

**10. Subroutines**

- Subroutine Instructions 10-1
- Subroutine Parameters 10-2
- Return Values 10-5
- Documentation 10-6
- Program Examples 10-7
- Problems 10-24
- References 10-26

**11. Input/Output**

- I/O and Memory 11-2
- I/O Device Categories 11-3
  - Interfacing Slow Devices 11-3
  - Interfacing Medium-Speed Devices 11-6
  - Interfacing High-Speed Devices 11-9
- Timing Intervals (Delays) 11-10
  - Basic Software Delay 11-10
  - Program Examples 11-11
- Z8000 I/O Instructions 11-14
  - I/O Instruction Examples 11-16
- The Z80 Parallel I/O Device (PIO) 11-43
  - PIO Addresses 11-44
  - PIO Mode Control 11-47
  - Configuring the PIO 11-49
- Complex I/O Devices 11-51
  - Program Example 11-54
  - USART/UART 11-77
- The Z80 Serial I/O Devices (SIO) 11-78
  - SIO Addresses 11-78
  - SIO Registers 11-78
  - Special Features of SIO 11-85
- Standard Interfaces 11-88
- Other Interface Devices 11-89
  - Problems 11-90
- References 11-94

**12. Interrupts**

- Interrupt Enable 12-2
- Non-Maskable Interrupts 12-4
- Maskable Interrupts 12-4
- Interrupt Priority 12-5
- Vectoring 12-6
- Polling 12-6
- Disadvantages of Interrupts 12-6
- The Z8000 Interrupt System 12-8
  - Program Status Area 12-9
  - Z8000 Interrupt Acknowledgment 12-10
  - Interrupt Identifiers 12-17
  - Interrupt Priorities 12-17
  - Return from Interrupt 12-18

The Z8000 Reset	12-18
Z80 PIO Interrupt Logic	12-19
Interrupt Vector Address Register	12-19
Z80 PIO Modes	12-20
Z80 PIO Interrupt Configuration	12-22
Z80 PIO Interrupt Priority	12-23
Z80 SIO Interrupt Logic	12-24
Program Examples	12-27
Problems	12-45
References	12-47

### **13. Large Configurations**

Z8010 Memory Management Unit	13-2
Memory Address Translation	13-2
Segment Descriptor Registers	13-5
Control Registers	13-8
Configuring Z8010 Memory Management Units	13-10
Program Examples	13-13
Segmentation Trap Acknowledgement	13-16
Multiple CPU Configurations	13-21
Program Example	13-22
References	13-25

### **14. Problem Definition and Program Design**

Stages of Software Development	14-2
Problem Definition	14-5
Inputs	14-5
Outputs	14-6
Processing Section	14-6
Error Handling	14-7
Human Interaction	14-8
Examples: Problem Definition	14-9
Review	14-18
Program Design	14-19
Basic Principles	14-19
Examples: Flowchart	14-22
Modular Programming	14-30
Examples: Modular Programming	14-32
Structured Programming	14-35
Examples: Structured Programming	14-41
Review of Structured Programming	14-47
Top-Down Design	14-48
Examples: Top-Down Design	14-49
Review of Top-Down Design	14-53
Chapter Review	14-54
References	14-54

# 12

## *Interrupts*

**Interrupts provide external logic with a means of modifying the sequence in which programs are executed by a microprocessor.**

Without interrupts, external logic has no way of directly controlling program execution sequences. Without interrupts, an external device that requires execution of some specific program must attract the microprocessor's attention by modifying some suitable external status flag. The external logic must then wait until the microprocessor gets around to checking the status flag. This is referred to as "polling." Polling is frequently used in simple microcomputer configurations. But polling will not work when an external device needs the microprocessor's immediate attention. By the time the microprocessor gets around to checking the external device's status flag it may be too late. Data that the external device had ready for the microprocessor may have been overwritten; information the external device needed from the microprocessor may not have arrived in the allotted time; or perhaps the microprocessor has continued to execute some I/O operation long after the external device detected a fatal error and tried to report it. These are three typical examples of situations where external logic must take an active role, forcing the microprocessor to stop whatever it is doing and attend to some more pressing need. Interrupts are the mechanism used by external logic to achieve this goal.

## **INTERRUPT ENABLE**

External logic transmits interrupt requests to the microprocessor via appropriate signals, generally referred to as "Interrupt Request" signals. The microprocessor tests these interrupt request signals once during the execution of every instruction. Some interrupts can be enabled or disabled under program control; others cannot. The microprocessor ignores a disabled interrupt request; it services an enabled interrupt requests as follows:

1. It stops executing the current program.
2. It executes a special program that caters to the needs of the interrupting external logic.
3. It continues executing the current program from the point where the interrupt occurred.

## **Interrupt Acknowledgment**

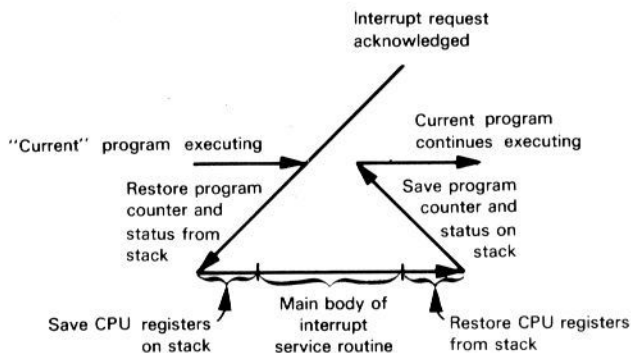
Step 1 above is frequently referred to as the **Interrupt Acknowledge step**. During an interrupt acknowledgment the microprocessor must save the program counter and the Flag and Control Word contents on the stack. The program counter addresses the next sequential instruction; this is the instruction which would have been executed had the interrupt not occurred. This is also the instruction which will be executed as soon as the interrupt has been serviced in Step 3. For Step 2, execution branches to a special program dedicated to a particular interrupt being acknowledged.

## **Interrupt Service Routine**

The program executed on behalf of the acknowledged interrupt is referred to as an **interrupt service routine**. This routine normally begins by saving additional information, information that was not automatically saved during the acknowledgment process. For example, the contents of all CPU registers are frequently pushed onto the stack before any register contents are modified by the body of the interrupt service routine. The interrupt service routine then performs operations required by the acknowledged interrupt.

## Return from Interrupt

Finally, in Step 3, a return from interrupt occurs; this is the exact inverse of the interrupt acknowledgment process. If the interrupt service routine saved any additional information before starting to execute, then it will restore this information before triggering the actual return from interrupt. For example, if the interrupt service routine initially saved the contents of all CPU registers on the stack, then it will restore all these registers' contents from the stack. A "Return-from-Interrupt" instruction is then executed; it restores the program counter and Flag and Control Word contents that were saved during the acknowledgment process, causing the interrupted program to continue execution from the point where it was interrupted. This sequence may be illustrated as follows:



An interrupted program is not affected in any way by the occurrence of the interrupt, except that there is a pause in program execution. The interrupted program appears to enter a state of "suspended animation," at the end of which it continues execution, unaffected by the interrupt process per se.

The repeated Z8000 instructions (such as CPIR) test and acknowledge enabled interrupts between repeated executions; therefore once **an interrupt request** occurs, **if enabled**, the interrupt **will be serviced no more than one instruction execution time later**.

**Why use interrupts?** Interrupts allow events to receive fast microprocessor attention. An alarm, a power failure, the end of a specified time delay, a fast peripheral's need to transmit or receive data, these are all candidates for interrupt logic. The only alternative would be for the microprocessor to execute a program that polled each potential interrupt source. That could be very time consuming and might easily result in important events being missed.

## NON-MASKABLE INTERRUPTS

Some interrupts are so important that they cannot be disabled. These are called **non-maskable interrupts**. A microprocessor will always acknowledge and service a non-maskable interrupt, whatever it is doing when the interrupt request occurs. **Non-maskable interrupts are frequently used to flag a power failure**; a microprocessor can usually execute a few hundred instructions between the time a power failure is detected and the time when insufficient power remains to operate the microprocessor. These instructions can "tidy up" the program, enabling an orderly restart when power is turned on again.

## MASKABLE INTERRUPTS

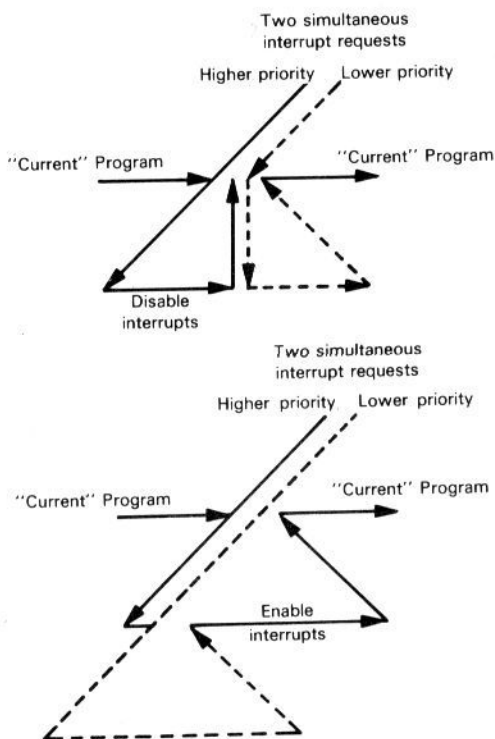
**Interrupts that can be disabled or enabled are referred to as maskable interrupts.** All interrupts encountered during normal program execution should be maskable interrupts.

The interrupt event sequence itself is valuable. Therefore microprocessors frequently allow the event sequence to be triggered by logic internal to the microprocessor. This can occur in one of two ways:

1. **A condition detected during instruction execution may trigger an event sequence that is equivalent to an interrupt request; this is called a software trap.** For example, if an unknown instruction object code is fetched from memory, the microprocessor might respond by executing a trap.
2. Many microprocessors have one or more **instructions that are designed to cause an interrupt sequence to occur.** These are referred to as **software interrupts**.

## INTERRUPT PRIORITY

A microprocessor may receive many different interrupt requests; this being the case, **two or more simultaneous interrupt requests may occur. Which one is to be serviced first? Interrupt priority logic answers this question.** Interrupt priority logic applies only during the interrupt acknowledgment step; it does not apply over the entire period that the interrupt is being serviced. For example, if two interrupt requests occur simultaneously, and the higher priority interrupt request is acknowledged, then the interrupt service routine executed on behalf of the higher priority interrupt must keep the lower priority interrupt disabled. If the higher priority interrupt service routine does not keep the lower priority interrupt disabled, the lower priority interrupt will be acknowledged within the higher priority interrupt service routine. This may be illustrated as follows:



Once the higher priority interrupt's service routine starts executing, only the lower priority interrupt request is pending, as shown in the above illustrations. Being the only interrupt request, it will be acknowledged, even within the higher priority interrupt's service routine, if interrupts are enabled.

