to get the interrupt type from the external device. At the start of the first interrupt-acknowledge machine cycle, the 8086 floats the data bus lines, AD0–AD15, and sends out an interrupt-acknowledge pulse on its INTA output pin. This pulse essentially tells the 8259A to "get ready." During the second interrupt-acknowledge machine cycle, the 8086 sends out another pulse on its INTA output pin. In response to this second INTA pulse, the 8259A puts the interrupt type (number) on the lower eight lines of the data bus, where it is read by the 8086.

Once the 8086 receives the interrupt type, it pushes the flag register on the stack, clears TF and IF, and pushes the CS and IP values of the next instruction on the stack. It then uses the type it read in from the external device to get the CS and IP values for the interrupt-service procedure from the interrupt-pointer table in memory. The IP value for the procedure will be put at an address equal to 4 times the type number, and the CS value will be put at an address equal to 4 times the type number plus 2, just as is done for the other interrupts.

The advantage of having an external device insert the desired interrupt type is that the external device can "funnel" interrupt signals from many sources into the INTR input pin on the 8086. When the 8086 responds with INTA pulses, the external device can send to the 8086 the interrupt type that corresponds to the source of the interrupt signal. As you will see later, the external device can also prevent an argument if two or more sources send interrupt signals at the same time.

PRIORITY OF 8086 INTERRUPTS

As you read through the preceding discussions of the different interrupt types, the question that may have occurred to you is, What happens if two or more interrupts occur at the same time? The answer to this question is that the highest-priority interrupt will be serviced first, and then the next-highest-priority interrupt will be serviced. Figure 8-7 shows the priorities of the 8086 interrupts as shown in the Intel data book. Some examples will show you what these priorities actually mean.

As a first example, suppose that the INTR input is enabled, the 8086 receives an INTR signal during execution of a Divide instruction, and the divide operation produces a divide-by-zero interrupt. Since the internal interrupts—such as divide error, INT, and INTO—have higher priority than INTR, the 8086 will do a divide error (type 0) interrupt response first. Part of the type 0 interrupt response is to clear IF. This disables the INTR input and prevents the INTR signal from interrupting the higher-priority type 0 interrupt-service procedure. An IRET instruction at the end of the type 0 procedure will restore the flags to what they were before the type 0 response. This will reenable the INTR input, and the 8086 will do an INTR interrupt response. A similar sequence of operations will occur if the 8086 is executing an INT or INTO instruction and an interrupt signal arrives at the INTR input.

INTERRUPT	PRIORITY
DIVIDE ERROR, INT n, INTO	HIGHEST
NMI	
INTR	
SINGLE-STEP	LOWEST

FIGURE 8-7 Priority of 8086 interrupts. (*Intel Corporation*)

As a second example of how this priority works, suppose that a rising-edge signal arrives at the NMI input while the 8086 is executing a DIV instruction, and that the division operation produces a divide error. Since the 8086 checks for internal interrupts before it checks for an NMI interrupt, the 8086 will push the flags on the stack, clear TF and IF, push the return address on the stack, and go to the start of the divide error (type 0) service procedure. However, because the NMI interrupt request is not disabled, the 8086 will then do an NMI (type 2) interrupt response. In other words, the 8086 will push the flags on the stack, clear TF and IF, push the return address on the stack, and go execute the NMI interrupt-service procedure. When the 8086 finishes the NMI procedure, it will return to the divide error procedure, finish executing that procedure, and then return to the mainline program.

To finish our discussion of 8086 interrupt priorities, let's see how the single-step (trap, or type 1) interrupt fits in. If the trap flag is set, the 8086 will do a type 1 interrupt response after every mainline instruction. When the 8086 responds to any interrupt, however, part of its response is to clear the trap flag. This disables the single-step function, so the 8086 will not normally single-step through the instructions of the interrupt-service procedure. The trap flag can be set again in the single-step procedure if single-stepping is desired in the interrupt-service procedure.

Now that we have shown you the different types of 8086 interrupts and how the 8086 responds to each, we will show you a few examples of how the 8086 hardware interrupts are used. Other applications of interrupts will be shown throughout the rest of the book.

HARDWARE INTERRUPT APPLICATIONS

Simple Interrupt Data Input

One of the most common uses of interrupts is to relieve a CPU of the burden of polling. To refresh your memory, polling works as follows.

The strobe or data ready signal from some external device is connected to an input port line on the microcomputer. The microcomputer uses a program loop to read and test this port line over and over until the data ready signal is found to be asserted. The microcomputer then exits the polling loop and reads in the data from the external device. Data can also be output on a polled basis.

The disadvantage of polled input or output is that while the microcomputer is polling the strobe or data

preceding example. In other words, the single interrupt-service routine can be used to keep track of several different time intervals. By counting a different number of interrupts or applying a different frequency signal to the interrupt input, this technique can be used to time many different tasks in a microcomputer system.

GENERATING AN ACCURATE TIME BASE FOR TIMING INTERRUPTS

The 555 timer that we used for the 4-min timer just described was accurate enough for that application, but for many applications—such as a real-time clock—it is not. For more precise timing, we usually use a signal derived from a crystal-controlled oscillator. The processor clock signal is generated by a crystal-controlled oscillator, so it is stable, but this signal is obviously too high in frequency to drive a processor interrupt input directly. The solution is to divide the clock signal down with an external counter device to the desired frequency for the interrupt input. Most microcomputer manufacturers have a compatible device which can be programmed with instructions to divide an input frequency by any desired number. Besides acting as programmable frequency dividers, these devices have many important uses in microcomputer systems. Therefore, the next section describes how an Intel 8254 Programmable Counter operates, how an 8254 can easily be added to an SDK-86 board, and how an 8254 is used in a variety of interrupt applications. Also in the next section, we use the 8254 discussion to show you the general procedure for initializing any of the programmable peripheral devices we discuss in later chapters.

8254 SOFTWARE-PROGRAMMABLE TIMER/COUNTER

Because of the many tasks that they can be used for in microcomputer systems, programmable timer/counters are very important for you to learn about. As you read through the following sections, pay particular attention to the applications of this device in systems and the general procedure for initializing a programmable device such as the 8254. Read lightly through the discussions of the different counter modes to become aware of the types of problems that the device can solve for you. Later, when you have a specific problem to solve, you can dig into the details of these discussions.

Another important point to make to you here is that the discussions of various devices throughout the rest of this book are not intended to replace the manufacturers' data sheets for the devices. Many of the programmable peripheral devices we discuss are so versatile that each requires almost a small book to describe all the details of its operations. The discussions here are intended to introduce you to the devices, show you what they can be used for, and show you enough details about them that you can do some real jobs with them. After you become familiar with the use of a device in some simple applications, you can read the data sheets to learn further "bells and whistles" that the devices have.

Basic 8253 and 8254 Operation

The Intel 8253 and 8254 each contain three 16-bit counters which can be programmed to operate in several different modes. The 8253 and 8254 devices are pin-for-pin compatible, and they are nearly identical in function. The major differences are as follows:

1. The maximum input clock frequency for the 8253 is 2.6 MHz; the maximum clock frequency for the 8254 is 8 MHz (10 MHz for the 8254-2).
2. The 8254 has a *read-back feature* which allows you to latch the count in all the counters and the status of the counter at any point. The 8253 does not have this read-back feature.

To simplify reading of this section, we will refer only to the 8254. However, you can assume that the discussion also applies to the 8253 except where we specifically state otherwise.

As shown by the block diagram of the 8254 in Figure 8-13, the device contains three 16-bit counters. In some ways these counters are similar to the TTL presettable counters we reviewed in Chapter 1. The big advantage of these counters, however, is that you can load a count in them, start them, and stop them with instructions in your program. Such a device is said to be software-programmable. To program the device, you send count bytes and control bytes to the device just as you would send data to a port device.

If you look along the left side of the block diagram in Figure 8-13, you will see the signal lines used to interface the device to the system buses. A little later we show how these are actually connected in a real system. The main points for you to note about the 8254 at the moment are that it has an 8-bit interface to the data bus, it has a $\overline{CS}$ input which will be asserted by an

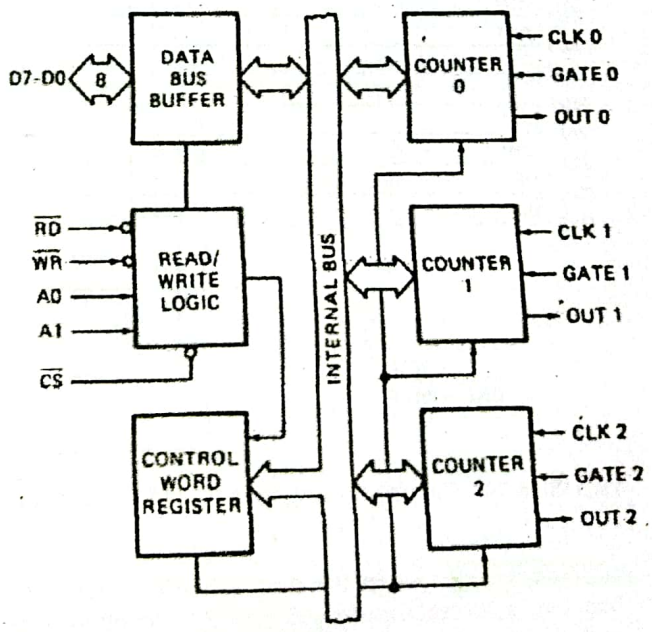


FIGURE 8-13 8254 internal block diagram. (*Intel Corporation*)

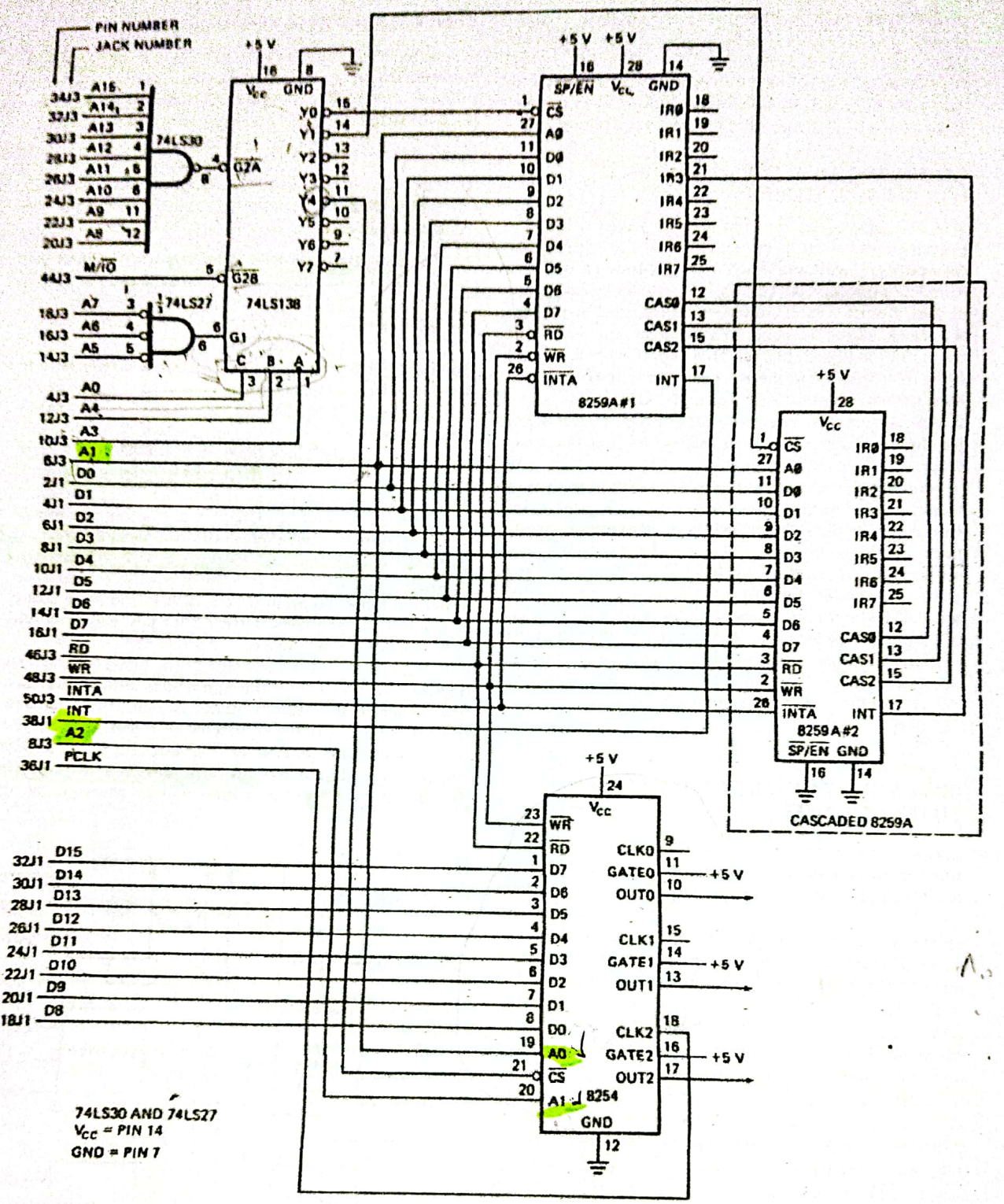


FIGURE 8-14 Circuit showing how to add an 8254 and 8259A(s) to an SDK-86 board.

address decoder when the device is addressed, and it has two address inputs, A0 and A1, to allow you to address one of the three counters or the control word register in the device.

