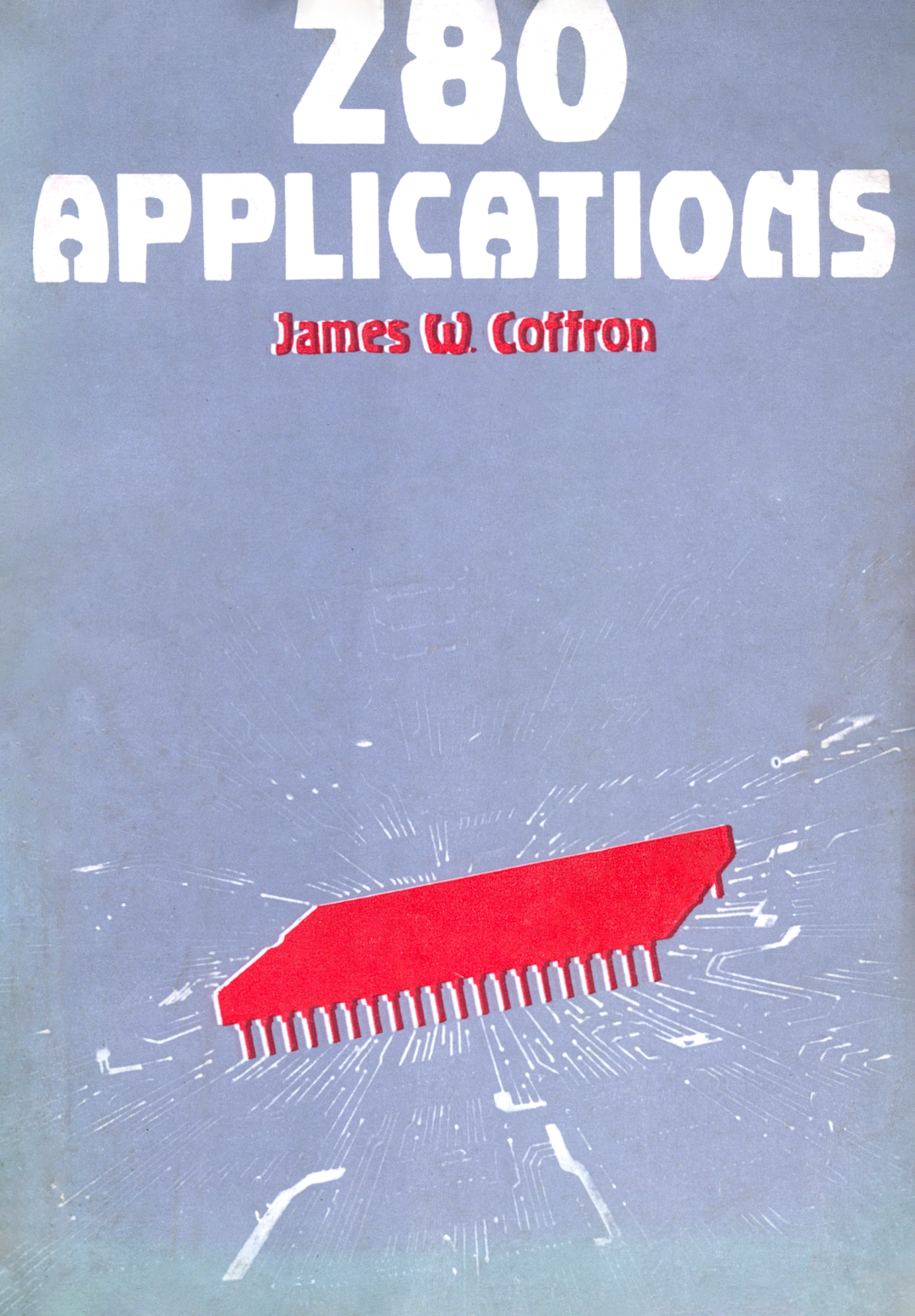
Z80 APPLICATIONS

James W. Coffron



Z80 APPLICATIONS

James W. Coffron

Berkeley • Paris • Düsseldorf

Cover Art by Jean Francois Penichoux
Layout and technical illustrations by Gaynelle B. Grover

In particular, technical characteristics and prices are subject to rapid change. Comparisons and evaluations are presented for their educational value and for guidance principles. The reader is referred to the manufacturer's data for exact specifications.

Library of Congress Card Number: 83-60950
ISBN: 0-89588-094-6
Printed in the United States of America
Printing 10 9 8 7 6 5 4 3 2 1

The following figures, taken from *Component Data Catalogue,* are reprinted by permission of Intel Corporation, Copyright 1981.

Figure 1.13, page 16
Figure 2.6, page 33
Figure 6.1, page 116
Figure 6.5, (redrawn), page 122
Figure 6.7, pages 125-26
Figure 7.1, page 142
Figure 7.2, page 143
Figure 7.24, page 161
Figure 10.8, page 218
Figure 10.10, page 220
Figure 10.16, page 229
Figure 10.17 (redrawn), page 230
Figure 10.18 (redrawn), page 232

The following figures, taken from *Linear Interface Integrated Circuits,* 1979, appear courtesy of Motorola, Inc.

Figure 10.13, page 225
Figure 10.14, page 226

The following figures are taken from the *1982/1983 Z80 Designers Guide,* copyright 1982 United Technologies' Mostek Corporation. Reprinted by permission.

Figure 1.16, page 18
Figure 4.2, page 69
Figure 8.1, page 164
Figure 8.2, page 165
Figure 8.15a, b, page 178
Figure 9.1, page 190
Figure 9.2, page 191
Figure 9.3, page 192
Figure 9.6, page 197
Figure 9.7, (redrawn) page 199
Figure 9.8, (redrawn) page 201
Figure 9.10, page 205
Figure 11.1, page 240
Figure 11.2, page 241
Appendix, pages 275-291

The following figure, taken from *IC Memories,* appears courtesy of Hitachi, Ltd., Tokyo, Japan.

Figure 2.19, pages 46-49

For Carol

Acknowledgements

I wish to thank Salley Oberlin for her excellent job in editing my original manuscript; and Bob Schuchard for his review and helpful suggestions. Finally, a hearty thanks to the entire staff at Sybex for their splendid efforts.

Contents

Introduction

Have you ever imagined a new application for a microprocessor-controlled system, only to have the idea vanish under a torrent of technical details found in the data sheets? If so, you're not alone. Several wonderful applications for microprocessor-controlled systems have experienced this fate. The idea was sound, but the person with the idea was not familiar enough with the details of the microprocessor to fully realize the project.

This book has been written to help Z80 system users overcome this dilemma. It has been designed to provide users with an understanding of the Z80 microprocessor. In general, it explains in detail how the Z80 microprocessor can be connected with the system hardware to comprise a system. In particular, it discusses ROM, static RAM, dynamic RAM, and input/output. In addition, later chapters examine certain special peripheral I/O devices. Some of these devices, like the SIO (serial input and output) and the PIO (peripheral input and output) chips were made to be used with the Z80; while others, like the 8253 timer chip and the 8255 peripheral chip, were not.

Written for the novice as well as the experienced programmer, this book provides instructive text, clear precise diagrams, and thorough examples. It has been designed to provide users with adequate information to realize and implement their own applications and ideas. Although this book does not show all the applications of the Z80 microprocessor—the individual applications must come from you—it provides the information necessary to realize any new application.

Contents

Chapter 1, *Using the Z80 with ROM,* starts off with an explanation of the different types of read-only memory. It then goes on to show how common ROM and EPROM devices can be interfaced to the Z80 CPU.

Chapter 2, *Using Static RAM with the Z80,* shows how static random-access memory can be used in a Z80 application. It discusses both common and separate I/O RAM chips and presents complete schematics of two static RAM systems.

Chapter 3, *Z80 Input and Output,* explains how the Z80 communicates electrically with input and output devices. Standard input and output ports are shown and discussed in detail.

Chapter 4, *Using Dynamic RAMs with the Z80,* shows how dynamic RAM systems can be used with the Z80. This chapter begins with a discussion of a typical dynamic RAM device. It the goes on to explain how you can interface RAM to the Z80, by taking advantage of the Z80 internal architecture.

Chapter 5, *Interrupts for the Z80,* discusses the important topic of interrupts. All three interrupt modes are examined, and examples are given of each type.

Chapter 6, *Using the 8255 PIO with the Z80,* examines the use of the 8255 peripheral I/O chip—a device that is interfaced to the Z80 buses. Each operating mode for the 8255 is discussed, and many programming examples are given.

Chapter 7, *Using the 8253 Programmable Timer with the Z80,* shows how the 8253 programmable timer chip can be used with the Z80. An explanation is given on interfacing the device to the Z80. In addition, several general programming examples are presented.

Chapter 8, *Using the Z80 PIO,* continues the discussion of peripheral chips, focusing on the Z80-PIO. It offers a block diagram of the device and a discussion of the operating modes and registers. In addition, many programming examples for using the Z80-PIO are given.

Chapter 9, *Using the Z80-CTC,* discusses the counter timer chip and shows you how to interface it to the Z80. In addition, many programming examples for using the CTC are discussed.

Chapter 10, *Introduction to Serial Communication,* discusses the topic of serial communication. Concepts such as baud rate, start bit, stop bit, and marking levels are defined. As a practical example, you are shown how to interface and use the 8251 USART with the Z80.

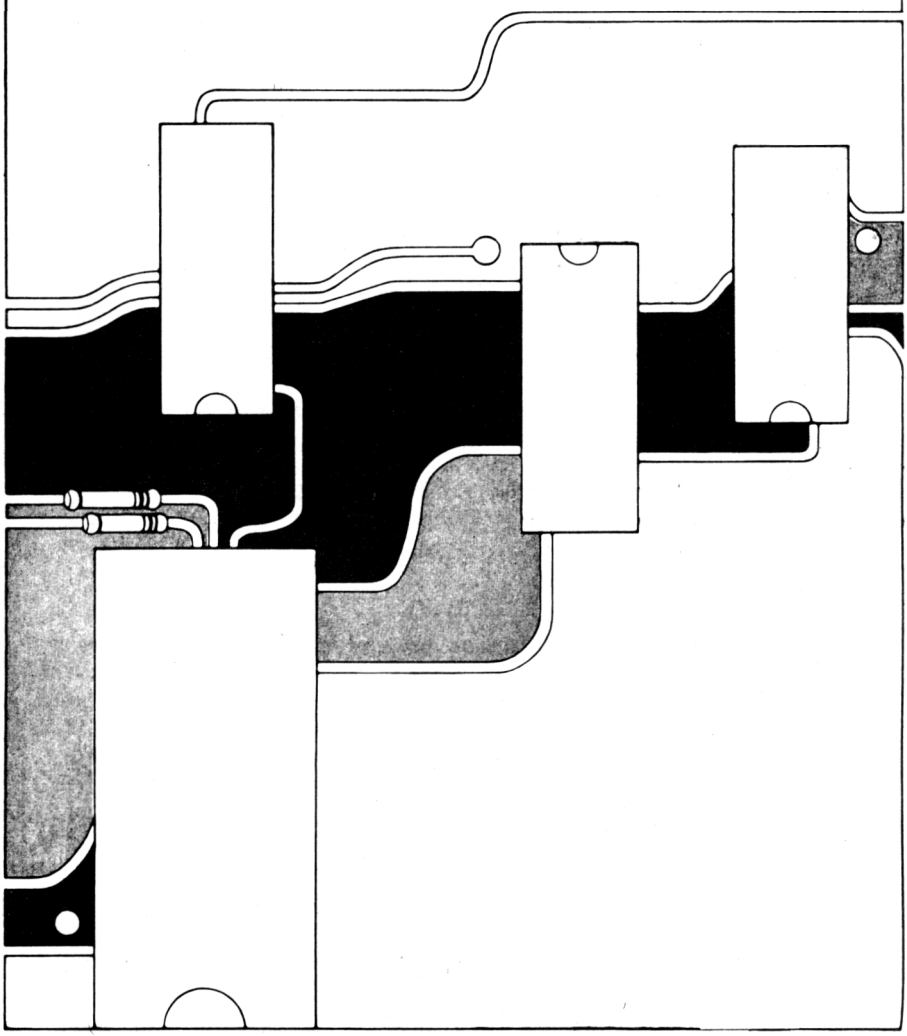
Chapter 11, *Using the Z80-SIO,* continues with serial communication by discussing the Z80-SIO chip. Several practical examples of using and programming the Z80-SIO are given.

Chapter 12, *Static Stimulus Testing for the Z80,* concludes the text with a discussion of static stimulus testing (SST) for the Z80. It shows how you can use this method to electrically debug your system without software. Further, if you are adding a new interface to an existing Z80 system, you can use this technique to check out the new interface quickly and easily.

Finally, Appendix A, *Z80-SIO Internal Register Descriptions,* describes the operation of each internal register of the Z80-SIO.

As you can see, this book offers information on all the important topics for understanding the Z80 microprocessor. With this knowledge you should be able to pursue any idea or application for a microprocessor-controlled system. As you begin implementing your own ideas and applications, you will find that using and applying the Z80 microprocessor is not only easy but can be a great deal of fun as well. So let your imagination go, and read on and learn how the Z80 can be made to work for you.

Using the Z80 with ROM



Chapter 1

INTRODUCTION

We will begin our study of the Z80 microprocessor by learning how to connect it with read-only memory (ROM). We will start with a general introduction to the operation of ROMs. We will then examine the important aspects of using ROM with the Z80. By the end of this chapter you should have a complete understanding of how ROM communicates electrically with the Z80, and you should be able to apply this knowledge to your system, regardless of your overall application.

1-1: What Is ROM?

In most microprocessor systems there is a certain amount of memory that can store information, and this information will not be lost when power is turned off. This type of memory is called *read-only memory* or *ROM*. Information in ROM can be read out but not altered. ROM is useful in a system because it allows the central processing unit (or CPU) of the system to initialize all of the peripheral hardware to the proper logical states when power is first turned on.

There are several types of non-volatile memory that can be used in a microprocessor system. These include Read-Only Memory (ROM), Programmable Read-Only Memory (PROM), Erasable Programmable Read-Only Memory (EPROM), and Electrically Alterable Read-Only Memory (EAROM). (See Figure 1.1.) Normally, in a typical Z80 system only one type of non-volatile

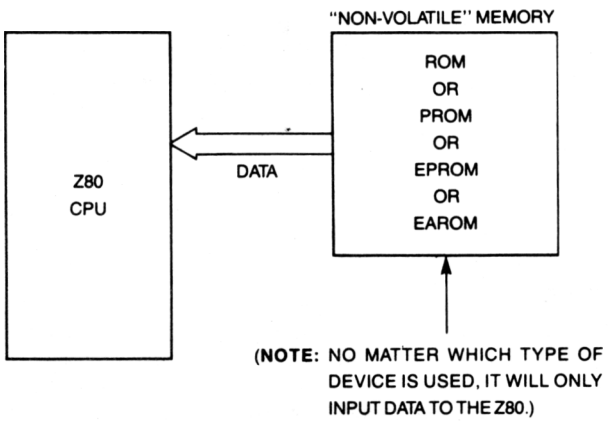


Figure 1.1: A block diagram showing a typical application of a non-volatile memory device that might be used in a microprocessor system. The non-volatile ROM usually contains "boot up" programs that are executed when the system is first powered up.

memory is used. However, since it is possible that your application may require any one of these memory devices, we will now describe them all:

ROM With ROM, or "read-only memory," data is programmed into the memory device by the manufacturer. ROM is used whenever there is a non-changing data base of high volume. (Note: programming data into ROM is a very expensive process.)

PROM With PROM or "programmable read-only memory," data is programmed into the memory device by the user. High voltage pulses actually "blow apart" metal strips or polycrystaline silicon inside the integrated circuit, forcing logical 1's and 0's into specific address locations in memory. Once programmed into the device, the data cannot be altered. This memory device usually operates at a much faster speed than other programmable memories.

EPROM With EPROM, or "erasable programmable read-only memory," data is programmed by the user into the memory device, by

applying high voltage signals. This is similar to the programming of a PROM. Data can be erased by shining ultraviolet light onto a transparent window that covers the integrated circuit. After a specified time of exposure to the light, all data will be erased and the device can then be reprogrammed with entirely new data. These devices are often used in development work where the program may undergo many changes.

EAROM With EAROM, or "electrically alterable read-only memory," data is programmed into the memory device by the user in a fashion similar to the EPROM. The major difference is that the data in an EAROM can be erased electrically, without the ultraviolet light.

No matter which non-volatile memory you are using in your system application, the operating characteristics are all very similar. We will now discuss the important parameters of this group of memories. (*Note:* throughout this text we will use the word ROM to refer to any of the non-volatile memory devices that we have just described.)

1-2: Important Operating Characteristics of ROM

Let's now review some important operating characteristics of ROM. To begin, it is important to note that ROM can only be *read* by the CPU; hence, the name "read-only memory." Second, information in the memory is fetched whenever an address is applied to the address input lines. The number of address input lines a ROM has depends on the internal organization of the data in memory.

It is possible to determine the internal organization of data in memory by noting the specification for the ROM. For example, a ROM may be organized as 1024 × 8, or 2048 × 8, or 4096 × 8—to name just a few types of memory organizations. The first number in the specification indicates the number of unique address locations contained within the single integrated circuit. The second indicates the number of bits of parallel data that can be read from the ROM at each unique address location.

The following steps help determine the number of address lines available with a particular ROM. If we start with a description of the device, say 2048 × 8, we already know that the 2048 indicates the number of unique address locations, and that this number is equal to the number of different binary combinations existing on the address lines. In other words, $2^X = 2048$, or $X = 11$, where X equals the number of address lines. Therefore, if the memory is described as 4096 × 8, the number of address lines is $2^X = 4096$, or $X = 12$. As you can see, this equation works regardless of the number of address locations in the ROM.

Note that when describing a ROM, the exact number of address locations is usually shortened to the nearest thousand and abbreviated by the single letter "K" for Kilo. In other words, a 1024 × 8 ROM would be described as a 1K device. A 4096 × 8 ROM would be described as a 4K device, etc.

A final point about ROM operation is that all data is read from ROM in a parallel fashion. In other words, while the data outputs are being electrically enabled onto the system data bus, the data bits are being strobed by the microprocessor into an internal register.

Figure 1.2 displays a functional block diagram of ROM operation. In this figure, we can see an input, called *chip select,* that we have not yet discussed. The function of this input line is to turn on and off the data output lines of the ROM device. When the chip select input line is active, the ROM data outputs are active, and are either logical 1 or 0, depending on the data programmed into the device. When the chip select line is not active, the

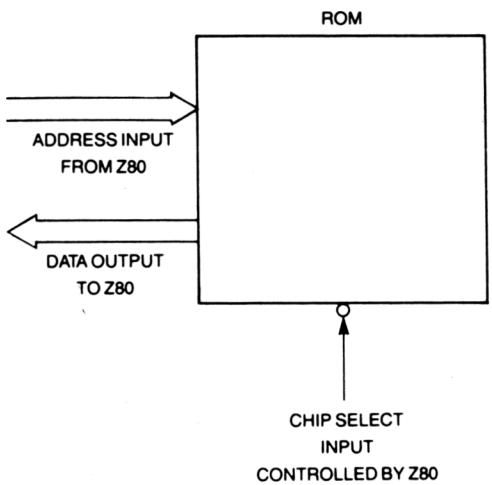


Figure 1.2: *A functional block diagram of a typical ROM device. Address lines are input to select the internal data to be output. Data is output when the device is selected with the chip select line.*

data outputs are placed into a *tri-state,* or high impedance state. In other words, the chip select input line either electrically enables or disables the outputs of the ROM device. Later in this chapter we will show exactly how this line is used.

1-3: Sequence of Electrical Events for Reading Data From ROM

Let's discuss the block diagram in Figure 1.2. A sequence of electrical events (listed below) occurs each time data is to be read from the device. Keep in mind that this general sequence must be followed regardless of the type of CPU used in the application. Let's examine the sequence. (Later on, we will discuss specifically how the Z80 follows this sequence.)

1. An address is input to the ROM from the CPU. This address indicates the internal location of the device from which data will be read. There are thousands of bytes of data stored in the ROM. The address lines select one byte of data to be output.

2. The CPU then waits for a finite amount of time, called *access time*—which is approximately 100–300 nanoseconds, depending on the type of ROM used—thus, allowing the memory device to decode the address that is input, and the output data to reach the data output lines of the device.

3. The chip select is made active to enable the data outputs onto the system data bus. At this time the data from ROM is present on the system data bus and the CPU data input pins. The CPU next strobes the data into an internal register.

4. The chip select is made inactive to remove the ROM data from the system data bus.

This general sequence is followed each time the CPU reads data from ROM. Figure 1.3 shows a general timing diagram of this sequence.

CPUs are designed to operate within these boundaries. Therefore, in most cases the user does not have to think about this sequence because the microprocessor handles it with the help of internal timing circuits. However, it is important that you understand this sequence, as it makes using ROM, as well as understanding how the circuits operate, much simpler.

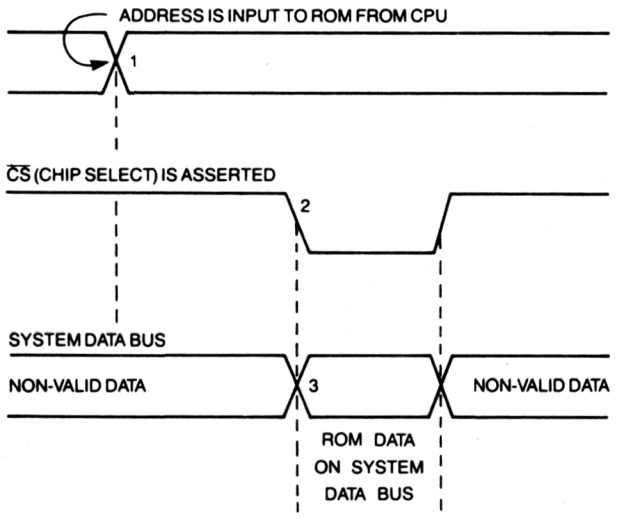


Figure 1.3: A general timing diagram for reading data from a ROM device:
1. Address is input to the device from the CPU.
2. The chip select is asserted.
3. The ROM data outputs are enabled onto the data bus.

1-4: Connecting the Z80 Buses

Let's now connect the Z80 to ROM. Figure 1.4 shows the physical connections of the Z80 data and address lines to ROM. These connections are very straightforward. Notice in Figure 1.4 that the chip select line of the 2716 EPROM is not connected. This is because we plan to first discuss *address mapping*.

1-5: Address Mapping

The Z80 has 16 physical address output lines, labeled A0 – A15. This means that a total of 2^{16} or 65,536 unique storage locations can be accessed directly by the Z80. For example, a 2716 EPROM has 2048 physical address locations that can be accessed. Therefore, a method for selecting only 2048

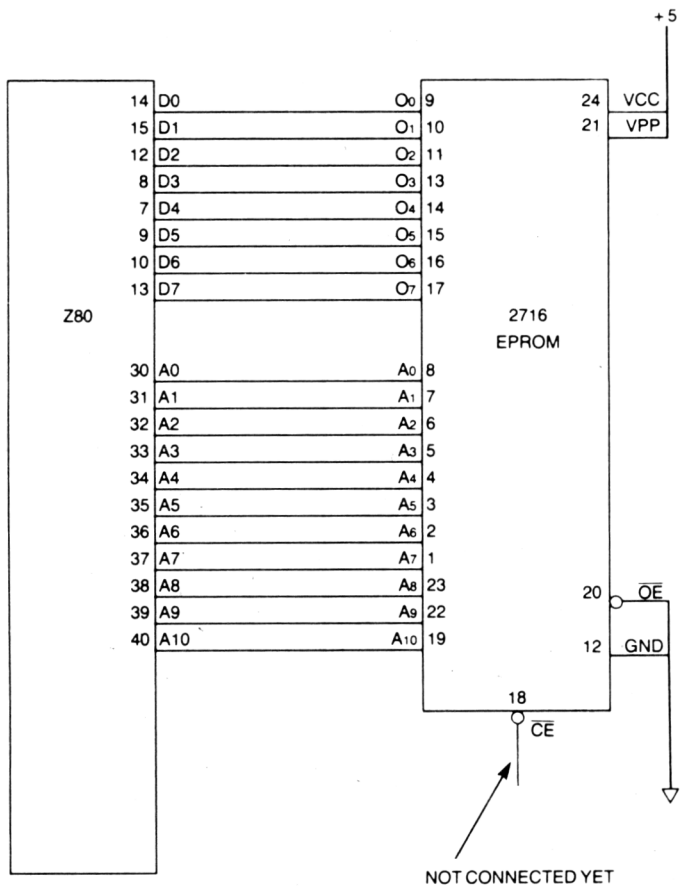


Figure 1.4: *A schematic diagram showing the address and data line connections between a Z80 microprocessor and a 2716 EPROM using non-buffered data and address lines.*

locations out of the total 65,536 or 64K possibilities is needed. In other words, which 2048 locations do we choose? There are 65,536 divided by 2048, or 32 possible blocks of 2048 locations from which we can select.

Before any hardware is built for a system, the system designer must first construct a *memory map.* A memory map is used to indicate which address locations are specified for any particular ROM, RAM, input, or output device. Figure 1.5 shows a typical memory map. We can see in this figure that the entire 65,536 address locations are divided into functional blocks. These blocks indicate the address locations that are reserved for specific ROMs and RAMs.

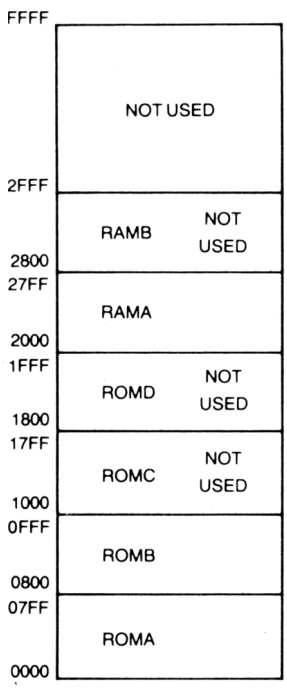


Figure 1.5: *A memory map of a typical microprocessor system. Before any hardware is designed, a memory map is normally constructed that indicates how the memory space is to be allocated.*

Note: As shown in Figure 1.5, the lowest value of address in a Z80 system (i.e., 0000 in hexadecimal) is usually reserved for ROM. This is because whenever the Z80 is reset from an external reset switch or from a power-on reset circuit, the address bus always expects the first instruction op-code to reside at address 0000 hexadecimal.