## VECTORIZING

Once an interrupt has been acknowledged, there are two ways in which the microprocessor can identify the source of the interrupt. **If the interrupt is vectored then no identification process is required**, since the acknowledgment sequence handles this identification step. Each vectored interrupt has one interrupt service routine, and the interrupt acknowledgment process includes some mechanism for identifying this interrupt service routine.

## POLLING

**In simpler cases two or more devices may share a single interrupt request. Then the microprocessor has to poll the devices sharing the single interrupt request** in order to determine which one (or more) is actually requesting an interrupt. The microprocessor reads Status register contents at the various devices — which is what the microprocessor would do if there were no interrupts. So why interrupt? Because the interrupt triggers the polling sequence. Without the interrupt the microprocessor would have to poll continuously, or on some scheduled basis that might, from time to time, not be frequent enough.

## DISADVANTAGES OF INTERRUPTS

**There are some disadvantages to using interrupts; they include:**

1. Interrupts require additional hardware external to the microprocessor.
2. Once an interrupt has been acknowledged, it must still be serviced by the normal process of executing a program. Interrupts have no inherent speed advantages equivalent to the speed advantages offered by giving external devices direct memory access.
3. Interrupts are difficult to debug since they are random events which occur sporadically; for example, when an I/O operation is complete or following error conditions. The occurrence of I/O completion or errors is unpredictable.
4. Interrupt service routines can spend a lot of time saving and restoring the contents of CPU registers.



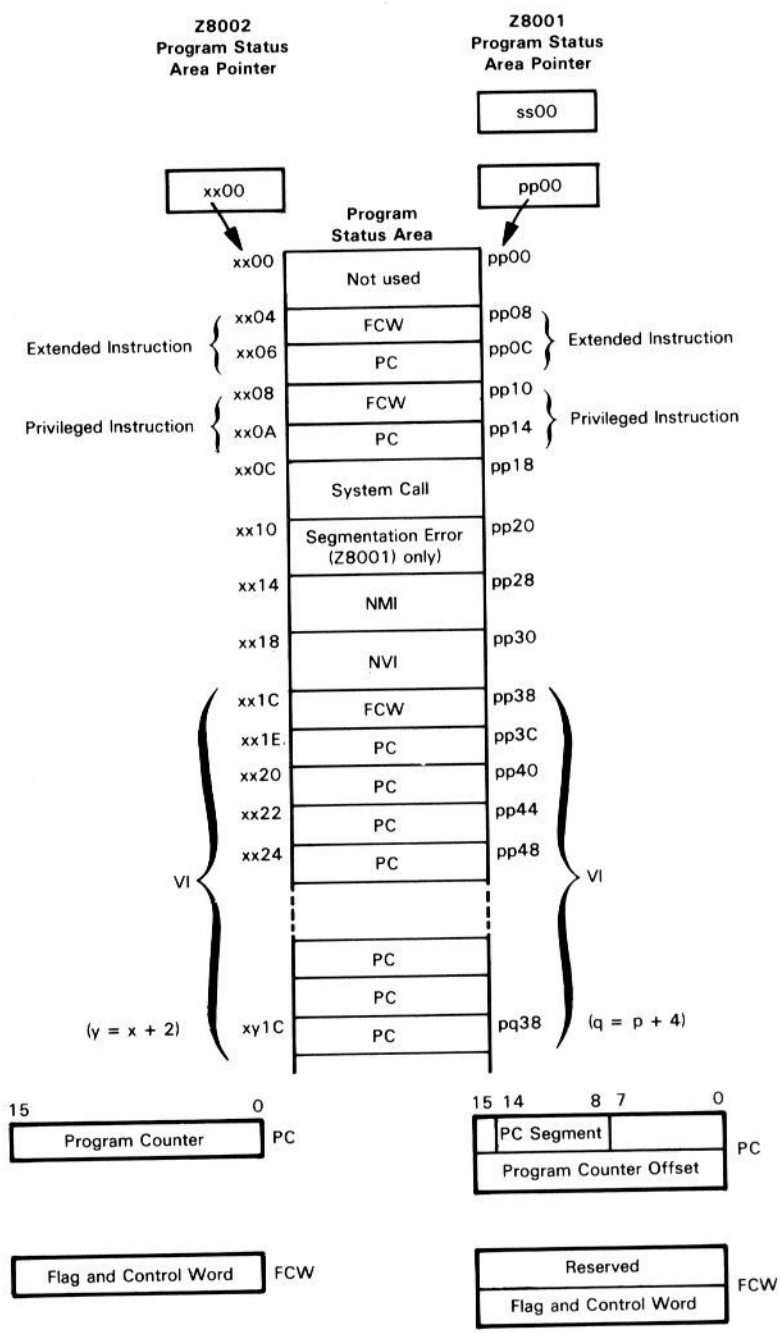


Figure 12-1. Z8000 Program Status Area

## THE Z8000 INTERRUPT SYSTEM

The Z8000 microprocessor has a very complete set of external interrupts and several internal interrupts, all supported by extensive interrupt handling logic. The following interrupts are available:

1. **Software interrupts.** Execution of the System Call instruction (SC) generates a software interrupt.
2. **Software traps.** There are two software traps. An extended instruction trap occurs whenever the Z8000 CPU is presented with the instruction object code of an extended instruction while the EPA bit of the Flag and Control Word is zero. A privileged instruction trap occurs whenever the Z8000 attempts to execute a privileged instruction in normal mode.
3. **Non-maskable interrupt**
4. **Maskable, vectored interrupt**
5. **Maskable, non-vectored interrupt**
6. **Segmentation trap**

Other software traps which are desirable but are not provided by the Z8000 include:

1. Arithmetic errors, such as overflow or division by zero. The Z8000 program must test the flags.
2. Illegal addressing, such as using an odd address for a jump or for a word or long word memory reference; or using a register number other than the defined ones in a multiple-word register operation. The Z8000 jump goes to the next lower even address; the result of the other instructions is undefined.
3. Undefined instruction, that is, a combination of object code bits which specifies a disallowed addressing mode or is not defined as an instruction (e.g., 0C03). The Z8000 result is undefined.

Any possible externally detected condition may be handled by any of the external interrupts 3 to 5 above. Almost 200,000 distinct external interrupts may be easily identified. Interrupt 6 is reserved for memory management errors, and is supported only by the Z8001. It is discussed in Chapter 13.

## PROGRAM STATUS AREA

Z8000 interrupt logic assumes the existence in memory of a **Program Status Area**; this memory area is illustrated in Figure 12-1. The base address in memory for the Program Status Area is held in the Program Status Area Pointer register; you initialize this register using an appropriate LDCTL instruction. Here are appropriate instruction sequences for the Z8002 and the Z8001:

```

! Initialize program status area for the Z8002
LDA      R0,BASEADDR      ! BASEADDR is a name
                                !
LDCTL    PSAP,R0          ! representing
                                !
                                ! the base address
                                !
HALT

! Initialize program status area for the Z8001
LDA      R0,NPSADR        ! NPSADR is a name
                                !
LDCTL    PSAPSEG,R0       ! representing ...
                                !
                                ! ...the segmented base
                                !
                                ! address
                                !
LDCTL    PSAPOFF,R1
HALT

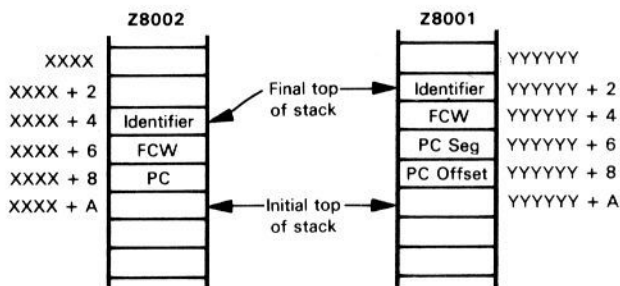
```

The Program Status Area contains information used by interrupt acknowledgment logic; for all interrupts, acknowledgment logic transfers control to an appropriate interrupt service routine using an address taken from the Program Status Area.

## Z8000 INTERRUPT ACKNOWLEDGMENT

When the Z8000 microprocessor acknowledges an interrupt, the following event sequence occurs:

1. The first word of the next instruction's object code is fetched in the usual way, as if no interrupt was being acknowledged. This instruction object code is discarded and the program counter is not incremented. The discarded instruction will be fetched again, after the interrupt has been serviced, becoming the first instruction to be executed following the return from interrupt. This aborted instruction fetch requires three clock periods.
2. Following the aborted instruction fetch an interrupt acknowledge machine cycle is executed. During the interrupt acknowledge machine cycle the interrupting device sends the Z8000 microprocessor a 16-bit data word which the microprocessor interprets as an interrupt identifier. This is an automatic event; no program steps are required to make it happen. Normally the interrupt acknowledge machine cycle lasts for 10 clock periods, but it may sometimes last longer.
3. Current contents of the program counter and the Flag and Control Word are pushed onto the stack. The identifier is then pushed onto the stack. New values are loaded into the Flag and Control Word and the program counter, these new values being taken from the Program Status Area. For the Z8002 and the Z8001, information is pushed onto the stack in the following sequences:



Note carefully that you do not write any instructions to enable the interrupt acknowledgment operations described above. The interrupt service routine entered after Step 3 simply assumes that information will be at the top of the stack, as illustrated. The interrupt service routine entered after Step 3 is identified by an entry address stored in the Program Status Area. At the same time data taken from the Program Status Area is loaded into the Flag and Control Word; this determines status conditions that will apply upon entry into the interrupt service routine.

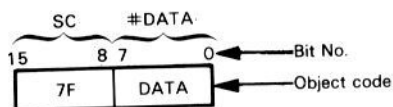
Vectored and non-vectored interrupts are separately enabled and disabled via the VI and NVI bits of the Flag and Control Word. Therefore **the Flag and Control Word, as stored in the Program Status Area, determines whether interrupts will be enabled or disabled when the interrupt service routine begins execution.** Normally an interrupt service routine will begin execution with interrupts disabled. Therefore the VI and NVI bits of Flag and Control Words stored in the Program Status Area will be 0.