The right side of the 8254 block diagram in Figure 8-13 shows the counter inputs and outputs. You can apply a signal of any frequency from dc to 8 MHz to each of the counter clock inputs, labeled CLK in the diagram.

The GATE input on each counter allows you to start or stop that counter with an external hardware signal. If the GATE input of a counter is high (1), then that counter is enabled for counting. If the GATE input is low, the counter is disabled. The output signal from each counter appears on its OUT pin. Now let's see how a programmable peripheral device such as the 8254 is connected in a system.

System Connections for an 8254 Timer/Counter

An 8254 is a very useful device to have in a microcomputer system, but, in order to keep the cost down, the SDK-86 was not designed with one on the board. Figure 8-14 shows the circuit connections for adding an 8254 Counter and an 8259A Priority Interrupt Controller to an SDK-86 board. We discuss the 8259A in a later section of this chapter.

If you use wire-wrap headers for connectors J1 and J3 on an SDK-86 board, the circuitry shown can easily be wire-wrapped on the prototyping area of the board. Install the WAIT-state jumper to insert one WAIT state. As explained in Chapter 7, a WAIT state is needed because of the added delay of the decoders and buffers.

The 74LS138 in Figure 8-14 is used to produce chip select ($\overline{CS}$) signals for the 8254, the 8259A, and any other I/O devices you might want to add. Let's look first at the circuitry around this device to determine the system base address which selects each device.

In order for any of the outputs of the 74LS138 to be asserted, the G1, $\overline{G2A}$, and $\overline{G2B}$ enable inputs must all be asserted. The G1 input will be asserted (high) if system address lines A5, A6, and A7 are all low. The $\overline{G2A}$ input will be asserted (low) if system address lines A8 through A15 are all high. As shown by the truth table in Figure 8-15, these two inputs therefore will be asserted for a system base address of FF00H. The $\overline{G2B}$ input of the 74LS138 will be asserted (low) if the M/$\overline{IO}$ line is low, as it will be for a port read or write operation.

Now, remember from Chapter 7 that only one of the Y outputs of the 74LS138 will ever be asserted at a time. The output asserted is determined by the 3-bit binary code applied to the A, B, and C select inputs. In the circuit in Figure 8-14, we connected system address line A0 to the C input, address line A4 to the B input, and address line A3 to the A input. The truth table in Figure 8-15 shows the system base addresses that will enable each of the 74LS138 Y outputs. As you will see a little later, system address lines A1 and A2 are used to select internal parts of the 8254 and 8259A.

We connected A0 to the C input so that half of the Y outputs will be selected by even addresses and half of the Y outputs will be selected by odd addresses. We did this so that loading on the two halves of the data bus will be equal as we add peripheral devices such as the 8254 and 8259A. To see how this works, note that the peripheral devices have only eight data lines. For an odd-addressed device we connect these data lines to the upper eight system data lines, and for an even-addressed device, we connect these to the lower eight system data lines. By alternating between odd- and even-selected outputs as we add peripheral devices, we equalize loading on the bus.

As shown by the truth table in Figure 8-15, the system base address of the added 8254 is FF01H. Other connections to the 8254 are the system $\overline{RD}$ and $\overline{WR}$ lines used to enable the 8254 for reading or writing; eight data lines, used to send control bytes, status bytes, and count values between the CPU and the 8254; and system address lines A1 and A2, used to select the control register or one of the three counters in the 8254. Now that you see how an 8254 is connected in a system, we will show you how to initialize an 8254 to do some useful work for you.

Initializing an 8254 Programmable Peripheral Device

When the power is first turned on, programmable peripheral devices such as the 8254 are usually in *undefined states*. Before you can use them for anything, you have to initialize them in the *mode* you need for your specific application. Initializing these devices is not usually difficult, but it is very easy to make errors if you do not do it in a very systematic way. To initialize any programmable peripheral device, you should always work your way through the following series of steps.

1. Determine the system base address for the device. You do this from the address decoder circuitry or the address decoder truth table. From the truth table

A8-A15	A5-A7	A4	A3	A2	A1	A0	M/$\overline{IO}$	Y OUTPUT SELECTED	SYSTEM BASE ADDRESS	DEVICE
1	0	0	0	X	X	0	0	0	FF00	8259A #1
1	0	0	1	X	X	0	0	1	FF08	8259A #2
1	0	1	0	X	X	0	0	2	FF10	
1	0	1	1	X	X	0	0	3	FF18	
1	0	0	0	X	X	1	0	4	FF01	8254
1	0	0	1	X	X	1	0	5	FF09	
1	0	1	0	X	X	1	0	6	FF11	
1	0	1	1	X	X	1	0	7	FF19	
ALL OTHER STATES								NONE		

FIGURE 8-15 Truth table for 74LS138 address decoder in Figure 8-14.

A_2 A_1 actually

A1	A0	SELECTS
0	0	COUNTER 0
0	1	COUNTER 1
1	0	COUNTER 2
1	1	CONTROL WORD REGISTER

(a)

SYSTEM ADDRESS	8254 PART
F F 0 1	COUNTER 0
F F 0 3	COUNTER 1
F F 0 5	COUNTER 2
F F 0 7	CONTROL REG

(b)

FIGURE 8-16 8254 addresses. (a) Internal. (b) System.

In Figure 8-15, the system base address of the 8254 in our example here is FF01H.

2. Use the device data sheet to determine the internal addresses for each of the control registers, ports, timers, status registers, etc., in the device. Figure 8-16a shows the internal addresses for the three counters and the control word register for the 8254. A0 in this table represents the A0 input of the device, and A1 represents the A1 input of the device. Note in the schematic in Figure 8-14 that system address line A1 is connected to the A0 input of the 8254, and system address line A2 is connected to the A1 input. We could not use system address line A0 as one of these because, as described before, we used system address line A0 as one of the inputs to the address decoder.

3. Add each of the internal addresses to the system base address to determine the system address of each of the parts of the device. You need to do this so that you know the actual addresses where you have to send control words, timer values, etc. Figure 8-16b shows the system addresses for the three timers and the control register of the 8254 we added to the SDK-86 board. Note that the addresses all have to be odd because the device is connected on the upper half of the data bus.

4. Look in the data sheet for the device for the format of the control word(s) that you have to send to the device to initialize it. For different devices, incidentally, the control word(s) may be referred to as command words or mode words. To initialize the 8254, you send a control word to the control register for each counter that you want to use. Figure 8-17 shows the format for the 8254 control word.

5. Construct the control word required to initialize the device for your specific application. You construct this control word on a bit-by-bit basis. We have found it helpful to actually draw the eight little boxes shown at the top of Figure 8-17 so that we don't miss any bits. (An easy way to draw the eight boxes is to draw a long rectangle, divide it in half, divide each resulting half in two, and finally divide each resulting quarter in two.) To help keep track of the meaning of each bit of a control word, write under each bit the meaning of that bit. A little later we show you how to do this for an 8254 control word. Documentation of this sort is very valuable when you are trying to debug a program or modify an old program for some new application.

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC – SELECT COUNTER:

SC1	SC0	
0	0	SELECT COUNTER 0
0	1	SELECT COUNTER 1
1	0	SELECT COUNTER 2
1	1	READ-BACK COMMAND (SEE READ OPERATIONS)

RW – READ/WRITE:

RW1	RW0	
0	0	COUNTER LATCH COMMAND (SEE READ OPERATIONS)
0	1	READ/WRITE LEAST SIGNIFICANT BYTE ONLY.
1	0	READ/WRITE MOST SIGNIFICANT BYTE ONLY.
1	1	READ/WRITE LEAST SIGNIFICANT BYTE FIRST, THEN MOST SIGNIFICANT BYTE.

M – MODE:

M2	M1	M0	
0	0	0	MODE 0 – INTERRUPT ON TERMINAL COUNT
0	0	1	MODE 1 – HARDWARE ONE-SHOT
X	1	0	MODE 2 – PULSE GENERATOR
X	1	1	MODE 3 – SQUARE WAVE GENERATOR
1	0	0	MODE 4 – SOFTWARE TRIGGERED STROBE
1	0	1	MODE 5 – HARDWARE TRIGGERED STROBE

BCD:

0	BINARY COUNTER 16-BITS
1	BINARY CODED DECIMAL (BCD) COUNTER (4 DECADES)

NOTE: DON'T CARE BITS (X) SHOULD BE 0 TO INSURE COMPATIBILITY WITH FUTURE INTEL PRODUCTS.

FIGURE 8-17 8254 control word format. (*Intel Corporation*)

6. Finally, send the control word(s) you have made up to the control register address for the device. In the case of the 8254, you also have to send the starting count to each of the counter registers.

Now that you have an overview of the initialization process, let's take a closer look at how you do the last two steps for an 8254.

A separate control word must be sent for each counter that you want to use in the device. However, according to Figure 8-16a, the 8254 has only one control register address. The trick here is that the control words for all three counters are sent to the same address in the

device. As shown in Figure 8-17, you use the upper 2 bits of a control word to tell the 8254 which counter you want that control word to initialize. For example, if you are making up a control word for counter 0 in the 8254, you make the SC1 bit of the control word a 0 and the SC0 bit a 0. Later we will explain the meaning of the read-back command, specified by a 1 in each of these bits.

The 16-bit counters in the 8254 are down counters. This means that the number in a counter will be decremented by each clock pulse. You can program the 8254 to count down a loaded number in BCD (decimal) or in binary. If you make the D0 bit of the control word a 0, then the counter will treat the loaded number as a pure binary number. In this case the largest number that you can load in is FFFFH. If you make the D0 bit of the control word a 1, then the largest number you can load in the counter is 9999H, and the counter will count a loaded number down in decimal (BCD). Actually, because of the way the 8254 counts, the "largest" number you can load in for both cases is 0000, but thinking of FFFFH and 9999H makes it easier to remember the difference between the two modes.

Now let's take a brief look at the mode bits (M2, M1, and M0) in the control word format in Figure 8-17. The binary number you put in these bits specifies the effect that the gate input will have on counting and the waveform that will be produced on the OUT pin. For example, if you specify mode 3 for a counter by putting 011 in these 3 bits, the counter will be put in a square-wave mode. In this mode, the output will be high for the first half of the loaded count and low for the second half of the loaded count. When the count reaches 0, the original count is automatically reloaded and the countdown repeated. The waveform on the OUT pin in this mode will then be a square wave with a frequency equal to the input clock frequency divided by the count you wrote to the counter. A little later we will discuss and show applications for some of the six different modes.