1-6: Generating the Chip Selects for ROM

Based on the information in Figure 1.5, we can see that ROMA is enabled whenever addresses between 0000 and 07FF, inclusive, are output by the Z80. An electrical signal in the system is needed to indicate that the address being output on the address bus is within these limits. Figure 1.6 shows such a circuit.

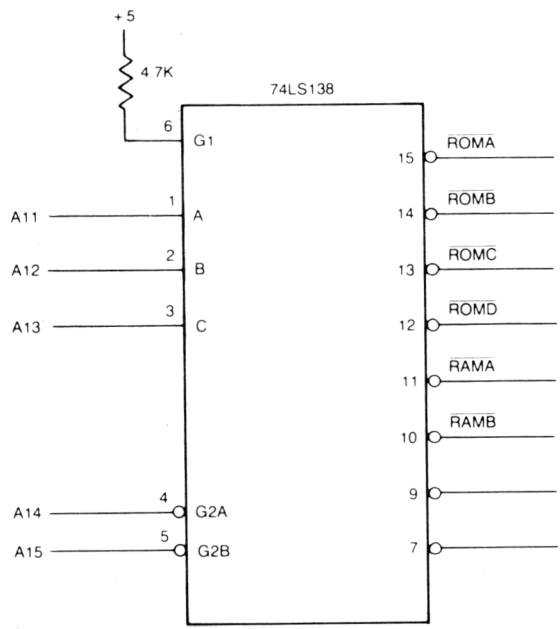


Figure 1.6: *A hardware circuit that realizes some of the memory mapping in Figure 1.5.*

In the circuit in Figure 1.6, whenever A11, A12, A13, A14, and A15 are logical 0's, output pin 15 of the 74LS138 is a logical 0. Pin 15 remains a logical 0 between the address limits specified. The table in Figure 1.7 shows how the output pins of the 74LS138 respond to the address output on the address bus.

We can see in Figure 1.7 that when A11 goes to a logical 1, pin 14 of the 74LS138 goes to a logical 0, and pin 1 to a logical 1. The result being that pin 1 is a logical 0 if and only if the address output from the Z80 is between the limits 0000 – 07FF. This same reasoning may be applied to the other output pins shown in the table in Figure 1.7.

It is important to keep in mind that the circuit shown in Figure 1.6 is only one way (out of many) of selecting a unique address block out of the available address space.

A15	A14	A13	A12	A11	A10 ---------- A0	HEX	PIN # = 0
0	0	0	0	0	0 ---------- 0	0000	15
0	0	0	0	0	1 ---------- 1	07FF	15
0	0	0	0	1	0 ---------- 0	0800	14
0	0	0	0	1	1 ---------- 1	0FFF	14
0	0	0	1	0	0 ---------- 0	1000	13
0	0	0	1	0	1 ---------- 1	17FF	13
0	0	0	1	1	0 ---------- 0	1800	12
0	0	0	1	1	1 ---------- 1	1FFF	12
0	0	1	0	0	0 ---------- 0	2000	11
0	0	1	0	0	1 ---------- 1	27FF	11
0	0	1	0	1	0 ---------- 0	2800	10
0	0	1	0	1	1 ---------- 1	2FFF	10

Figure 1.7: Memory Map of Figure 1.6

1-7: Generating the Memory Read Signal

At this point we are nearly ready to connect the chip select input pin to the Z80. However, we must first learn about the timed control line signal from the

Z80 that starts and stops the data transfer. This signal is called the *memory read* (or the $\overline{\text{MEMR}}$ signal). The $\overline{\text{MEMR}}$ signal and the memory select signals work together to enable the ROM data outputs onto the system data bus at the correct time. Figure 1.8 shows one way of generating the $\overline{\text{MEMR}}$ signal with the Z80.

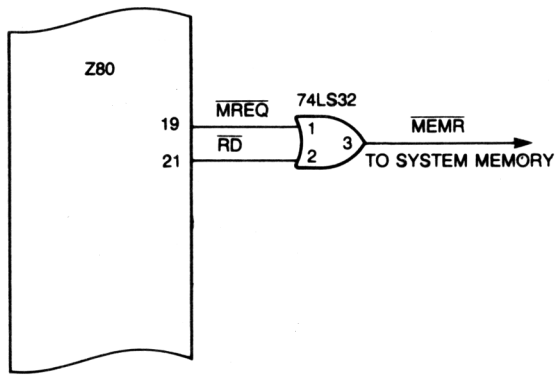


Figure 1.8: *A schematic circuit showing how the system memory read control signal is generated by ORing the control signals from the Z80 microprocessor.*

In Figure 1.8 the $\overline{\text{MREQ}}$ line is set to a logical 0 whenever the Z80 is communicating with memory. The software instructions dictate this action. $\overline{\text{RD}}$ is a timed control signal output by the Z80. It becomes a logical 0 whenever the Z80 is electrically prepared to receive data from memory or from an input/output device.

1-8: Connecting the Chip Select Lines

Figure 1.9 shows a complete schematic for connecting the Z80 to ROM. When both the memory select and the memory read ($\overline{\text{MEMR}}$) signals are logical 0, the chip select input to the 2716 EPROM is active logical 0. Figure 1.10 shows the timing that occurs during a ROM read operation and the important signal relationships.

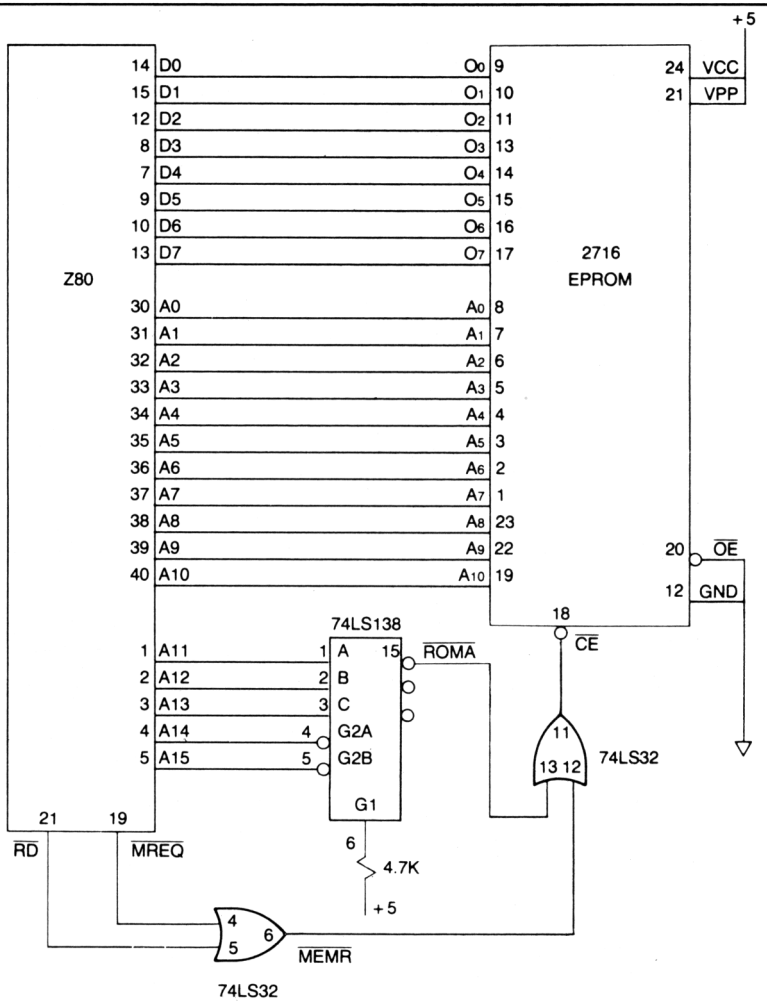


Figure 1.9: A complete schematic showing the connection between a 2716 EPROM and the Z80 microprocessor. This schematic includes the logic used for enabling the memory device according to the memory map.

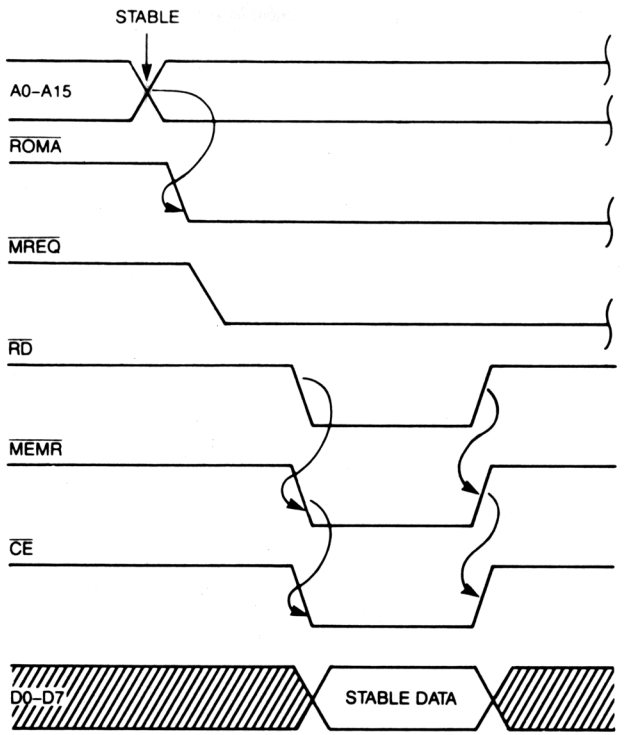


Figure 1.10: A timing diagram showing the important signal relationships of the schematic diagram of Figure 1.9.

1-9: A Variation

Figure 1.11 shows yet another technique that can be used to enable the ROM data outputs onto the system data bus at the correct time. This technique takes advantage of the internal decoding located inside the 2716 device package. In this application both the $\overline{OE}$ pin 20 and the $\overline{CE}$ pin 18 must be a logical 0 before the data outputs are enabled onto the system data bus.

Recall that with the previous technique, the $\overline{OE}$ input pin 20 was connected to ground potential which was the active state for this input.

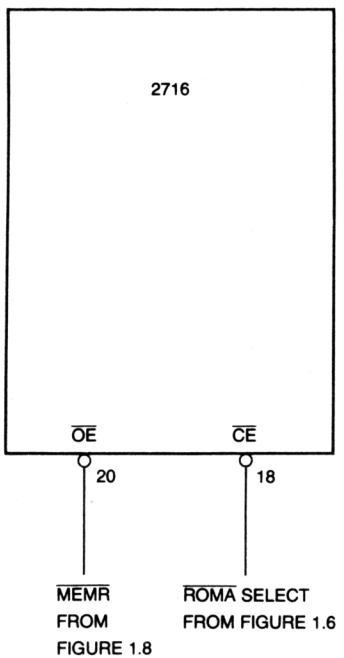


Figure 1.11: A schematic diagram showing how the 2716 device can be enabled using the output enable pin 20 or the chip select pin 18.

1-10: Adding More ROM

Figure 1.12 shows that it is possible to add more ROM devices to the system by using the enabling technique of Figure 1.11. This technique allows us to add more physical ROM devices without having to add external gates. (Recall that the decoding technique described in Figure 1.10 requires additional gates when adding more ROM to the system.)

Recall that each ROM has a specific address block reserved for its use. However, the Z80 CPU makes no separation when executing the software. It assumes that all the ROM space is available. In other words, a three-byte instruction may have one byte located in the last location of ROMA, and the next two bytes located in the first locations of ROMB.

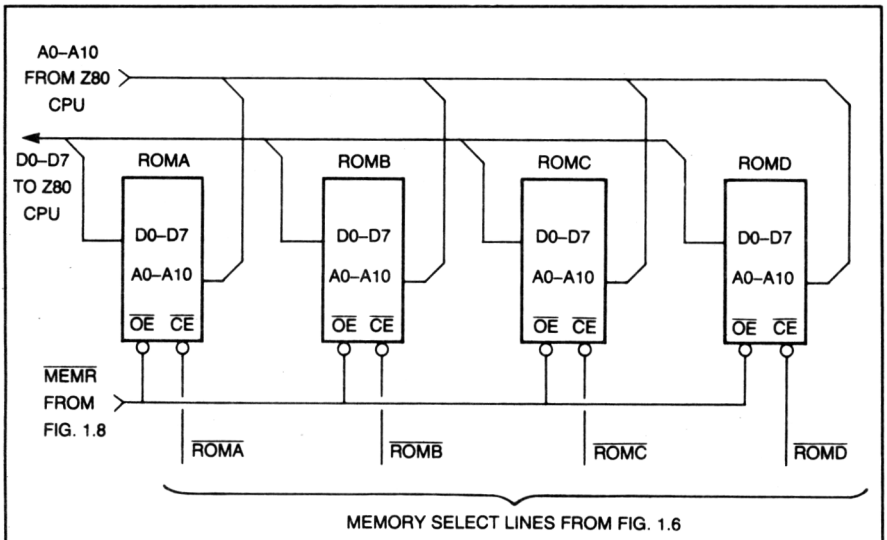


Figure 1.12: A schematic diagram showing how you can add multiple memory devices to the memory hardware without using extra logic gates. The use of the $\overline{OE}$ line of the 2716 is employed in this memory decoding technique.

1-11: Memory Mapping Larger ROMs

There are two larger ROMs that are widely used in system applications: the 2732 and the 2764. These devices are organized as 4096 × 8 and 8192 × 8, respectively. Figure 1.13 shows the pinouts of these two devices.

A point of interest regarding the pinouts of the 2732 and the 2764 is that they can be interchanged in a physical socket even though the pin count is completely different. This can be accomplished by using 28-pin sockets in the system and connecting Vcc to pins 28 and 26. (See Figure 1.14.)

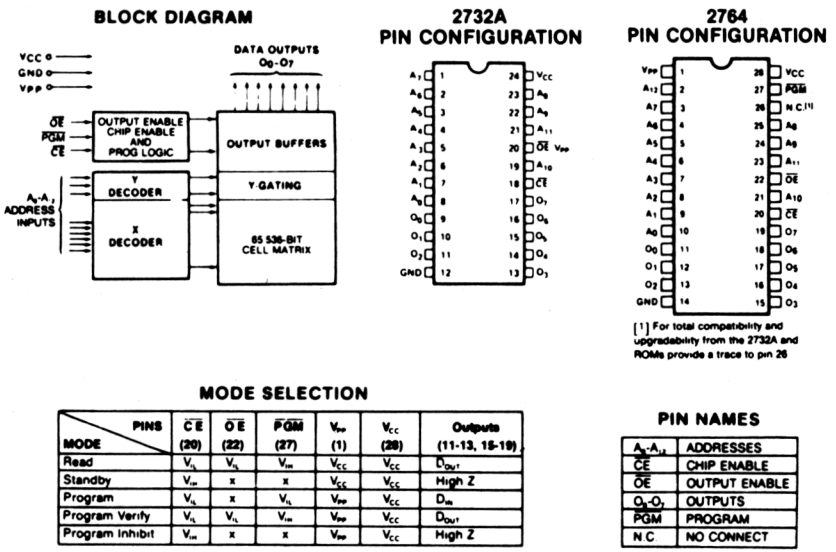


Figure 1.13: Pinouts and a block diagram of the 2732 and the 2764 EPROM devices that are widely used in microprocessor applications.

These devices are designed so that you can elect to start with the 2732 and then later upgrade to the 2764. This allows you to double your available ROM space without increasing any physical board space. If the larger memories are chosen, the memory map will change from the one shown in Figure 1.5 to the one shown in Figure 1.15. Notice in Figure 1.15 that only one physical device is used for the address space from 0000 – 0FFF. (Recall that the map in Figure 1.5 required two.) The hardware used to generate the memory select lines is similar to the circuit hardware shown for the previous map that used the 2716 EPROMs.

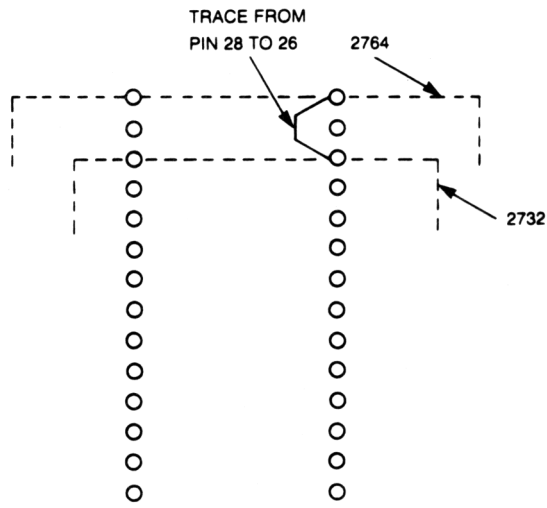


Figure 1.14: A block diagram showing how a single physical socket can be used to apply either the 2732 or the 2764 device. In this application, a 28-pin socket has been used.

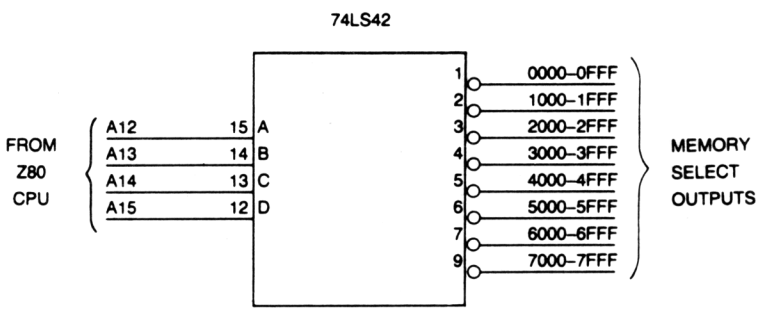


Figure 1.15: A hardware schematic showing the memory select decoding logic using a 74LS42 BCD to Decimal decoder.

1-12: Adding Address Buffers

The memory system that we have presented is very basic. It is, however, widely used in industry. Note that there is a critical area in this type of system design: the load on the Z80 address bus lines.

If we examine a typical Z80 data sheet as presented in Figure 1.16, the specification for I_{OL} (output current logical 0) and I_{OH} (output current logical 1) for an address output line is 1.8 mA and $-250\,\mu A$, respectively. (The minus sign indicates that the current is being supplied by the Z80 and is leaving the CPU device.) This specification tells us that only 1 standard TTL (transistor transistor logic) load can be driven by any address output line. A TTL load is equal to 1.6 mA in the logical 0 state and 40 microamperes in the logical 1 state. See Figure 1.16.

In the memory system designs presented thus far, the only TTL device used has been the 74LS138. The LS indicates low power Schottky and is equivalent to approximately 1/3 of a standard TTL load. This load was only present on the upper address lines, A11 – A15. All other Z80 address output lines had loads of the memory address inputs only. Memory address input loads are typically 10 microamperes in the logical 1 and 0 states. However, the system address line may be connected to other circuits besides memory. The address line may be connected to I/O devices, cables, or even through edge connectors to another circuit board.

In these cases and in any application where the output load on the Z80

D.C. CHARACTERISTICS
T_A = 0°C to 70°C, V_{CC} = 5V ± 5% unless otherwise specified

SYMBOL	PARAMETER	MIN.	TYP.	MAX.	UNIT	TEST CONDITION
V_{ILC}	Clock Input Low Voltage	−0.3		0.8	V	
V_{IHC}	Clock Input High Voltage	Vcc−.6		Vcc+.3	V	
V_{IL}	Input Low Voltage	−0.3		0.8	V	
V_{IH}	Input High Voltage	2.0		V_{CC}	V	
V_{OL}	Output Low Voltage			0.4	V	I_{OL} = 1.8mA
V_{OH}	Output High Voltage	2.4			V	I_{OH} = −250 μA
I_{CC}	Power Supply Current			150*	mA	
I_{LI}	Input Leakage Current			±10	μA	V_{IN} = 0 to V_{CC}
I_{LOH}	Tri-State Output Leakage Current in Float			10	μA	V_{OUT} = 2.4 to V_{CC}
I_{LOL}	Tri-State Output Leakage Current in Float			−10	μA	V_{OUT} = 0.4V
I_{LD}	Data Bus Leakage Current in Input Mode			±10	μA	$0 \leq V_{IN} \leq V_{CC}$

Figure 1.16: *A partial data sheet for the Z80 microprocessor showing the I_{OL} and I_{OH}, sink and source currents for the CPU output lines.*

address lines exceeds the specified rating, *address buffers* will be needed. Address buffers increase the output current drive capability of the Z80 address bus. Address buffers should be used any time the address lines are required to drive a cable, even if the DC load does not exceed the specification. This is due to the capacitive load a cable places on the address line. Figure 1.17 shows one way that you can add address buffers to a Z80 system. Note, however, that not all Z80 systems require address buffers. The use of buffers is dependent on the system application.

1-13: Memory Data Buffering

In some system applications, the data outputs from the system ROM do not have enough current drive capability to adequately control the system data bus lines. In such cases, it is necessary to use *memory data buffers.*

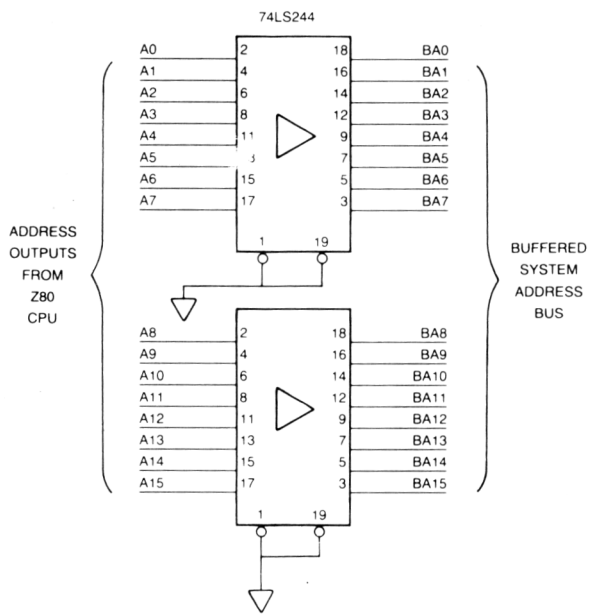


Figure 1.17: *A schematic diagram showing one technique for adding address buffers to a Z80 system.*

Memory data buffers perform a function similar to the memory address buffers discussed in the previous section.

The memory data buffers must be capable of a *tri-state* operation. A tri-state operation is necessary because the memory data must be electrically removed from the system data bus when the Z80 is not electrically requesting it. Figure 1.18 presents a schematic diagram showing how memory data buffers can be installed in a typical ROM system. In this figure, the ROM data outputs need only drive the data buffer inputs. The outputs of the memory data buffers will drive the entire load of the system data bus.

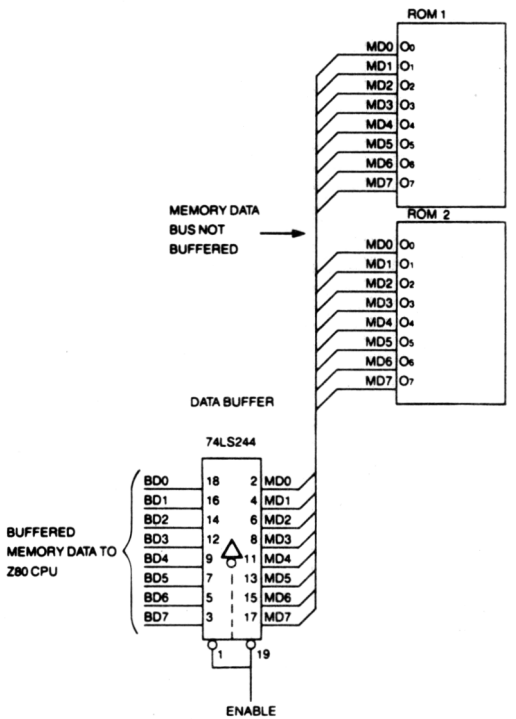


***Figure 1.18:** A schematic diagram showing how memory data output buffers can be used to buffer the memory data prior to driving the system data bus.*

1-14: Three Complete ROM System Examples

The schematics shown in Figures 1.19, 1.20, and 1.21 are all complete ROM systems based on the concepts presented in this chapter. The memory

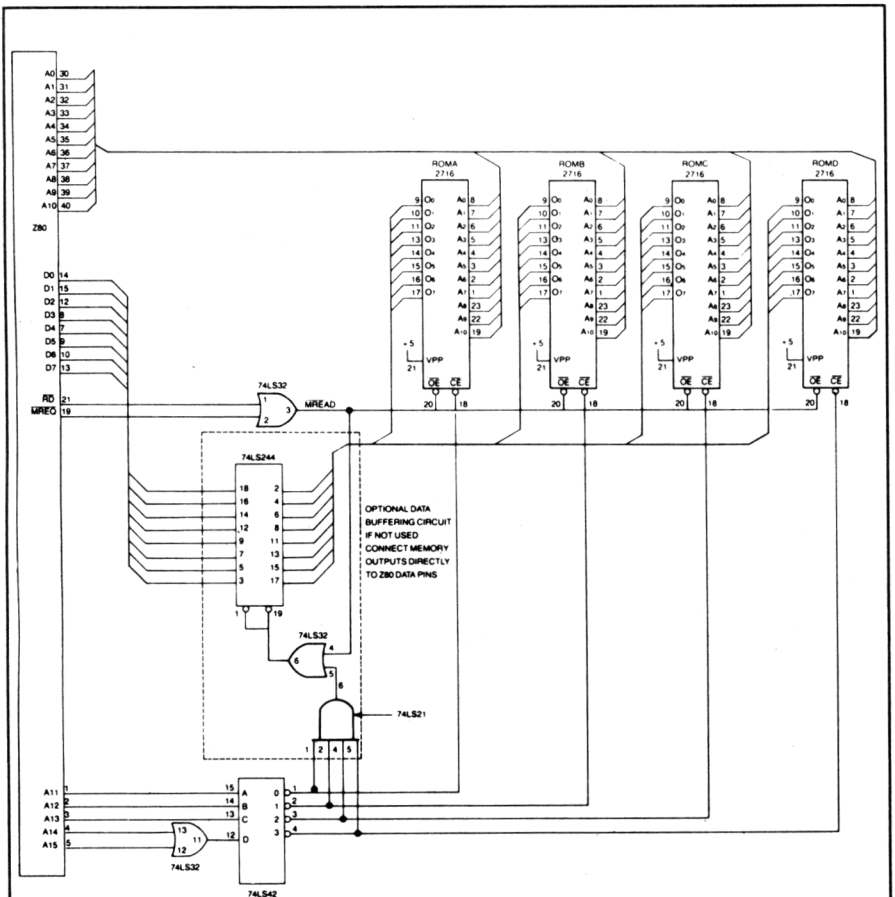


Figure 1.19: A complete memory system using the memory map shown in Figure 1.22. This system employs 2716 devices.