**Interrupt service routines do not need to reenale interrupts before returning control to the main program.** When a Return-from-Interrupt instruction is executed, the program counter and the Flag and Control Word values saved on the stack are returned to their respective CPU registers. **The returning Flag and Control Word determines whether interrupts will be enabled or disabled following the Return-from-Interrupt.**

We will now examine the specific acknowledgment sequence associated with each interrupt type supported by the Z8000 microprocessor.

## System Call Interrupt Acknowledgment

A software interrupt is executed in response to a System Call instruction. This instruction has the following source and object code:



The one-word object code becomes the interrupt identifier; therefore, it will be at the top of the stack when the System Call interrupt service routine begins execution. The System Call instruction takes new values for the program counter and the Flag and Control Word from these specific memory locations within the Program Status Area:

Z8002		Z8001	
[FCW]	from xx0C	[FCW]	from sspp1A
[PC]	from xx0E	[PC Segment]	from sspp1C
		[PC Offset]	from sspp1E

(See Figure 12-1.)

Suppose the Program Status Area begins at memory address 0200<sub>16</sub>. The System Call software interrupt service routine is entered at memory location 3C80<sub>16</sub>, in system mode, with all interrupts disabled and status flags reset to 0. You could initialize the Program Status Area for a Z8002 as follows:

```

LDA    R1,%0200      ! Initialize PSAP      !
LDCTL  PSAP,R1
LD     %C(R1),##4000  ! Initialize System Call !
                        ! service           !
LD     %E(R1),##3C80  ! routine FCW and entry !
                        ! address          !
HALT'
```

For a Z8001 the sequence would be:

```

LDA     RR2,<<0>>%0200
LDCTL   PSAPOFF,R2
LDCTL   PSAPSEG,R3      ! PSAP in Segment 0      !
LD      %1A(RR2),##4000  ! FCW                !
LD      %1C(RR2),#0      ! Service Routine Segment !
LD      %1E(RR2),##3C80  ! and entry              !
```

These initialization instructions would be executed once, during system initialization, shortly after your program begins execution. In some microcomputers the Program Status Area will be permanently defined and held in read-only memory.

**The System Call interrupt service routine can recognize up to 256 different System Call instructions.** Remember, the SC instruction has an 8-bit immediate operand that appears in the low-order byte of the SC instruction's object code. The SC instruction's object code is treated as the interrupt identifier, therefore it appears on the top of the stack when the System Call interrupt service routine begins execution. Instructions at the beginning of the interrupt service routine could branch to different procedures for each of the 256 different SC instructions as follows:

```

! System Call interrupt service routine      !
ENTER: LD    R1,@R15      ! Load identifier into R1  !
      CLRB   RH1          ! Convert R1 into a jump  !
                        ! table index          !
      SLL    R1
      LD     R1,JTABL(R1)  ! Load start address from !
                        ! jump table          !
      JP     @R1          ! Jump to selected procedure !
      .
      .
! Start of jump table holding addresses for individual !
! procedures executed on behalf of SC instructions    !
      internal
      JTABL ARRAY [256 WORD]:= [
          ADDR0
          ADDR1
          ADDR2
          .
          .
          .
      ]
```

## Software Trap Interrupt Acknowledgment

The extended instruction and privileged instruction software traps, when acknowledged, give control to different interrupt service routines. Each trap has its own entries in the Program Status Area from which new program counter and Flag and Control Word contents are taken. This portion of the Program Status Area may be illustrated as follows:

Z8002			Z8001		
Illegal instruction:	[FCW]	from xx04	[FCW]	from sspp0A	
	[PC]	from xx06	[PC Segment]	from sspp0C	
Privileged instruction:			[PC Offset]	from sspp0E	
	[FCW]	from xx08	[FCW]	from sspp12	
	[PC]	from xx0A	[PC Segment]	from sspp14	
			[PC Offset]	from sspp16	

(See Figure 12-1.)

The privileged instruction trap occurs if a program attempts to execute a privileged instruction in normal mode. The privileged instruction trap identifier is the first word of the object code for the privileged instruction which caused the software trap to occur. A privileged instruction software trap routine can use the identifier, together with the saved program counter value, to locate the normal mode program which was attempting to execute a privileged instruction. This can be useful when debugging a program that is being developed, or it can be used as part of program security logic in a working system.

An extended instruction trap occurs if a program attempts to execute an extended instruction with the EPA bit of the Flag and Control Word set to zero. The following instruction object codes are defined as extended instruction operation codes:

0Fxx  
4Fxx  
8Exx  
8Fxx

The symbol xx represents any two hexadecimal digits. Following an extended instruction interrupt, one of these object codes will be at the top of the stack. This trap could be used to allow simulation of an instruction not available on the microcomputer.

## Non-Maskable and Non-Vectored Interrupts

When a non-maskable interrupt or a maskable non-vectored interrupt is acknowledged, control transfers to a single interrupt service routine using a unique set of entries in the Program Status Area, which can be illustrated as follows:

	Z8002		Z8001	
Non-maskable interrupt:	[FCW]	from xx14	[FCW]	from sspp2A
	[PC]	from xx16	[PC Segment] [PC Offset]	from sspp2C from sspp2E
Non-vectored interrupt:	[FCW]	from xx18	[FCW]	from sspp32
	[PC]	from xx1A	[PC Segment] [PC Offset]	from sspp34 from sspp36

Both the non-maskable interrupt and the maskable non-vectored interrupt return a 16-bit identifier which is provided by the external device requesting the interrupt. The maskable non-vectored interrupt is usually shared by a number of devices capable of requesting an interrupt. These devices use the identifier to identify themselves. The interrupt service routine executed for the non-vectored interrupt will examine the identifier and use it to identify the requesting device.

When more than one emergency condition can generate a non-maskable interrupt, the microprocessor will use the identifier to determine which condition caused the non-maskable interrupt to occur.

Let us look at non-vectored interrupt program logic in more detail. Suppose the non-vectored interrupt service routine entry point is at memory location NVIR; it must be entered in system mode, with interrupts disabled and all status flags reset to 0. The Program Status Area would be initialized as follows:

```
! Initialize non-vectored interrupt in program status area      !
LD      NPSAP+%18,%4000    ! NPSAP is a name                !
                                ! identifying                  !
LD      NPSAP+%1A,#NVIR    ! the program status              !
                                ! area starting address !
```

Suppose devices requesting a non-vectored interrupt each have a dedicated interrupt service routine. How is the microprocessor to identify the interrupt service routine? A good method would be for the interrupt service routine's entry address to be the sum of the identifier, and the contents of an Index register. Changing the value in the Index register would allow for changes in the memory location of the group of non-vectored interrupt service routines; but the routines would have to be moved as a group. Here are the instructions that would select the correct service routine:

```
! Entry point for all non-vectored interrupts                  !
NVIR: LD      R2,@R15      ! Get identifier                  !
      LDA     R2,NVIX(R2)  ! First routine address + !
                                ! identifier                  !
      JP      @R2
```

**Do not pop the identifier off the stack;** this will be done by the IRET instruction, to be described later.



There is a more flexible way of computing the interrupt service routine's entry address. Let devices requesting a non-vectored interrupt return an index in the identifier. The index should start at 0 for the first device, then increment by 2 for subsequent devices. We will use a jump table to compute the procedure starting address as follows:

```
! Entry point for all non-vectored interrupts !
NVIR: LD R1,@R15 ! Get identifier !
      LD R1,JTABL(R1)
      JP @R1 ! Jump to procedure !
      .
      .
      .
! Start of non-vectored interrupts entry address table !
internal
JTABL ARRAY [256 WORD]:= [
      ADDR0
      ADDR1
      ADDR2
      .
      .
      .
]
```

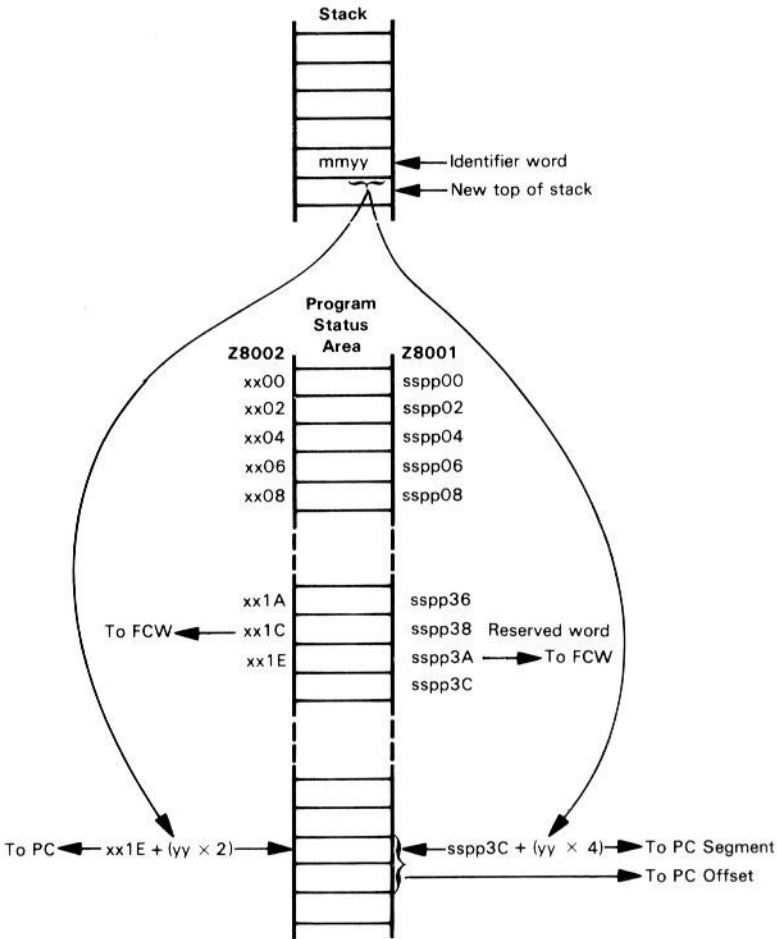
## Vectored Interrupt Acknowledgment

When a vectored interrupt request is acknowledged, the Z8000 microprocessor transfers control directly to one of 256 interrupt service routines. The low-order byte of the identifier word returned by the acknowledged device (yy) is used to perform this vectoring step as follows:

For the Z8002: New PC comes from  $xx1E + (yy \times 2)$   
New FCW comes from  $xx1C$

For the Z8001: New PC comes from  $sspp3C + (yy \times 4)$   
New FCW comes from  $sspp3A$

This may be illustrated as follows:



As illustrated above, all vectored interrupts take the same new Flag and Control Word from the Program Status Area. The new program counter contents, however, are selected using the low-order byte of the identifier as an index into the Program Status Area.