First, let's finish looking at the control word bits and see how you send the control word and a count to the device.

The RW1 and RW0 bits of the control word are used to specify how you want to write a count to a counter or to read the count from a counter. If you want to load a 16-bit number into a counter, you put 1's in both these bits in the control word you send for that counter. After you send the control word, you send the low byte of the count to the counter address and then send the high byte of the count to the counter address. In a later paragraph we show an example of the instruction sequence to do this. In cases where you only want to load a new value in the low byte of a counter, you can send a control word with 01 in the RW bits and then send the new low byte to the counter. Likewise, if you want to load only a new high byte value in the counter, you can send a control word with 10 in the RW bits, and then send only the new high byte to the counter.

You can read the number in one of the counters at any time. The usual way to do this is to first latch the current count in some internal latches by sending a control word with 00 in the RW bits. Send another control word with 01, 10, or 11 in the RW bits to specify how you want to read out the bytes of the latched count. Then read the count from the counter address.

As a specific example of initializing an 8254, suppose that we want to use counter 0 of the 8254 in Figure 8-14 to produce a stable 78.6-kHz square-wave signal for a UART clock by dividing down the 2.45-MHz PCLK signal available on the SDK-86 board. To do this, we first connect the SDK-86 PCLK signal to the CLK input of counter 0 and tie the GATE input of the counter high to enable it for counting. To produce 78.6 kHz from 2.45 MHz, we have to divide by 32 decimal, so this is the value that we will eventually load into counter 0. First, however, we have to determine the system addresses for the device, make up the control word for counter 0, and send the control word.

As shown in Figure 8-16*b*, the system address for the control register of this 8254 is FF07H. This is where we will send the control word. For our control word we want to select counter 0, so we make the SC1 and SC0 bits both 0's. We want the counter to operate in square-wave mode. This is mode 3, so we make the mode bits of the control word 011. Since we want to divide by 32 decimal, we tell the counter to count down in decimal by making the BCD bit of the control word a 1. This makes our life easier, because we don't have to convert the 32 to binary or hex. Finally, we have to decide how we want to load the count into the counter. Since the count that we need to load in is less than 99, we only have to load the lower byte of the counter. According to Figure 8-17, the RW1 bit should be a 0 and the RW0 bit a 1 for a write to only the lower byte (LSB). The complete control word then is 00010111 in binary. Here are the instructions to send the control word and count to counter 0 of the 8254 in Figure 8-14. Note how the bits of the control word are documented.

```
MOV AL,00010111B  ; Control word for counter 0
                  ; Read/write LSB only, mode 3, BCD countdown
                  ; 00  01  011  1
                  ;  |   |   |   |_____ BCD countdown
                  ;  |   |   |_________ Mode 3
                  ;  |   |_____________ R/W LSB only
                  ;  |_________________ Select counter 0
MOV DX,0FF07H     ; Point at 8254 control register
OUT DX,AL         ; Send control word
MOV AL,32H        ; Load lower byte of count
MOV DX,0FF01H     ; Point to counter 0 count register
OUT DX,AL         ; Send count to count register
```

Note that since we set the RW bits of the control word for read/write LSB only, we do not have to include instructions to load the MSB of the counter. Programmed in this way, the 8254 will automatically load 0's in the upper byte of the counter.

If you need to load a count that is larger than 1 byte, make the RW bits in the control word both 1's. Send the lower byte of the count as shown above. Then send the high byte of the count to the count register by adding the instructions

```
MOV AL,HIGH_BYTE_OF_COUNT ; Load MSB of count
OUT DX,AL                 ; Send MSB to
                            count register
```

Note that the high byte of the count is sent to the same address that the low byte of the count was sent.

For each counter that you want to use in an 8254, you repeat the preceding series of six or eight instructions with the control word and count for the mode that you want. Before going on with this chapter, review the six initialization steps shown at the start of this section to make sure these are firmly fixed in your mind. In the next section we discuss and show some applications of the different modes in which an 8254 counter can be operated, but we do not have space there to show all the steps for each of the modes.

8254 Counter Modes and Applications

As we mentioned previously, an 8254 counter can be programmed to operate in any one of six different modes. The Intel data book uses timing diagrams such as those in Figure 8-18 to show how a counter functions in each of these modes. Since these waveforms may not be totally obvious to you at first glance, we will work our way through some of them to show you how to interpret them. We will also show some uses of the different counter modes. As you read through this section, don't try to absorb all the details of the different modes. Concentrate on learning to interpret the timing waveforms and on the different types of output signals you can produce with an 8254.

MODE 0—INTERRUPT ON TERMINAL COUNT

First read the Intel notes at the bottom of Figure 8-18; then take a look at the top set of waveforms in the figure. For this first example, the GATE input is held high so that the counter is always enabled for counting. The first dip in the waveform labeled $\overline{WR}$ represents the control word for the counter being written to the 8254. CW = 10 over this dip indicates that the control word written is 10H. According to the control word format in Figure 8-17, this means that counter 0 is being initialized for binary counting, mode 0, and a read/write of only the LSB. After the control word is written to the control register, the output pin of counter 0 will go low. The next dip in the $\overline{WR}$ waveform represents a count of 4 being written to the count register of counter 0. Before this count can be counted down, it must be transferred from the count register to the actual counter. If you look at the count values shown under the OUT waveform in the timing diagram, you should see that the count of 4 is transferred into the counter by the next clock pulse after $\overline{WR}$ goes high. Each clock pulse after this will decrement the count by 1. When the count is decremented to 0, the OUT pin will go high. If you write a count N to a counter in mode 0, the OUT pin will go high after N + 1 clock pulses have occurred. Note that the counter decrements from 0000 to FFFFH on the next clock pulse unless you load some new count into the

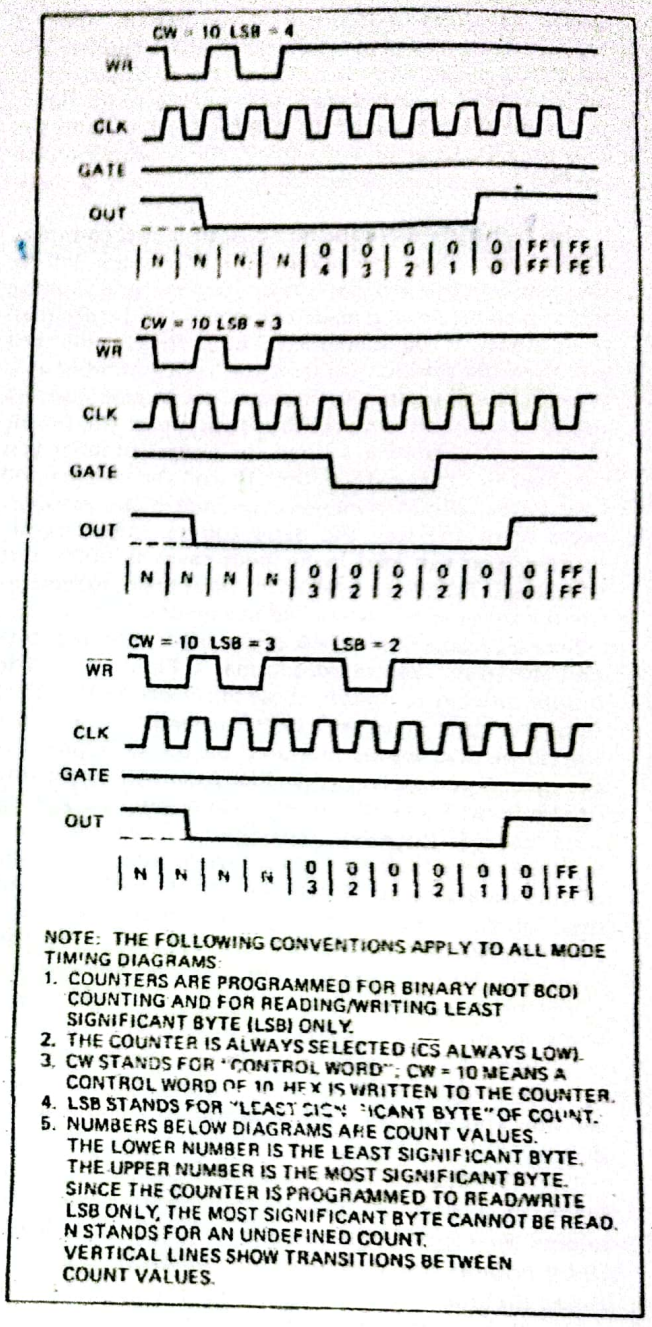
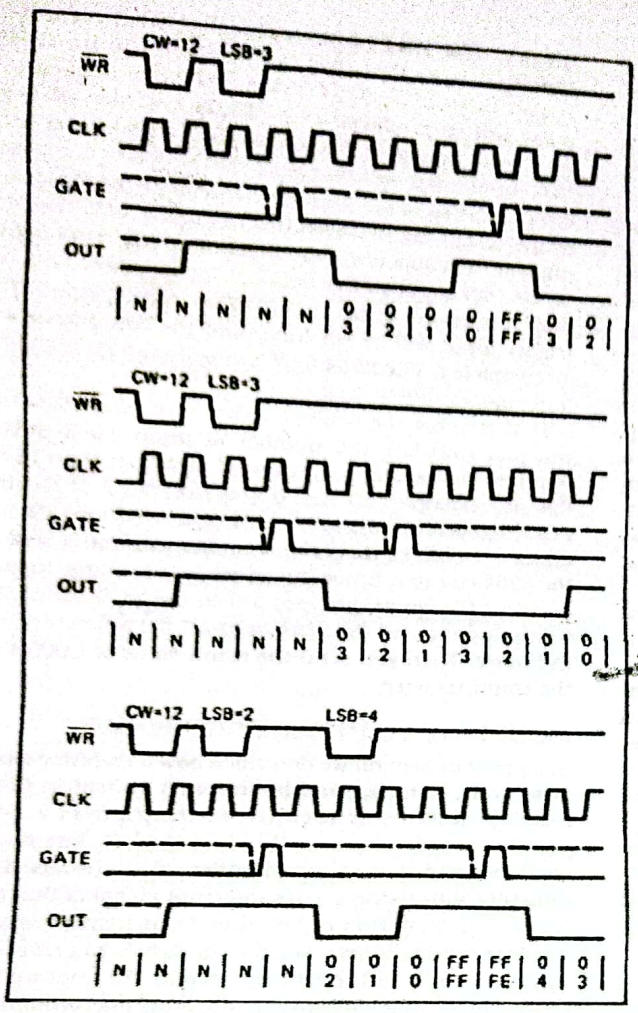


FIGURE 8-18 8254 MODE 0 example timing diagrams. (*Intel Corporation*)



MODE 1

FIGURE 8-19 8254 MODE 1 example timing diagrams. (*Intel Corporation*)

counter. If the OUT pin is connected to an 8259A IR input or the NMI interrupt input of an 8086, then the processor will be interrupted when the counter reaches 0 (terminal count).

The second set of waveforms in Figure 8-18 shows that if the GATE input is made low, the counter value will be held. When the GATE input is made high again, the counter continues to decrement by 1 for each clock pulse.

The third set of waveforms in Figure 8-18 shows that if a new count is written to a counter, the new count will be loaded into the counter on the next clock pulse. Following clock pulses will decrement the new count until it reaches 0.

As an example of what you can use this mode for, suppose that as one of its jobs you want to use an 8086 to control some parking lot signs around an electronics factory. The main parking lot can hold 1000 cars. When it gets full, you want to turn on a sign which directs people to another lot. To detect when a car enters the lot, you can use an optical sensor such as the one shown in Figure 8-10. Each time a car passes through, this circuit will produce a pulse. You could connect the signal from this sensor directly to an interrupt input and have the processor count interrupts, as we did for the printed-circuit-board-making machine in a previous example. However, the less you burden the processor with trivial tasks such as this, the more time it has available to do complex work for you. Therefore, you let a counter in an 8254 count cars and interrupt the 8086 only when it has counted 1000 cars.