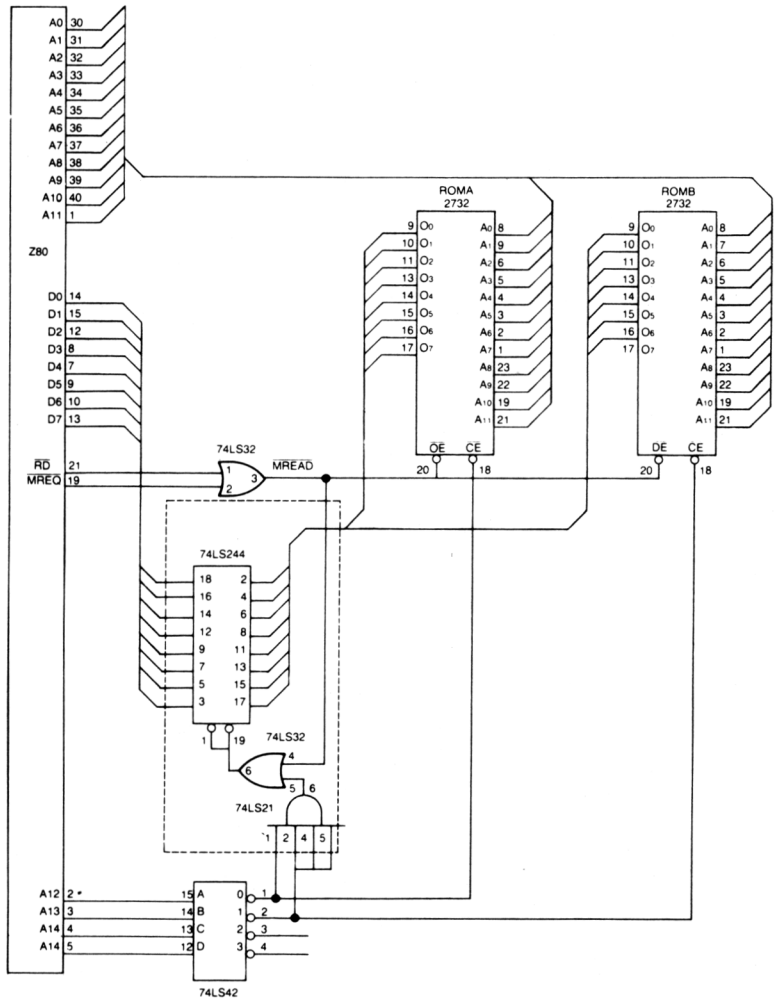


***Figure 1.20:** A complete memory system using the memory map shown in Figure 1.22. This system employs 2732 devices.*

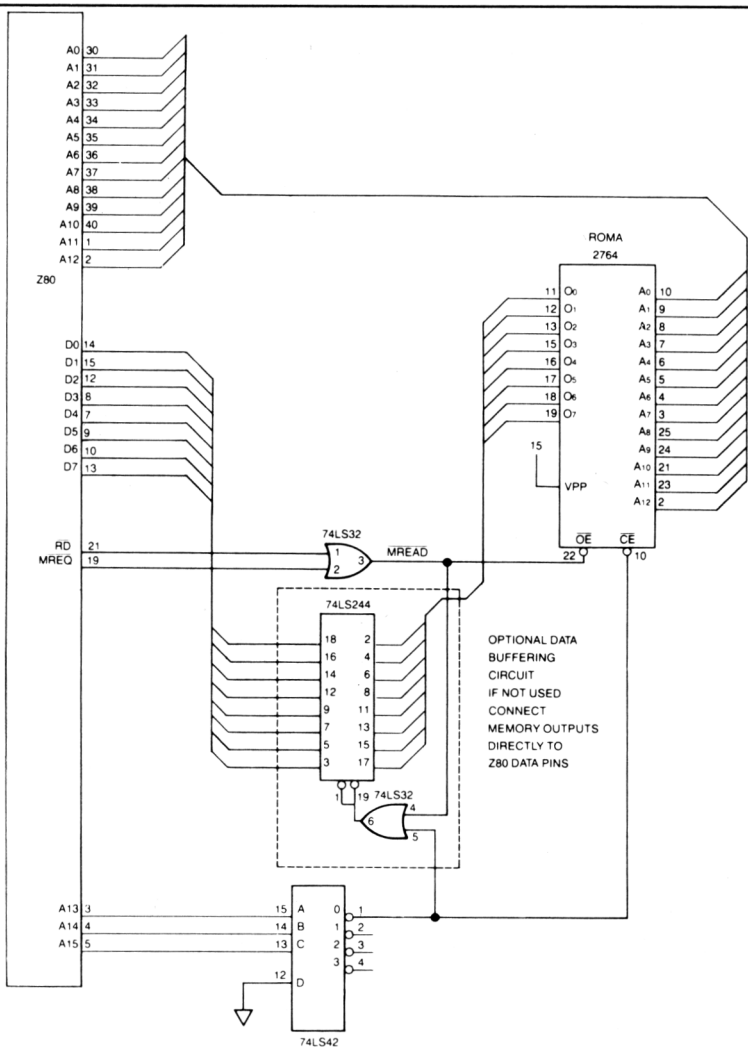


Figure 1.21: *A complete memory system using the memory map shown in Figure 1.22. This system employs a 2764 device.*

map for these systems appears in Figure 1.22. Memory systems using these devices are in wide general use today. The EPROMs are available from many suppliers and are easy to program and use.

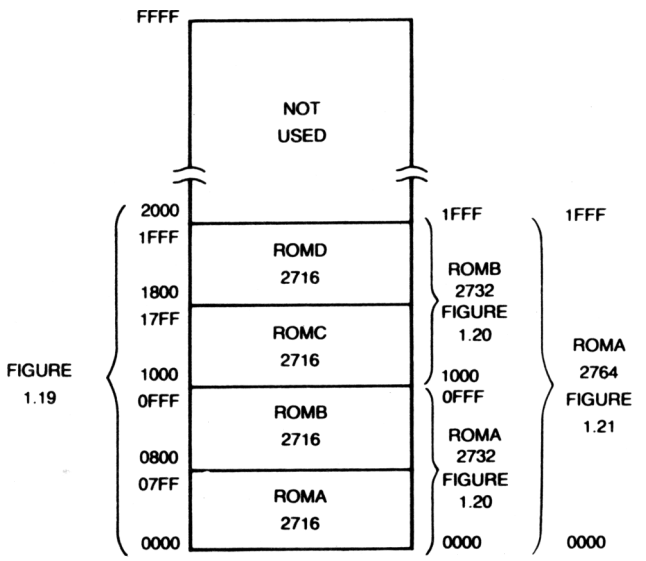
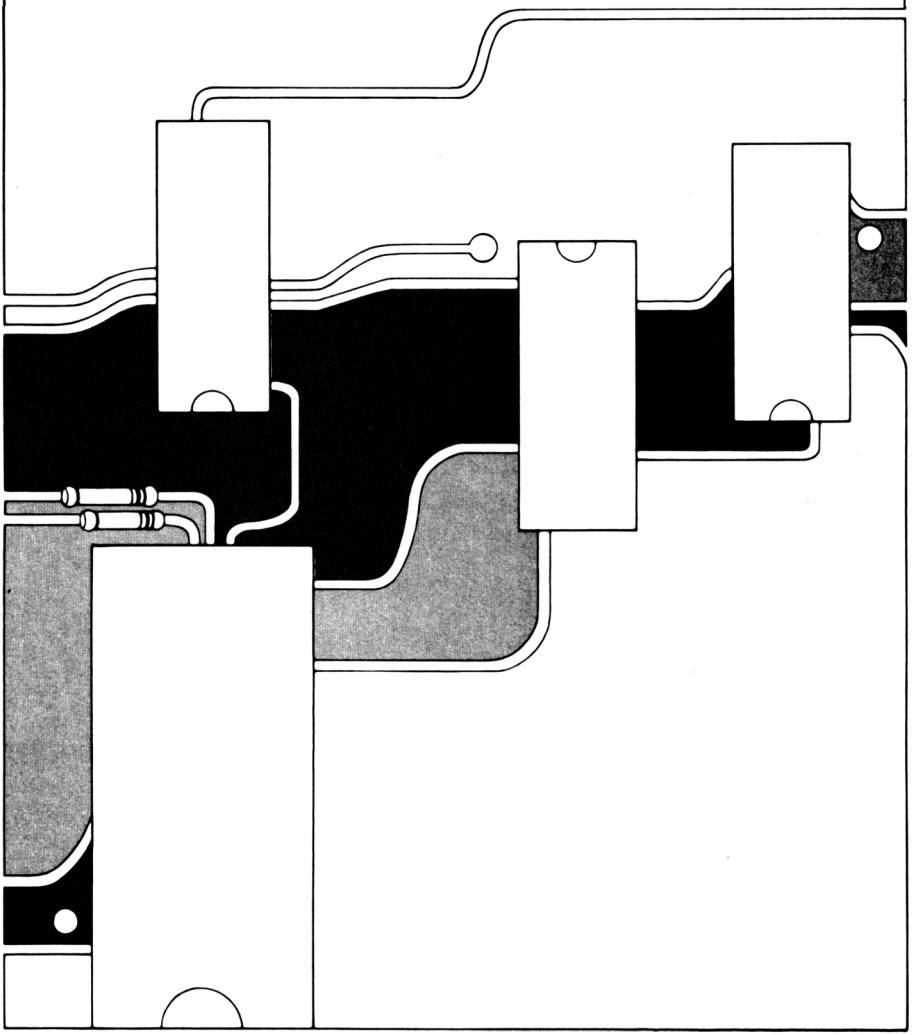


***Figure 1.22:** A typical memory map showing how the address space relates to different physical memory devices. Notice that this map requires four 2716, two 2732 and only one 2764 memory device.*

CHAPTER SUMMARY

In this chapter we have presented useful information regarding the use of ROM with the Z80. We have presented basic information concerning ROM operation and described the fundamental elements that give a reliable connection between the Z80 and ROM. The examples in this chapter have shown that using ROM with the Z80 can be a relatively simple task. There are certain guidelines, however, that must be followed.

Using Static RAM with the Z80



Chapter 2

INTRODUCTION

We will now continue with our study of the Z80 microprocessor by learning how to interface it with *random access memory,* also known as RAM. RAM is used in a system for temporary storage of programs, data, or variables. Unlike ROM (or read-only memory—discussed in Chapter 1), RAM is volatile and information is lost when power is turned off.

The electrical interface between the Z80 and static RAM is not a difficult one to realize. There are certain guidelines, however, that should be followed. In this chapter we will examine these guidelines and cover all the important interfacing topics relating to static RAM. We will also display and discuss two popular static RAM systems often used in industry. One is comprised of 2114, 1K by 4-bit memories, and the other of 6116, 1K by 8-bit memories.

2-1: Overview of Static RAM Communication

We shall begin our discussion by explaining how a semiconductor read and write memory is communicated with in general. Once you understand this process, you will find that it is an easy and straightforward jump to understanding how the Z80 communicates with RAM. If you are already familiar with this communication process, you may want to skip the first part of this chapter.

To start, a random access memory in a microprocessor system is capable of having data written to it and read from it by the CPU. Specific electrical signals are necessary for performing these two types of communication. Figure 2.1 displays a block diagram of a typical RAM device and shows the major electrical signals.

Let's first examine the power supply connections. A RAM must be powered up, and the most common voltages are +5 and ground. (We suggest that you refer to the manufacturer's data sheet for the exact voltages.) For our purposes, however, we will assume that the RAMs we are using require +5 volts and a ground.

In Figure 2.1, the lines labeled "Data In" are the physical wires that allow the electrical information to be written into the RAM. The lines labeled "Data Out" are the wires that allow the information stored in the RAM to be electrically read out. A RAM must have at least one data input line and one data output line. The exact number depends on the internal organization of the device.

In Chapter 1 we described the internal organization of ROM. Much of the same information and reasoning can be used to describe RAM. Like ROM, much information can be gleaned from the description of a RAM. For example, a RAM may be described as being a 256 × 1, or a 256 × 4, or a 1024 × 8, etc. The second number indicates the number of data input and

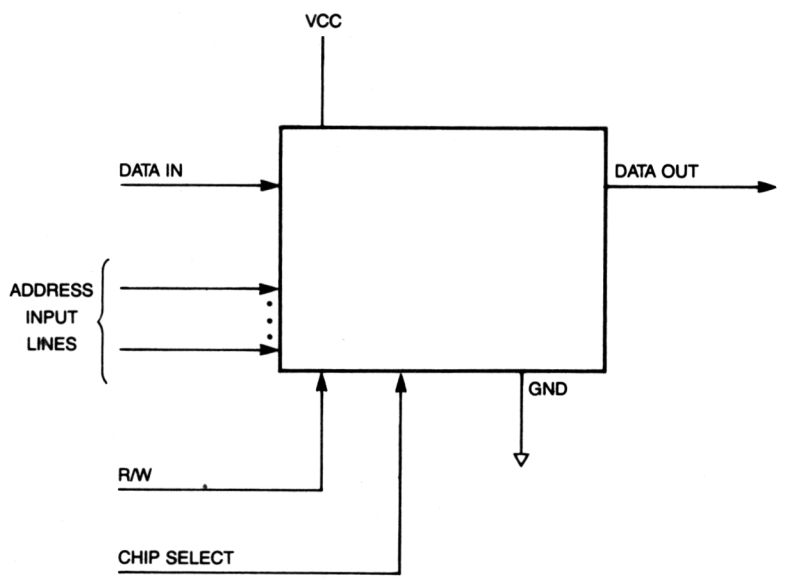


Figure 2.1: A block diagram showing the major electrical signals of a typical RAM device.

output lines the memory has. In other words, every time the memory is electrically written to or read from, the second number in the description determines the number of parallel data bits that are written or read.

Let's now examine the address input lines shown in Figure 2.1. (We previously examined these lines in Chapter 1 during our discussion of ROM.) Address lines of the static RAM have the electrical function of selecting the internal location of the memory device from which data will be read or written to. It is possible to determine the number of unique storage locations available in a RAM by simply counting the number of address lines. The converse of this is also true. That is, you can determine the number of address lines in a device by determining the stated number of unique storage locations. For example, a 1024 × 4 static RAM will have 1024 or 1K unique storage locations available, and at each storage location, 4 bits will be written or read at the same time.

The final signal line in the figure (labeled R/W) electrically determines whether the RAM will have data written to it or read from it.

Let's now turn our attention to the general sequence of events that occurs whenever any semiconductor memory is written to or read from, that is, whenever the microprocessor communicates with RAM. Let's first examine the events that occur during a read operation.

2-2: Sequence of Events for a RAM Read

1. First, the address lines are input to the memory. At this time the internal location from which the data will be read is logically decoded by the RAM.
2. The R/W control line is placed in the correct logical condition for a memory read. On some memories this is a logical 1; on others it is a 0. (The manufacturer's data sheets will help you determine the exact logical condition for a memory read for your system.)
3. The system must wait for a certain length of time, called *read access time,* to allow the internal circuits of the memory chip to decode the address and select the addressed data.
4. After the wait time, the data is available on the memory output lines, and can then be read by the system microprocessor. If the microprocessor reads the data too soon—that is, if it does not wait for the read access time—invalid data may be read from the device.

A timing diagram of this sequence appears in Figure 2.2. We shall refer back to this sequence when we examine communication between the Z80 and the semiconductor RAM. The sequence for reading data from memory

is exactly the same for RAM as it was for ROM. In fact, the microprocessor system does not electrically distinguish between reading data from ROM and reading it from RAM.

Let's now discuss the sequence of events that occurs during the writing of data to RAM.

2-3: Sequence of Events for a RAM Write

1. First, the address input lines to the memory are set to the logical conditions of the internal memory location to which data will be written.
2. Next, the data that will be written into the memory are placed on the data input lines.

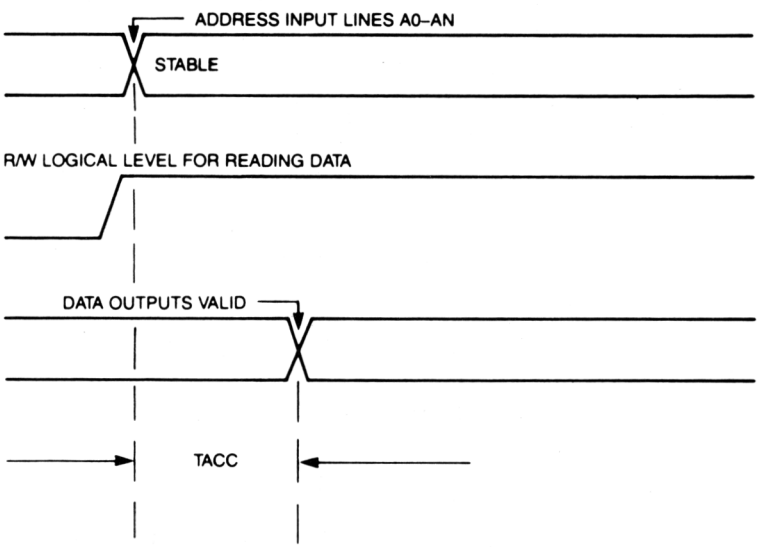


Figure 2.2: A timing diagram showing the general sequence of events for reading data from system memory.

3. The system must wait a finite amount of time, called *write access time* (usually less than a few hundred nanoseconds), to allow the internal decoding circuits to stabilize.
4. After the wait time, the R/W control line of the memory is set to the logical level, or pulsed, to allow the data present at the data input lines to be written into the RAM.

Figure 2.3 shows a timing diagram representation of this sequence of events. The hardware required to follow this sequence varies with each microprocessor.

2-4: A Real Memory Device

Let's now examine a typical RAM memory that is widely used in industry. We will begin by examining the important specifications of the device. We will then show how data is read from it and written to it. It is essential, when using RAMs, to understand the electrical specifications of the device. However, once you understand the workings of one RAM device in detail, it is easy to understand the workings of other RAMs used in other microprocessor applications.

The RAM that we have chosen for this discussion is the 2114, 1024 × 4, static, common I/O RAM. The term *common I/O* refers to the electrical configuration of the data *input* and *output* lines of the RAM. A RAM may

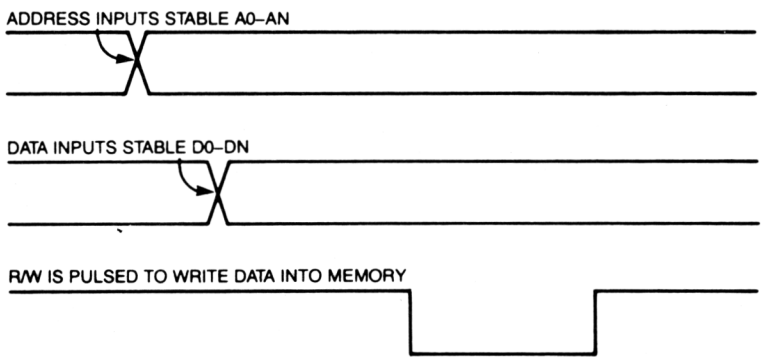


***Figure 2.3:** A timing diagram showing the sequence of events necessary when writing data to a typical RAM device.*

have either separate or common I/O. *Separate I/O* means that there are unique data input and output lines for the RAM (as shown in Figure 2.4). *Common I/O* means that the data input and output lines of the device are the same pin (see Figure 2.5).

Since RAM can only be written to and read from, only data input and output lines are needed. Further, these two functions are mutually exclusive: you can never read and write to memory at the same time. Thus, the input and output lines are never used at the same time. This fact allows for a reduction in the number of hardware pins needed to perform the read and write functions in RAM. It also allows for a common I/O. In addition, the data input and output lines can be *time-multiplexed*. This means that during

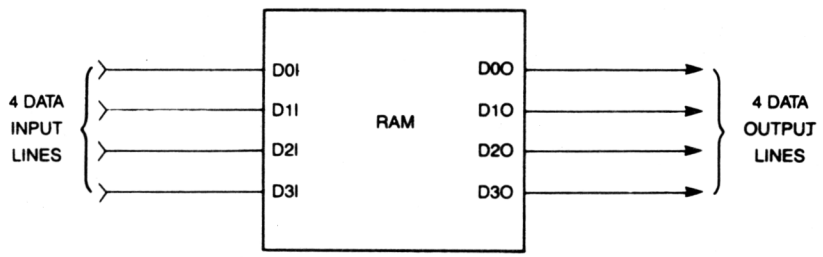


Figure 2.4: A block diagram of a RAM with separate input and output lines.

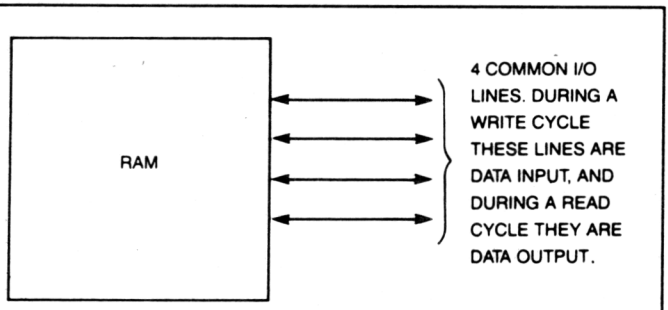


Figure 2.5: A block diagram of a RAM with common input and output lines.

a memory read operation, the data I/O lines are used as data outputs and during a write operation, they are used as inputs. The logical state of the R/W control line determines the definition of the data I/O pins at any given time for the memory. Figure 2.6 shows a pinout and functional block diagram for a 2114 memory device.

2114A 1024 X 4 BIT STATIC RAM

	2114AL-1	2114AL-2	2114AL-3	2114AL-4	2114A-4	2114A-5
Max. Access Time (ns)	100	120	150	200	200	250
Max. Current (mA)	40	40	40	40	70	70

- **HMOS Technology**
- **Low Power, High Speed**
- **Identical Cycle and Access Times**
- **Single +5V Supply ±10%**
- **High Density 18 Pin Package**
- **Completely Static Memory - No Clock or Timing Strobe Required**
- **Directly TTL Compatible: All Inputs and Outputs**
- **Common Data Input and Output Using Three-State Outputs**
- **2114 Upgrade**

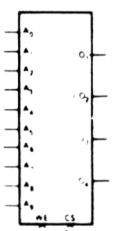
The Intel® 2114A is a 4096-bit static Random Access Memory organized as 1024 words by 4-bits using HMOS, a high performance MOS technology. It uses fully DC stable (static) circuitry throughout, in both the array and the decoding, therefore it requires no clocks or refreshing to operate. Data access is particularly simple since address setup times are not required. The data is read out nondestructively and has the same polarity as the input data. Common input/output pins are provided.

The 2114A is designed for memory applications where the high performance and high reliability of HMOS, low cost, large bit storage, and simple interfacing are important design objectives. The 2114A is placed in an 18-pin package for the highest possible density.

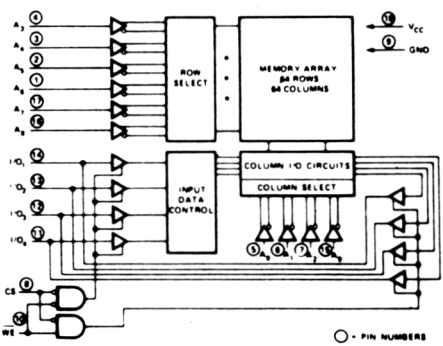
It is directly TTL compatible in all respects: inputs, outputs, and a single +5V supply. A separate Chip Select ($\overline{CS}$) lead allows easy selection of an individual package when outputs are or-tied.

PIN CONFIGURATION

LOGIC SYMBOL



BLOCK DIAGRAM



PIN NAMES

A_0–A_9	ADDRESS INPUTS	V_{CC}	POWER +5V
$\overline{WE}$	WRITE ENABLE	GND	GROUND
$\overline{CS}$	CHIP SELECT		
I/O_1–I/O_4	DATA INPUT OUTPUT		

Figure 2.6: A partial data sheet and block diagram for the 2114, 1K × 4, common I/O static RAM.

Let's now examine the important parameters of this memory. (*Note:* the information given in this section may be applied to most semiconductor memories, even if they do not use common I/O.)

For this example, we will consider the 2114 at the user level only. Further, we will assume that the memory access time is consistent with the overall system speed, and that we won't need to generate any wait states for slow memory.

Figure 2.7 shows a partial data sheet and general timing diagrams for electrical communication with the 2114 memory. Let's first discuss writing

A.C. CHARACTERISTICS T_A = 0°C to 70°C V_{CC} = 5V ± 10% unless otherwise noted

READ CYCLE [1]

SYMBOL	PARAMETER	2114AL-1 Min.	2114AL-1 Max.	2114AL-2 Min.	2114AL-2 Max.	2114AL-3 Min.	2114AL-3 Max.	2114A-4/L-4 Min.	2114A-4/L-4 Max.	2114A-5 Min.	2114A-5 Max.	UNIT
t_{RC}	Read Cycle Time	100		120		150		200		250		ns
t_A	Access Time		100		120		150		200		250	ns
t_{CO}	Chip Selection to Output Valid		70		70		70		70		85	ns
t_{CX}	Chip Selection to Output Active	10		10		10		10		10		ns
t_{OTD}	Output 3-state from Deselection		30		35		40		50		60	ns
t_{OHA}	Output Hold from Address Change	15		15		15		15		15		ns

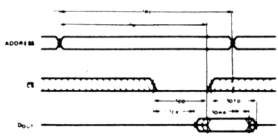
WRITE CYCLE [2]

SYMBOL	PARAMETER	2114AL-1 Min.	2114AL-1 Max.	2114AL-2 Min.	2114AL-2 Max.	2114AL-3 Min.	2114AL-3 Max.	2114A-4/L-4 Min.	2114A-4/L-4 Max.	2114A-5 Min.	2114A-5 Max.	UNIT
t_{WC}	Write Cycle Time	100		120		150		200		250		ns
t_W	Write Time	75		75		90		120		135		ns
t_{WR}	Write Release Time	0		0		0		0		0		ns
t_{OTW}	Output 3-state from Write		30		35		40		50		60	ns
t_{DW}	Data to Write Time Overlap	70		70		90		120		135		ns
t_{DH}	Data Hold from Write Time	0		0		0		0		0		ns

NOTES
1. A Read occurs during the overlap of a low $\overline{CS}$ and a high $\overline{WE}$.
2. A Write occurs during the overlap of a low $\overline{CS}$ and a low $\overline{WE}$. t_W is measured from the latter of $\overline{CS}$ or $\overline{WE}$ going low to the earlier of $\overline{CS}$ or $\overline{WE}$ going high.