## INTERRUPT IDENTIFIERS

We can summarize identifier interpretations for the different Z8000 interrupts as follows:

Interrupt	Identifier
System Call	zzzz
Extended Instruction Trap	zzzz
Privileged Instruction Trap	zzzz
Non-Maskable Interrupt	xxxx
Non-Vectored Interrupt	xxxx
Vectored Interrupt	xyyy
Segmentation Error Trap	xxxx

The summary given above lists the identifier contents as a sequence of four hexadecimal digits. Letters are used as follows:

The symbol xxxx represents device dependent information; Z8000 interrupt logic does not specify the way in which this data will be interpreted.

The symbol yy is an offset used to generate an address in the program status area.

The symbol zzzz represents the first word of the object code of the instruction causing the interrupt or trap.

## INTERRUPT PRIORITIES

When the Z8000 detects two or more simultaneous interrupt requests, it uses the following priority in order to determine which interrupt to acknowledge:

- Software interrupt or trap (highest priority)
- Non-maskable interrupt
- Segmentation error trap (for the Z8001 only)
- Maskable vectored interrupt
- Maskable non-vectored interrupt (lowest priority)

Note that only one of the software interrupts or traps can exist at any time since each is the product of a different instruction's execution, and only one instruction can be executed at any time. That is why software traps and interrupts are grouped together in the highest priority category.

## RETURN FROM INTERRUPT

**You must use the Return-from-Interrupt instruction (IRET) to return from an interrupt service routine to the interrupted program.** Every interrupt service routine must include one or more Return-from-Interrupt (IRET) instructions. When IRET is executed, this is what happens:

1. The identifier is popped from the stack and discarded.
2. The Flag and Control Word is popped from the stack and loaded into the FCW register; but the new FCW contents do not become effective until the next instruction begins execution.
3. The saved program counter contents are popped from the stack and loaded into the program counter, effecting the actual return from interrupt.

**Do not enable microprocessor interrupts before executing IRET. The data word popped into the FCW register determines whether vectored and/or non-vectored interrupts will be enabled after the Return-from-Interrupt.**

## THE Z8000 RESET

**Z8000 reset logic is very similar to non-maskable interrupt logic.** A Z8000 microcomputer system is reset to initialize all portions of the system before restarting program execution from the lowest level starting point. **Following a reset, prior contents of the program counter and the Flag and Control Word are not saved on the stack. But new values for these registers are taken from the beginning of program memory, as follows:**

Z8002		Z8001	
[FCW]	from 0002	[FCW]	from 0002
[PC]	from 0004	[PC Segment]	from 0004
		[PC Offset]	from 0006

This is very similar to the standard interrupt acknowledgment process.

## Z80 PIO INTERRUPT LOGIC

The Z80 PIO which we described in Chapter 11 has two I/O ports; each I/O port has its own interrupt logic with these component parts:

1. **An 8-bit Interrupt Vector Address register.** The contents of this register are returned to the Z8000 as the low-order byte of the identifier word during an interrupt acknowledgment. The low-order bit of the Interrupt Vector register must be 0. If the PIO is connected to a vectored interrupt this value is used as an index into the Program Status Area.
2. **An Interrupt Enable bit.** This bit can be used by the microprocessor to enable or disable the I/O port's interrupt logic.
3. **An Interrupt Control register.** Contents of this register determine the circumstances that will cause an interrupt request to be generated by the I/O port.
4. **An Interrupt Mask register.** Individual data lines can generate interrupt requests at an I/O port that is operating in control mode. The Interrupt Mask register selects the data lines capable of generating interrupt requests.

### INTERRUPT VECTOR ADDRESS REGISTER

The Interrupt Vector Address register at each I/O port is selected by writing to the I/O Port Control address. An 8-bit data value must be written to the Interrupt Vector Address register; the low-order bit of the data value must be 0 in order to select the Interrupt Vector Address register. (Recall that different locations are accessed via the single I/O port control address; the low-order bits of the data written to the I/O port control address determine which location is selected.)

From our discussion of Z8000 vectored interrupts, recall that the low-order byte of the identifier is left shifted (one bit for the Z8002, or two bits for the Z8001) before being used as an index into the Program Status Area. Since the Z80 PIO always has a 0 in the low-order bit of each I/O port interrupt vector address, alternate entries in the Program Status Area will be used. For example, suppose the two I/O ports of a particular Z80 PIO have the adjacent values  $3C_{16}$  and  $3E_{16}$  in their Interrupt Vector Address registers. A Z8002 will compute Program Status Area addresses as follows:

$$\begin{aligned} xx1E + 3C \times 2 &= xx96 \\ xx1E + 3E \times 2 &= xx9A \end{aligned}$$

The address in  $xx98_{16}$  has been skipped.

## Z80 PIO MODES

The Z80 PIO generates interrupt requests in different ways for different modes. Z80 PIO modes are described in Chapter 11.

### Mode 0 Interrupts

In Mode 0 either I/O port generates an interrupt request when external logic acknowledges having received data output.

### Mode 1 Interrupts

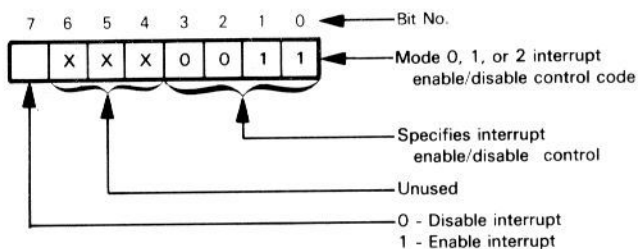
In Mode 1 either I/O port generates an interrupt request when external logic transmits a new data word to the I/O port.

### Mode 2 Interrupts

In Mode 2 an I/O Port A interrupt request is generated when external logic acknowledges having received data output via I/O Port A; an I/O Port B interrupt request is generated when external logic transmits input data to I/O Port A.

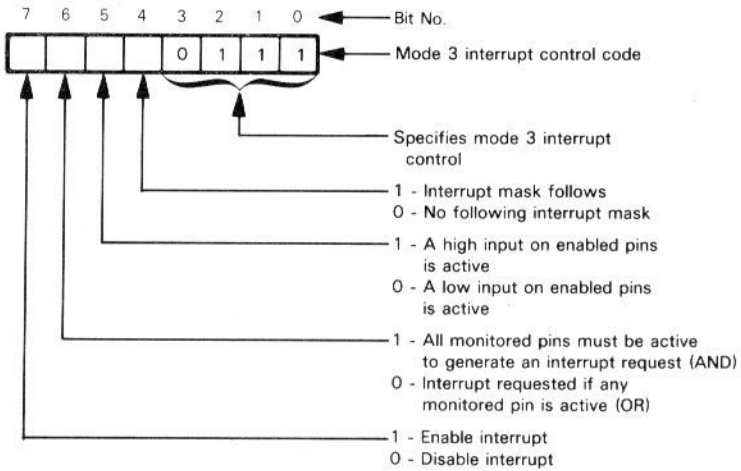
## Interrupt Enable/Disable Control Code

You must enable or disable interrupt logic associated with an I/O port; each I/O port has its own interrupt enable/disable logic. The interrupt control code is output to the control address of each I/O port. Control code bits are interpreted as follows:

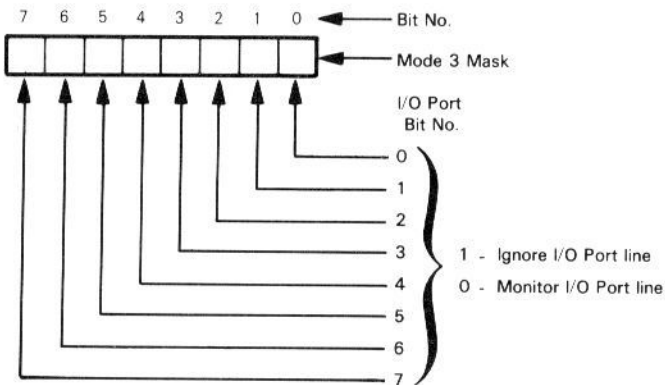


## Mode 3 Interrupts

When an I/O port is operating in Mode 3, you must write a special Mode 3 control code, optionally followed by a mask code. These two outputs are directed to the control address for the I/O port. The Mode 3 interrupt control code is interpreted as follows:



If bit 4 of the Mode 3 control code is 1, then the next byte written to the control address is interpreted as a mask identifying the data lines of the Mode 3 I/O port which are to be monitored for interrupts. Mask bits are interpreted as follows:



## Z80 PIO INTERRUPT CONFIGURATION

We will now look at some Z80 PIO configuration programming examples.

Suppose I/O Port B is to be initialized as an output port (Mode 0), with interrupts enabled and an interrupt vector of  $80_{16}$ ; initialization requires these instructions:

```
LDB    RLO, #80F          ! Make Port B output      !
OUTB   PIOADR+7, RH0      ! Port B control address !
LD     R0, #8083          !
OUTB   PIOADR+7, RH0      ! Vector Address 80      !
OUTB   PIOADR+7, RLO      ! Enable interrupts      !
```

An interrupt request will be generated when external logic acknowledges data received from I/O Port B.

If I/O Port A is bidirectional (Mode 2) then I/O Port B must be operated in control mode (Mode 3). I/O Port A has interrupts enabled and an interrupt vector address of  $40_{16}$ . I/O Port B has interrupts enabled; an interrupt occurs when lines 1, 3, and 5 are all high. The Port B interrupt vector address is  $42_{16}$ . Initialization requires these instructions:

```
LDB    RLO, #8F          ! Make Port A bidirectional !
OUTB   PIOADR+5, RLO      ! Port A control address !
LD     R0, #4083          !
OUTB   PIOADR+5, RH0      ! Vector address 40      !
OUTB   PIOADR+5, RLO      ! Enable interrupts      !
LD     R0, #CF2A          !
OUTB   PIOADR+7, RH0      ! Make Port B control    !
OUTB   PIOADR+7, RH0      ! with pins 1, 3, and 5  !
                          ! inputs                          !
LD     R0, #F72A          !
OUTB   PIOADR+7, RH0      ! Enable interrupt when all !
                          ! high, mask follows      !
OUTB   PIOADR+7, RLO      ! Monitor lines 1, 3, and 5 !
```

Try rewriting this instruction sequence using OUTIRB. Does it save execution time or memory space?