You connect the output from the optical sensor circuit to the CLK input of, say, counter 1 of an 8254. You tie the GATE input of counter 1 to +5 V so it will be enabled for counting and connect the OUT pin of counter 1 to an interrupt input on an 8259A or the NMI input on the 8086.

In the mainline program, you initialize counter 1 for mode 0, BCD counting, and read/write LSB then MSB with a control word of 01110001 binary. You want the counter to produce an interrupt after 1000 pulses from the sensor, so you will send a count of 999 decimal to the counter. The reason that you want to send 999 instead of 1000 is that, as shown in Figure 8-18, the OUT pin will go high N + 1 clock pulses after the count value is written to the counter. Since you initialized the counter for read/write LSB then MSB, you send 99H and then 09H to the address of counter 1. By initializing the counter for BCD counting, you can just send the count value as a BCD number instead of having to convert it to hex.

The service procedure for this interrupt will contain instructions which turn on the parking-lot-full sign, close off the main entrance, and return to the mainline program. For this example you don't have to worry that the counter decrements from 0000 to FFFFH, because after you shut the gate, the counter will not receive any more interrupts.

MODE 1—HARDWARE-RETRIGGERABLE ONE-SHOT

The basic principle of a *one-shot* is that when a signal is applied to the trigger input of the device, its output will be asserted. After a fixed amount of time the output will automatically return to its unasserted state. For a TTL one-shot such as the 74LS122, the time that the output is asserted is determined by the time constant of a resistor and a capacitor connected to the device. For an 8254 counter in one-shot mode, the time that the output is asserted low is determined by the frequency of an applied clock and by a count loaded into the counter. The advantage of the 8254 approach is that the output pulse width can be changed under program control and, if a crystal-controlled clock is used, the output pulse width can be very accurately specified.

Figure 8-19 shows some example timing waveforms for an 8254 counter in mode 1. Let's take a look at the top set of waveforms. Again, the first dip in the $\overline{WR}$ waveform represents the control word of 12H being sent to the 8254. Use Figure 8-17 to help you determine how this control word initializes the device. You should find

that a control word of 12H programs counter 0 for binary count, mode 1, read/write LSB only. When the control word is written to the 8254, the OUT pin goes high.

The second dip in the $\overline{WR}$ waveform represents writing a count to the counter. Note that, because the GATE input is low, the counter does not start counting down immediately when the count is written, as it does in mode 0. For mode 1, the GATE input functions as a trigger input. When the GATE/trigger input is made high, the count will be transferred from the count register to the actual counter on the next clock pulse. Each following clock pulse will decrement the counter by 1. When the counter reaches 0, the OUT pin will go high again. In other words, if you load a value of N in the counter and trigger the device by making the GATE input high, the OUT pin will go low for a time equal to N clock cycles. The output pulse width is then N times the period of the signal applied to the CLK input. Incidentally, the dashed sections of the GATE waveforms in Figure 8-19 mean that the GATE/trigger input signal can go low again any time during that time interval.

The second set of waveforms in Figure 8-19 demonstrates what is meant by the term *retriggerable*. If another trigger pulse comes before the previously loaded count has been counted down to 0, the original count will be reloaded on the next clock pulse. The countdown will then start over and continue until another trigger occurs or until the count reaches 0. If trigger pulses continue to come before the count is decremented to 0, the OUT pin will remain low.

The bottom set of waveforms in Figure 8-19 shows that if you write a new count to a count register while the OUT pin is low, the new count will not be loaded into the counter and counted down until the next trigger pulse occurs.

For an example of the use of mode 1, we will show you how to make a circuit which produces an interrupt signal if the ac power fails. This circuit could be connected to the NMI input of an 8086 to call an interrupt procedure which saves parameters in battery-backed RAM when the ac power fails.

Figure 8-20 shows a circuit which uses an optical coupler (an LED and a phototransistor packaged together) to produce logic-level pulses at power line frequency. The 74LS14 inverters sharpen the edges of these pulses so that they can be applied to the GATE/trigger input of an 8254. For a 60-Hz line frequency, a pulse will be produced every 16.66 ms. Now what we want to do here is to load the counter with a value such that the counter will always be retriggered by the power line pulses before the countdown is completed. As shown by the second set of waveforms in Figure 8-19, the OUT pin will then stay low and not send an interrupt signal to the NMI input of the 8086. If the ac power fails, no more pulses come in to the 8254 trigger input. The trigger input will be left high, and the countdown will be completed. The 8254 OUT pin will then go high and interrupt the 8086.

To determine the counter value for this application, you just calculate the number of input clock pulses required to produce a countdown time longer than 16.66 ms—for example, 20 ms. If you use the 2.4576-MHz PCLK signal on an SDK-86 board, 20 ms requires 49,152 cycles of PCLK, so this is the number you would load in the 8254 counter. Since this number is too large to load in as a BCD count, you put a 0 in the BCD bit of the control word to tell the 8254 to count the number down in binary. Then you send the count value of C000H to the count register.

MODE 2—TIMED INTERRUPT GENERATOR

In a previous section we described how a real-time clock of seconds, minutes, and hours could be kept in three memory locations by counting interrupts from a 1-Hz pulse source. We also described how the 1-Hz interrupts could be used to measure off other time intervals. The difficulty with using a 1-Hz interrupt signal is that the maximum resolution of any time measurement is 1 s. In other words, if you use a 1-Hz signal, you can only measure times to the nearest second. To improve the resolution of time measurements, most microcomputer systems use a higher-frequency signal such as 1 kHz for a real-time clock interrupt. With a 1-kHz interrupt signal, the time resolution is 1 ms. An 8254 counter operating in mode 2 can be used to produce a stable 1-kHz signal by dividing down the processor clock signal.

Figure 8-21 shows the waveforms for an 8254 counter operating in mode 2. Let's look at the top set of waveforms

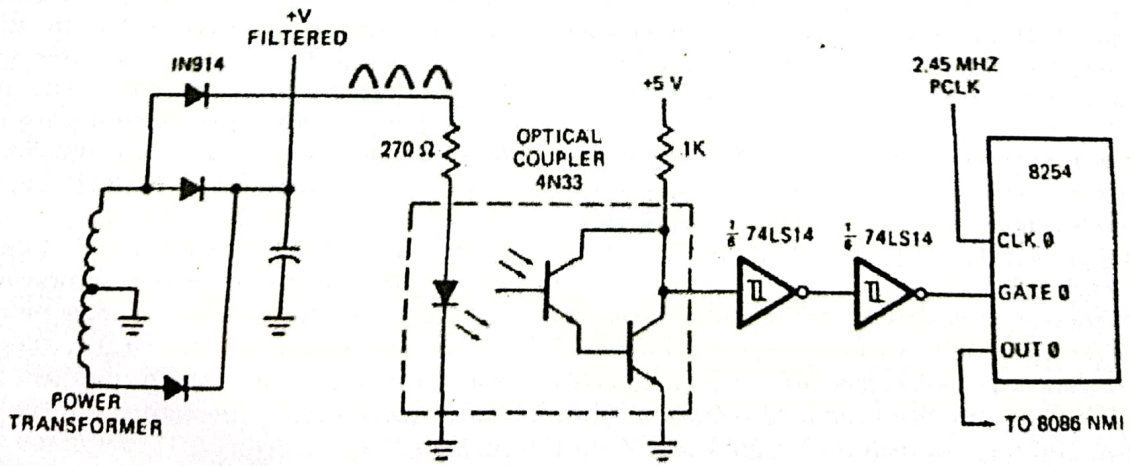


FIGURE 8-20 Circuit to produce logic-level pulses at power line frequency.

first. The two dips in the $\overline{WR}$ waveform represent a control word and the LSB of a count being written to the count register. The next clock pulse after the count is written will transfer the count from the count register to the actual counter. Since the GATE input is high, succeeding clock pulses will count down this value until it reaches 1. When the count reaches 1, the OUT pin, which was previously high, will go low for one clock pulse time. The falling edge of the next clock pulse will cause the OUT pin to go high again and the original count to be loaded into the counter again. Successive clock pulses will cause the countdown and load cycle to repeat over and over. If the counter is loaded with a number N, the OUT pin will go low for one clock cycle every N input clock pulses. The frequency of the output waveform then will be equal to the input clock frequency divided by N.

Now, for a specific example, suppose that you want to produce a 1-kHz signal for a real-time clock from an 8-MHz processor clock signal. To do this, you connect the processor clock signal to the CLK input on one of the 8254 counters and tie the GATE input of that counter high. You initialize that counter for BCD counting, mode 2, and read/write LSB then MSB. Since you want to divide the 8 MHz by 8000 decimal to get 1 kHz, you then write 00H to the counter as the LSB and 80H to the counter as the MSB.

A question that may occur to you at this point is, How do I count seconds if the interrupts are coming in every millisecond? The answer to the question is that you set aside a memory location as a milliseconds counter and initialize that location with 1000 decimal (3E8H). The interrupt-service procedure decrements this count each time an interrupt occurs and checks to see if the count is down to 0 yet. If the count is not 0, then execution is simply returned to the mainline. If the count is down to 0, 1000 interrupts or 1 s has passed. The milliseconds counter location is then reloaded with 3E8H, and the seconds-minutes-hours procedure is called to update the count of seconds, minutes, and hours. An exercise in the accompanying lab manual gives you a chance to develop a real-time clock in this way. Incidentally, the 1-kHz interrupt-service procedure can be used to measure off several different time intervals that are multiples of 1 ms.

The middle set of mode 2 waveforms in Figure 8-21 demonstrates that if the GATE input is made low while the counter is counting, counting will stop. If the GATE input is made high again, the original count will be reloaded into the counter by the next clock pulse. Succeeding clock pulses will decrement the loaded count.

The bottom set of mode 2 waveforms in Figure 8-21 shows that if a new count is written to the count register, this new count will not be transferred to the counter until the previously loaded count has been decremented to 1.

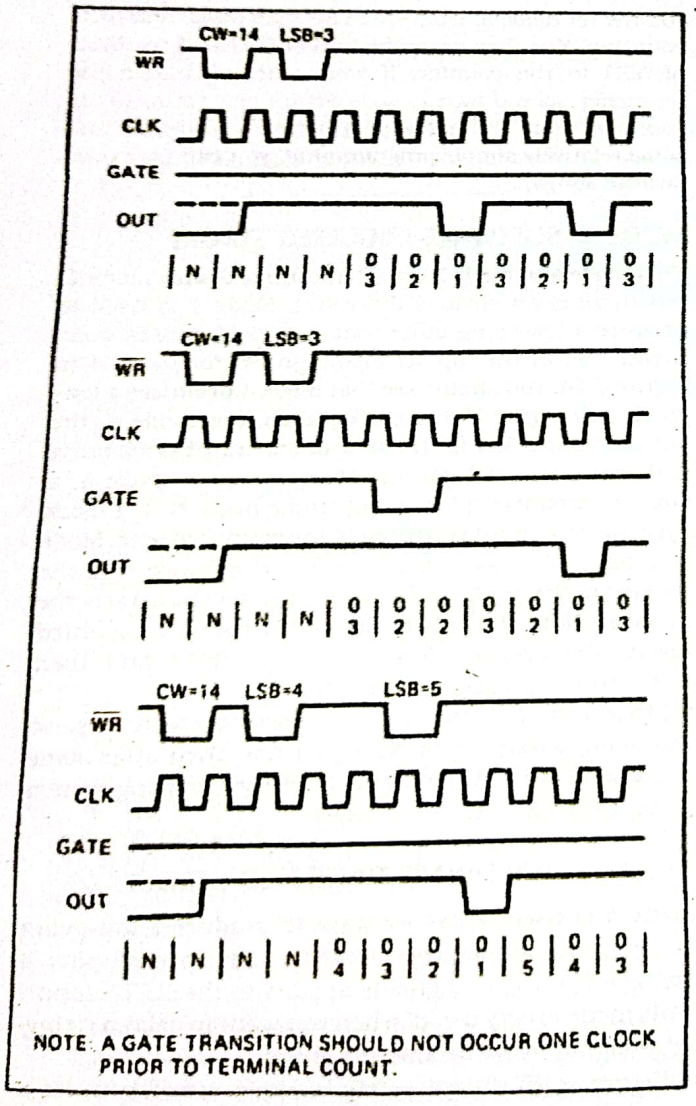


FIGURE 8-21 8254 MODE 2 example timing waveforms. *(Intel Corporation)*

MODE 3—SQUARE-WAVE MODE

If an 8254 counter is programmed for mode 3 and an even number is written to its count register, the waveform on the OUT pin will be a square wave. The frequency of the square wave will be equal to the frequency of the input clock divided by the number written to the count register. If an odd number is written to a counter programmed for mode 3, the output waveform will be high for one more clock cycle than it is low, so the waveform will not be quite symmetrical. Figure 8-22, p. 230, shows some example waveforms for mode 3. By now these waveforms should look quite familiar to you.