WAVEFORMS

READ CYCLE ③



NOTES
3. $\overline{WE}$ is high for a Read Cycle.
4. If the $\overline{CS}$ low transition occurs simultaneously with the $\overline{WE}$ low transition, the output buffers remain in a high impedance state.
5. $\overline{WE}$ must be high during all address transitions.

WRITE CYCLE

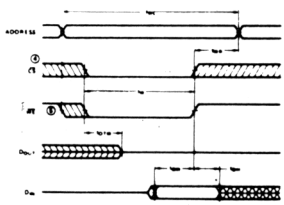


Figure 2.7: Excerpted data sheet showing the timing parameters for the 2114 memory device.

data to the memory. These events follow the general sequence described in Section 2.3 of this chapter. However, now we will describe the process in more detail and include the actual specifications of the memory under consideration.

2-5: Sequence of Events for Writing Data to the 2114

1. Address inputs are set to the address to which data will be written.
2. Data is applied to the data I/O lines of the device. These lines are tri-stated. If this were not the case, there would be a conflict between the data being applied to the memory and the data stored in memory; i.e., each would try to control the I/O line. Figure 2.8 shows a diagram of such a conflict.
3. Next, the $\overline{WE}$ and $\overline{CS}$ inputs to the memory are asserted at approximately the same instant in time. When the $\overline{WE}$ input is a logical 0, the I/O lines act as inputs. Therefore, the memory device should not be selected until the $\overline{WE}$ input is asserted. This is shown in Figure 2.9.

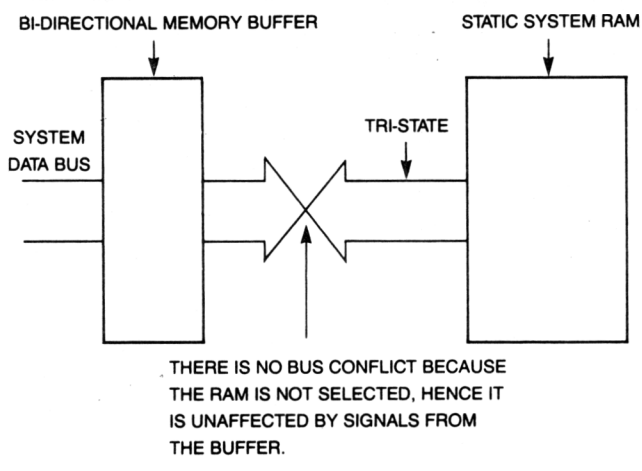


***Figure 2.8:** A block diagram of a possible bus conflict between the 2114 data output lines and the memory data buffer outputs. This conflict can be avoided through proper control of the memory data buffer direction control line.*

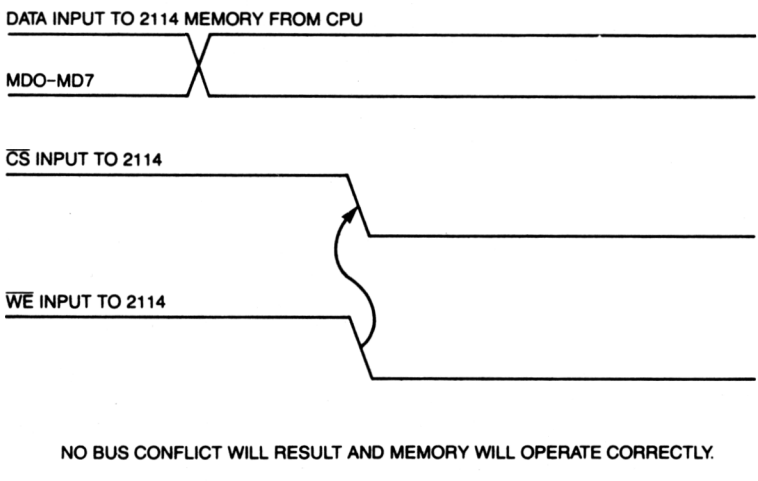


Figure 2.9: A timing diagram showing the correct way to control the memory data buffers. In this case, the bus conflict shown in Figure 2.8 will be avoided.

Now that we are familiar with the events that occur each time the 2114 memory has data written to it by the CPU, let's go on to explore the events that occur when the 2114 memory has data read from it.

2-6: Sequence of Events for Reading Data from the 2114

1. The address is input to the memory from which data will be read.

2. Next the $\overline{CS}$ is asserted. At this time the $\overline{WE}$ input line is a logical 1. The data I/O lines of the 2114 memory are enabled.

3. The data I/O lines output the data that has been previously stored at the address location.

We state again that the reading of data from static RAM works in the same way as the reading of data from ROM. That is, all of the same timing and buffering rules apply.

Now that we understand how the 2114 device works, let's connect it to the Z80 microprocessor.

2-7: Connecting the Address Lines to the Z80

To illustrate how the Z80 communicates with RAM, we will now present a complete design of a general-purpose RAM system. As we examine this system, we will explore potential problem areas that can arise when using RAM with the Z80. Although the information presented here is general, it is important information for anyone who wishes to use static RAM in a Z80 microprocessor application.

We can see how the Z80 address bus connects to the RAM system, by examining the system memory map. Figure 2.10 shows a partial map. A memory map can help determine where in the (possible) 64K address space the RAM is to reside. The map in Figure 2.10 shows the RAM memory space in locations 3000 to 3FFF in hexadecimal. This amount of memory space gives 4096 or 4K × 8 locations of RAM memory for the system.

For this example we will construct a 4K × 8 RAM memory that communicates with the Z80. If we examine the data sheet in Figure 2.6, we can see

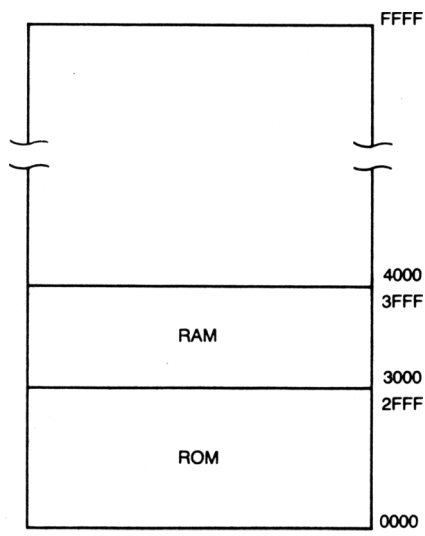


Figure 2.10: A partial memory map showing the ROM and RAM space of the system memory.

that each memory device has four I/O lines. Since the Z80 microprocessor communicates using eight data lines in parallel—that is, a byte at a time—we must use two 2114 memories in parallel, to form a complete 8-bit word. (See Figure 2.11.)

The first thing we must do then is to determine the number of memories needed to construct the entire RAM space. We can calculate this in the following way. The available RAM space is equal to 4096 bytes. Each 2114 memory stores up to 1024 nibbles; therefore, for each 1024 bytes of storage using 2114 memories, we need two physical devices. Thus, we need eight 2114 memory devices to realize the entire RAM space. Of course, we do not necessarily need to use all of the available memory space.

For this example, our Z80 system will require only 1K bytes of RAM. We will assume, however, that the system was designed with 4K bytes of memory space. (*Note:* it is normally a good idea to include more memory space

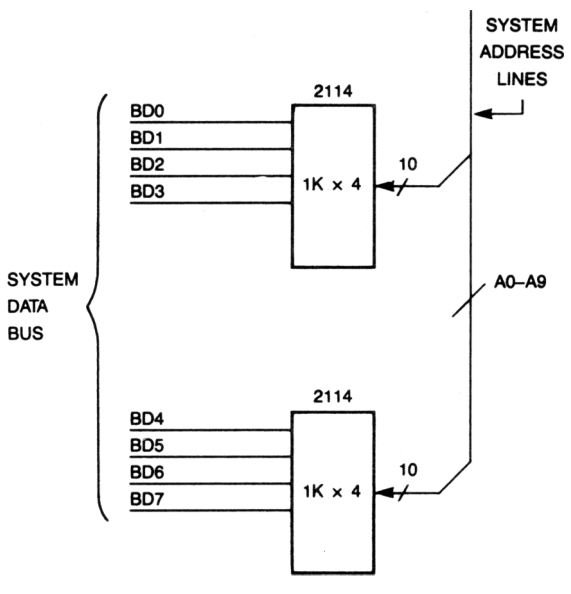


Figure 2.11: *Two 2114 memory devices are used in a parallel fashion to construct a complete 8-bit wide memory system.*

in your design than you feel you will use, if your system permits it. This allows you to easily expand the memory in the future, if your application requires it. Bear in mind that adding memory to a system that has not been designed with that concept in mind can be a difficult task.)

We now know that our system requires eight 2114 memories to realize the entire memory space. Each 2114 has 10 address input lines, A0–A9; and all of these lines must be connected together in a parallel fashion. That is, A0 of all eight devices must be connected; A1 of all eight devices must be connected, and so on. (See Figure 2.12.)

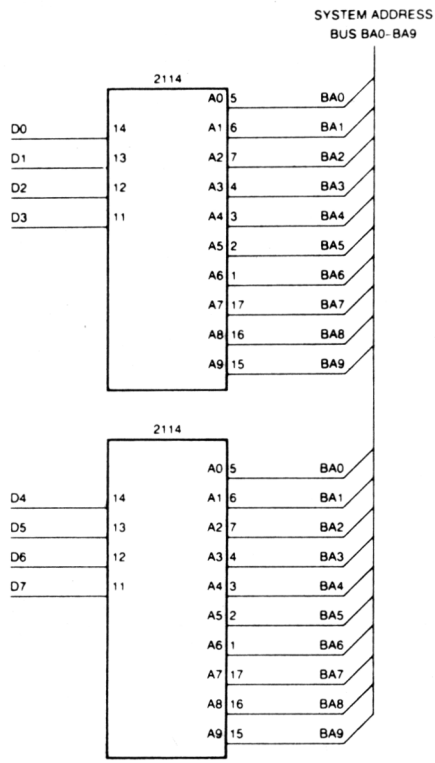


***Figure 2.12:** Address input lines A0–A9 are connected to each memory in a parallel fashion.*

The remaining address bus lines, A10-A15, are used to decode the RAM memory space from all other system memory space. Two levels of decoding are used. Address lines A10 and A11 will be used to select the correct pair of 2114 memories, and address lines A11-A15 will be used to select the memory space C000-D000 from all available memory space.

There are several different techniques that can be used to decode address lines for any particular application. The circuit shown in Figure 2.13 is a typical one. It is provided to give you a general idea of how logical memory selection can be accomplished. Although these circuits do work, they are not meant to be the ultimate solution. All solutions must be tailored to the system application and normally depend on the system speed, bus loading and final memory map. Figure 2.13 shows the hardware required to realize the two-level decoding of the RAM space for this example.

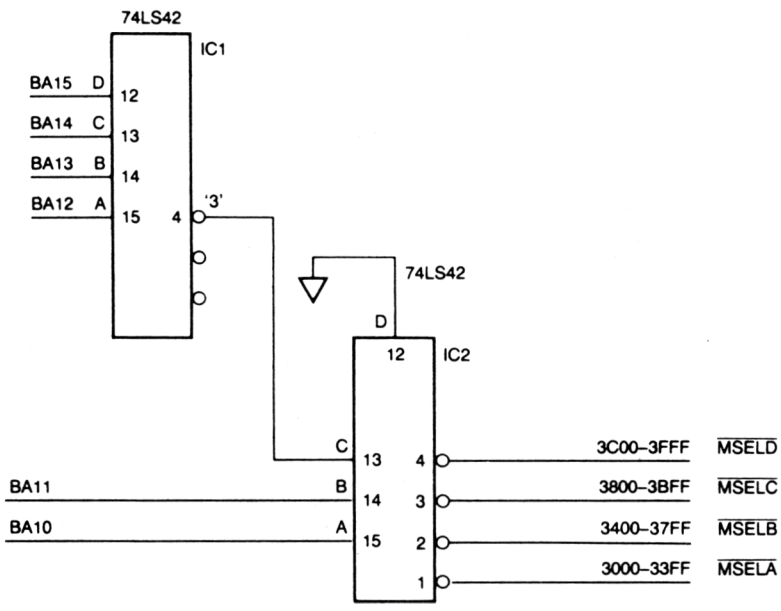


***Figure 2.13:** A schematic diagram showing the decoding circuits for realizing the memory select logic.*

Note here that address buffers may or may not be used. Recall that we explained the use of address buffers in Chapter 1. The same loading and speed rules apply to both the RAM and ROM devices.

2-8: Connecting the Data Lines—Non-Buffered

We will now show you how to connect the memory data outputs to the Z80 data bus. For this example we will use no data buffers; we will assume that the loading on the Z80 data bus lines does not exceed the specified output drive capability of the memory data I/O lines. Thus, we will simply connect the 2114 memory outputs in a parallel fashion, and then connect the resulting eight data I/O lines to the Z80 data bus. The connection is straightforward. The process is shown in Figure 2.14.

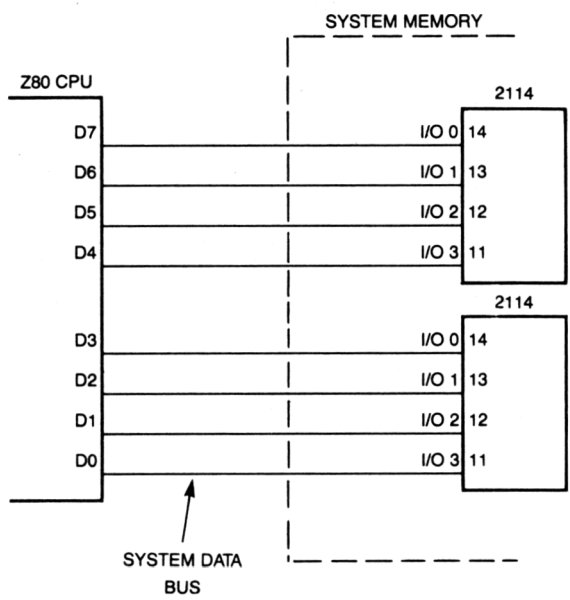


Figure 2.14: A diagram showing how the data lines are physically connected between the 2114 and the Z80 microprocessor.

It is important to recall that the I/O lines of the 2114 memory system are controlled by the $\overline{WE}$ and the $\overline{CS}$ inputs to the device. If your application is such that data buffers are not required, a simple direct connection from the memory output lines to the Z80 data bus lines is all that is needed.

2-9: Generating the Memory Read and Write Control Lines

When the Z80 microprocessor communicates electrically with memory, there are three physical output lines, $\overline{MREQ}$, $\overline{RD}$ and $\overline{WR}$, that are used to generate the memory control lines labeled $\overline{MEMR}$ and $\overline{MEMW}$. Figure 2.15 shows a typical logic circuit that generates a system memory read and write signal, based on the logic state of the three Z80 output lines.

We will now use the output lines in Figure 2.15 with the memory address decoding circuits in Figure 2.13 to control the $\overline{WE}$ and $\overline{CS}$ inputs to the system RAM. (*Note:* recall that the information presented here is general and applies *not* only to the 2114 memory device but to most static memory devices. The 2114 is used simply as a means of transferring information.)

Figure 2.16 shows how the $\overline{MEMR}$ and $\overline{MEMW}$ lines connect with the 2114 memory $\overline{WE}$ and $\overline{CS}$ inputs. This circuit will assert the $\overline{WE}$ and $\overline{CS}$ inputs to

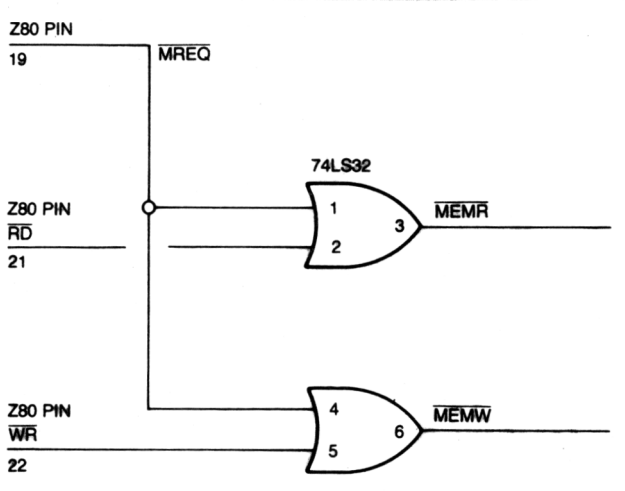


Figure 2.15: *The logic circuits for generating the system memory read and memory write control signals.*

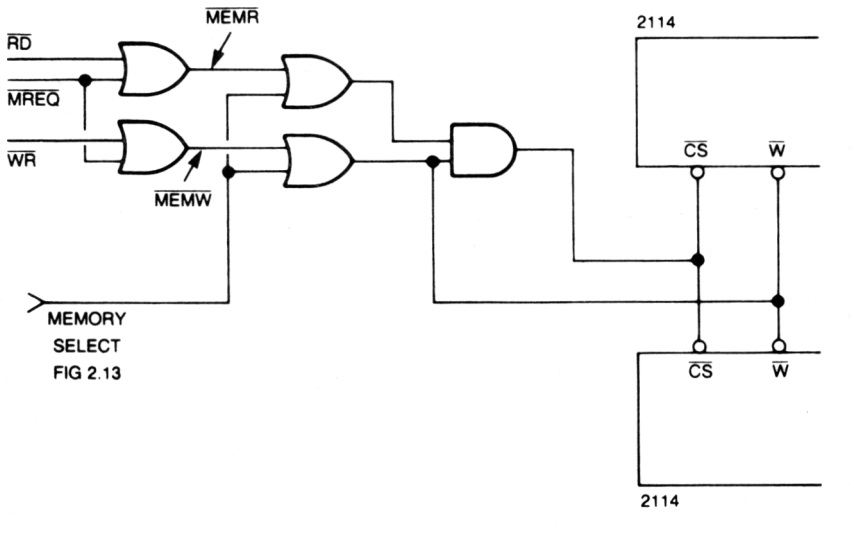


Figure 2.16: A schematic diagram used for asserting the $\overline{CS}$ and $\overline{WE}$ inputs to the 2114 devices.

the memory at approximately the same instant that a memory write occurs. During the memory read operation, however, only the $\overline{CS}$ input will be asserted to the 2114 memory.

2-10: Using Buffered Data Lines with Static RAM

In the previous section we connected the 2114 memory I/O lines directly to the Z80 data lines. This connection is electrically valid if the overall system application does not require data buffering on the Z80 data bus. We will now, however, assume that your application requires a bi-directional data buffer on the Z80 data bus line. If this is the case, a data buffer must be installed on the memory data output lines. This is due to the fact that during a memory read operation, the memory data outputs must electrically drive the entire data bus load.

Figure 2.17 shows one way to install data buffers on a static RAM system. Note that the 2114 memory I/O lines are all connected as before; however they are not connected directly to the Z80 data lines. Instead, they are connected to one side of a bi-directional buffer. The direction control for the

data buffer is dependent on the logical state of the $\overline{MEMR}$ signal. Whenever this signal is a logical 0 and the memory space is enabled, the memory data buffers will then buffer data from the RAM outputs to the system data bus.

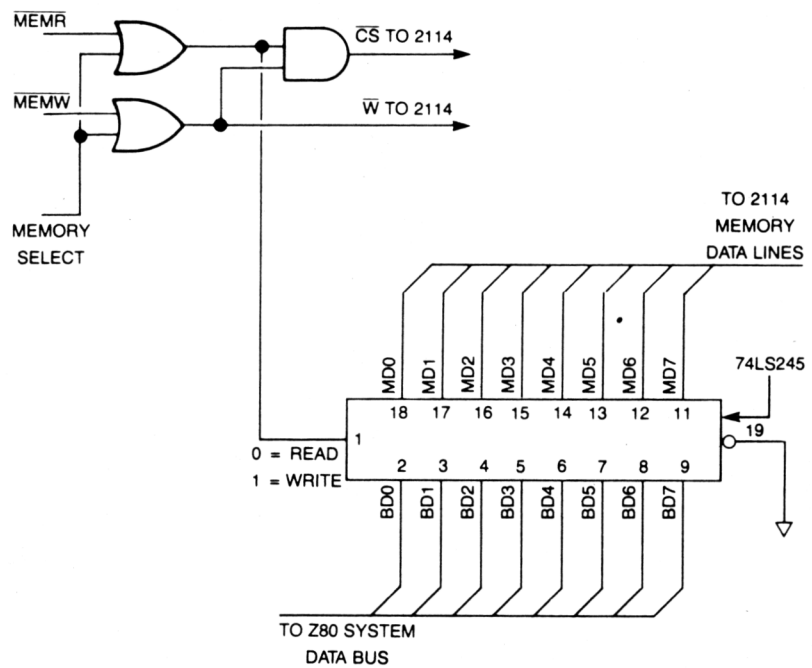


Figure 2.17: A partial schematic for using memory data buffers for the 2114 device. Notice that a 74LS245 bi-directional buffer is used. This is due to the common I/O characteristics of the memory device.

2-11: Complete 4K × 8-Bit Static RAM

Figure 2.18 shows a complete static RAM system that will communicate with the Z80 microprocessor. Note that this system includes a memory data buffer. If your application does not require it, you can eliminate it by simply connecting the MD lines directly to the system data bus lines.

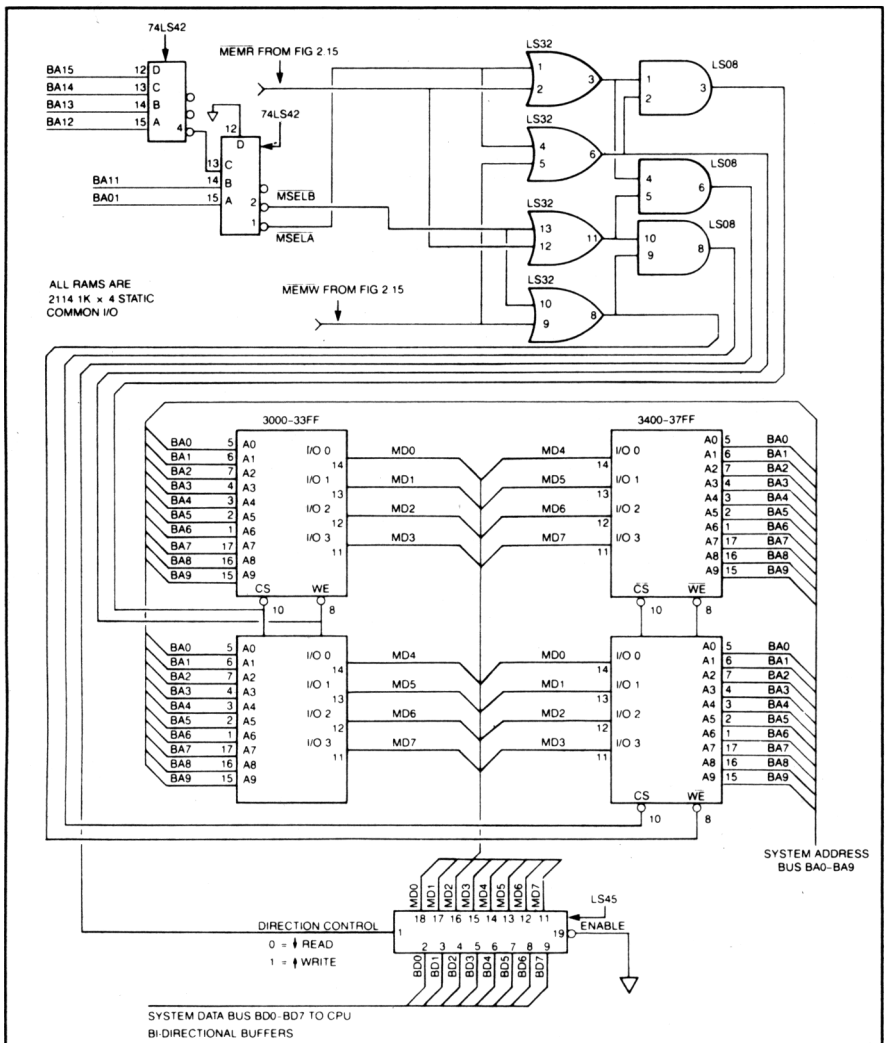


***Figure 2.18:** A complete schematic diagram for a 2K × 8-bit memory system for the Z80 microprocessor.*

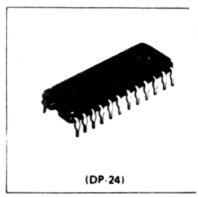
2-12: The 6116: Another Static RAM Device

In this section we will present another popular static RAM that can be easily used in a Z80-controlled system: the 6116—a 2K × 8, common I/O static RAM. Figure 2.19 shows a partial data sheet for this device. The 6116 has operating characteristics that are similar to the 2114 static RAM.

HM6116P-2, HM6116P-3, HM6116P-4

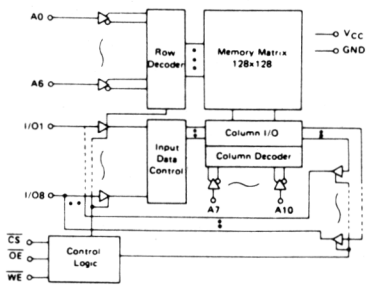
2048-word X 8-bit High Speed Static CMOS RAM

- **FEATURES**
- Single 5V Supply and High Density 24 pin Package
- High Speed: Fast Access Time 120ns/150ns/200ns (max.)
- Low Power Standby and Low Power Operation: Standby: 100μW (typ.) Operation: 180mW (typ.)
- Completely Static RAM: No clock or Timing Strobe Required
- Directly TTL Compatible: All Input and Output
- Pin Out Compatible with Standard 16K EPROM/MASK ROM
- Equal Access and Cycle Time

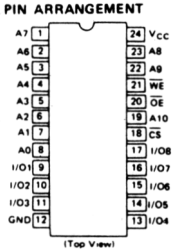


(DP-24)

- **FUNCTIONAL BLOCK DIAGRAM**



- **PIN ARRANGEMENT**



- **ABSOLUTE MAXIMUM RATINGS**

Item	Symbol	Rating	Unit
Voltage on Any Pin Relative to GND	V_{IN}	−0.5 to +7.0	V
Operating Temperature	T_{opr}	0 to +70	°C
Storage Temperature	T_{stg}	−55 to +125	°C
Temperature Under Bias	T_{bias}	−10 to +85	°C
Power Dissipation	P_T	1.0	W

- **TRUTH TABLE**

$\overline{CS}$	$\overline{OE}$	$\overline{WE}$	Mode	V_{CC} Current	I/O Pin	Ref. Cycle
H	X	X	Not Selected	I_{SB}, I_{SB1}	High Z	
L	L	H	Read	I_{CC}	D_{out}	Read Cycle (1) ~ (3)
L	H	L	Write	I_{CC}	D_{in}	Write Cycle (1)
L	L	L	Write	I_{CC}	D_{in}	Write Cycle (2)

Figure 2.19: A data sheet for the 6116—a 2K by 8-bit, common I/O static RAM.