## Z80 PIO INTERRUPT PRIORITY

If interrupt requests occur simultaneously at I/O Ports A and B, then the I/O Port A interrupt request has priority.

When two or more Z80 PIO devices are present in a single microcomputer system, daisy chained interrupt priority logic is frequently employed. The Z80 PIO design provides for this, but how it is implemented is a function of the way in which Z80 PIO devices have been connected in your particular microcomputer configuration. You should therefore check the documentation provided with your microcomputer to determine whether the information which follows applies to your case.

When any Z80 PIO I/O port's interrupt request is acknowledged, the Z80 PIO device outputs a signal which permanently disables interrupt requests from all lower priority Z80 PIO devices. Lower priority Z80 PIO device interrupt requests remain disabled until the higher priority Z80 PIO device removes its disable signal. This occurs when the Z80 PIO device receives an interrupt acknowledgment. Z80 PIO devices were originally designed as support parts for the Z80 microprocessor. Therefore Z80 PIO devices assume that they have received an interrupt acknowledgment when they detect a Z80 Return-from-Interrupt instruction (RETI) being executed. Unfortunately the Z8000 microprocessor has no equivalent logic, nor is it possible to simulate a Z80 RETI instruction using Z8000 program steps only. Some additional external hardware is required. Depending on the nature of this hardware, you will have to execute one or more Z8000 instructions in order to provide the Z80 PIO device with its interrupt acknowledgment. For example, the standard Z8000 development system provided by Zilog requires seven instructions to be executed; this sequence causes external hardware to provide the Z80 PIO device with its interrupt acknowledgment. The way in which you create an interrupt acknowledgment for Z80 PIO devices will depend on the way your microcomputer system has been designed, and whether your microcomputer system does indeed use Z80 PIO devices for its parallel I/O ports.

## Z80 SIO INTERRUPT LOGIC

The Z80 SIO device was introduced in Chapter 11. **The Z80 SIO device has very complex interrupt logic.** There are forty different ways in which interrupt requests can be generated by the Z80 SIO device; interrupt requests which are active at any time depend on operating options which you specify by writing appropriately to seven Write registers. You subsequently monitor operations by reading from the device's three Read registers. But the complexities of the Z80 SIO device all pertain to the different types of serial I/O protocols which are supported; they have no impact whatsoever on programming techniques for using the device. Therefore **in the discussion that follows we will examine only the routine interrupt servicing steps for the Z80 SIO device. A detailed description of this device can be found elsewhere.**<sup>6</sup>

**Transmitter interrupts are enabled by writing 1 to bit 1 of Write Register 1.** Each of the two channels has its own Write Register 1; transmitter interrupts must be enabled or disabled separately for each channel.

**Each Z80 SIO channel's Read and Write registers are written via the channel's control port.** Read Register 0 and Write Register 0 are normally selected. However, if you write any other register number to the three low-order bits of Write Register 0, then the next access will select the register identified in the three low-order bits of Write Register 0; the three low-order bits of Write Register 0 are then reset to 0. While this may sound complicated, from the programmer's viewpoint it is quite straightforward. Just assume that the first time you write to a Z80 SIO control channel register, or read from it, you will access Write Register 0 or Read Register 0. To access any other register, first write the other register's number, then execute another write or read. For example, the following instruction writes data to Write Register 0 of Channel A:

```
LDB    RL0,#DATA
OUTB   SIOADR+5,RL0    ! Channel A control address !
```

To enable transmitter interrupts by writing to Write Register 1, as described previously, execute these instructions:

```
LDB    RL0,#1
OUTB   SIOADR+5,RL0    ! Select Channel A Write    !
                                ! register 1                !

LDB    RL0,#2
OUTB   SIOADR+5,RL0    ! Write to Channel A Write    !
                                ! register 1                !

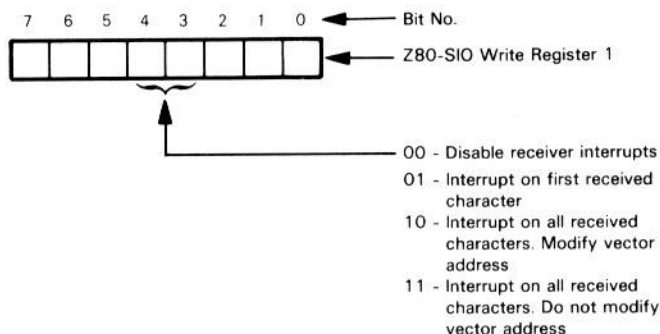
! The next write to SIOADR+5 will select Write Register 0 again !
```

If you wish to read from Read Register 2 execute these two instructions:

```
LDB    RL0,#2
OUTB   SIOADR+5,RL0    ! Select Channel A Read    !
                                ! Register 2                !

INB    RL0,SIOADR+5    ! Input from Read Register 2 !
```

Receiver interrupts are enabled and disabled by writing appropriately to bits 3 and 4 of Write Register 1 at each channel. In addition to enabling and disabling interrupts these two bits specify the receive condition which generates an interrupt request as follows:



The Z80 SIO device has an 8-bit interrupt vector address which is returned as the low-order byte of the identifier following an interrupt acknowledge. The Z80 SIO device does not return any high-order identifier byte. The high-order identifier byte will be defined by logic specific to your microcomputer system, or it will be undefined. One interrupt vector address is shared by both channels of the Z80 SIO device. This address is held in Write Register 2 of Channel B. There is no Channel A Write Register 2.

If you use non-vectored interrupts to service a single Z80 SIO device, instructions at the beginning of the interrupt service routine can read the identifier from the top of the stack to determine what caused the interrupt.

If there is more than one Z80 SIO device in your configuration, and the priority logic is not connected, then you will have to determine which device requested the interrupt. You can poll Z80 SIO devices by reading Read Register 0 of Channel A at each Z80 SIO device. Bit 1 of this register will be set to 1 if there is an active interrupt request anywhere in the device. Subsequently, you should read the Interrupt Vector Address register contents directly from the Z80 SIO device by reading the contents of Read Register 2 of Channel B. Do not read the identifier from the top of the stack. Two or more Z80 SIO devices may have requested interrupts simultaneously. Without performing further analysis you will not know which device's vector address is on top of the stack.

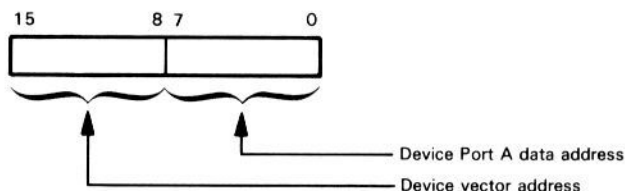
The following instruction sequence polls four Z80 SIO devices with adjacent I/O port addresses. On detecting a device with an active interrupt request, this program fetches the vector address, returning the device number in RH1 and the vector address in RL1.

```

                LDA    R2,SIOADR4+5    ! Load largest Port A control !
                ! address !
LOOP:          LDB    RH1,#4            ! Load device number (4 to 1) !
                INB    RL0,@R2         ! Poll device !
                BITB   RL0,#1          ! Test interrupt request bit !
                JR     NZ,FOUND         ! Active interrupt found !
                DEC    R2,#8           ! Decrement address for next !
                ! device !
                DBJNZ  RH1,LOOP        ! Decrement device number. !
                ! Return if not zero !
                CLRB   RL1            ! Therefore no SIO interrupt !
                ! requests !
FOUND:         HALT
                INC    R2,#2           ! Interrupt found. Select !
                ! Port B !
                LDB    RL1,#2          ! Select Register 2 !
                OUTB   @R2,RL1        ! !
                INB    RL1,@R2        ! Interrupt Vector Address !
                HALT

```

Try rewriting this program to poll a variable number of Z80 SIO devices, where the number of devices is represented by a name. Return a list of all Z80 SIO devices with active interrupt requests; the list should consist of 16-bit words having the following format:



Push these words onto a data stack addressed by Register R1. Return the number of stack entries in Register RL0.

When both channels of a Z80 SIO device request interrupts simultaneously, Channel A interrupts have priority over Channel B interrupts.

We will not discuss programming aspects of Z80 SIO interrupt service routines. The programming logic itself is very simple; you have to clear interrupt conditions by writing to appropriate registers and you must read or transmit data in a routine fashion. The hard part is understanding the serial I/O protocol being supported by the Z80 SIO device.

## PROGRAM EXAMPLES

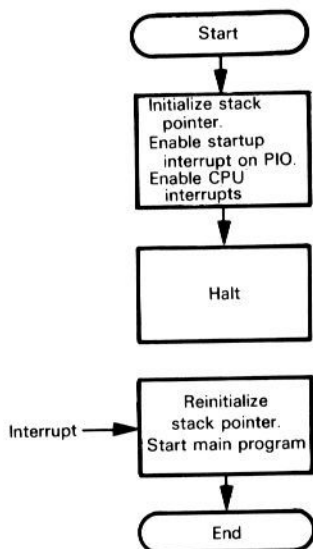
### 12-1. A Startup Interrupt

**Purpose:** Following a reset, the program performs initialization steps, then waits for a PIO interrupt to occur before starting actual operations.

Following a reset, the processor fetches a program status from low memory and transfers program control to the specified address. On the Z8001 the status occupies four words beginning at segment 0, offset 0. (The first word of the status is not used by current production chips.) On the Z8002, the status occupies two words beginning at address 2.

When a reset occurs, the program should initialize the stack pointer, enable the startup interrupt, then execute a HALT instruction. Remember that a reset disables microprocessor interrupts and power-on disables all PIO interrupts. In flowchart 12-1, hardware inputs an interrupt request to initiate the startup.