The top set of waveforms shows that after a control word is written to the control register and a count is written to the count register, the count is transferred to the counter on the next clock pulse. As shown by the count sequence under the OUT waveform, each additional clock pulse decrements the counter by 2. When the count is down to 2, the OUT pin goes low and the original count is reloaded. The OUT pin stays low while the loaded count is again counted down by 2's. When the count is down to 2, the OUT pin goes high again and the original count is again loaded into the counter. The cycle then repeats.

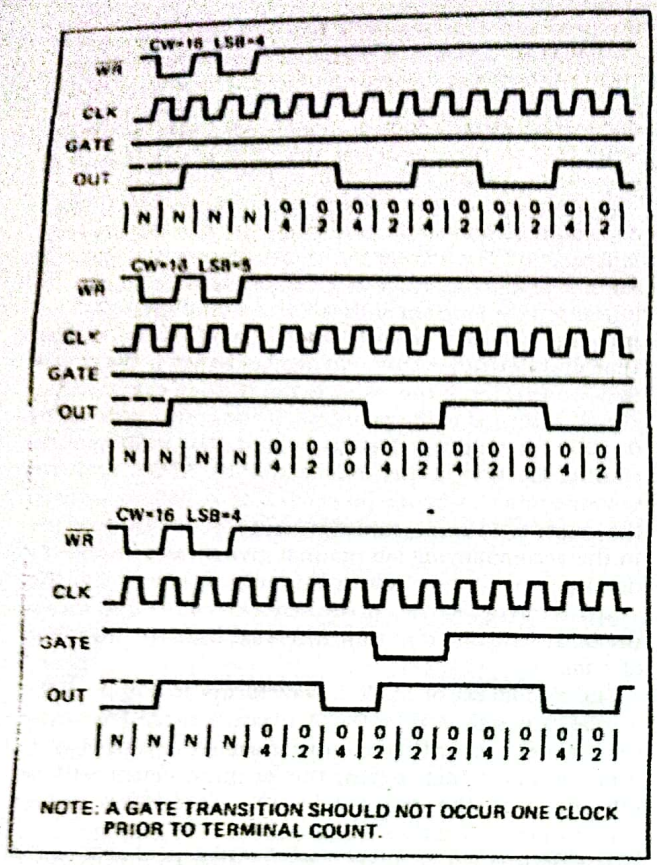


FIGURE 8-22 8254 MODE 3 example timing waveforms. (*Intel Corporation*)

The center set of waveforms in Figure 8-22 shows what happens if an odd number is written to the count register. As you can see from this waveform, the number of clock cycles for each waveform is still equal to the number loaded into the count register. However, as we mentioned before, the clock is high for one more clock cycle than it is low.

The bottom set of waveforms in Figure 8-22 shows that counting stops if the gate is made low at any time. After the GATE input is made high again, the countdown will continue.

Mode 3 can be used for any case where you want a repetitive square-wave-type signal. An 8254 counter operating in mode 3 can be used to generate the baud rate clock for a USART such as the 8251A. Also, mode 3 could be used to generate interrupt pulses for a real-time clock as we described for mode 2. The square-wave signal has the advantage that it is more easily observed with a scope than the narrow pulse produced by mode 2 operation.

Another use of 8254 counters operating in mode 3 is as programmable audio-tone generators. For this application, a high-frequency clock such as the 2.4576-MHz PCLK signal on an SDK-86 board is connected to the counter CLK input, the GATE input is tied high,

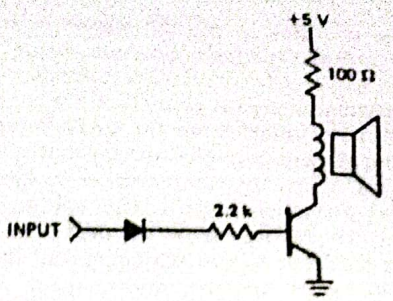


FIGURE 8-23 Audio speaker buffer for 8254 timer output or port.

and the OUT pin is connected to an audio buffer such as that shown in Figure 8-23.

As an example of this application, suppose that you want to produce a tone that is a musical A of 440 Hz from the 2.4576-MHz PCLK signal. Dividing the PCLK signal by 5585 will give the desired 440 Hz. Therefore, you simply send a control word which programs a counter for mode 3, read/write LSB then MSB, and BCD counting. You then write the LSB of 85H and the MSB of 55H to the counter. If you want to change the frequency, all you have to do is write a new count to the count register. With a few programmable counters and some relatively simple programming, you can play your favorite songs.

MODE 4—SOFTWARE-TRIGGERED STROBE

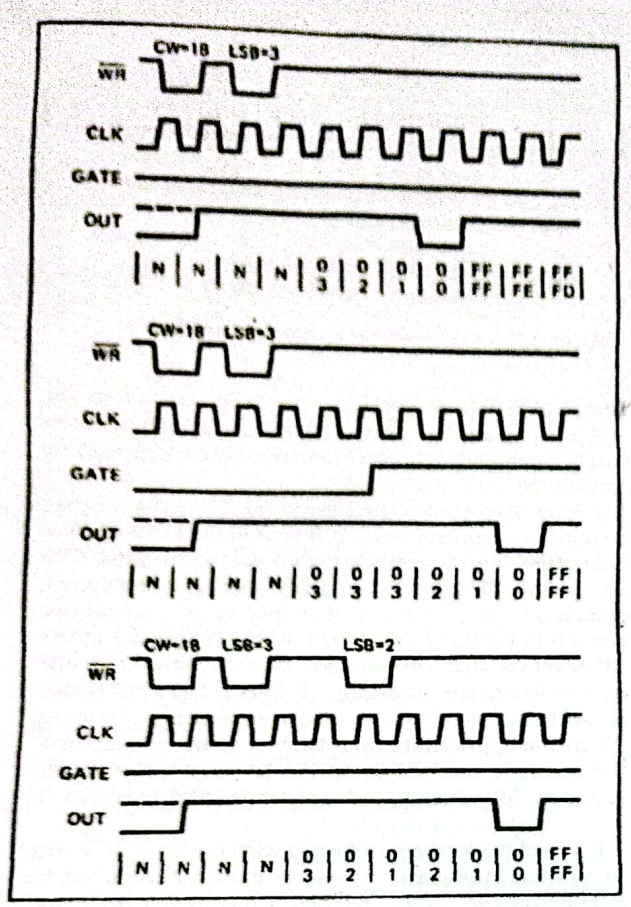
This mode and mode 5 are often confused with mode 1, but there is an obvious difference. Mode 1 is used to produce a low-going pulse that is N clock pulses wide. If you look at the top set of waveforms for mode 4 in Figure 8-24, you should see that mode 4 produces a low-going pulse *after* N + 1 clock pulses. For mode 4, the output pulse is low for the time of one input clock pulse and then returns high. In other words, in mode 4, a counter produces a low-going strobe pulse N + 1 clock cycles after a count is written to the count register. Mode 4 is referred to as *software-triggered* because it is the writing of the count to the count register that starts the process. Note that after the loaded count is counted down, the counter decrements to FFFFH and then continues to decrement from there.

Mode 4 can be used in a case where you want to send out some parallel data on a port and then after some delay send out a strobe signal to let the receiving system know that the data is available.

MODE 5—HARDWARE-TRIGGERED STROBE

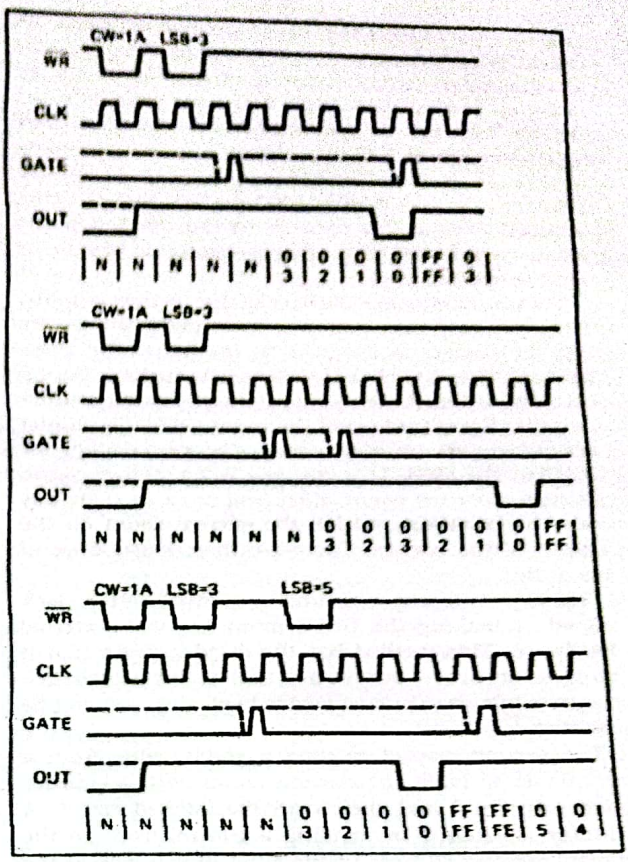
Mode 5 is used where we want to produce a low-going strobe pulse some programmable time interval after a rising-edge trigger signal is applied to the GATE input. This mode is very useful when you want to delay a rising-edge signal by some amount of time.

Figure 8-25 shows some example waveforms for a counter operating in mode 5. For a start, let's look at the top set of waveforms. As usual, we write a control word and the desired count to a counter. As shown



MODE 4

FIGURE 8-24 8254 MODE 4 example timing waveforms. (Intel Corporation)



MODE 5

FIGURE 8-25 8254 MODE 5 example timing waveforms. (Intel Corporation)

by the count sequence under the OUT waveform, however, the count does not get transferred to the counter until the GATE (trigger) is made high. When the trigger input is made high, the count will be transferred to the counter on the next clock pulse. Succeeding clock pulses will decrement the counter. When the counter reaches 0, the OUT pin will go low for one clock pulse time. The OUT pin will go low $N + 1$ clock pulses after the trigger input goes high.

The second set of waveforms in Figure 8-25 shows that if another trigger pulse occurs during the countdown time, the original count will be reloaded on the next clock pulse and the countdown will start over. The OUT pin will remain high until the count is finally counted down. If trigger pulses continue to come before the countdown is completed, the OUT pin will continue to stay high. Therefore, you can use a counter in mode 5 to produce a power fail signal, as we showed in the previous discussion of mode 1. Note that for mode 5, however, the OUT pin will be high if the power is on and go low when the power fails.

The bottom set of waveforms in Figure 8-25 shows that if a new count is written to a counter, the new count will not be loaded into the counter until a new trigger pulse occurs.

USING A NONSYSTEM CLOCK WITH AN 8254 IN MODES 2 AND 3

If you are applying a signal which is not derived from the system clock to the CLK input of an 8254 in mode 2 or mode 3, then a small note in the Intel data sheet indicates that the GATE input of the counter must be pulsed low just after the count is written to the counter. An easy way to do this is to connect the GATE input of the counter to an otherwise unused output port pin. You can then pulse the GATE by outputting a low and then outputting a high to that port pin.