■ **RECOMMENDED DC OPERATING CONDITIONS** (T_a = 0 to +70°C)

Item	Symbol	min	typ	max	Unit
Supply Voltage	V_{CC}	4.5	5.0	5.5	V
	GND	0	0	0	V
Input Voltage	V_{IH}	2.2	3.5	6.0	V
	V_{IL}	−1.0*	–	0.8	V

* Pulse Width: 50 ns, DC: V_{IL} min. = −0.3V

■ **DC AND OPERATING CHARACTERISTICS** (V_{CC} = 5V ±10%, GND = 0V, T_a = 0 to +70°C)

Item	Symbol	Test Conditions	HM6116P-2			HM6116P-3/-4			Unit
			min	typ*	max	min	typ*	max	
Input Leakage Current	$\|I_{LI}\|$	V_{CC} = 5.5V, V_{in} = GND to V_{CC}	–	–	10	–	–	10	μA
Output Leakage Current	$\|I_{LO}\|$	$\overline{CS}$ = V_{IH} or $\overline{OE}$ = V_{IH} $V_{I/O}$ = GND to V_{CC}	–	–	10	–	–	10	μA
Operating Power Supply Current	I_{CC}	$\overline{CS}$ = V_{IL}, $I_{I/O}$ = 0mA	–	40	80	–	35	70	mA
	I_{CC1}**	V_{IN} = 3.5V, V_{IL} = 0.6V, $I_{I/O}$ = 0mA	–	35	–	–	30	–	mA
Average Operating Current	I_{CC2}	Min. cycle, duty = 100%	–	40	80	–	35	70	mA
Standby Power Supply Current	I_{SB}	$\overline{CS}$ = V_{IH}	–	5	15	–	5	15	mA
	I_{SB1}	$\overline{CS} \geq V_{CC}$ −0.2V, $V_{in} \geq V_{CC}$ −0.2V or $V_{in} \leq 0.2V$	–	0.02	2	–	0.02	2	mA
Output Voltage	V_{OL}	I_{OL} = 4mA	–	–	0.4	–	–	–	V
		I_{OL} = 2.1mA	–	–	–	–	–	0.4	
	V_{OH}	I_{OH} = −1.0mA	2.4	–	–	2.4	–	–	V

* V_{CC} = 5V, T_a = 25°C
** Reference Only

■ **AC CHARACTERISTICS** (V_{CC} = 5V ±10%, T_a = 0 to +70°C)

● **AC TEST CONDITIONS**

Input Pulse Levels: 0.8 to 2.4V
Input Rise and Fall Times: 10 ns

Input and Output Timing Reference Levels: 1.5V
Output Load: 1TTL Gate and C_L = 100pF (including scope and Jig)

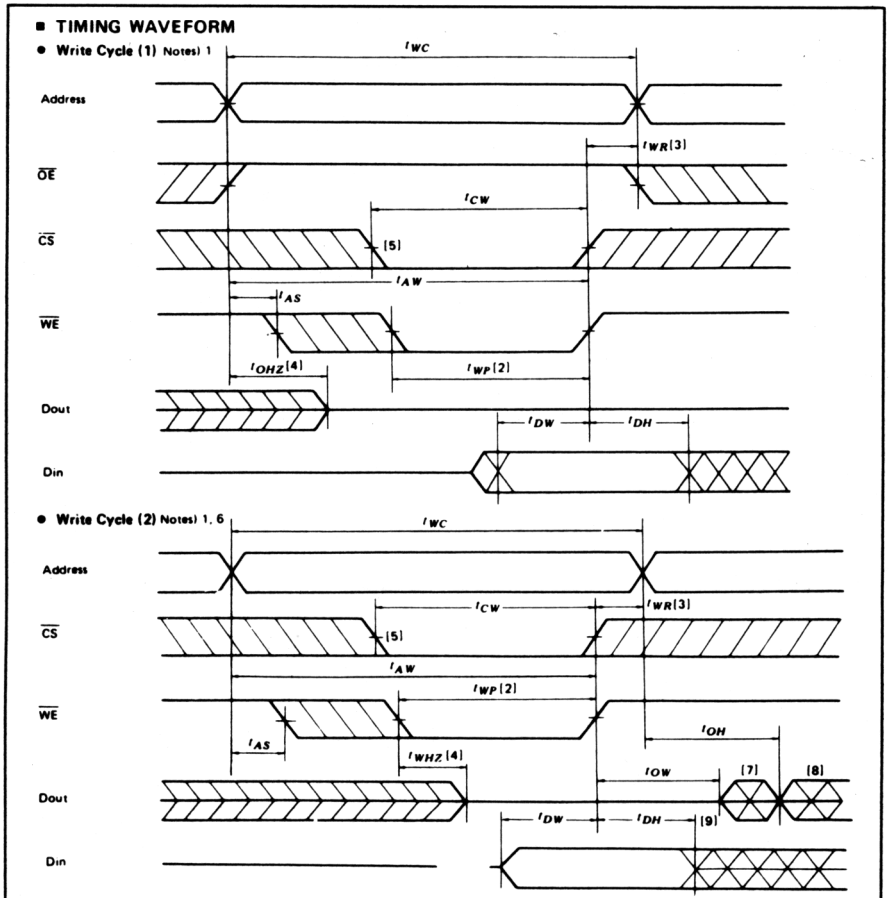
● **READ CYCLE**

Item	Symbol	HM6116P-2		HM6116P-3		HM6116P-4		Unit
		min	max	min	max	min	max	
Read Cycle Time	t_{RC}	120	–	150	–	200	–	ns
Address Access Time	t_{AA}	–	120	–	150	–	200	ns
Chip Select Access Time	t_{ACS}	–	120	–	150	–	200	ns
Chip Selection to Output in Low Z	t_{CLZ}	10	–	15	–	15	–	ns
Output Enable to Output Valid	t_{OE}	–	80	–	100	–	120	ns
Output Enable to Output in Low Z	t_{OLZ}	10	–	15	–	15	–	ns
Chip deselection to Output in High Z	t_{CHZ}	0	40	0	50	0	60	ns
Chip Disable to Output in High Z	t_{OHZ}	0	40	0	50	0	60	ns
Output Hold from Address Change	t_{OH}	10	–	15	–	15	–	ns

● **WRITE CYCLE**

Item	Symbol	HM6116P-2		HM6116P-3		HM6116P-4		Unit
		min	typ	min	max	min	max	
Write Cycle Time	t_{WC}	120	–	150	–	200	–	ns
Chip Selection to End of Write	t_{CW}	70	–	90	–	120	–	ns
Address Valid to End of Write	t_{AW}	105	–	120	–	140	–	ns
Address Set Up Time	t_{AS}	20	–	20	–	20	–	ns
Write Pulse Width	t_{WP}	70	–	90	–	120	–	ns
Write Recovery Time	t_{WR}	5	–	10	–	10	–	ns
Output Disable to Output in High Z	t_{OHZ}	0	40	0	50	0	60	ns
Write to Output in High Z	t_{WHZ}	0	50	0	60	0	60	ns
Data to Write Time Overlap	t_{DW}	35	–	40	–	60	–	ns
Data Hold from Write Time	t_{DH}	5	–	10	–	10	–	ns
Output Active from End of Write	t_{OW}	5	–	10	–	10	–	ns

Figure 2.19: Data sheet continued.



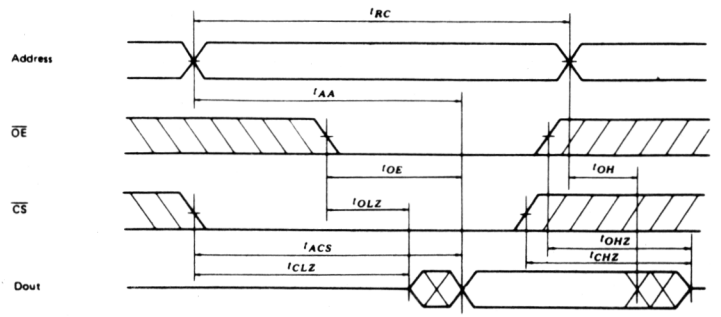
NOTES: 1. $\overline{WE}$ must be high during all address transitions.
2. A write occurs during the overlap (t_{WP}) of a low $\overline{CS}$ and a low $\overline{WE}$.
3. t_{WR} is measured from the earlier of $\overline{CS}$ or $\overline{WE}$ going high to the end of write cycle.
4. During this period, I/O pins are in the output state so that the input signals of opposite phase to the outputs must not be applied.
5. If the $\overline{CS}$ low transition occurs simultaneously with the $\overline{WE}$ low transitions or after the $\overline{WE}$ transition, output remain in a high impedance state.
6. $\overline{OE}$ is continuously low. ($\overline{OE}$ = V_{IL})
7. D_{out} is the same phase of write data of this write cycle.
8. D_{out} is the read data of next address.
9. If $\overline{CS}$ is Low during this period, I/O pins are in the output state. Then the data input signals of opposite phase to the outputs must not be applied to them.

Figure 2.19: Data sheet continued.

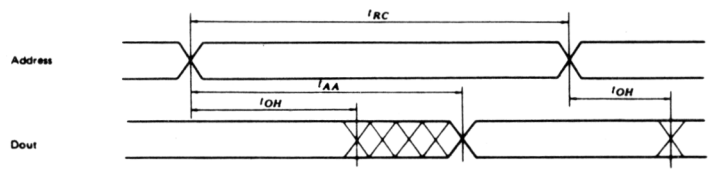
■ **CAPACITANCE** (f = 1MHz, T_a = 25°C)

Item	Symbol	Test Conditions	typ.	max.	Unit
Input Capacitance	C_{in}	V_{in} = 0V	3	5	pF
Input/Output Capacitance	$C_{I/O}$	$V_{I/O}$ = 0V	5	7	pF

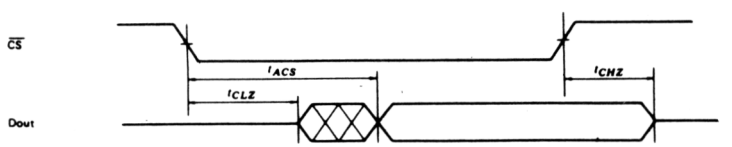
● **Read Cycle (1)** Notes) 1, 5



● **Read Cycle (2)** Notes) 1, 2, 4, 5



● **Read Cycle (3)** Notes) 1, 3, 4, 5



NOTES: 1. $\overline{WE}$ is High for Read Cycle.
2. Device is continuously selected, $\overline{CS}$ = V_{IL}.
3. Address Valid prior to or coincident with $\overline{CS}$ transition Low.
4. $\overline{OE}$ = V_{IL}.
5. When $\overline{CS}$ is Low, the address input must not be in the high impedance state.

Figure 2.19: Data sheet continued.

Figure 2.20 shows one way of connecting the 6116 RAM to a Z80-controlled system. Because of the organization of the device, only one device is required for a complete RAM section of memory. This fact makes the 6116 an ideal choice for both small and large system applications.

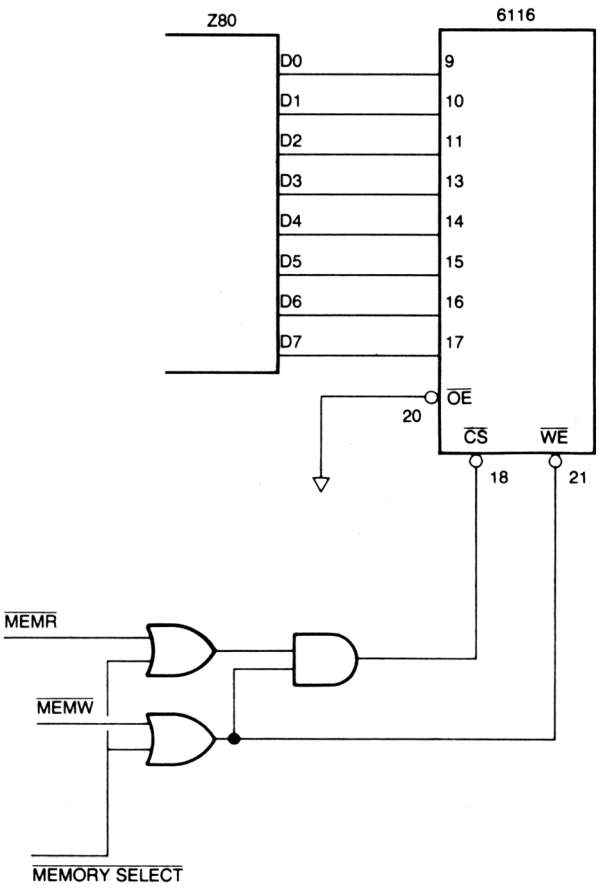
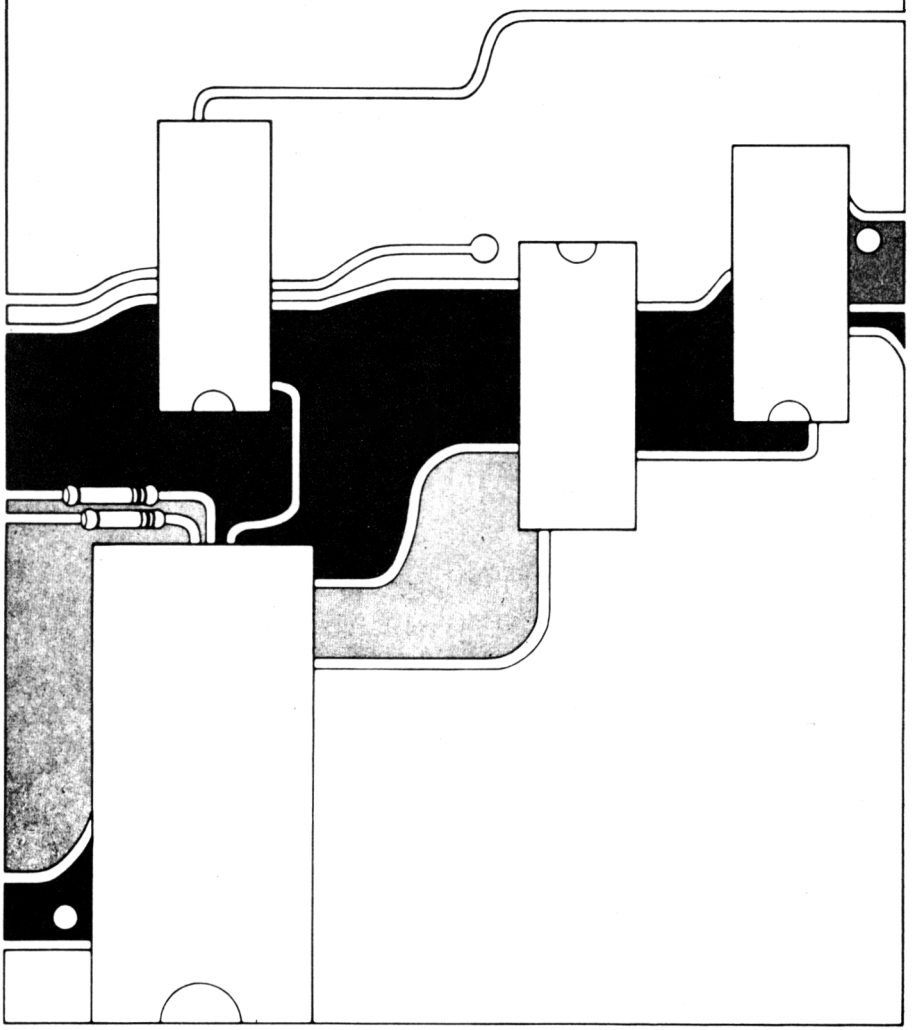


Figure 2.20: A schematic diagram showing the interconnections between the 6116 and the Z80 microprocessor.

CHAPTER SUMMARY

In this chapter we have presented important information regarding the use of static RAM with the Z80 microprocessor. Our discussion began with a description of a general sequence of events for a memory read and write operation. We then examined a typical RAM system comprised of 2114, 1K × 4, common I/O memories. Important memory parameters were presented and details for reading and writing to memory were shown. We then went on to examine a complete 2K × 8 memory system that uses 2114 memories as the physical memory devices. After that, we realized the same memory system with a 6116 memory device.

Z80
Input and Output



Chapter 3

INTRODUCTION

Two major hardware operations that a microprocessor performs electrically are reading data from an input device and writing data to an output device. In this chapter, we will learn how the Z80 communicates electrically with input and output devices. In addition, we will construct a general input and output (I/O) port using discrete logic devices.

We have chosen to discuss input/output operations at this point because electrical communication with I/O is similar to that of static RAM. In fact, as you progress through this chapter you will see strong similarities in communication between the microprocessor and static RAM, and the microprocessor and I/O devices.

The information presented in this chapter is important. You will need a basic understanding of I/O to understand the material presented in the chapters that follow.

3-1: Overview of Z80 Input and Output

The Z80 can input data from and output data to an I/O device. These two types of operations are shown in Figures 3.1a and 3.1b. Whenever the Z80 performs an I/O operation, a certain sequence of electrical events occurs in the system hardware; that is, certain Z80 address, data, and control lines become asserted in a specified order. This sequence is similar to the electrical events that occur when reading and writing data to memory.

There are certain software instructions that, when executed by the Z80, start the sequence of electrical events for an I/O operation. (We will examine these instructions in detail later on when we learn about interfacing different I/O devices to the Z80.) But, no matter which software instructions are used, the sequence of events for the system hardware is always the same.

As shown in Figures 3.1a and 3.1b, the Z80 can input (read) eight bits of data from an input device and output (write) eight bits of data to an output device. Let's now examine this subject in more detail.

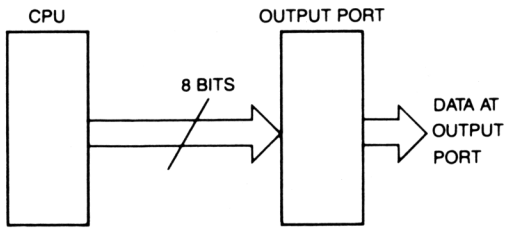


Figure 3.1a: A block diagram showing a microprocessor outputting eight bits of parallel data.

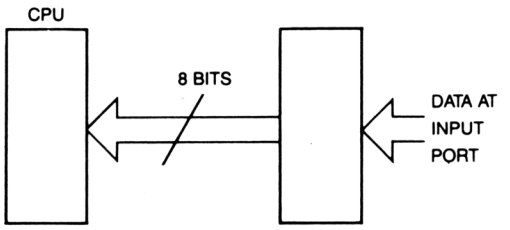


Figure 3.1b: A block diagram showing a microprocessor inputting eight bits of parallel data from an external source.

3-2: Port Address

In a Z80 system, eight of the sixteen address lines are used for I/O. This means that the Z80 can support up to 256, 8-bit I/O ports. (This is assuming that the system is using standard I/O mapped architecture and that each port has a unique address. If *linear select I/O* is used (where a single address line is used as the I/O select line) there are eight unique I/O ports. Linear select I/O is very useful in a system that has a small number of I/O devices.

(*Note:* in this chapter, we will discuss I/O mapped I/O. We believe that once you understand this type of I/O architecture, you can easily understand other types, since they are simply a variation of this general case.)

Recall from Chapter 2 that I/O addressing and memory addressing in a Z80 system use the same address lines. It is up to the microprocessor to electrically separate memory requests from I/O requests. In addition, we know that system memory does not always use all sixteen address lines (A0–A15). The same is true for the I/O system—it often does not use all eight address lines (A0–A7). For example, if there are only five different I/O ports in the system, only three address lines will be needed (since three lines will give a total of eight possible combinations). The general I/O port that we will examine in this chapter will, however, use all eight address lines for port decoding.

Recall that each I/O port in a microprocessor system will respond to a unique combination of eight bits on the system address lines A0–A7. The address combination that a port will respond to electrically is known as the *port address.* FF is the port address for the I/O device that we will be discussing in this chapter. Figure 3.2 shows a schematic diagram of a circuit that will detect this—and only this—port address.

In the diagram in Figure 3.2 we can see that the output pin 8 of the 74LS30 is a logical 0 if, and only if, all input lines are a logical 1. Note that the output pin 8 of the 74LS30 (labeled *port select line*) is active logical 0 whenever the system address bus is logically equal to the unique select code of the port. In this example the port select line is active logical 0.

It is important to remember that the port select line can become active even when there is no I/O communication occurring in the system. This is because system memory uses the same address lines. As an example, let's

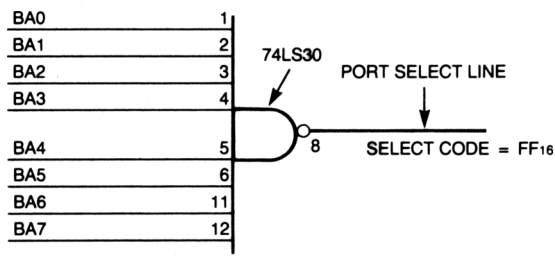


Figure 3.2: *A schematic diagram of a circuit that detects FF on the address bus as the port address.*

assume that the microprocessor is reading data from memory address XXFFh and the port select line suddenly becomes active. This can occur because the address lines A0–A7 are equal to the port select code. In other words, if the port select line in your system begins to signal that it is being selected at a time when software instructions are not indicating system communication with the port—do not be alarmed—the condition is valid.

3-3: Generation of the $\overline{IOW}$ and $\overline{IOR}$ Control Lines

Whenever the Z80 is performing I/O operations, the output control line labeled $\overline{IORQ}$ is asserted. (Note that this line is similar to the $\overline{MREQ}$ output line discussed previously.) The two timed-control lines, $\overline{RD}$ and $\overline{WR}$, are then logically combined with $\overline{IORQ}$ to provide the $\overline{IOR}$ (I/O read) and $\overline{IOW}$ (I/O write) system control line.

Figure 3.3 presents a schematic showing how the $\overline{IOR}$ and $\overline{IOW}$ system control signals are generated in a Z80 system. Note that these signals are active during all input read or output write operations, as shown in Figures 3.4a and 3.4b.

The port write signal for a Z80 system is defined as "the write enable strobe for a selected output port." This signal is generated whenever the port

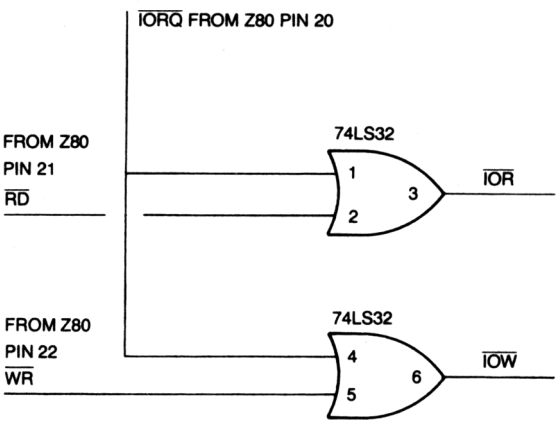


Figure 3.3: A schematic diagram of the hardware required to generate the $\overline{IOR}$ and $\overline{IOW}$ system control signals.

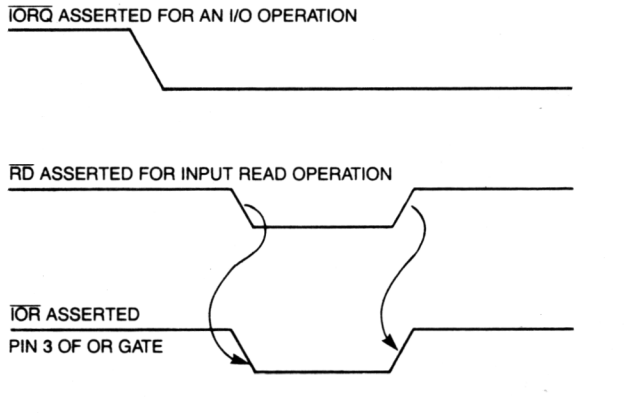


Figure 3.4a: *A timing diagram showing how the circuit of Figure 3.3 responds during an input read operation.*

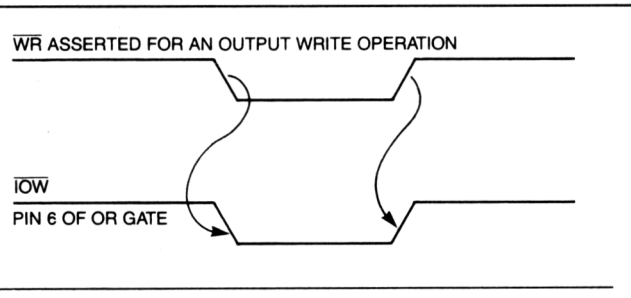


Figure 3.4b: *A timing diagram showing how the circuit of Figure 3.3 responds during an output write operation.*

select line and the $\overline{\text{IOW}}$ control signal are both active. The port write signal provides the active digital signal that specifies that the data from the microprocessor is to be latched or written to an output port.

The timing diagram in Figure 3.5 shows the general timing sequence that occurs when data is written to an output port. In this figure, all of the signals except $\overline{\text{WR}}$ can be thought of as static logic levels. The signal voltage

levels remain stable for the entire hardware operation. We have used this concept and the timing diagram in Figure 3.5 to create the schematic in Figure 3.6. This schematic shows one way that the port write strobe can be generated using hardware.

Let's examine this diagram and see how the circuit operates. Understanding this operation will help you to better understand how each Z80 signal is used. (Remember, however, that there are other ways that a port strobe can be generated.)