Flowchart 12-1:



**Program 12-1:**

```

                                internal
                                $abs      2
0002 4000 RESFCW WORD := %4000      ! RESET program status: Z8002 !
0004 0006 RESPC  WORD := RESET     ! FCW: interrupts disabled !
                                      ! PC: the reset program !

                                internal
                                pgm_12_1 procedure
                                entry
                                $abs      6

0006 760F RESET: LDA      R15,%8000      ! Initialize stack pointer !
0008 8000                      LD        R0,%4F83
000A 2100                      LD        R0,%4F83
000C 4F83                      LD        R0,%4F83
000E 3A06 OUTB      PIOADR+5,RH0        ! PIO port A input mode !
0010 FF25                      OUTB      PIOADR+5,RL0
0012 3A86                      OUTB      PIOADR+5,RL0      ! Interrupts enabled !
0014 FF25                      EI
0016 7C04                      EI        ! Enable processor interrupts !
0018 7A00                      HALT      ! Wait for first interrupt !

                                ! Interrupt service routine: !

                                $abs      INTRP

4200 760F LDA      R15,%8000      ! Reinitialize stack !
4202 8000                      LD        R0,%4F83
4204 5E08 JP        START        ! Enter main program !
4206 4600

4208                      end      pgm_12_1

```

First we enable the startup-of-PIO interrupt, then we execute the EI instruction to enable CPU interrupts. Remember, reset automatically disables Z8000 CPU interrupts.

The program executed following the reset must initialize the stack pointer, since the program counter contents will be pushed onto the stack with the next interrupt. We show the service routine reinitializing the stack pointer after the next interrupt, and before the actual startup routine is executed. Instead we could increment the stack pointer by 4 to bypass the current stack contents.

The exact location of the interrupt service routine varies with the microcomputer. If your microcomputer has no monitor, you can start the interrupt service routine wherever you wish. Of course, you should place the routine so that it does not interfere with fixed addresses or with other programs.

Most microcomputers do have a monitor. A monitor is a program which provides the lowest level, most primitive interface between the microcomputer and a user. Often the monitor is held in read-only memory; its execution is triggered by a reset. In our example, the startup PIO interrupt will probably reexecute the monitor. The monitor is a conversational program, with commands that let you execute other programs, then return to the monitor. This may be done by CALLing the program, or an interrupt may be used to exit from the monitor to the program you wish to execute. This requires that you enter an execution address (probably from a keyboard), then trigger an interrupt request, possibly by depressing a control key. There are many ways in which this logic could be implemented. The execution address could be held in an Index register, for example, in which case the interrupt service routine might begin with the instruction:

```

JP      @Rn      ! Enter identified program !

```

You might want to exit the monitor to execute a user-defined program, as described above, or to execute a program out of a permanent library. Programs in such a library perform a variety of routine tasks, such as editing, assembling, compiling, etc. We could hold execution addresses for library programs in an address list. Suppose RL0 holds 0 if a library program is to be executed, or 1 if a user program is to be executed. R1 holds the library program number, or the user program execution address. This is how the monitor might select the program to be executed:

	TESTB	RL0	! User program?	!
	JR	NZ, PROG	! Yes	!
	LD	R1, LIBADR(R1)	! No. Get library address	!
PROG:	JP	@R1		
LIBADR:			! Library program execution !	
			! addresses stored here	!

## 12-2. A Keyboard Interrupt

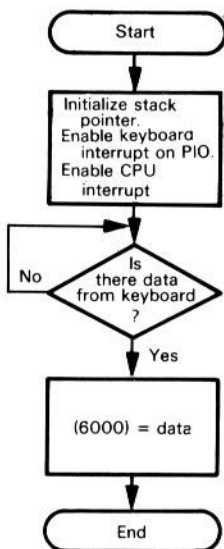
**Purpose:** The computer waits for a keyboard interrupt and places the data from the keyboard into memory location 6000.

**Sample Problem:**

Keyboard data = 06

Result: (6000) = 06

**Flowchart 12-2:**



There is nothing in the main program that pertains specifically to the keyboard interrupt; it is shown to illustrate the interrupt process. In this instance, the JR HERE endless loop will be interrupted by the keyboard interrupt service routine, among others.

**Instead of returning to JR HERE, you might want to return elsewhere.** To return to the instruction following JR HERE increment the return address by 2, since JR HERE has two object code bytes. This single instruction appearing before IRET will suffice (for the Z8002):

```
INC    4(R15),#2      ! Increment return PC !
```

What single instruction is needed for the Z8001? What instructions would you add to give a completely new address?

Since the Z8000 does not automatically save registers' contents during an interrupt acknowledgment, you can use them to pass parameters and results between the main program and the interrupt service routine. This is, however, a dangerous practice that should be avoided in all but the most trivial programs. In most applications, the processor is using its registers during normal program execution; having the interrupt service routines randomly change the contents of one or more registers would surely cause



## Program 12-2a:

```

internal
  pgm_12_2a procedure
  entry
    $abs      IBASE

! Main Program !

4600 760F          LDA      R15,%8000      ! Initialize stack !
4602 8000
4604 2101          LD       R1,##4F83
4606 4F83
4608 3A16          OUTB     PIOADR+5,RH1    ! PIO port A input mode !
460A FF25
460C 3A96          OUTB     PIOADR+5,RL1    ! Enable interrupts !
460E FF25
4610 7C04          EI       ! Enable processor interrupts !
4612 E8FF  HERE:    JR      HERE           ! Loop until PIO interrupts !

! Keyboard interrupt service routine !

4614 93F0          PUSH     @R15,R0        ! Save a register !
4616 3A84          INB      RL0,PIOADR+1    ! Read character !
4618 FF21
461A 6E08          LDB      %6000,RL0      ! Store into memory !
461C 6000

! (PIO interrupt acknowledgement sequence here) !

461E 97F0          POP      R0,@R15
4620 7B00          IRET                     ! Enable interrupts upon exit !

4622              end      pgm_12_2a

```

havoc. In general, no interrupt service routine should ever alter any register unless that register's contents are saved prior to its alteration and restored at the completion of the routine.

Note that you must not explicitly reenale microprocessor interrupts at the end of the service routine. The IRET instruction restores the interrupt enable status in effect at the time of the interrupt. The special PIO interrupt acknowledgment sequence deactivates the interrupt signal so that the same interrupt is not serviced again.

An alternative approach, shown in program 12-2b, would have the main program initialize a keyboard buffer. The interrupt service routine will store keyboard input in the buffer until a carriage return character is detected or the buffer fills up.

Can you see how the INIB and CPB instructions work? Try rewriting the program so that an INB instruction loads data from Port A to RL0. Do you save object code or execution time?

When the processor receives a carriage return, it leaves the interrupt system disabled while it handles the line using a program entered at LPROC.

An alternative approach would be to fill another buffer while LPROC handles the first one; this approach is called double buffering.

In a real application, the CPU could perform other tasks between interrupts. It could, for instance, edit, move, or transmit a line from one buffer while the interrupt was filling another buffer.

## Program 12-2b:

```

internal
  pgm_12_2b procedure
  entry
    $abs      IBASE

    ! Main Program !

4600 760F          LDA      R15,%8000          ! Initialize stack !
4602 8000
4604 2101          LD       R1,#%4F83
4606 4F83
4608 3A16          OUTB     PIOADR+5,RH1       ! PIO port A input mode !
460A FF25
460C 3A96          OUTB     PIOADR+5,RL1       ! Enable interrupts !
460E FF25

    ! Complete line returns to next instruction. Main !
    ! program falls through first time !
LININIT:LD        %6000,#KEYBUF              ! Buffer starting address !

4610 4D05          LD       %6002,#SIZEOF KEYBUF ! Length of buffer !
4612 6000
4614 6000
4616 4D05          LD       %6002,#SIZEOF KEYBUF ! Length of buffer !
4618 6002
461A 0050
461C 7C04          EI
461E E8FF          HERE:   JR      HERE          ! Enable processor interrupts !
                                         ! Loop until PIO interrupts !

    ! Keyboard interrupt service routine !

4620 93F1          PUSH     @R15,R1            ! Save R1, R2 and R3 !
4622 91F2          PUSHL    @R15,RR2
4624 6101          LD       R1,%6000          ! Next free byte in buffer !
4626 6000
4628 6102          LD       R2,%6002          ! Remaining space in buffer !
462A 6002
462C 2103          LD       R3,#PIOADR+1      ! Port address !
462E FF21
4630 3A30          INIB     @R1,@R3,R2        ! Next byte into buffer !
4632 0218

    ! (PIO interrupt acknowledgement sequence here) !

4634 8D24          TEST     R2                ! End of buffer? !
4636 E60B          JR      EQ,LPROC
4638 4C11          CPB      -1(R1),#CR        ! Was last character CR? !
463A FFFF
463C 0D0D
463E E607          JR      EQ,LPROC          ! Yes !
4640 6F01          LD       %6000,R1          ! No, update pointer !
4642 6000
4644 6F02          LD       %6002,R2
4646 6002
4648 95F2          POPL     RR2,@R15          ! Restore registers !
464A 97F1          POP      R1,@R15
464C 7B00          IRET
                                         ! Re-enable interrupts on exit !

LPROC: ! (Process complete line) !

464E 95F2          POPL     RR2,@R15
4650 97F1          POP      R1,@R15
4652 A9F5          INC      R15,#6            ! Remove status (PC etc)... !
                                         ! ...pushed by interrupt !
4654 5E08          JP      LININIT          ! Set up for next line !
4656 4610

4658              end          pgm_12_2b

```

## 12-3. A Printer Interrupt

**Purpose:** The computer waits for a printer interrupt and sends a data byte from memory location 6000 to the printer.

**Sample Problem:**

(6000) = 51<sub>16</sub>

Result: Printer receives a 51<sub>16</sub> (ASCII Q) when it is ready.

**Program 12-3a:**

```

internal
  pgm_12_3a procedure
  entry
    $abs      IBASE

! Main program !

4600 760F          LDA      R15,%8000          ! Set up stack !
4602 8000
4604 2100          LD       R0,%0F83
4606 0F83
4608 3A06          OUTB     PIOADR+5,RH0       ! PIO Port A output !
460A FF25
460C 3A86          OUTB     PIOADR+5,RL0       ! Interrupts enabled !
460E FF25
4610 7C04          EI
4612 E8FF          HERE:   JR      HERE

! Interrupt service routine !

4614 93F0          PUSH     @R15,R0
4616 6008          LDB      RL0,%6000          ! Get data to be sent !
4618 6000
461A 3A86          OUTB     PIOADR+1,RL0       ! Send byte to printer !
461C FF21

! (PIO interrupt acknowledgement steps here) !

461E 97F0          POP      R0,@R15
4620 7B00          IRET

4622              end      pgm_12_3a

```

Here, as with the keyboard, you could have the printer continue to interrupt until it transferred an entire line of text. The main program and the service routine are shown in program 12-3b.