READING THE COUNT FROM AN 8254 COUNTER

For many counter applications, we want to be able to read the current count in the counter. Suppose, for example, that we are using an 8254 counter to count the cars coming into a parking lot, as we did in our example for mode 0 in an earlier section. In that case

we used the counter to produce an interrupt when the parking lot was full, so we could shut the gate. Now further suppose that as part of a traffic flow study, we want to find out how many cars have come into the lot by 7:30 A.M. An interrupt-driven real-time clock procedure can, at 7:30 A.M., call a procedure which reads in the current count from the counter. Since the counter was initially loaded with 1000 decimal and is being counted down as cars come in, we can simply subtract the current count from 1000 to determine how many cars have come in.

The counters in an 8254 have latches on their outputs. When you read the count from a counter, what you are actually reading is the data on the outputs of these latches. These latches are normally enabled during counting so that the latch outputs just follow the counter outputs. If you try to read the count while the counter is counting, the count may change between reading the LSB and the MSB. This may give you a strange count. To read a correct count, then, you must in some way stop the counting or latch the current count on the output of the latches. There are three major ways of doing this.

The first is to stop counting by turning off the clock signal or making the GATE input low with external hardware. This method has the disadvantages that it requires external hardware and that a clock pulse which occurs while the clock is disabled will obviously not be counted.

The second way of reading a stable value from a counter is to latch the current count with a counter latch command and then read the latched count. A counter is latched by sending a control word to the control register address in the 8254. If you look at the format for the 8254 control word in Figure 8-17, you should see that a counter latch command is specified by making the RW1 and RW0 bits both 0. The SC1 and SC0 bits specify which counter we want to latch. The lower 4 bits of the control word are "don't cares" for a counter latch command word, so we usually make them 0's for simplicity. As an example, here is the sequence of instructions you would use to latch and read the LSB and MSB from counter 1 of the 8254 in Figure 8-14. We assume that the counter was already programmed for read/write LSB then MSB when the device was initialized. If the counter was programmed for only LSB or only MSB, then only that byte can be read.

```
MOV AL,01000000B    ; Counter 1 latch command
MOV DX,0FF07H       ; Point at 8254 control register
OUT DX,AL           ; Send latch command
MOV DX,0FF03H       ; Point at counter 1 address
IN AL,DX            ; Read LSB of latched count
MOV AH,AL           ; Save LSB of latched count
IN AL,DX            ; Read MSB of latched count
XCHG AH,AL          ; Count now in AX
```

When a counter latch command is sent, the latched count is held until it is read. When the count is read from the latches, the latch outputs return to following the counter outputs.

The third method of reading a stable count from a counter is to latch the count with a read-back command.

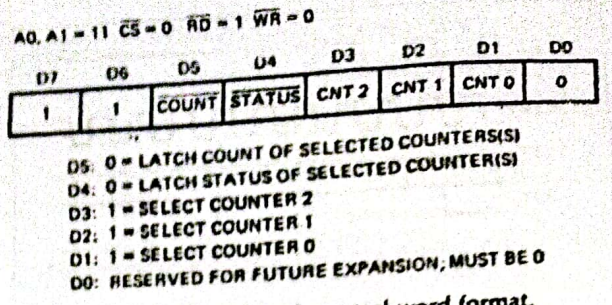


FIGURE 8-26 8254 read-back control word format.

This method is available in the 8254 but not in the 8253. It is essentially an enhanced version of the counter latch command approach described in the preceding paragraphs.

Figure 8-26 shows the format for the 8254 counter read-back command word. It is sent to the same address that other control words are for a particular 8254. The 1's in bits D7 and D6 identify this as a read-back command word. To latch the count on a counter, you put a 0 in bit D5 of the control word and put a 1 in the bit position that corresponds to that counter in the control word. The advantage of this control word is that you can latch one, two, or all three counters by putting 1's in the appropriate bits. Once a counter is latched, the count is read as shown in the previous example program. After being read, the latch outputs return to following the counter outputs.

If a read-back command word with bit D4 = 0 is sent to an 8254, the status of one or more counters will be latched on the output latches. Consult the Intel data sheet for further information on this latched status.

The preceding sections have shown how 8254 counters can be used to do a wide variety of tasks around microcomputers. Many of these applications produce an interrupt signal which must be connected to an interrupt input on the microprocessor. In the next section we show how a *priority interrupt controller* device, the Intel 8259A, is used to service multiple interrupts.

8259A PRIORITY INTERRUPT CONTROLLER

Previous sections of this chapter show how interrupts can be used for a variety of applications. In a small system, for example, we might read ASCII characters in from a keyboard on an interrupt basis; count interrupts from a timer to produce a real-time clock of seconds, minutes, and hours; and detect several emergency or job-done conditions on an interrupt basis. Each of these interrupt applications requires a separate interrupt input. If we are working with an 8086, we have a problem here because the 8086 has only two interrupt inputs, NMI and INTR. If we save NMI for a power failure interrupt, this leaves only one interrupt input for all the other applications. For applications where we have interrupts from multiple sources, we use an external device called a *priority interrupt controller* (PIC) to "funnel" the interrupt signals into a single interrupt input

on the processor. In this section, we show how a common PIC, the Intel 8259A, is connected in an 8086 system, how it is initialized, and how it is used to handle interrupts from multiple sources.

8259A Overview and System Connections

To show you how an 8259A functions in an 8086 system, we first need to review how the 8086 INTR input works. Remember from Figure 8-5 and a discussion earlier in this chapter that if the 8086 interrupt flag is set and the INTR input receives a high signal, the 8086 will

1. Send out two interrupt acknowledge pulses on its $\overline{\text{INTA}}$ pin to the $\overline{\text{INTA}}$ pin of an 8259A PIC. The $\overline{\text{INTA}}$ pulses tell the 8259A to send the desired interrupt type to the 8086 on the data bus.
2. Multiply the interrupt type it receives from the 8259A by 4 to produce an address in the interrupt vector table.
3. Push the flags on the stack.
4. Clear IF and TF.
5. Push the return address on the stack.
6. Get the starting address for the interrupt procedure from the interrupt-vector table and load that address in CS and IP.
7. Execute the interrupt-service procedure.

Now let's take a little closer look at how the 8259A functions during this process. To start, study the internal block diagram of an 8259A in Figure 8-27. In the figure, first notice the 8-bit data bus and control signal pins in the upper left corner of the diagram. The data bus allows the 8086 to send control words to the 8259A and read a status word from the 8259A. The $\overline{\text{RD}}$ and $\overline{\text{WR}}$ inputs control these transfers when the device is selected by asserting its chip select ($\overline{\text{CS}}$) input low. The 8-bit data bus also allows the 8259A to send interrupt types to the 8086.

Next, in Figure 8-27, observe the eight interrupt inputs labeled IR0 through IR7 on the right side of the diagram. If the 8259A is properly enabled, an interrupt signal applied to any one of these inputs will cause the 8259A to assert its INT output pin high. If this pin is connected to the INTR pin of an 8086 and if the 8086 interrupt flag is set, then this high signal will cause the previously described INTR response.

The $\overline{\text{INTA}}$ input of the 8259A is connected to the $\overline{\text{INTA}}$ output of the 8086. The 8259A uses the first $\overline{\text{INTA}}$ pulse from the 8086 to do some activities that depend on the mode in which it is programmed. When it receives the second $\overline{\text{INTA}}$ pulse from the 8086, the 8259A outputs an interrupt type on the 8-bit data bus, as shown in Figure 8-6. The interrupt type that it sends to the 8086 is determined by the IR input that received an interrupt signal and by a number you send the 8259A when you initialize it. The point here is that the 8259A "funnels" interrupt signals from up to eight different sources into the 8086 INTR input, and it sends the 8086 a specified interrupt type for each of the eight interrupt inputs.

At this point the question that may occur to you is, What happens if interrupt signals appear at, for example, IR2 and IR4 at the same time? In the *fixed-priority mode* that the 8259A is usually operated in, the answer to this question is quite simple. In this mode, the IR0 input has the highest priority, the IR1 input the next highest, and so on down to IR7, which has the lowest priority. What this means is that if two interrupt signals occur at the same time, the 8259A will service the one with

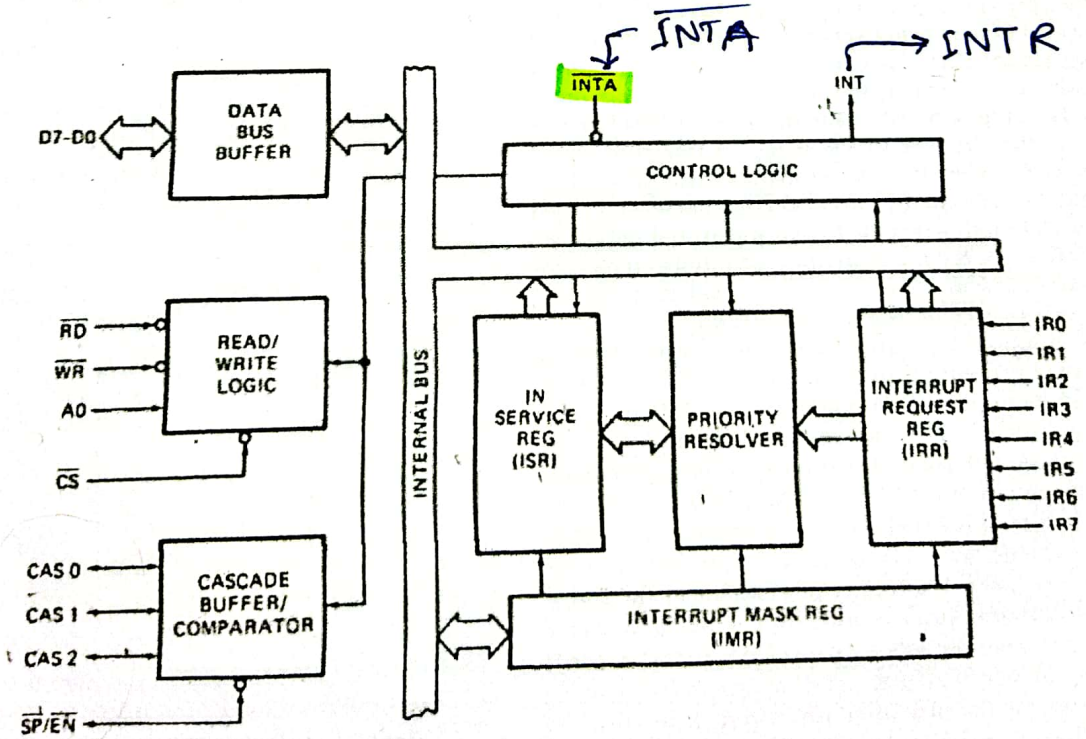


FIGURE 8-27 8259A internal block diagram. (*Intel Corporation*)

the highest priority first, assuming that both inputs are unmasked (enabled) in the 8259A.

Now let's look again at the block diagram of the 8259A in Figure 8-27 so we can explain in more detail how the device will respond to multiple interrupt signals. In the block diagram note the four boxes labeled *interrupt request register* (IRR), *interrupt mask register* (IMR), *in-service register* (ISR), and *priority resolver*.

The interrupt mask register is used to disable (mask) or enable (unmask) individual interrupt inputs. Each bit in this register corresponds to the interrupt input with the same number. You unmask an interrupt input by sending a command word with a 0 in the bit position that corresponds to that input.

The interrupt request register keeps track of which interrupt inputs are asking for service. If an interrupt input has an interrupt signal on it, then the corresponding bit in the interrupt request register will be set.

NOTE: An interrupt signal must remain high on an IR input until after the falling edge of the first INTA pulse.

The in-service register keeps track of which interrupt inputs are currently being serviced. For each input that is currently being serviced, the corresponding bit will be set in the in-service register.

The priority resolver acts as a "judge" that determines if and when an interrupt request on one of the IR inputs gets serviced.