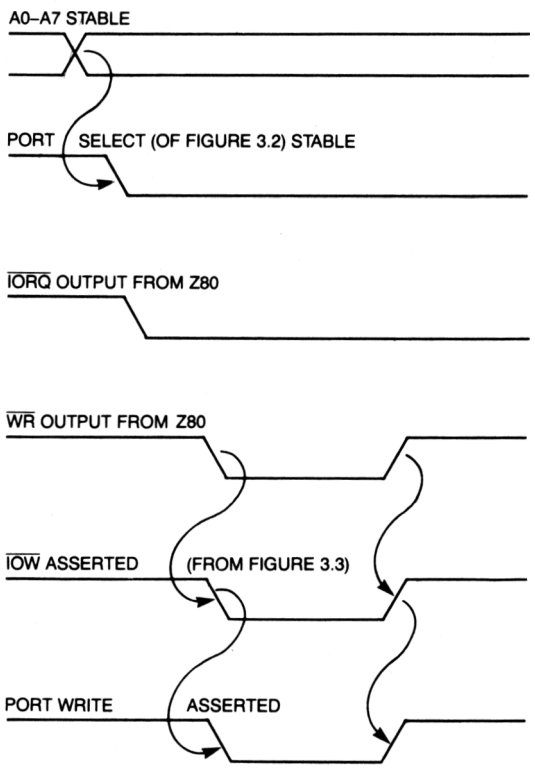


Figure 3.5: A general timing sequence of the events that occur during an output write operation.

The electrical objective of the circuit in Figure 3.6 is to provide an active logical 0 strobe whenever the CPU is writing data to the specified output port. In this figure, the system address lines A0–A7 are input to an 8 input NAND gate. This is the same circuit that was shown in Figure 3.2.

When all of the address lines (A0–A7) are a logical 1 (which is the port select code OFF), the output of the NAND gate is a logical 0. This port select line is connected to one input of the 74LS32 OR gate. The other input of the OR gate is connected to the $\overline{\text{IOW}}$ strobe signal (displayed in Figure 3.3). When the $\overline{\text{IOW}}$ strobe signal is a logical 0, it is indicating electrically that the CPU is performing the function of outputting a byte to a system output port.

Therefore, if the $\overline{\text{IOW}}$ signal is a logical 0 and the port select is a logical 0, the CPU must be electrically outputting data to the selected output port. In effect, we have electrically qualified the $\overline{\text{IOW}}$ signal with the port select line. The result is a unique active logical 0 strobe that occurs if, and only if, the Z80 CPU is performing an output operation to the system output port FF. This resulting strobe signal is called the *port write strobe*. It is used to strobe data into the selected output port.

Note that this is only one way that an output port strobe can be realized. The sequence of electrical events and the concept of qualifying the $\overline{\text{IOW}}$ strobe are, however, common to most microprocessor systems.

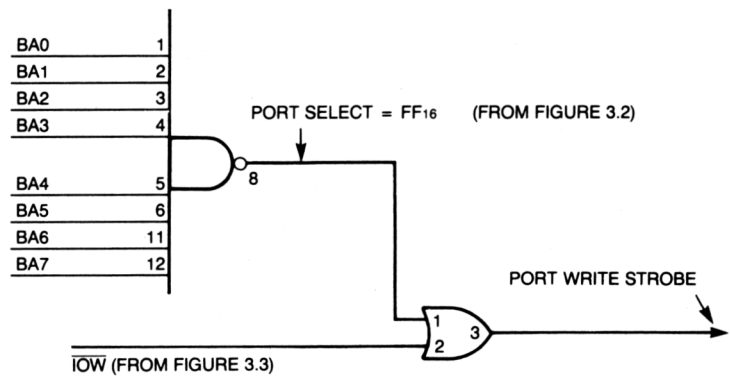


Figure 3.6: *A schematic diagram showing the hardware required to realize the port write control signal. Note that this signal is a qualified $\overline{\text{IOW}}$ control signal. It is qualified by the port address select line.*

3-4: Generation of the Port Read Signal

Let's now explain how the Z80 reads data from an 8-bit input port. The timing diagram displayed in Figure 3.7 shows a general sequence of electrical

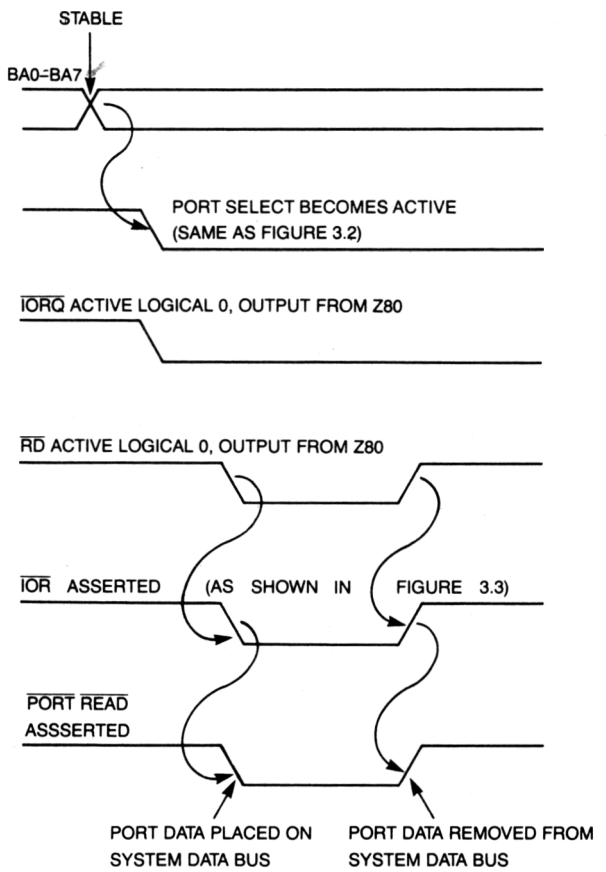


Figure 3.7: A timing diagram showing the sequence of electrical events that will occur in Figure 3.6.

events that occurs during an input port read operation. In this diagram, the port select $\overline{IOR}$ signal is similar to the $\overline{IOW}$ signal used for an output operation.

Let's now examine the electrical effect of the $\overline{RD}$ signal shown in Figure 3.7. $\overline{RD}$ is the timed control signal that generates the $\overline{IOR}$ signal for the input port hardware. This signal electrically starts the data transfer. When $\overline{RD}$ goes to a logical 0, the data from the input port is placed on the system data bus and is strobed by the Z80 into an internal register. At a later point in time, $\overline{RD}$ will go to a logical 1. When this occurs, the input port data will be electrically removed from the system data bus and the hardware transfer will be complete.

Figure 3.8 shows one of several ways a port read signal can be generated, based on the logical conditions given in Figure 3.7. We can see in Figure 3.8 that this circuit is very similar to the logic shown in Figure 3.6. Indeed, the only difference between the two circuits is that one uses the $\overline{IOW}$ signal and the other uses the $\overline{IOR}$ signal. Apart from that, they are identical. The port read strobe becomes active if, and only if, the Z80 is electrically inputting data from the specified port, FF.

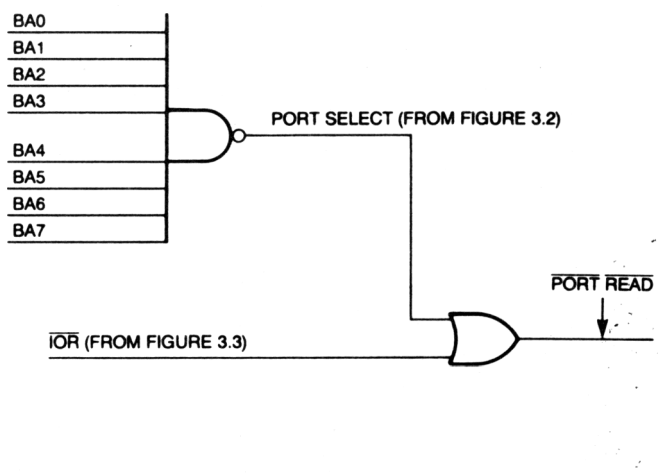


Figure 3.8: A schematic diagram showing the hardware required to realize the port read control signal.

3-5: A Complete Schematic for an I/O Port

In this section we will cover the events that occur during an I/O read and an I/O write operation. As we discuss these events, you may want to refer to the schematic in Figure 3.9, showing a general 8-bit I/O port. Let's first examine how the Z80 performs an output operation.

3-6: Sequence of Events for an Output Write

1. For an output write, the address lines A0–A7 are first set to the desired output address under control of the Z80. At this time, the output address lines A0–A7 are decoded by the port select hardware. In Figure 3.9 output pin 8 of IC1 becomes active logical 0.
2. Next, the Z80 outputs the electrical data to be written to the output port on the data bus lines D0–D7. Notice in Figure 3.9 that the data inputs D0–D7 are now valid at the 74LS374 octal latch (IC4).
3. Next, the $\overline{\text{IORQ}}$ output line from the Z80 is set to a logical 0. This indicates that the Z80 is performing an I/O operation. At this point, all of the static decoding from the Z80 output signals has occurred.
4. Next, the $\overline{\text{WR}}$ timed control output line from the Z80 is set to a logical 0. This action asserts the $\overline{\text{IOW}}$ system control line. With $\overline{\text{IOW}}$ going to a logical 0, the port write strobe also goes to a logical 0. Next, the $\overline{\text{WR}}$ output from the Z80 goes to a logical 1. At this point the data that was present on the D0–D7 data lines is written to the 74LS374. Since the signal is now a logical 1, the output operation is complete.

In this output port the writing of different data to the port turns the LEDs on and off, as shown in Figure 3.9. A logical 0 written to a specified data bit will turn an LED on, a logical 1 written to the port will turn it off.

The chapters that follow will show how writing different output data to a specified port can completely alter the port operation.

3-7: Input Port Read Operation

Let's now discuss the sequence of events performed by the Z80 when reading data from an input port (as shown in Figure 3.9). The major events that occur in an I/O read are the same as those in an I/O write, except that the $\overline{\text{RD}}$ timed control output line is used, rather than the $\overline{\text{WR}}$ control line.

1. First, the address lines A0–A7 are placed on the system address bus. When this occurs the input port decoding logic, IC1, will have a

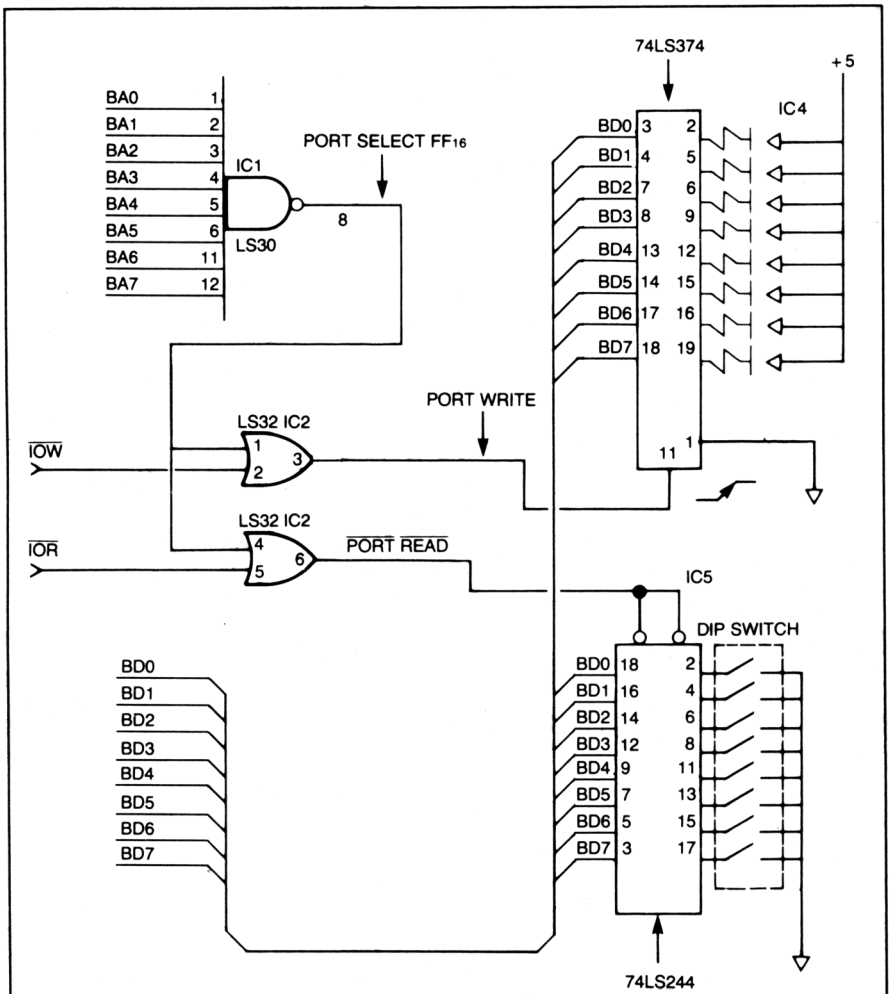


Figure 3.9: A complete schematic of a general 8-bit input and output port for the Z80. The port address for the input and output port is FF.

logical 0 on pin 8. The data from the input port will not yet be enabled onto the system data bus. The hardware is waiting for $\overline{IORQ}$ and the timed control signal $\overline{RD}$ to become active.

2. Next, $\overline{IORQ}$ is set to a logical 0 level by the Z80. This action electrically informs the system that the Z80 will be communicating with I/O, rather than memory.

3. Next, the timed control signal $\overline{RD}$ is set to a logical 0 by the Z80. This action asserts the $\overline{IOR}$ system control line.

4. When the $\overline{IOR}$ control line is asserted, the microprocessor is electrically ready to receive data from the input port. Since the $\overline{RD}$ output from the Z80 is at a logical 0 level, the enable input pins 1,19 of the 74LS244 IC5 are set to a logical 0. A logical 0 on these input pins enables the 74LS244 outputs to control the system data bus. When this occurs any logical values set by the switches and input to the 74LS244 buffers are now output to the system data bus. During the time the 74LS244 buffers are enabled, the Z80 strobes the information on the system data bus into an internal register.

5. Finally, the $\overline{RD}$ is set to a logical 1 under control of the Z80. When $\overline{RD}$ goes to a logical 1 level, the data from the input port is electrically removed from the system data bus. The system operation is now complete.

3-8: Summary of Electrical Sequences

Let's now quickly review the electrical events that occur in the system hardware whenever the Z80 performs an input or output operation. This information will be useful later on when we examine several I/O devices in detail.

3-9: Output Write Sequence

1. A0-A7 are set to the correct output port address.

2. D0-D7 are set to the correct data to be written to the output port.

3. $\overline{IORQ}$ is set to a logical 0.

4. $\overline{WR}$ is set to a logical 0.

5. $\overline{WR}$ is set to a logical 1.

3-10: Input Read Sequence

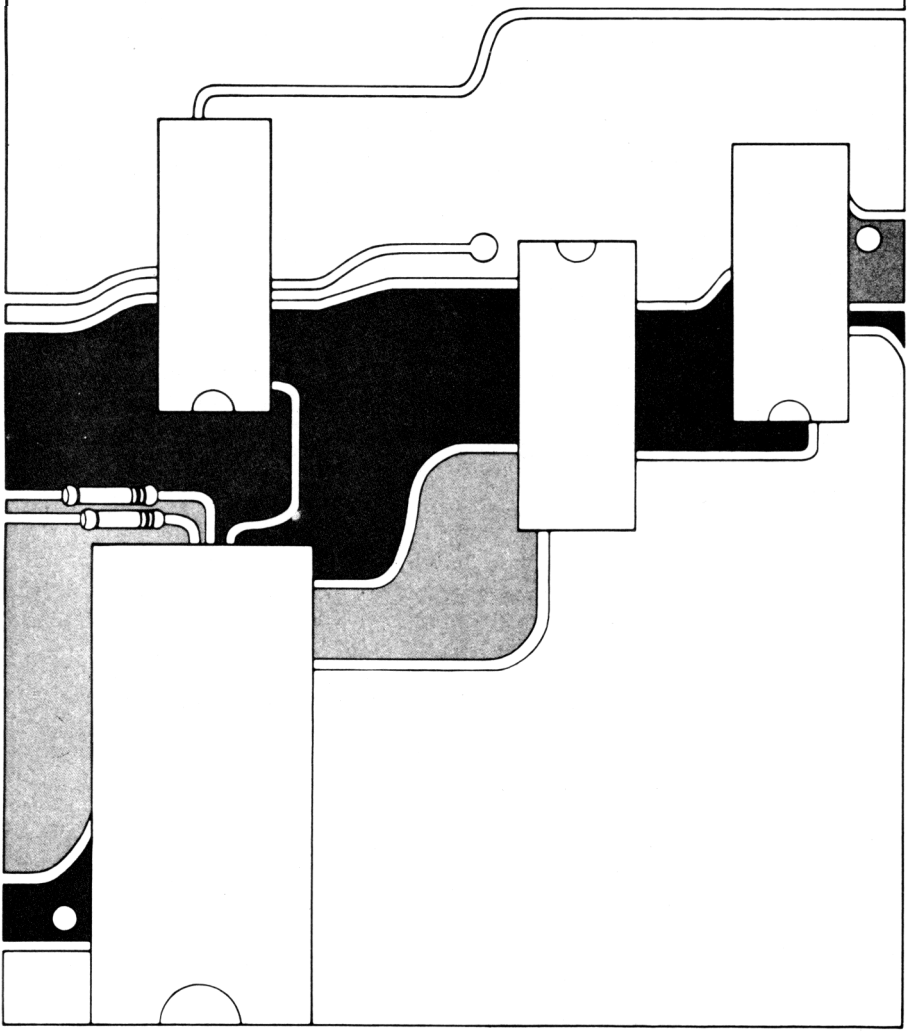
1. A0–A7 is set to the correct input port address.
2. $\overline{\text{IORQ}}$ is set to a logical 0 under control of the Z80.
3. $\overline{\text{RD}}$ is set to a logical 0.
4. $\overline{\text{RD}}$ is set to a logical 1.

CHAPTER SUMMARY

In this chapter we have discussed the details of how the Z80 performs general input and output operations. We have explored several hardware decoding techniques, and we have examined the events that occur during an input and output operation.

If you plan to troubleshoot, debug or design hardware for a Z80 system, you will want to be familiar with how input and output operations occur. Therefore, the information we have presented in this chapter is important. This information will be very useful in later chapters when we examine some of the hardware necessary to interface the Z80 with other devices, and when we use special LSI I/O devices.

Using Dynamic RAMS with the Z80



Chapter 4

INTRODUCTION

In this chapter we will show how dynamic RAMs can be used with the Z80. We will first examine a typical dynamic RAM. We will then show how the Z80 address, data, and control buses all communicate electrically with dynamic RAM devices. Finally, we will examine a complete dynamic RAM system for the Z80.

The dynamic RAM that we will be discussing in this chapter is the 4116, 16K × 1. We have chosen this device because it has general operating characteristics applicable to most dynamic RAMS used today. Once you understand how this device operates with the Z80, you should be able to easily understand how most other dynamic RAM devices operate, including the 64K × 1.

4-1: Overview of the 4116

The 4116 is organized as a 16,384 (16K) by 1 RAM. The 4116 uses separate data input and output lines. A memory with these characteristics needs 14 address lines (to give the total of 16,384 address locations), 1 data input line and 1 data output line.

If this RAM is to have data written to it, there must be a write enable line. In addition, to operate, the 4116 requires power supplies of +12 and +5 volts, −5 volts, and a ground connection. Thus, if we add up the necessary pins on this device, we find:

14 address lines
1 data input line
1 data output line
4 power supply lines
1 write enable line

for a total of 21 physical lines. See Figure 4.1 for a diagram showing these lines.

The 4116 is packaged in a 16-pin DIP (dual in-line package) form as shown in Figure 4.2. Although the 16-pin DIP may at first appear to be incomplete, there is no mistake. The manufacturers of this device have made use of the time-multiplex technique to save physical device pins. *Time-multiplexing* with the 4116 means that the required 14 address lines are input in two separate groups, with seven lines in each group. One group of address lines is input first and then the next. Note that there are seven address input pins to the 4116. These lines are labeled A0–A6, as shown in Figure 4.3. When these lines input the lower order microprocessor address outputs, A0–A6, they are labeled *row address*. They are labeled *column address*, when the 4116 address inputs (A0–A6) have the upper order microprocessor address lines, A7–A13, input on them. See Figure 4.4.

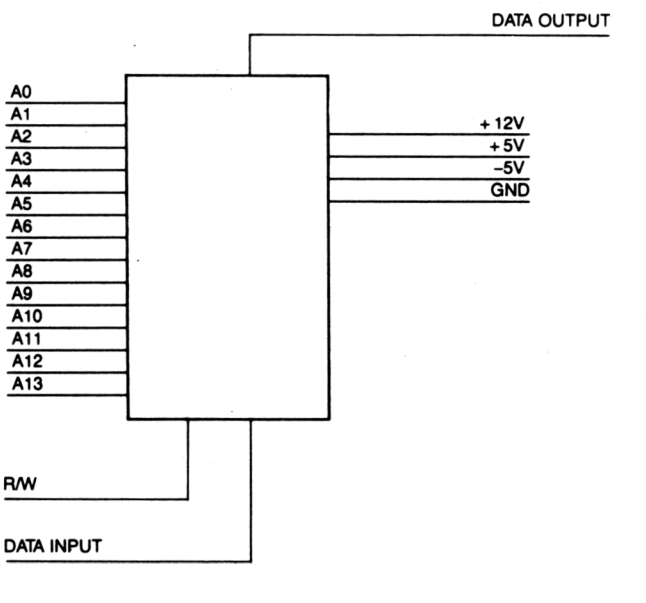


***Figure 4.1:** A block diagram showing the physical address lines, data input, data output and R/W lines required for a 16K × 1 RAM. The 4116 must somehow meet these physical conditions.*

16,384 X 1-BIT DYNAMIC RAM

MK4116(J/N/E)-2/3

FEATURES

- Recognized industry standard 16-pin configuration from MOSTEK
- 150ns access time, 320ns cycle (MK 4116-2) 200ns access time, 375ns cycle (MK 4116-3)
- ± 10% tolerance on all power supplies (+12V, ±5V)
- Low power: 462mW active, 20mW standby (max)
- Output data controlled by $\overline{CAS}$ and unlatched at end of cycle to allow two dimensional chip selection and extended page boundary
- Common I/O capability using "early write" operation
- Read-Modify-Write, $\overline{RAS}$-only refresh, and Page-mode capability
- All inputs TTL compatible,low capacitance, and protected against static charge
- 128 refresh cycles
- ECL compatible on VBB power supply (-5.7V)
- MKB version screened to MIL-STD-883
- JAN version available to MIL-M-38510/240

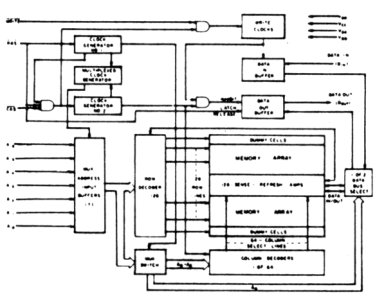
DESCRIPTION

The MK 4116 is a new generation MOS dynamic random access memory circuit organized as 16,384 words by 1 bit. As a state-of-the-art MOS memory device, the MK 4116 (16K RAM) incorporates advanced circuit techniques designed to provide wide operating margins, both internally and to the system user, while achieving performance levels in speed and power previously seen only in MOSTEK's high performance MK 4027 (4K RAM).

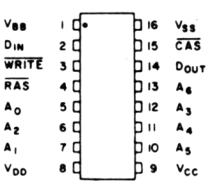
The technology used to fabricate the MK 4116 is MOSTEK's double-poly, N-channel silicon gate, POLY II⊖ process. This process, coupled with the use of a single transistor dynamic storage cell, provides the maximum possible circuit density and reliability, while maintaining high performance capability. The use of dynamic circuitry throughout, including sense amplifiers, assures that power dissipation is minimized without any sacrifice in speed or operating margin. These factors combine to make the MK 4116 a truly superior RAM product.

Multiplexed address inputs (a feature pioneered by MOSTEK for its 4K RAMS) permits the MK 4116 to be packaged in a standard 16-pin DIP. This recognized industry standard package configuration, while compatible with widely available automated testing and insertion equipment, provides highest possible system bit densities and simplifies system upgrade from 4K to 16K RAMs for new generatior applications. Non-critical clock timing requirement allow use of the multiplexing technique while mair taining high performance.

FUNCTIONAL DIAGRAM



PIN CONNECTIONS



PIN NAMES

A_0-A_6	ADDRESS INPUTS	$\overline{WRITE}$	READ/WRITE INPUT
$\overline{CAS}$	COLUMN ADDRESS STROBE	V_{BB}	POWER (-5V)
D_{IN}	DATA IN	V_{CC}	POWER (+5V)
D_{OUT}	DATA OUT	V_{DD}	POWER (+12V)
$\overline{RAS}$	ROW ADDRESS STROBE	V_{SS}	GROUND

Figure 4.2: A partial data sheet showing the physical layouts and pin connections for the 4116, 16K × 1 dynamic RAM.

The labels, row address and column address, stem from the fact that these groups of address lines perform the named function in the internal memory array of the device. Storage cells of a memory device are organized in a matrix. Each cell in the matrix has a unique row and column address. When the multiplexing feature is used, the number of required address input pins for the 4116 is reduced from 14 to 7.

When the time-multiplexing of digital signals is performed, the system hardware needs an electrical signal that will indicate when the address input lines represent each of these two groups. In other words, the 4116 must be informed electrically when the information at the address input pins A0–A6 is the row address and when it is the column address. To accomplish this task, there are two additional inputs to the device. In Figure 4.5, they are labeled $\overline{RAS}$ (row address strobe) and $\overline{CAS}$ (column address strobe). When the $\overline{RAS}$ input is asserted (logical 0), the information on the address input lines A0–A6 is strobed or latched internally and used as the lower 7 bits of the total 14-bit address. When the $\overline{CAS}$ input to the 4116 is asserted (logical 0), the information at the memory address pins A0–A6 is used as the upper 7 bits of the complete 14-bit address. Figure 4.5 shows a timing diagram of the $\overline{RAS}$ and $\overline{CAS}$ being used to strobe in the memory address on a 4116 device.

Memory address lines A0–A6 of the 4116 are inputs. This means that the memory device itself is passive for this function. It is the job of the external

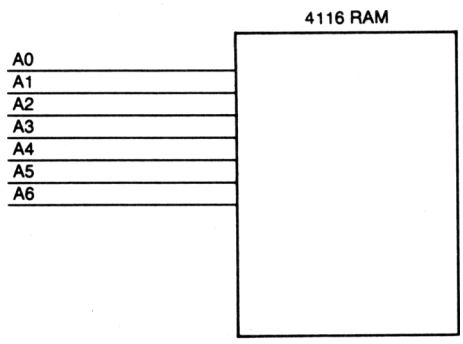


Figure 4.3: A block diagram showing the actual number of physical address input lines available on the 4116. The 14 total address input lines are placed on the 7 physical lines in two separate groups.

hardware to provide the memory address inputs. In other words, external hardware must provide the following to the 4116: the row address inputs, the $\overline{RAS}$, the column address inputs, and the $\overline{CAS}$. Furthermore, the external hardware must time these signals properly. It is then up to the memory device to make use of these properly-timed signals.