**Program 12-3b:**

```

internal
  pgm_12_3b procedure
  entry
    $abs      IBASE

! Main Program !

4600 760F      LDA      R15,%8000      ! Initialize stack !
4602 8000
4604 2101      LD       R1,##0F83
4606 0F83
4608 3A16      OUTB     PIOADR+5,RH1    ! PIO port A output mode !
460A FF25
460C 3A96      OUTB     PIOADR+5,RL1    ! Enable interrupts !
460E FF25

! Complete line returns to next instruction. Main !
! program falls through first time !

4610 4D05      LININIT:LD      %6000,#LPBUF    ! Buffer starting address !
4612 6000
4614 6000
4616 4D05      LD       %6002,#SIZEOF LPBUF ! Length of buffer !
4618 6002
461A 0050
461C 7C04      EI              ! Enable processor interrupts !
461E E8FF      HERE:  JR      HERE          ! Loop until PIO interrupts !

! Printer interrupt service routine !

4620 91F0      PUSHL     @R15,RR0        ! Save R0 and R1 !
4622 6101      LD       R1,%6000        ! Next free byte in buffer !
4624 6000
4626 2018      LDB      RL0,@R1         ! Get next character !
4628 3A86      OUTB     PIOADR+1,RL0     ! Send to printer !
462A FF21

! (PIO interrupt acknowledgement sequence here) !

462C 0A08      CPB      RL0,#CR         ! Watch for carriage return !
462E 0D0D
4630 E607      JR       EQ,LPROC
4632 6900      INC      %6000          ! Advance to next character !
4634 6000
4636 6B00      DEC      %6002          ! Adjust length !
4638 6002
463A E602      JR       EQ,LPROC
463C 95F0      POPL     RR0,@R15        ! Watch for end of buffer !
463E 7B00      IRET

LPROC: ! (End of line processing takes place here) !

4640 95F0      POPL     RR0,@R15
4642 A9F5      INC      R15,#6          ! Remove status (PC etc.)... !
! ...pushed by interrupt !
4644 E8E5      JR       LININIT

4646          end      pgm_12_3b

```

Note that this program is simpler than the equivalent keyboard program. It is also shorter and will execute faster. So beware of complex instructions.

Comments following the equivalent keyboard program also apply to this printer program.

## 12-4. A Real-Time Clock Interrupt

**Purpose:** The computer waits for an interrupt from a real-time clock.

A real-time clock simply provides a regular series of pulses. The interval between the pulses can be used as a time reference. Real-time clock interrupts can be counted to give any multiple of the basic time interval. A real-time clock can be produced by dividing down the CPU clock, by using a separate timer or a programmable timer like the Z80-CTC or Z8036 CIO, or by using external sources such as the AC line frequency.

Note the tradeoffs involved in determining the frequency of the real-time clock. A high frequency (say 10 kHz) allows the creation of a wide range of time intervals of high accuracy. On the other hand, the overhead involved in counting real-time clock interrupts may be considerable. The choice of frequency depends on the precision and timing requirements of your application. The clock may, of course, consist partly of hardware; a counter may count high frequency pulses and interrupt the processor only occasionally. A program will have to read the counter to measure time to high accuracy.

Synchronizing operations with the real-time clock is not straightforward. Clearly, the time interval will not be precise if the CPU starts the measurement randomly during a clock period, rather than exactly at the beginning of a clock period. Some ways to synchronize operations are:

1. Start the CPU and clock together. A reset or a startup interrupt can start the clock as well as the CPU.
2. Allow the CPU to start and stop the clock under program control.
3. Use a high-frequency clock so that an error of less than one clock period will be small.
4. Line up the clock (by waiting for a change in some signal or an interrupt) before starting the measurement.

A real-time clock interrupt should have very high priority, since the precision of the timing intervals will be affected by any delay in servicing the interrupt. Usually the real-time clock is the highest priority interrupt, except for power failure. The clock interrupt service routine is generally kept extremely short so that it does not interfere with other CPU activities.

**a. Wait for Real-Time Clock****Program 12-4a:**

```

                internal
                pgm_12_4a procedure
                entry
                $abs      IBASE

4600 760F          LDA      R15,%8000
4602 8000
4604 2100          LD       R0,%4F83
4606 4F83
4608 3A06          OUTB     PIOADR+5,RH0      ! PIO Port A input mode !
460A FF25
460C 3A86          OUTB     PIOADR+5,RL0      ! Interrupts enabled !
460E FF25
4610 7C04          EI
4612 E8FF  HERE:      JR      HERE

                ! Real time clock interrupt service routine !

4614 7A00          HALT                                ! Clock interrupt !

4616              end      pgm_12_4a

```

Program 12-4a reduces the real-time clock concept to its most trivial case. A real-time clock interrupt occurs on the rising edge of a PIO STROBE signal at Port A. The main program executes the JR HERE instruction waiting for the real-time clock interrupt. This interrupt halts the microprocessor.

If the real-time clock interrupt is to be useful, the interrupt service routine must do something useful; and the main program will do more than prepare for the interrupt.

The next problem is a small step in the direction of usefulness.

## b. Count 10 Real-Time Clock Interrupts

## Program 12-4b:

```

internal
  pgm_12_4b procedure
  entry
    $abs      IBASE

    ! Main Program !

4600 760F      LDA      R15,%8000
4602 8000
4604 2100      LD       R0,#%4F83
4606 4F83
4608 3A06      OUTB     PIOADR+5,RH0      ! PIO Port A input mode !
460A FF25
460C 3A86      OUTB     PIOADR+5,RL0      ! Interrupts enabled !
460E FF25
4610 4D05      LD       %6000,#10        ! Initialize clock counter !
4612 6000
4614 000A
4616 7C04      EI
4618 4D04      WTTEN:   TEST      %6000      ! 10 interrupts yet? !
461A 6000
461C EEF0      JR       NZ,WTTEN          ! No, wait !
461E 7A00      HALT
                                ! Yes, done !

    ! Real time clock interrupt service: !

4620 6B00      DEC      %6000            ! Count interrupts !
4622 6000
4624 7B00      IRET

4626          end      pgm_12_4b

```

The interrupt service routine shown in program 12-4b merely updates the counter in memory location 6000. The main program continues to execute.

**A more realistic real-time clock interrupt routine could maintain real time in several memory locations.** For example, the following routine uses addresses 6000 through 6003 as follows:

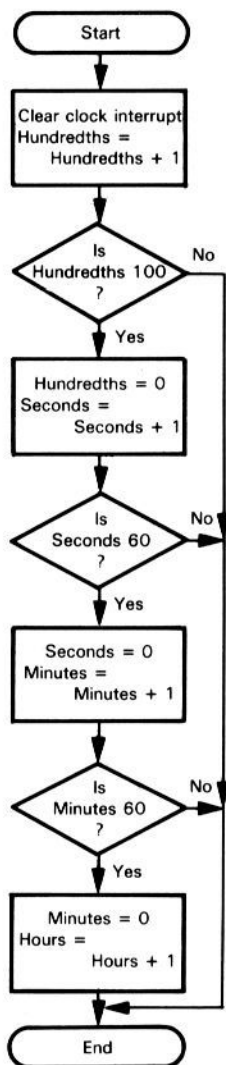
```

6000 - hundredths of seconds
6001 - seconds
6002 - minutes
6003 - hours

```

We assume that the interrupt is triggered by a 100 Hz clock.

Flowchart 12-4b:



The “Count 10 Real-Time Clock Interrupts” main program is used here, however memory locations 6000 through 6003 are cleared using the two instructions:

```

CLR    %6000
CLR    %6002

```

The interrupt service routine is shown in program 12-4c.



## Program 12-4c:

```

internal
pgm_12_4c procedure
entry
    $abs      IBASE
4600 91F0      PUSHL    @R15,RR0
4602 7601      LDA      R1,%6004      ! Pointer to time data !
4604 6004
4606 5400      LDL      RR0,%6000      ! Fetch 4 bytes of time data !
4608 6000
460A A800      INCB     RH0
460C 0A00      CPB      RH0,#100      ! Hundredths of a second !
460E 6464      ! Count up to 99 !
4610 E70C      JR       ULT,DONE
4612 8C08      CLRB     RH0
4614 A880      INCB     RL0
4616 0A08      CPB      RL0,#60      ! At 100, reset and bump secs !
4618 3C3C      ! 60 seconds in an minute !
461A E707      JR       ULT,DONE
461C 8C88      CLRB     RL0
461E A810      INCB     RH1
4620 0A01      CPB      RH1,#60      ! Increment minutes !
4622 3C3C      ! 60 minutes in an hour !
4624 E702      JR       ULT,DONE
4626 8C18      CLRB     RL1
4628 A890      INCB     RL1
462A 5D00      DONE:   LDL      %6000,RR0      ! Increment the hour !
462C 6000      ! Return time to memory !
462E 95F0      POPL     RR0,@R15
4630 7B00      IRET
4632          end      pgm_12_4c

```

A main program can now compute delays by testing the time as stored in memory. The following routine causes a 300 ms delay in the main program:

```

          LDB      RL0,%6000      ! Get current hundredths of !
          ! a second !
          ADDB     RL0,#30        ! Add 30 !
          CPB      RL0,#100      ! If the sum is more than !
          ! 100 ... !
          JR       ULT,WT30
          SUBB     RL0,#100      ! ...then normalize !
WT30:    CPB      RL0,%6000      ! Desired time reached? !
          JR       NE,WT30      ! No, wait !

```

Of course, the program could perform other tasks and check the elapsed time only occasionally. How would you produce a delay of seven seconds? Of three minutes?

Sometimes you may want to keep time either as BCD digits or as ASCII characters. How would you revise the program to handle these alternatives?

You can disable the clock interrupt (or any other interrupt) when it is no longer needed in any of the following ways:

1. By executing a DI VI instruction in the main program. This disables all vectored interrupts.
2. By clearing bit 7 of the Interrupt Control Word for the Z80 PIO I/O port. This disables interrupts at the selected I/O port only.
3. By changing the VI status bits in the Flag and Control Word stored on the stack, before returning from the interrupt service routine. However, it is a dangerous practice to modify a main program's interrupt enable/disable status from within an interrupt service routine; the main program can no longer assume that it knows which interrupts are or are not enabled. You are asking for trouble.

## 12-5. A Teletypewriter Interrupt

**Purpose:** The computer waits for a character to be received from a teletypewriter and stores the character in memory location 6000.