As an example of how this works, suppose that IR2 and IR4 are unmasked and that an interrupt signal comes in on the IR4 input. The interrupt request on the IR4 input will set bit 4 in the interrupt request register. The priority resolver will detect that this bit is set and check the bits in the in-service register (ISR) to see if a higher-priority input is being serviced. If a higher-priority input is being serviced, as indicated by a bit being set for that input in the ISR, then the priority resolver will take no action. If no higher-priority interrupt is being serviced, then the priority resolver will activate the circuitry which sends an interrupt signal to the 8086. When the 8086 responds with INTA pulses, the 8259A will send the interrupt type that was specified for the IR4 input when the 8259A was initialized. As we said before, the 8086 will use the type number it receives from the 8259A to find and execute the interrupt-service procedure written for the IR4 interrupt.

Now, suppose that while the 8086 is executing the IR4 service procedure, an interrupt signal arrives at the IR2 input of the 8259A. This will set bit 2 of the interrupt request register. Since we assumed for this example that IR2 was unmasked, the priority resolver will detect that this bit in the IRR is set and make a decision whether to send another interrupt signal to the 8086. To make the decision, the priority resolver looks at the in-service register. If a higher-priority bit in the ISR is set, then a higher-priority interrupt is being serviced. The priority resolver will wait until the higher-priority bit in the ISR is reset before sending an interrupt signal to the 8086 for the new interrupt input. If the priority resolver finds that the new interrupt has a higher priority than the highest-priority interrupt currently being serviced, it will set the appropriate bit in the ISR and activate the circuitry which sends a new INT signal to the 8086. For our example here, IR2 has a higher priority than IR4, so the priority resolver will set bit 2 of the ISR and activate the circuitry which sends a new INT signal to the 8086. If the 8086 INTR input was reenabled with an STI instruction at the start of the IR4 service procedure, as shown in Figure 8-28*a*, then this new INT signal will interrupt the 8086 again. When the 8086 sends out a second INTA pulse in response to this interrupt, the 8259A will send it the type number for the IR2 service procedure. The 8086 will use the received type number to find and execute the IR2 service procedure.

At the end of the IR2 procedure, we send the 8259A a command word that resets bit 2 of the in-service register so that lower-priority interrupts can be serviced. After that, an IRET instruction at the end of the IR2 procedure sends execution back to the interrupted IR4 procedure. At the end of the IR4 procedure, we send the 8259A a command word which resets bit 4 of the in-service register so that lower-priority interrupts can be

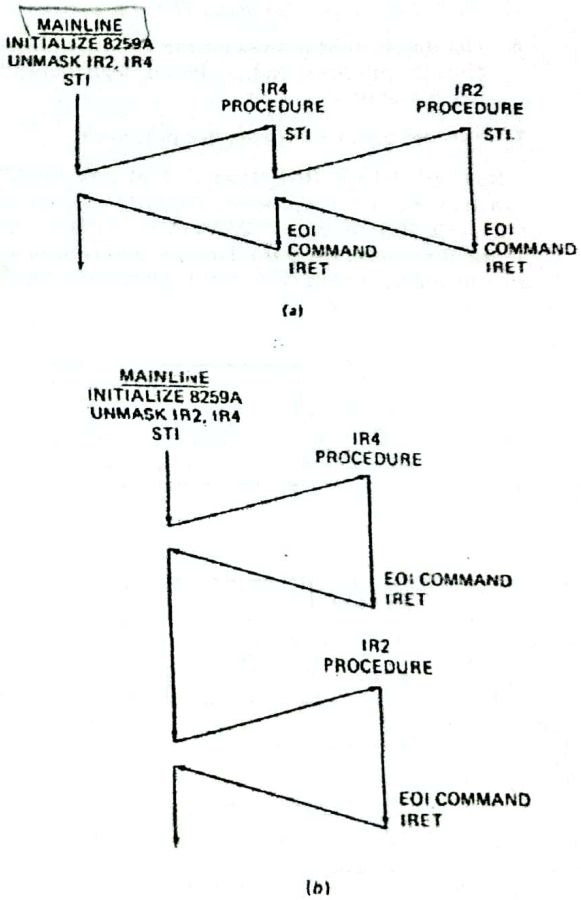


FIGURE 8-28 8259A and 8086 program flow for IR4 interrupt followed by IR2 interrupt. (*a*) Response with INTR enabled in IR4 procedure. (*b*) Response with INTR not enabled in IR4 procedure.

serviced. An IRET instruction at the end of the IR4 procedure returns execution to the mainline program. This all sounds very messy, but it is really just a special case of nested procedures. Incidentally, if the IR4 procedure did not reenable the 8086 INTR input with an STI instruction, the 8086 would not respond to the IR2-caused INT signal until it finished executing the IR4 procedure, as shown in Figure 8-28b.

We can't describe all the possible cases, but the main point here is that the 8086 and the 8259A can be programmed to respond to interrupt signals from multiple sources in almost any way you want them to. Now, before we can show you how to initialize and write programs for an 8259A, we need to show you more about how one or more 8259As are connected in a microcomputer system.

8259A System Connections and Cascading

Figure 8-14 shows how an 8259A can be added to an SDK-86 board. As shown by the truth table in Figure 8-15, the 74LS138 address decoder will assert the $\overline{CS}$ input of the 8259A when an I/O base address of FF00H is on the address bus. The A0 input of the 8259A is used to select one of two internal addresses in the device. This pin is connected to system address line A1, so the system addresses for the two internal addresses of the 8259A are FF00H and FF02H. The eight data lines of the 8259A are always connected to the lower half of the 8086 data bus because the 8086 expects to receive interrupt types on these lower eight data lines. $\overline{RD}$ and $\overline{WR}$ are connected to the system $\overline{RD}$ and $\overline{WR}$ lines. $\overline{INTA}$ from the 8086 is connected to $\overline{INTA}$ on the 8259A. The interrupt request signal, INT, from the 8259A is connected to the INTR input of the 8086. The multipurpose $\overline{SP/EN}$ pin is tied high because we are using only one 8259A in this system. When just one 8259A is used in a system, the cascade lines (CAS0, CAS1, and CAS2) can be left open. The eight IR inputs are available for interrupt signals. Unused IR inputs should be tied to ground so that a noise pulse cannot accidentally cause an interrupt. In a later section we will show you how to initialize this 8259A, but first we need to show you how more than one 8259A can be added to a system.

The dashed box on the right side of Figure 8-14 shows how another 8259A could be added to the SDK-86 system to give a total of 15 interrupt inputs. If needed, an 8259A could be connected to each of the eight IR inputs of the original 8259A to give a total of 64 interrupt inputs. Note that since the 8086 has only one INTR input, only one of the 8259A INT pins is connected to the 8086 INTR pin. The 8259A connected directly into the 8086 INTR pin is referred to as the *master*. The INT pin from the other 8259A connects into an IR input on the master. This secondary, or *cascaded*, device is referred to as a *slave*. Note that the $\overline{INTA}$ signal from the 8086 goes to both the master and the slave devices.

Each 8259A has its own addresses so that command words can be written to it and status bytes read from it. For the cascaded 8259A in Figure 8-14, the two system I/O addresses will be FF08H and FF0AH.

The cascade pins (CAS0, CAS1, and CAS2) from the master are connected to the corresponding pins of the slave. For the master, these pins function as outputs, and for the slave device, they function as inputs. A further difference between the master and the slave is that on the slave the $\overline{SP/EN}$ pin is tied low to let the device know that it is a slave.

Briefly, here is how the master and the slave work when the slave receives an interrupt signal on one of its IR inputs. If that IR input is unmasked on the slave and if that input is a higher priority than any other interrupt level being serviced in the slave, then the slave will send an INT signal to the IR input of the master. If that IR input of the master is unmasked and if that input is a higher priority than any other IR inputs currently being serviced in the master, then the master will send an INT signal to the 8086 INTR input. If the 8086 INTR is enabled, the 8086 will go through its INTR interrupt procedure and send out two $\overline{INTA}$ pulses to both the master and the slave. The slave ignores the first interrupt acknowledge pulse, but when the master receives the first $\overline{INTA}$ pulse, it outputs a 3-bit slave identification number on the CAS0, CAS1, and CAS2 lines. (Each slave in a system is assigned a 3-bit ID as part of its initialization.) Sending the 3-bit ID number enables the slave. When the slave receives the second $\overline{INTA}$ pulse from the 8086, the slave will send the desired interrupt type number to the 8086 on the lower eight data bus lines.

If an interrupt signal is applied directly to one of the IR inputs on the master, the master will send the desired interrupt type to the 8086 when it receives the second $\overline{INTA}$ pulse from the 8086.

Now that we have given you an overview of how an 8259A operates and how 8259As can be cascaded, the initialization command words for the 8259A should make some sense to you.

Initializing an 8259A

Earlier in this chapter, when we showed you how to initialize an 8254, we listed a series of steps you should go through to initialize any programmable device. To refresh your memory of these very important steps, we will work quickly through them again for the 8259A

The first step in initializing any device is to find the system base address for the device from the schematic or from a memory map for the system. In order to have a specific example here, we will use the 8259A shown in Figure 8-14. The base address for the 8259A in this system is FF00H.

The next step is to find the internal addresses for the device. For an 8259A the two internal addresses are selected by a high or a low on the A0 pin. In the circuit in Figure 8-14, the A0 pin is connected to system address line A1, so the internal addresses correspond to 0 and 2.

Next, you add the internal addresses to the base address for the device to get the system address for each internal part of the device. The two system addresses for this 8259A then are FF00H and FF02H.

Next, look at Figure 8-29a for the format of the command words that must be sent to an 8259A to

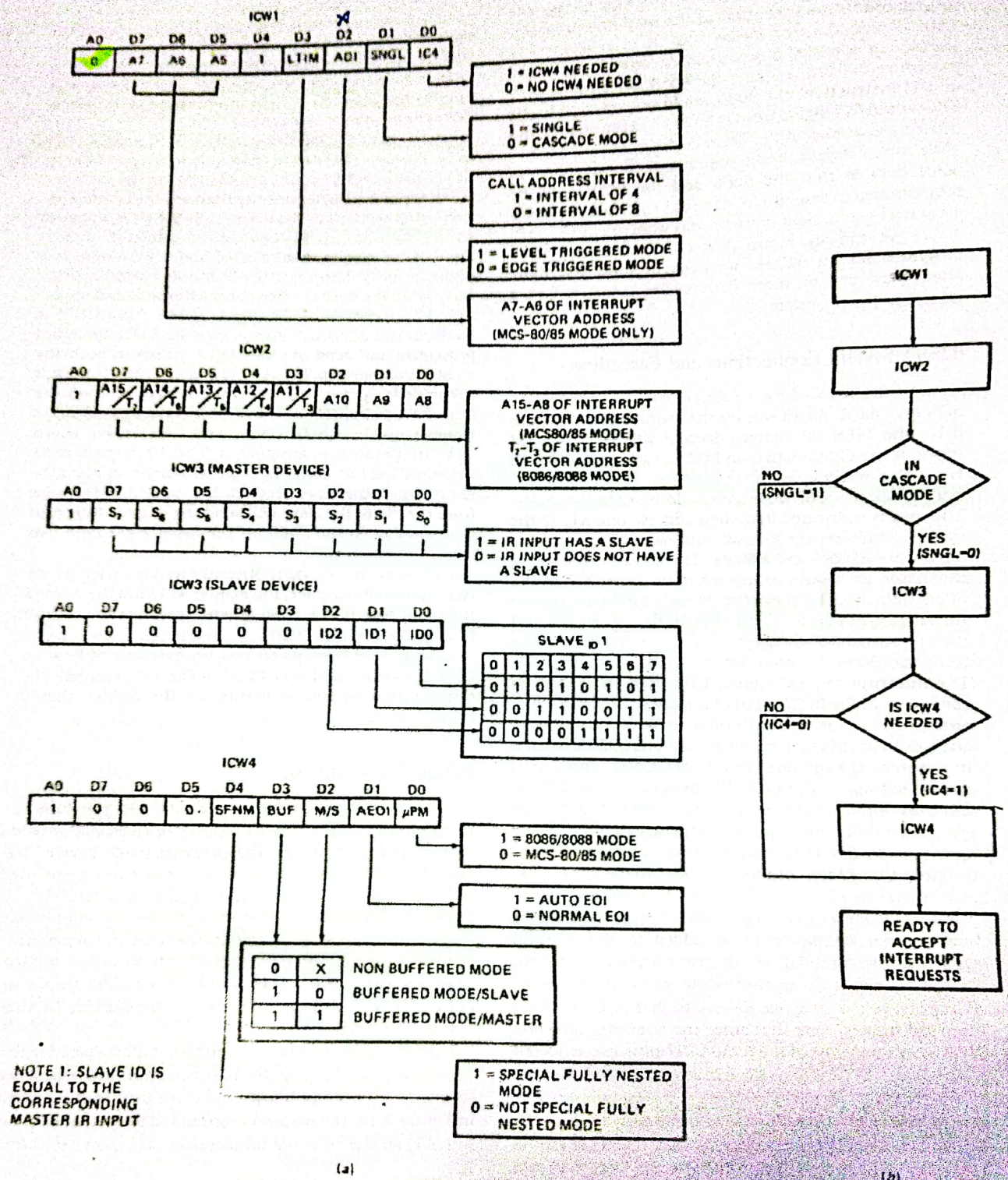


FIGURE 8-29 8259A initialization command word formats and sending order. (a) Formats. (b) Sending order and requirements. *(Intel Corporation)*