4-2: Multiplexing the Address Lines

A circuit similar to the one in Figure 4.6 can be used to perform the function of multiplexing the address inputs to the 4116. Let's see how the circuit operates. The 14 address lines, A0–A13, are input to the digital devices

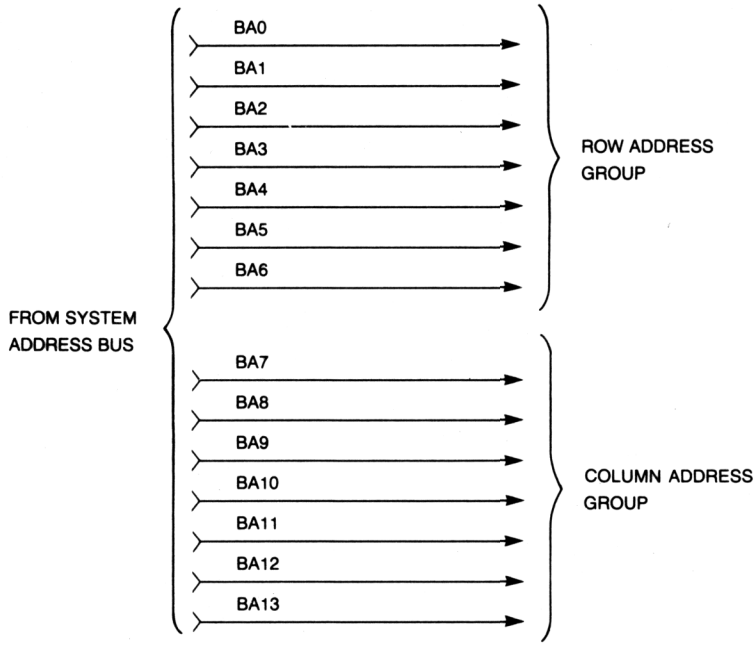


Figure 4.4: The system address lines are grouped into a row address and column address group. The address lines originate at the output of the system address buffers.

labeled multiplexers. When the control line to the multiplexers is in the logical 1 state, the microprocessor address lines A0–A6 are routed to the 4116 address input pins A0–A6. When the control line to the multiplexers is a logical 0, the remaining address lines, A7–A13, are routed to the 4116 address inputs A0–A6.

From this discussion we know that there are three main electrical events that must be performed by the hardware in order to input the entire 14-bit address to the 4116. They include:

1. generating the $\overline{RAS}$ at the correct time.
2. forcing the MUX control line to a logical 0 after the $\overline{RAS}$.
3. generating the $\overline{CAS}$ at the correct time after MUX.

Figure 4.7 shows these three events in a general timing diagram. (Later on we will learn how the Z80 accomplishes them.) Let's now learn how an address is input to the 4116 device.

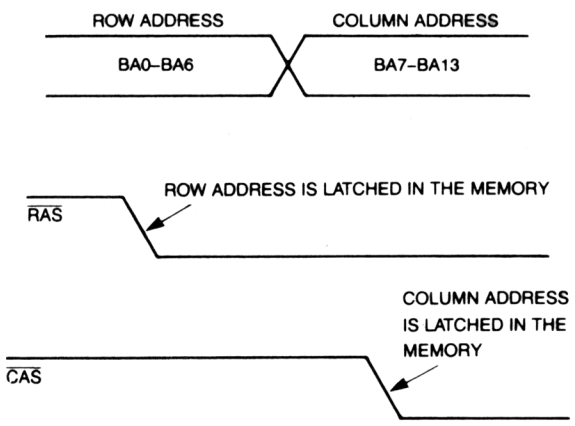


Figure 4.5: A timing diagram showing the timing relationship between the application of the row address group and the column address group to the 4116 RAM. Notice that the $\overline{RAS}$ signal is active during the row group and the $\overline{CAS}$ signal is active during the column group.

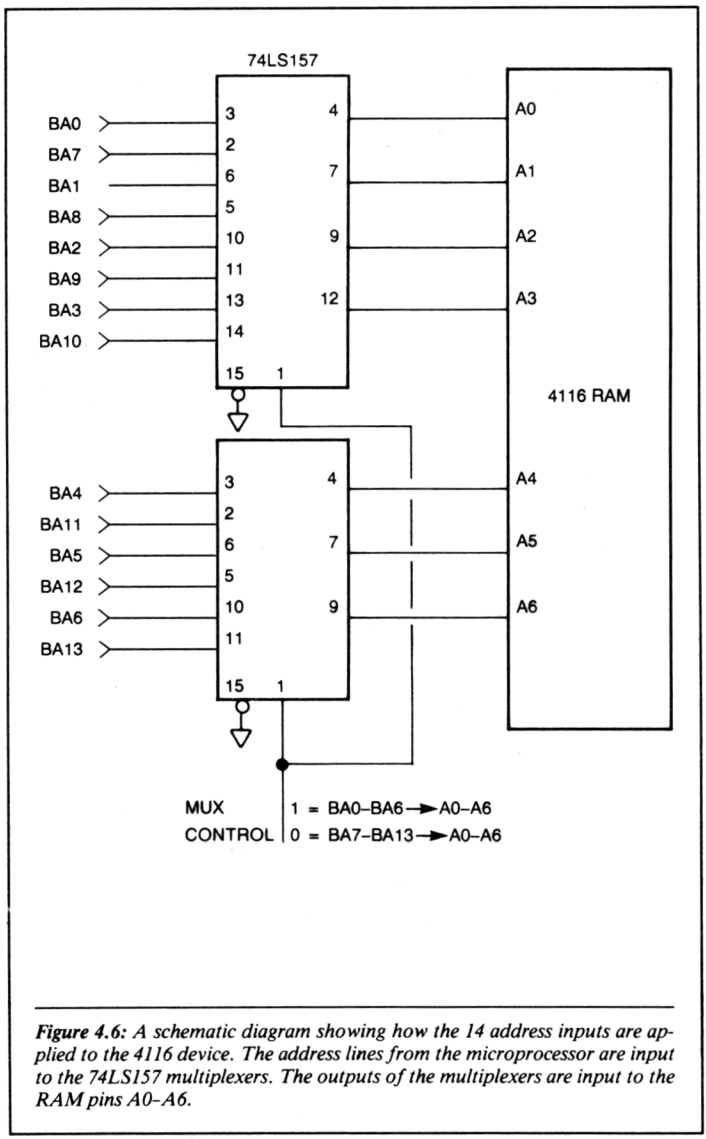


Figure 4.6: *A schematic diagram showing how the 14 address inputs are applied to the 4116 device. The address lines from the microprocessor are input to the 74LS157 multiplexers. The outputs of the multiplexers are input to the RAM pins A0–A6.*

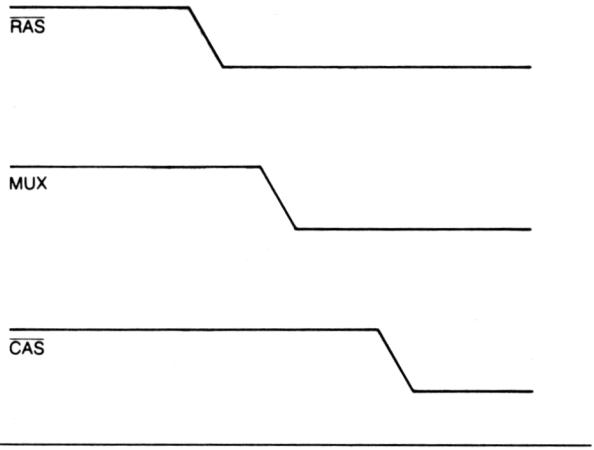


Figure 4.7: A timing diagram showing the relationship between the $\overline{RAS}$, MUX and $\overline{CAS}$ control signals for the dynamic RAM.

4-3: Block Diagram of the 16K × 8-Bit Dynamic RAM System

Figure 4.8 shows a block diagram of a 16K × 8-bit dynamic RAM system that can be used by the Z80. Let's now examine it and discuss the hardware for the circuits shown in the diagram.

In viewing this diagram it is important to remember that there are many ways to interface a microprocessor with dynamic RAMs. We have chosen the technique presented here because it is straightforward and can be understood by those with only a modest amount of digital hardware experience. An interface to a dynamic RAM can be very complicated, especially in cases of industrial uses. However, even the most complicated designs evolve from the information given in this chapter. Let's examine Figure 4.8.

The memory address shown in the figure is input with a circuit similar to the one in Figure 4.6 (although this block does not appear in the diagram in Figure 4.8). The $\overline{RAS}$ line is input in parallel to all memory devices in the system memory. The $\overline{CAS}$ input to the memory system may or may not be asserted during the memory operations. (We will discuss this further later on in the chapter.)

The logic for generating when the $\overline{CAS}$ signal becomes active is included in block A. This logic decodes the conditions of the memory select lines, and at the appropriate time asserts the $\overline{CAS}$ input line to the memory. The

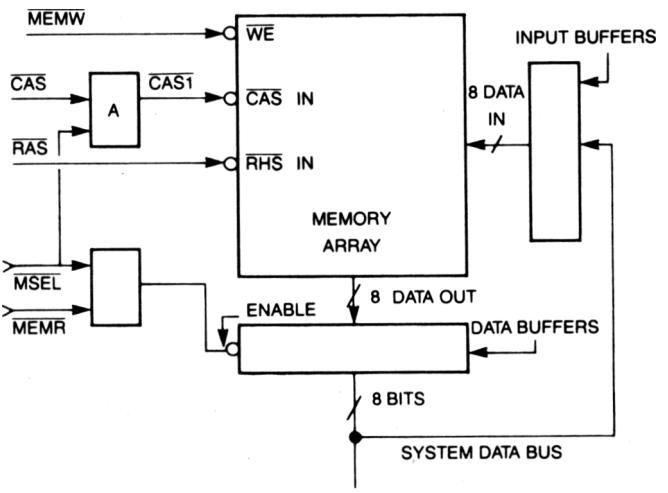


Figure 4.8: A block diagram showing the peripheral hardware associated with a dynamic RAM system.

memory select line is generated from the decoding of the system address.

The system address space for the dynamic RAM is equal to 16K bytes. The enable line for this space is decoded from the address lines A14 and A15. Whenever A14 and A15 are a logical 1, the RAM is communicating with the Z80 CPU.

In the block diagram shown in Figure 4.8, the memory will have the write enable input on active logical 0 whenever the $\overline{\text{MREQ}}$ and the $\overline{\text{WR}}$ are both logical 0. You might think that this could present problems; but this is not so. The $\overline{\text{MREQ}}$ and the $\overline{\text{WR}}$ are both logical 0 whenever the microprocessor is performing any system write operation to any system memory address space. Later in this discussion, we will show how the system avoids unwanted write operations to the dynamic RAM.

4-4: Generating the $\overline{\text{RAS}}$, $\overline{\text{CAS}}$ and MUX Signals

We will now learn how the Z80 generates the $\overline{\text{RAS}}$, $\overline{\text{CAS}}$ and MUX control signals. Again, there are many different techniques that can be used to do this.

The technique described in this section allows the beginner to understand exactly what must occur, and in what sequence. In addition, it works.

If we refer back to Figure 4.5 we can see the correct order in which the $\overline{RAS}$, MUX and $\overline{CAS}$ signals must occur. Figure 4.9 shows how we can generate these signals in the correct sequence. Here, the $\overline{RAS}$ signal is a buffered $\overline{MREQ}$ signal. Recall that the $\overline{RAS}$ signal must strobe the lower microprocessor address lines, A0–A6, into the internal latch of the 4116. Note that the $\overline{MREQ}$ line is active at the correct instant in time—that is, after the microprocessor address is stable on the address bus.

The purpose of the MUX signal is to switch the address input lines from the microprocessor outputs A0–A6 to the outputs A7–A13. This signal is generated from the $\overline{RAS}$ line. In fact, the MUX signal is the $\overline{RAS}$ signal after it has gone through a delay line. Any standard delay line can be used. Sometimes the delay line is made up of digital devices that are put in series with the $\overline{RAS}$ line. In this case, the propagation delay of the device or devices is used as the delay line. The actual delay time from $\overline{RAS}$ to MUX is approximately 50 nanoseconds.

After the MUX line has become active logical 0, the $\overline{CAS}$ line can be asserted. A delay line is used to generate the $\overline{CAS}$ signal as well.

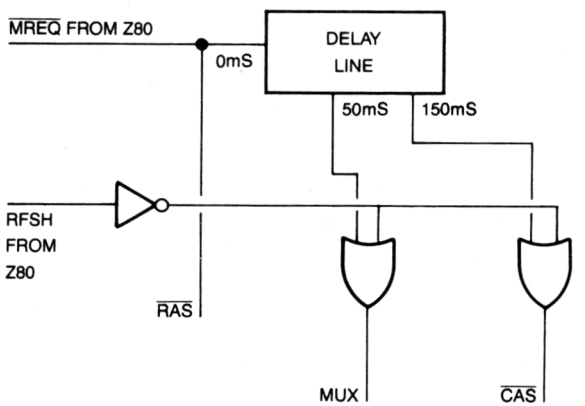


Figure 4.9: A schematic diagram showing how the $\overline{RAS}$, $\overline{CAS}$, and MUX control signals are generated.

This technique for signal generation is easy to realize with hardware. However, there are some drawbacks. The main one is that the memory must have a very fast access time if it is to run at maximum microprocessor speed, because the memory access time is measured from the falling edge of the $\overline{CAS}$. One way to avoid this problem is to slow down the system clock.

Although the system clock frequency must be slow enough to allow the memory sufficient access time to do its job properly, it cannot be too slow, due to refreshing considerations (explained later in this chapter). Figure 4.9 shows an exact timing diagram of the Z80. This diagram displays specific times for each signal. It also shows sequences and how certain Z80 signals are used to generate the $\overline{RAS}$ and $\overline{CAS}$ inputs to the RAM.

4-5: Data Input to the Dynamic RAM

The 4116 dynamic RAM makes use of separate I/O. Recall that separate I/O means that a different physical memory device pin is used for inputting and outputting data. Let's now see how data is physically input to the RAM from the microprocessor.

Figure 4.10 shows a buffer circuit that connects the system data bus to the 4116 data input pin. This figure shows that the buffers are always enabled.

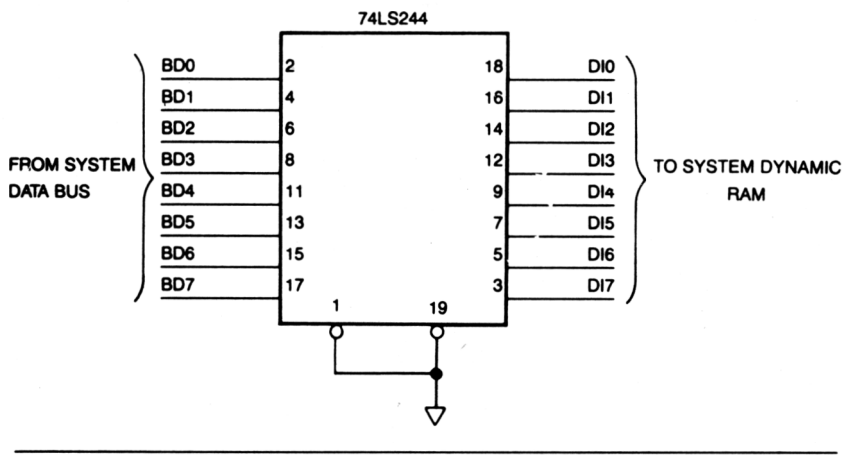


Figure 4.10: *The hardware required to input the data to the RAM input lines. In some systems, however, this input buffer may not be used. In such systems, the dynamic RAM inputs will be connected directly to the system data bus lines.*

Data from the system data bus appears (electrically) at the RAM input pins at all times. This occurrence is electrically valid because the data will not be stored in the RAM until a write enable signal is asserted to the device.

A natural question then is, "Why use buffers at all?" Since the data bus is input directly to the RAM input pins, it would appear that buffers may be eliminated. However, although it is true that some systems do not use data buffers, there are two reasons why the use of buffers is recommended:

- to isolate the RAM inputs from the entire system data bus. This helps in the debugging of a defective system. If a single RAM chip goes bad, then the entire data bus line may be affected. This problem is made even worse when multiple RAM chips are connected to the system data bus.
- to control noise. The system data bus is usually very "glitchy;" that is, there are normally several unwanted, spurious signals on the system data bus. The data bus buffers help to prevent the noise on the system data bus from reaching the RAM input pins.

Most manufacturers recommend the use of data input buffers.

4-6: Writing Data to the Dynamic RAM

In this section we will learn how the microprocessor writes data to the 4116. The block diagram in Figure 4.8 shows that the following signals must be active for a memory write operation to occur:

- $\overline{\text{MREQ}}$ must equal logical 0
- $\overline{\text{WR}}$ must equal logical 0

These two signals assert the write enable to the system memory. In addition, the following signals are used to indicate that the memory space is electrically selected:

- A14 and A15 must equal logical 1
- the dynamic RAM must occupy address space between C000–FFFF.

Under these system conditions the $\overline{\text{CAS}}$ is asserted to the system RAM and a write operation occurs.

Let's examine the timing diagram for a memory write operation under two different conditions: when the memory is enabled and when it is not. When the memory is not enabled, the microprocessor is writing data to some other memory space or to I/O.

Figure 4.11 shows the major system timing for a memory write operation when the dynamic RAM is enabled. The first event to occur is that the

memory select line is enabled. This occurs when address lines A14 and A15 go to a logical 1 under the control of the Z80. Following this event, the $\overline{RAS}$ to the system $\overline{RAS}$ is asserted. The $\overline{RAS}$ is generated by the $\overline{MREQ}$ output control line from the Z80. After the $\overline{RAS}$ is asserted, the MUX control to

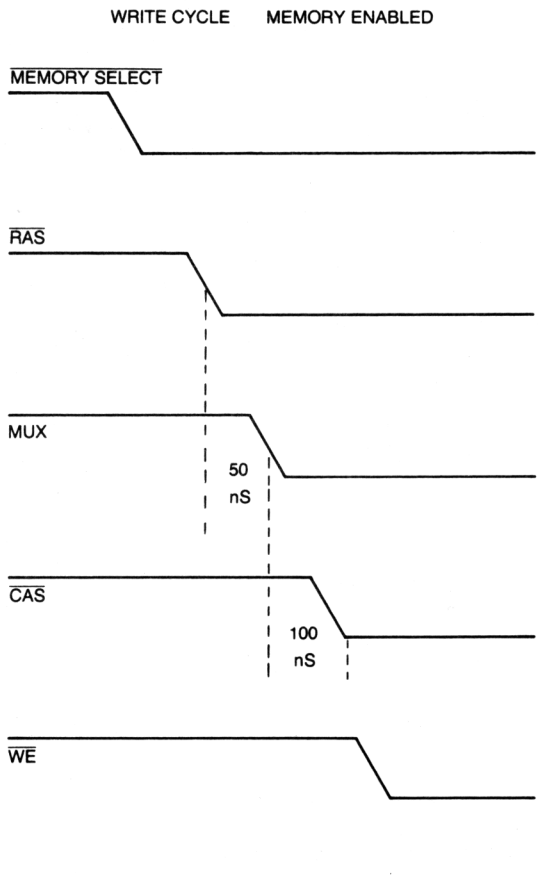


Figure 4.11: A timing diagram showing the sequence of signals that will become asserted during a memory write cycle when the dynamic RAM space is enabled.

the address multiplexers goes to a logical 0, after a delay of 50 nanoseconds from the falling edge of $\overline{RAS}$.

As the MUX control goes to a logical 0, the address inputs to the 4116 switch from the row address to the column address. After the MUX has switched, the $\overline{CAS}$ is asserted. The $\overline{CAS}$ signal is generated by delaying the MUX line. Next, the $\overline{WE}$ input to the memory is generated by the $\overline{WR}$ line from the Z80.

The timing diagram in Figure 4.11 shows the sequence of electrical events that occurs whenever the microprocessor is writing data to the system dynamic RAM. Figure 4.12 displays a timing diagram of the events that occur whenever the microprocessor is writing data to any memory that is not in the dynamic RAM address space. In other words, since the data is intended for elsewhere, the dynamic RAM section is disabled.

Let's look at the timing diagram in Figure 4.12. In this figure the memory

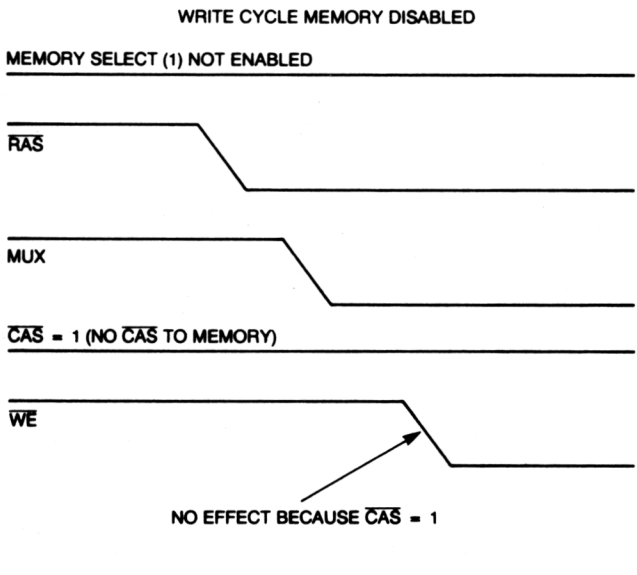


***Figure 4.12:** A timing diagram showing the sequence of signals that are asserted during a memory write cycle when the dynamic RAM is not enabled. Notice that the memory select line is a logical 1 during this operation. Also notice that no $\overline{CAS}$ signal is generated.*

select line is not active. This is due to the fact that *both* A14 and A15 are not logical 1's. The $\overline{RAS}$ is active because the $\overline{MREQ}$ control line on the Z80 is active. The MUX control line goes to a logical 0 because it is driven from the $\overline{RAS}$ control line. The system $\overline{CAS}$ line is asserted, but $\overline{CAS}$ to the memory is not generated. This is due to the fact that the memory select line is not active.

Finally, the $\overline{WE}$ to the system dynamic RAM becomes active. Data is not transferred to the RAM, however, because both the $\overline{CAS}$ and the $\overline{RAS}$ inputs to the 4116 must be active logical 0 for the memory chip to be electrically enabled, thus disabling the memory space for any memory write when its address space is not selected by the microprocessor.

4-7: Data Output from the Dynamic RAM

We will now learn how the dynamic RAM data output line is placed on the system data bus during a memory read operation. The technique used is similar to the placing of static RAM, ROM and I/O data onto the system data bus. Figure 4.13 shows one way hardware can be used to do this.

In this figure, a 74LS244 uni-directional buffer is used. The input to the buffers is the 4116 RAM data output pins. The output from the 74LS244 buffers are connected directly to the system data bus. When the microprocessor is electrically requesting data from the RAM, the buffers become enabled and place the dynamic RAM data onto the system data bus.

The addressing of the RAM for a memory read operation is identical to that of a memory write operation. The $\overline{RAS}$, MUX, and $\overline{CAS}$ have exactly the same timing. To enable the dynamic RAM data outputs onto the system data bus during a memory read operation, the following electrical events must occur:

1. $\overline{MREQ}$ output from the Z80 must be a logical 0, which indicates a memory read operation.

2. A15 and A14 must both be a logical 1, which asserts the memory select for the dynamic RAM address space.

3. $\overline{RD}$ output from the Z80 must be a logical 0, which indicates a read operation on the Z80 system data bus. This signal electrically indicates that the Z80 is prepared to receive data.

As we can see in Figure 4.13, when $\overline{MREQ}$ is a logical 0, and A15 and A14 are both logical 1's, the inputs to the OR gate pins 1 and 2 are logical 0's. When $\overline{RD}$ becomes asserted, in the logical 0 level, the 74LS244 is enabled. The microprocessor will then keep $\overline{RD}$ a logical 0 for a fixed period of time

determined by the Z80 clock frequency. When the Z80 sets $\overline{RD}$ to a logical 1, the RAM data is electrically removed from the system data bus. This action terminates the memory transfer.

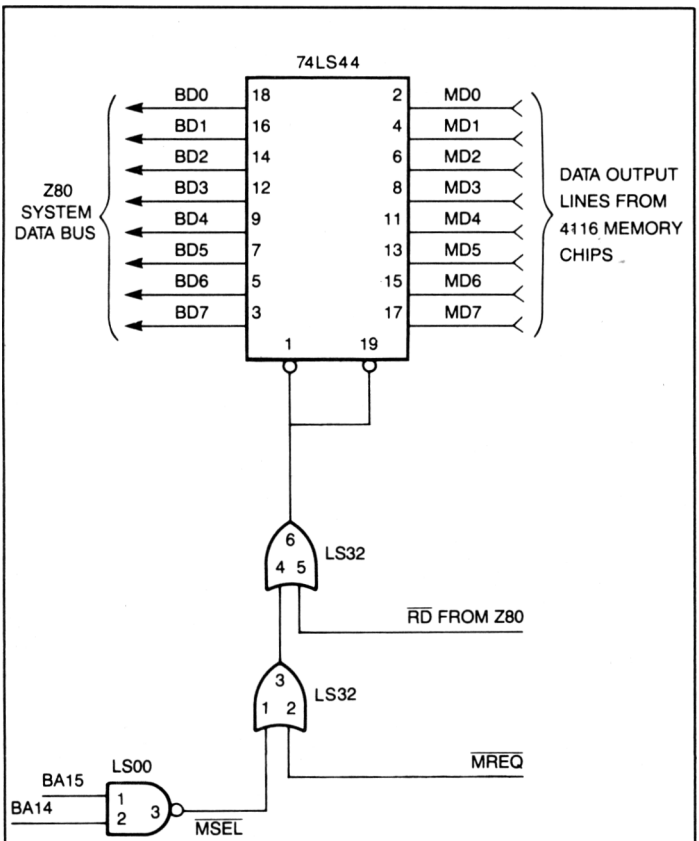


Figure 4.13: *A schematic diagram showing one way the data output lines of the dynamic RAM can be placed onto the system data bus during a memory operation to the RAM space.*

4-8: Refreshing the Dynamic RAM

Up to this point in our discussion of the dynamic RAM, we have neglected to state that the device must be refreshed. *Refreshing* means that the information contained in the dynamic RAM internal storage cells must be accessed (read from) periodically to keep it valid. Typically, the information in a dynamic RAM storage cell will remain valid for only a few (usually two) milliseconds, and if the cell is not accessed in that period of time, the data will be lost.