### a. Using an SIO (7-bit characters with odd parity)

#### Program 12-5a:

```

internal
  pgm_12_5a procedure
  entry
    $abs      IBASE

! Main Program !

4600 2100          LD      R0,##0441
4602 0441
4604 3A06          OUTB    SIOADR+5,RH0      ! Select write register 4 !
4606 FF45
4608 3A86          OUTB    SIOADR+5,RL0      ! x16 clock, odd parity !
460A FF45
460C 2100          LD      R0,##0341
460E 0341
4610 3A06          OUTB    SIOADR+5,RH0      ! Select write register 3 !
4612 FF45
4614 3A86          OUTB    SIOADR+5,RL0      ! 7 bit characters... !
4616 FF45          ! ...enable receiver !
4618 2100          LD      R0,##0118
461A 0118
461C 3A06          OUTB    SIOADR+5,RH0      ! Select write register 1 !
461E FF45
4620 3A86          OUTB    SIOADR+5,RL0      ! Interrupt every character !
4622 FF45
4624 7C04          EI
4626 E8FF  HERE:   JR      HERE

! Teletypewriter interrupt service routine: !

4628 93F0          PUSH    @R15,R0
462A 3A84          INB      RL0,SIOADR+1      ! Get character from SIO !
462C FF41
462E 6E08          LDB      %6000,RL0        ! Store in memory !
4630 6000
4632 97F0          POP      R0,@R15
4634 7B00          IRET

4636              end      pgm_12_5a

```

We could use the OTIRB instruction to initialize a Z80 SIO channel in the main program. Assuming that the six bytes 044103410118 are stored in memory, beginning at address SIOINIT, these instructions will initialize Channel A:

```

LD      R1,SIOADR+5      ! Load Channel A control !
                        ! address !
LDA      R2,SIOINIT      ! Load data base address !
LD      R0,#6            ! Load number of bytes !
OTIRB    @R1,@R2,R0

```

These four instructions occupy 8 words of object code; add 6 bytes for the data and we have a total of 11 object code words. Nine instructions, occupying 12 object code bytes, are replaced.

The service routine in program 12-5a assumes that only the receive interrupt from one channel of the SIO has been enabled. Otherwise, further decoding will be required based on control bits D2, D3, and D4 of Write Register 0 (see the discussion of SIO interrupts earlier in this chapter). Alternatively, the routine will have to examine the status bits in Read Register 0 to determine the activity. The key status bits are:

Bit 0 - Receive Character Available — 1 when at least one character is available in the receive buffers.

Bit 1 - Interrupt Pending (Read from Channel A only) — 1 if any interrupt is pending in the entire SIO.

Bit 2 - Transmit Buffer Empty — 1 if all characters in the transmit buffer have been sent.

The program establishes the SIO registers as follows:

**Write Register 4:**

Bit 7 = 0, bit 6 = 1 for X16 clock mode

Bit 1 = 0 to select odd parity

Bit 0 = 1 to enable parity generation

**Write Register 3:**

Bit 7 = 0, bit 6 = 1 to select 7-bit characters

Bit 0 = 1 to enable the receiver

**Write Register 1:**

Bit 4 = 1, bit 3 = 1 to produce an interrupt on all received characters with parity errors not affecting the vector address sent to the CPU during interrupt acknowledgment.

The CPU clears the Received Character Available bit by reading a character from the SIO Data register. The Interrupt Pending bit is cleared automatically when the interrupt is serviced.

If we use bit 7 of Z80 PIO Port A to receive serial data, as we did in Chapter 11, the programs needed to input data from the teletypewriter are shown in program 12-5b.

## Program 12-5b:

```

external
    TTYRCV procedure

internal
    pgm_12_5b procedure
    entry
        $abs      IBASE

! Main Program !

4600 2100          LD      R0,%%CF80
4602 CF80
4604 3A06          OUTB    PIOADR+5,RH0      ! Port A is control !
4606 FF25
4608 3A86          OUTB    PIOADR+5,RL0      ! Bit 7 is input !
460A FF25
460C 2100          LD      R0,%%977F
460E 977F
4610 3A06          OUTB    PIOADR+5,RH0      ! Enable int on start bit (0) !
4612 FF25
4614 3A86          OUTB    PIOADR+5,RL0      ! Mask out bits 0-6 !
4616 FF25
4618 7C04          EI
461A E8FF          HERE:   JR      HERE

! Teletypewriter interrupt service routine !

461C 030F          SUB     R15,#30           ! Room for 15 registers !
461E 001E
4620 1CF9          LDM     @R15,R0,#15      ! Save R0-R15 !
4622 000E
4624 C807          LDB     RL0,%%07
4626 3A86          OUTB    PIOADR+5,RL0      ! Disable start bit interrupt !
4628 FF25
462A 5F00          CALL    TTYRCV           ! Fetch data from TTY !
462C 0000*
462E C883          LDB     RL0,%%83
4630 3A86          OUTB    PIOADR+5,RL0      ! Enable start bit interrupt !
4632 FF25
! (PIO interrupt acknowledgement sequence here) !

4634 1CF1          LDM     R0,@R15,#15      ! Restore registers !
4636 000E
4638 010F          ADD     R15,#30
463A 001E
463C 7B00          IRET

463E              end      pgm_12_5b

```

Subroutine TTYRCV receives data from the teletypewriter; it is part of program 11-11a. It is good programming practice for subroutines to save and restore the contents of all CPU registers they use. This is particularly important for subroutines called by interrupt service routines.

The programs in program 12-5b assume that the monitor initializes the stack pointer. Otherwise it will have to be loaded in the main program.

The signal edge used to cause the interrupt is very important here. An interrupt must occur when the data line changes from the normal MARK or '1' state to the SPACE or '0' state, since this transition identifies the start of the transmission.

The service routine must disable the PIO interrupt, since otherwise each '1'-to-'0' transition in the character will cause an interrupt. Of course, you must reenable the PIO interrupt after the entire character has been read.

Note the use of the PIO in the control mode:

1. The PIO is placed in the control mode by establishing Mode 3.
2. The next control word defines which data lines are to be inputs ('1') and which are to be outputs ('0').
3. The interrupt control word has, besides the usual enable in bit 7,

bit 6 = 0 to perform a logical OR of the monitored data lines for an interrupt (not used in this case, since only one line is monitored)

bit 5 = 0 to define the active polarity of the data lines as low (for the start bit in this case)

bit 4 = 1 to indicate that a mask word follows.

4. The next control word contains the interrupt masks. Only those port lines with a mask bit of zero will be monitored for generating an interrupt.

The next result is for an interrupt to be generated if bit 7 is zero or changes from one to zero. Note that further interrupts occur only when the status of the bits being monitored changes.

Here again, the PIO could be configured by using a table and the repeated block output instruction.

## PROBLEMS

### 12-1. Wait for an Interrupt

**Purpose:** The computer waits for a PIO interrupt to occur, then executes the endless loop instruction:

```
HERE: JR HERE
```

until the next interrupt occurs.

### 12-2. A Keyboard Interrupt

**Purpose:** The computer waits for a 4-digit entry from a keyboard and places the digits into memory locations 6000 through 6003 (first one received in 6000). Each digit entry causes an interrupt. The fourth entry should also result in the disabling of the keyboard interrupt.

**Sample Problem:**

Keyboard data = 04, 06, 01, 07

Result:	(6000)	=	04
	(6001)	=	06
	(6002)	=	01
	(6003)	=	07

### 12-3. A Printer Interrupt

**Purpose:** The computer sends four characters from memory locations 6000 to 6003 (starting with 6000) to the printer. Each character is requested by an interrupt. The fourth transfer also disables the printer interrupt.

### 12-4. A Real-Time Clock Interrupt

**Purpose:** The computer clears memory location 6000 initially and then complements memory location 6000 each time the real-time clock interrupt occurs.

How would you change the program so that it complements memory location 6000 after every ten interrupts? How would you change the program so that it leaves memory location 6000 at zero for ten clock periods,  $FF_{16}$  for five clock periods, then zero again for ten, and so on continuously? You may want to change a display rather than memory location 6000 so that it will be easier to see.

## 12-5. A Teletypewriter Interrupt

**Purpose:** The computer receives TTY data from an interrupting SIO and stores the characters in a buffer starting in memory location 6000. The process continues until the computer receives a carriage return ( $0D_{16}$ ).

Assume that the characters are 7-bit ASCII with odd parity. How would you change your program to use a PIO? Assume that subroutine TTYRCV is available, as in the example. Include the carriage return as the final character in the buffer.



## REFERENCES

1. You may want to review the discussion of interrupts in A. Osborne, *An Introduction to Microcomputers: Volume 1 — Basic Concepts*. Osborne/McGraw-Hill, 1980.
2. For a discussion of designing with interrupts, see R. L. Baldridge, "Interrupts Add Power, Complexity to Microcomputer System Design," *EDN*, August 5, 1977, pp. 67-73.
3. See A. Osborne and J. Kane, *An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors*, Osborne/McGraw-Hill, 1980.
4. See reference 3 and L. Leventhal, *8080A/8085 Assembly Language Programming*, Osborne/McGraw-Hill, 1978.
5. R. L. Baldridge, "Interrupts Add Power, Complexity to Microcomputer System Design," *EDN*, August 5, 1977, pp. 67-73.  
 R. M. Pond, "Let Microprocessors Communicate," *Electronic Design*, November 8, 1977, pp. 88-90.  
 M. Shima, and R. Blacksher, "Improved Microprocessor Interrupt Capability," *Electronic Design*, April 26, 1978, pp. 96-100.  
 W. J. Weller, *Practical Microcomputer Programming: the Z80*, Northern Technology Books, Evanston, Ill., 1978.  
 A. W. Winston and T. B. Smith, "Use of the Z-80 in Data Collection and Control," *IECI '78 Proceedings — Industrial Applications of Microprocessors*, March 20-22, 1978, pp. 208-14.
6. J. Kane and A. Osborne, *An Introduction to Microcomputers: Volume 3 — Some Real Support Devices*, Chapter C4. Osborne/McGraw-Hill, 1978.
7. Zilog, Inc., *Z80-SIO Technical Manual*, Pub. 03-3033-01, August 1978.
8. The Proceedings of the IEEE's Industrial Electronics and Control Instrumentation Group's Annual Meeting on "Industrial Applications of Microprocessors" contains many interesting articles. Volumes (starting with 1975) are available from IEEE Service Center, CP Department, 445 Hoes Lane, Piscataway, NJ 08854.