initialize it. The sight of all these command words may seem overwhelming at first, but taken one at a time, they are quite straightforward. To help you see which initialization command words (ICWs) are needed for various 8259A applications, Figure 8-29*b* shows this in flowchart form. According to this flowchart, an ICW1 and an ICW2 must be sent to any 8259A in the system. If the system has any slave 8259As (cascade mode), then an ICW3 must be sent to the master, and a different ICW3 must be sent to the slave. If the system is an 8086 or if you want to specify certain special conditions, then you have to send an ICW4 to the master and to each slave. Now let's look at the formats for the different ICWs.

The first thing to notice about the ICW formats in Figure 8-29*a* is that the bit labeled A0 on the left end of each of these is not part of the actual command word. This bit tells you the internal address that the control word must be sent to. The A0 = 0 next to ICW1, for example, tells you that ICW1 must be sent to internal address 0, which for our 8259A corresponds to system address FF00H.

The next step in the initialization procedure is to make up the control words. The least significant bit of ICW1 tells the 8259A whether it needs to look for an ICW4 or not. Since we are using the device in an 8086 system, we need to send ICW4. Therefore we make bit D0 a 1. We only want to use one 8259A for now, so we make bit D1 a 1. When used with an 8086, bit D2 is a don't care, so we make it a 0. Bit D3 is used to specify level-triggered mode or edge-triggered mode. In level-triggered mode, service will be requested whenever a high level is present on an IR input. In edge-triggered mode, a signal on an IR input must go from low to high and stay high until serviced. We usually use the edge-triggered mode so that a signal such as a square wave will not cause multiple interrupts. Making bit D3 a 0 does this. Bit D4 has to be a 1. For operation in an 8086 system, bits D5, D6, and D7 are don't cares, so we make them 0's for simplicity. Therefore, the ICW1 for our example here is 00010011.

In an 8086 system, ICW2 is used to tell the 8259A the type number to send in response to an interrupt signal on the IR0 input. In response to an interrupt signal on some other IR input, the 8259A will automatically add the number of the IR input to this base number and send the result to the 8086 as the type number for that input. Because 8086 interrupt types 0 through 31 are either dedicated or reserved, type 32 (decimal) is the lowest type number available for us to use. If we send the 8259A an ICW2 of 00100000 binary or 32 decimal, the 8259A will send this number as the type to the 8086 in response to an IR0 interrupt. For an interrupt request on the IR1 input, the 8259A will send 00100001 binary or 33 decimal, and for an interrupt request on the IR2 input, the 8259A will send an interrupt type 00100010 binary or 34 decimal. The same pattern continues for interrupt requests on the remaining IR inputs. In any ICW2 you send the 8259A, the lowest three bits must always be 0's because the 8259A automatically supplies these bits to correspond to the number of the IR input.

Since we are not using a slave in this example, we don't need to send an ICW3. If you are using a slave 8259A in a system, you have to send an ICW3 to the master to tell it which IR inputs have slaves. The master has to be told this so that it knows for which IR input signals it has to send out a slave ID number on the CAS0, CAS1, and CAS2 lines. You have to send an ICW3 to a slave 8259A to give it an ID number. The ID number you give a slave is equal to the IR input of the master that its INT output is connected to. When the master sends out an ID number on the CAS lines, the slave will recognize its ID number and output the desired type to the 8086 when it receives an $\overline{\text{INTA}}$ pulse.

For our example here, the only reason we need to send an ICW4 is to let the 8259A know that it is operating in an 8086 system. We do this by making bit D0 of the word a 1. Another interesting bit in this command word is D1, the automatic end-of-interrupt bit. If this bit is set in ICW4, the 8259A will automatically reset the in-service register bit for the interrupt input that is being responded to when the second interrupt-acknowledge pulse is received. The effect of this is that the 8259A will then be able to respond to an interrupt signal on a lower-priority IR input. In other words, a lower-priority interrupt input could then interrupt a higher-priority procedure. Since we don't want automatic end of interrupt, the ICW4 for our example here is 00000001.

In addition to the initialization command words shown in Figure 8-29*a*, the 8259A has a second set of command words called *operation command words*, or OCWs. These are shown in Figure 8-30, p. 238. An OCW1 must be sent to an 8259A to unmask any IR inputs that you want it to respond to. For our example here, let's assume that we want to use only IR2 and IR3. Since a 0 in a bit position of OCW1 unmasks the corresponding IR input, we put 0's in these two bits and 1's in the rest of the bits. Our OCW1 is 11111100**1**1. OCW2 is mainly used to reset a bit in the in-service register. This is usually done at the end of the interrupt-service procedure, but it can be done at any time in the procedure. The effect of resetting the ISR bit for an interrupt level is that once the bit is reset, the 8259A can respond to interrupt signals of lower priority. In small systems we usually use the nonspecific End-of-Interrupt command word. The OCW2 for this is 00100000. When the 8259A receives this OCW, it will automatically reset the in-service register bit for the IR input currently being serviced. If you want to reset a specific ISR bit, you can send the 8259A an OCW2 with 011 in bits D7, D6, and D5, and the number of the ISR bit you want to reset in the lowest 3 bits of the word. You can also use OCW2 to tell the 8259A to rotate the priorities of the IR inputs so that after an IR input is serviced, it drops to the lowest priority. If you are interested, consult the Intel data sheet for more information on this and on the use of OCW3.

Now that we have made up the required ICWs and OCWs, the next step is to write the instructions to send these command words to the 8259A.

Figure 8-31, p. 239–40, shows an 8086 assembly language program which initializes an 8259A and demonstrates many of the concepts of this chapter. You can use this program as a pattern for writing programs

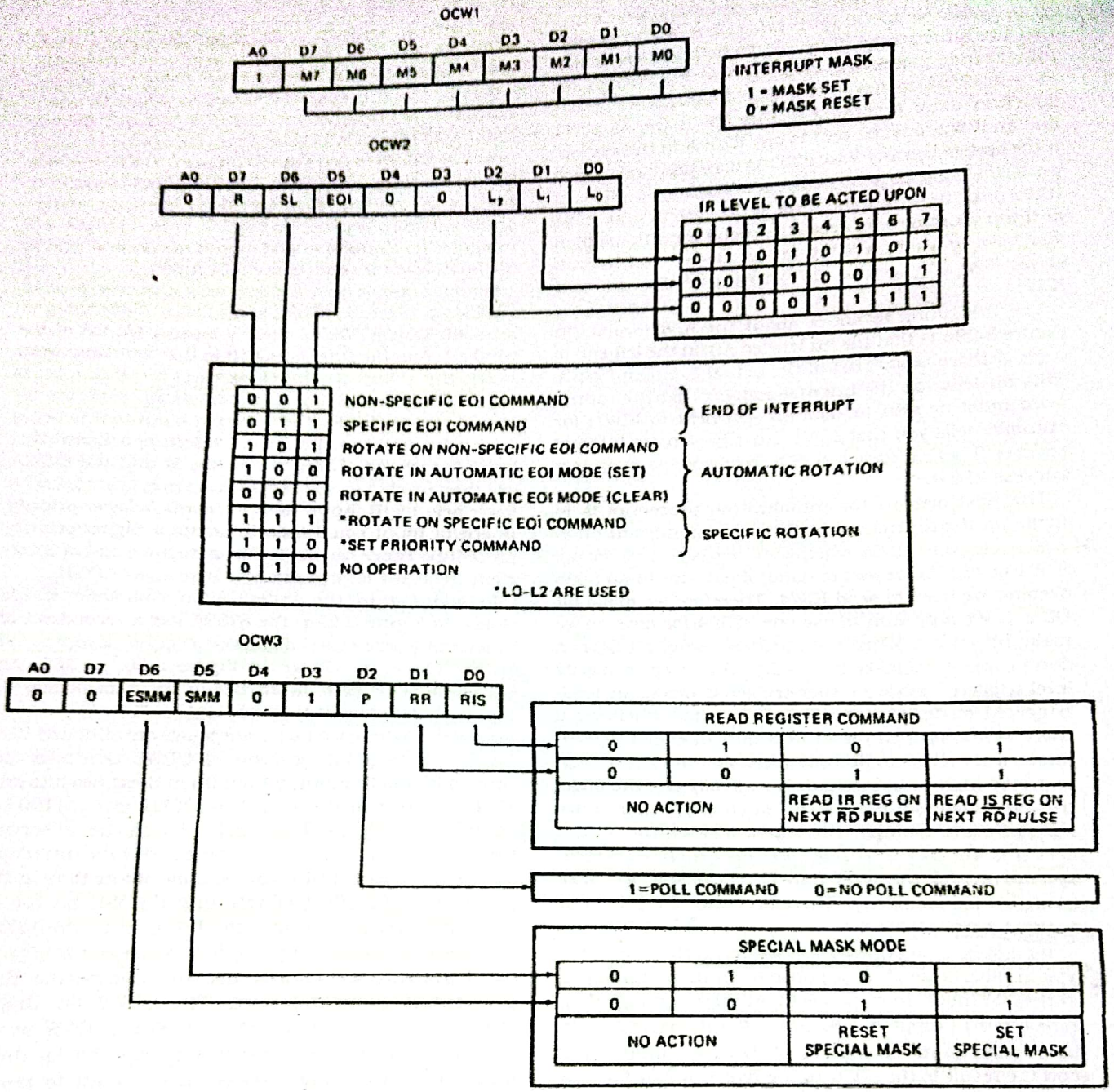


FIGURE 8-30 8259A operational command words. (*Intel Corporation*)

which service one or more interrupts. This program initializes the SDK-86 system in Figure 8-14 for generating a real-time clock of seconds, minutes, and hours from a 1-kHz interrupt signal and for reading ASCII codes from a keyboard on an interrupt basis. This program assumes that the 2.4576-MHz PCLK signal on the board is connected to the CLK input of the 8254 counter 0, the GATE input of the 8254 counter 0 is tied high, and the OUT pin of counter 0 is connected to the IR0 input of the 8259A. The program further assumes that the key-pressed strobe from the ASCII keyboard is connected to the IR2 input of the 8259A.

In the program, we first declare a segment called AINT_TABLE to reserve space for the vectors to the interrupt procedures. The statement TYPE_64 DW 2 DUP(0), for example, sets aside a word space for the offset of the type 64 procedure and a word for the segment base of the procedure. The statement TYPE_65 DW 2 DUP(0) sets aside a word for the offset of the type 65 procedure and a word space for the segment base