There are several techniques for refreshing data in a dynamic RAM. The technique shown here is used by the Z80. For this example, let's say that to retain the internal information, the dynamic RAM must have each storage cell accessed every two milliseconds. The accessing must occur even when the dynamic RAM space is not being used for program execution.

When data is read from the 4116, an entire row of internal cells is refreshed in a parallel fashion. An entire RAM can be refreshed by accessing every row address, A0–A6. This equals 128 row address inputs. With the two millisecond criterion then, we must cycle through every row address once every two milliseconds. Therefore, with 128 unique row addresses being accessed every two milliseconds, the microprocessor must access a different row address every sixteen microseconds.

The Z80 is an extremely attractive microprocessor to use with dynamic RAMs, because it has a built-in hardware feature, called the *refresh counter,* that allows the interface to dynamic memories to be quite simple. This counter can be set to output a refresh address on the system address bus at any specified time interval. Then, after the address has been output, the Z80 will automatically increment the counter to output the next address at the next specified time.

This internal feature of the Z80 takes care of the dynamic RAM system automatically. During the opcode fetch machine cycle of an instruction cycle, the refresh address is output on the Z80 address lines A0–A6. The address is output during the T4 time that is used internally by the Z80 to decode the opcode just fetched from memory. This type of refresh addressing is called *cycle stealing*.

During the time the refresh address is being output on the Z80 microprocessor, the RFSH pin on the Z80 is active. For the dynamic RAM, the refresh cycle requires that only the $\overline{\text{RAS}}$ become asserted. (Recall that the $\overline{\text{RAS}}$ control line strobes in the row address to the system memory.) We do not wish to assert the $\overline{\text{CAS}}$ control line because the memory will become enabled, and, to perform the internal refresh operation, the memory does not require the $\overline{\text{CAS}}$ input to be active. (You may want to refer to a complete data sheet on the 4116 to see the exact timing required for a refresh operation with this memory.)

Figure 4.14 shows the sequence of timing that is required for the 4116 in the refreshing mode. In this figure we can see that the refresh address is output on the A0–A6 address lines of the Z80, and that the internal refresh counter will automatically increment the refresh address by one each time a refresh cycle occurs.

During a refresh cycle the Z80 asserts the $\overline{\text{MREQ}}$ line. This forces the $\overline{\text{RAS}}$ to occur. However, during a refresh cycle the $\overline{\text{RD}}$ control line is not asserted. The logic of the $\overline{\text{CAS}}$ circuit keeps it from being asserted. Further, the lack of $\overline{\text{RD}}$ keeps the tri-state output buffers from becoming enabled, and thus from placing memory data on the system data bus.

It should be noted that the refreshing of memory will occur as long as the Z80 is executing a program; that is, as long as it is fetching opcodes. However, the time interval between opcode fetches can be no longer than the time interval required to refresh the memory. Some instructions take longer to execute than others. In a dynamic RAM system the "worst case" instructions must be accounted for when calculating the time between refresh addresses.

Whenever the Z80 is halted or the hold input is asserted, there is no refreshing of the system memory. Also, when the external reset is applied to the Z80 there is no refresh operation to the dynamic RAM. It is important that you are aware of these facts when considering the use of dynamic RAM with the Z80.

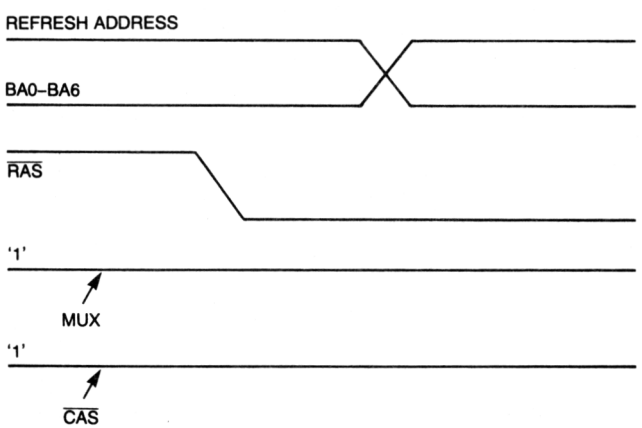


Figure 4.14: *A timing diagram showing the relationship of the main dynamic RAM control signals during a memory refresh operation.*

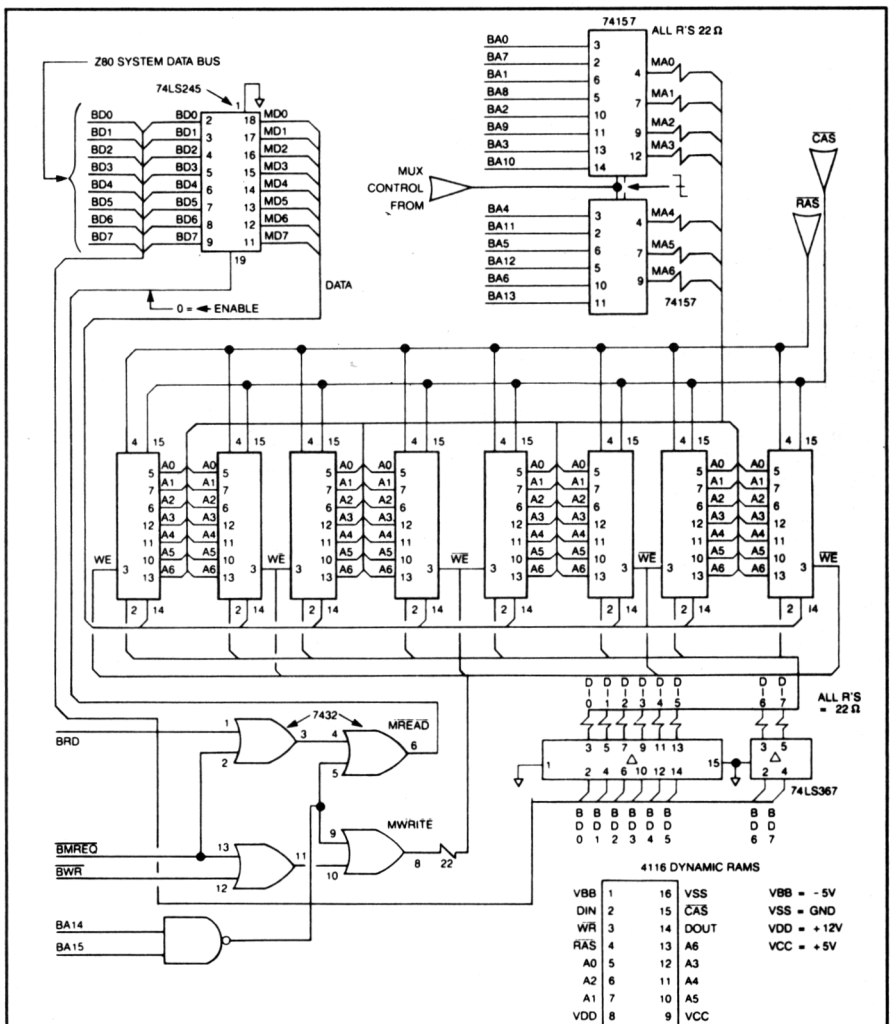


Figure 4.15: A complete schematic of a 16K × 8 dynamic RAM memory interfaced to a Z80 microprocessor.

4-9: Complete Schematic of a 16K × 8-Bit Dynamic RAM

Figure 4.15 shown on the previous page is a complete schematic for a dynamic RAM system that can be used with the Z80. Remember, though, that the hardware realization of the system can be accomplished in several ways. This design was chosen because it covers the various essential details that must be considered when interfacing the Z80 to a typical dynamic RAM device.

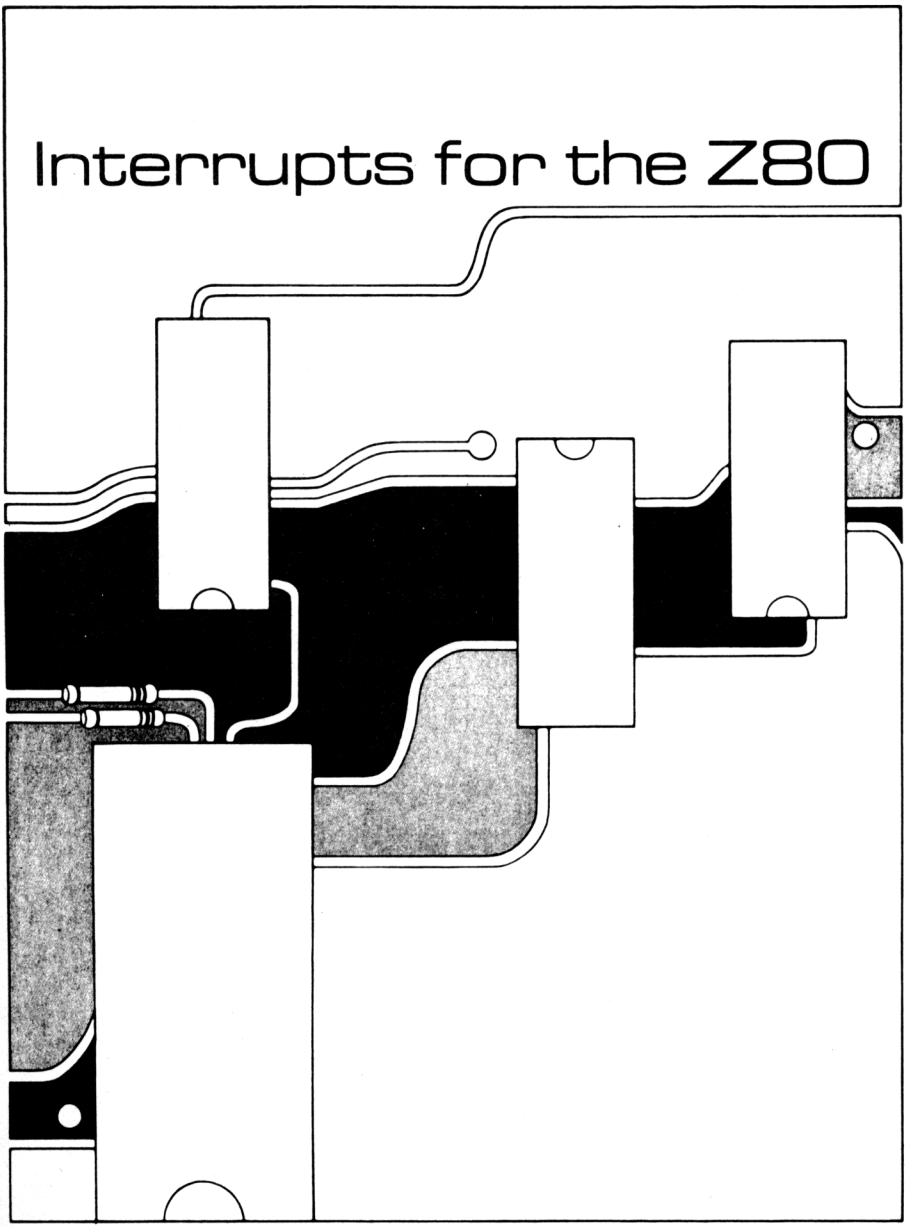
CHAPTER SUMMARY

In this chapter we have discussed the basics of electrical communication between the Z80 and dynamic RAM. We have examined a typical dynamic RAM—the 4116. This RAM is representive of a general class of dynamic RAM devices that use time-multiplexing for address inputs.

In this chapter, we have also explained how to perform address multiplexing using actual hardware. We have discussed the function of the $\overline{RAS}$, MUX and $\overline{CAS}$ control signals for the dynamic RAM, and we have learned how to generate each signal using the Z80 microprocessor.

Finally, we have learned about refreshing the dynamic RAM. This feature makes the Z80 CPU extremely popular for use with dynamic RAM devices.

Interrupts for the Z80



Chapter 5

INTRODUCTION

In this chapter we will discuss the general topic of interrupting the Z80 microprocessor. We will begin by introducing the concept of an interrupt. We will then show how each different type of interrupt is handled electrically by the Z80. For each new interrupt mode, we will present examples that will help you better understand what occurs during an interrupt operation.

It is important that you understand how the Z80 handles external interrupt requests in order to understand information in the chapters that follow. Therefore, if you aren't completely familiar with this topic, read this chapter carefully.

5-1: What Is an Interrupt?

To explain interrupts, envision the following situation. You are having a conversation with one other person. A second person walks up and speaks your name—thus, requesting your attention. The following is a list of possibilities for responding to this external request:

1. You can completely ignore the second person and continue your conversation as though that person were not there.
2. You can get to a convenient stopping place in your conversation and then turn your attention to the second person.
3. You can *immediately* end your conversation with the first person and start talking with the second person.

In any case, it is likely that when you have finished talking with the second person, you will want to continue the conversation you were having before the interruption occurred.

The preceding scenario may seem simplistic, but it accurately presents the concept of interrupts within a microprocessor system.

Think of it this way. You are the Z80 CPU. The person you were talking with initially is the main program being executed, and the second person is an external interrupt request (that is, it is some hardware in the system that wishes the CPU's attention). The Z80 CPU must handle this external request. There are several ways to do this. The three possibilities listed previously are the most common.

With this general introduction to interrupts, let's now examine the details of interrupts for the Z80.

5-2: Where Do the Interrupt Requests Come From?

A microprocessor system may be comprised of many different hardware components. It may include a CRT, a printer, a floppy or hard disk drive, a timer, a control motor, or a digital to analog converter (DAC)—just to name a few. Most of these external hardware components need the attention of the CPU only at certain times. At other times they function on their own.

For example, let's suppose your system has a clock as an external hardware device, and the clock is designed to display the time of day on the CRT screen. The clock digits are updated once each second. In other words, once each second the clock hardware requires the CPU to read the time and print it on the CRT screen. However, at all other times the CPU is available to perform the other tasks required of it.

The main point here is that the external hardware of the clock does not need the attention of the CPU *all of the time.* The clock hardware electrically requests the CPU to read it, only when required. This can be accomplished through the CPU's interrupt system, whereby, once each second the CPU receives an external electrical interrupt request from the clock hardware. At this time the CPU stops whatever it is doing and reads the clock time. After the clock has been read and the time displayed, the CPU will continue with the program it was executing prior to the interrupt.

This is a simple example of an interrupt. A similar procedure will be followed by all external hardware that issues an electrical interrupt request to the CPU. The answer to the initial question of where do the interrupt requests come from, then, is that they come from the microprocessor system hardware.

5-3: Non-Maskable Interrupts

There are four interrupt input lines on the Z80 microprocessor: $\overline{\text{INT}}$, $\overline{\text{NMI}}$, $\overline{\text{BUSRQ}}$ and $\overline{\text{RES}}$; each one can cause an interrupt to the Z80.

Figure 5.1 shows the four input lines. In this section we will discuss the $\overline{\text{NMI}}$ or non-maskable interrupt input line. We will start with this interrupt because it operates in only one way. Most of the points we will be discussing, however, will apply to the other types of interrupt inputs as well.

A *non-maskable interrupt* is one that is always recognized by the Z80 as soon as electrically possible. This type of interrupt falls under the third possibility in our list in Section 5-1. Since the Z80 has a non-maskable interrupt, it follows that it would also have a maskable interrupt (the $\overline{\text{INT}}$ input pin). (We will discuss this input pin in a later section of this chapter.)

Pin 17 of the Z80 is labeled $\overline{\text{NMI}}$ (for non-maskable interrupt). This interrupt input is active on the negative-going edge of the signal (see Figure 5.2). The $\overline{\text{NMI}}$ input is completely asynchronous to the Z80. This means that the external hardware can issue or assert its request at any time without regard for the operating state of the CPU.

The next question then is, "When is the $\overline{\text{NMI}}$ interrupt input electrically recognized by the CPU?" Generally, the $\overline{\text{NMI}}$ request is recognized by the Z80 at the end of a current instruction cycle. This allows the Z80 to finish a complete instruction sequence before any action must take place for the interrupt. By allowing the Z80 to act in this way, the internal logic of the CPU can be simplified. Further, this sequence does not significantly slow down the CPU response to interrupt requests.

The first action to occur during an $\overline{\text{NMI}}$ operation is that the PC register

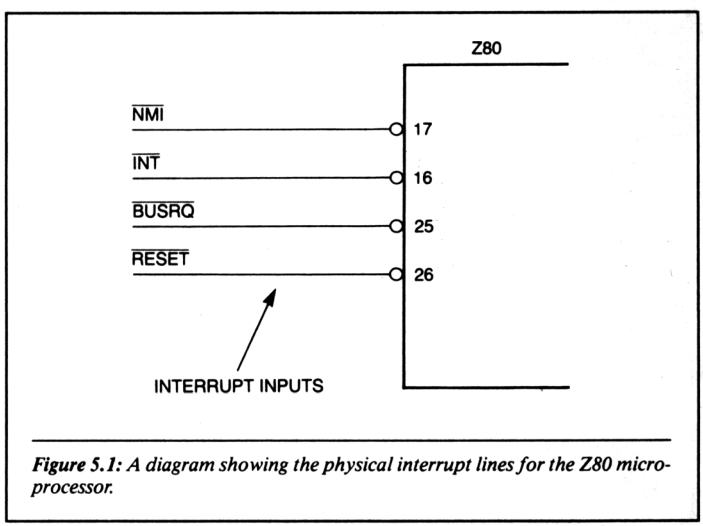


***Figure 5.1:** A diagram showing the physical interrupt lines for the Z80 microprocessor.*

is pushed onto the system stack. This saves in memory the address where the Z80 is to return after the interrupt servicing has been completed.

Next, the state of the interrupt flip-flop (IFF1) is stored in IFF2. IFF1 is labeled the "interrupt-enable flip-flop" and its function is to enable or disable interrupts at the $\overline{INT}$ input pin of the Z80. It is not used for non-maskable interrupts.

IFF1 is stored in IFF2 because the Z80 must save the logical state of this flip-flop during the handling of an $\overline{NMI}$. (IFF1 determines if the $\overline{INT}$ request will be accepted by the CPU.) This flip-flop is set to a logical 0 during the servicing of an $\overline{NMI}$ request. Doing this disables any interrupts from occurring during the servicing of an $\overline{NMI}$. (Later on when the Z80 returns from servicing the $\overline{NMI}$, the CPU must be restored to the operating state it was in prior to the NMI request. This means that IFF1 must be set to the state it was in prior to the occurrence of the interrupt.)

After IFF1 is stored in IFF2, and IFF1 is set to a logical 0, the Z80 will jump to location 0066H in memory.

Let's now quickly review the four events that occur in the Z80 when an $\overline{NMI}$ is accepted. In short:

1. The PC is pushed onto the STACK.
2. IFF1 is stored in IFF2.
3. IFF1 is set to a logical 0.
4. The Z80 jumps to location 0066H.

Location 0066H holds the service routine that must be executed during an $\overline{NMI}$. (Note that not all CPU registers are saved when an interrupt occurs. The programmer must save these registers, if wanted, during the interrupt service routine.) A typical start of an interrupt service routine is shown in Figure 5.3.

An alternative to saving all the registers is to execute the register exchange

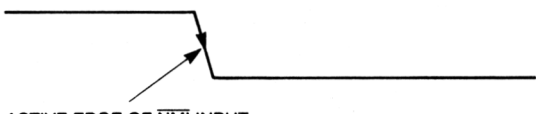


Figure 5.2: The $\overline{NMI}$ interrupt input line is active on the negative-going edge of the signal. When this line is asserted it sets an internal latch that remembers the interrupt request.

instructions. This allows the use of the alternate register set during the execution of an interrupt. Figure 5.4 shows the instructions for doing this.

```
PUSH AF
PUSH BC   } SAVE ALL REGISTERS
PUSH DE
PUSH HL
```

Figure 5.3: A typical start of an interrupt routine saves the status of the internal register. This is not always necessary and is dependent on the overall system application.

```
EX    AF,AF'  } SAVE REGISTERS
EXX
```

(NOTE: THESE TWO INSTRUCTIONS WILL SAVE THE CONTENTS OF ALL CPU REGISTERS.)

Figure 5.4: An alternative method for saving the internal registers of the Z80.

5-4: Clearing the $\overline{\text{NMI}}$ Request

The $\overline{\text{NMI}}$ electrical request was generated in hardware external to the Z80. One event that the Z80 must perform is to clear the condition (if possible or necessary) that caused the interrupt. One way to clear the interrupt request is by writing to an output port. (Later on we will show another way to do this.) The main point here is that the programmer is responsible for electrically removing the interrupt request.

5-5: End of the $\overline{\text{NMI}}$ Service Routine

At the end of the $\overline{\text{NMI}}$ service routine, the Z80 executes the RETN (return from non-maskable interrupt) instruction. At this time the two top data bytes on the stack are used for the return address. IFF2 is copied into IFF1. (*Note:* you can use the RET instruction for this operation, but IFF2 will not be copied into IFF1.)

5-6: An $\overline{\text{NMI}}$ Example

Here is an example that illustrates the use of the non-maskable interrupt. We will show a hardware technique that can be used to issue an $\overline{\text{NMI}}$ request, and we will present the basic software necessary to service the interrupt. Figure 5.5 shows a hardware circuit that can be used to request an $\overline{\text{NMI}}$. Let's examine it.

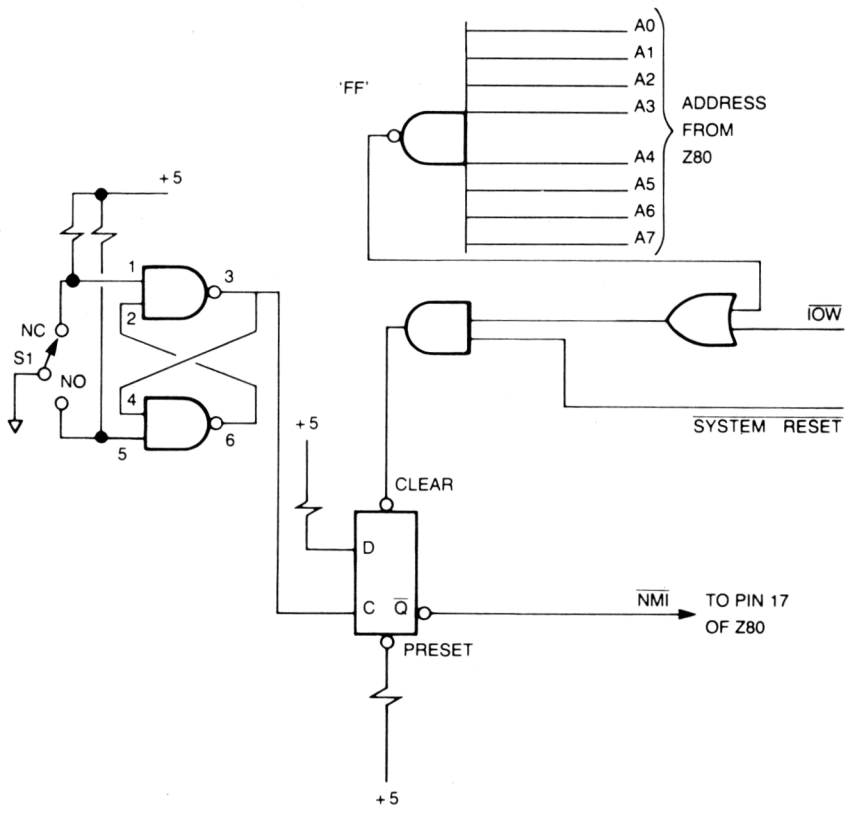


Figure 5.5: A schematic diagram showing one way an $\overline{\text{NMI}}$ request can be generated via the external hardware.

When the switch S1 (shown in Figure 5.5) is pressed, pin 3 of the 7400 will become a logical 0. This forces the clock input to the D flip-flop to a logical 0. Nothing happens at this time. When the switch S1 is released, the clock input to the D flip-flop goes to a logical 1. At this time the $\overline{Q}$ output is set to a logical 0. The $\overline{NMI}$ is electrically requested at the Z80 input pin 17.

The $\overline{NMI}$ input will remain a logical 0 until the Z80 performs an output write operation to port 0FFH. At that time the $\overline{Q}$ output of the D flip-flop will be set to a logical 1. This action will then set the $\overline{NMI}$ input to the Z80 to a logical 1 state.

It should be noted here that the $\overline{NMI}$ input is edge-sensitive not level-sensitive. This means that when the $\overline{NMI}$ input transitions from a logical 1 to a logical 0, the request is latched by the Z80 (as shown in Figure 5.6).

Because of the edge-triggered characteristic of the $\overline{NMI}$ input, a very simple interrupt request circuit could have been used. Such a circuit is shown in Figure 5.7. In this figure, when the switch is pressed, the $\overline{NMI}$ is accepted by the Z80. The switch must then be released and pressed again before another interrupt will occur. Note that unlike the circuit shown in Figure 5.5, the Z80 will not have to remove the interrupt request in this circuit.

5-7: $\overline{NMI}$ Summary

The preceding discussion has covered the important points of the $\overline{NMI}$ input on the Z80. Let's now review these points:

1. $\overline{NMI}$ is non-maskable, i.e., it will always be accepted.

2. $\overline{NMI}$ is edge-triggered on a transition from a logical 1 to a logical 0.

3. The Z80 pushes the PC onto the stack.

4. IFF1 is stored in IFF2, which saves the logical state of the interrupt flip-flop.

5. IFF1 is set to a logical 0, which disables other interrupts.

6. The Z80 jumps to memory address 0066H and starts to execute code.

7. RETN returns the PC address and restores the logical conditions of the interrupt-enable flip-flop.

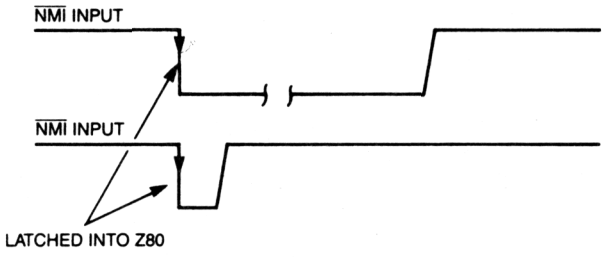


Figure 5.6: A timing diagram showing how the NMI request may be removed from the input line before the interrupt is electrically honored by the system microprocessor.

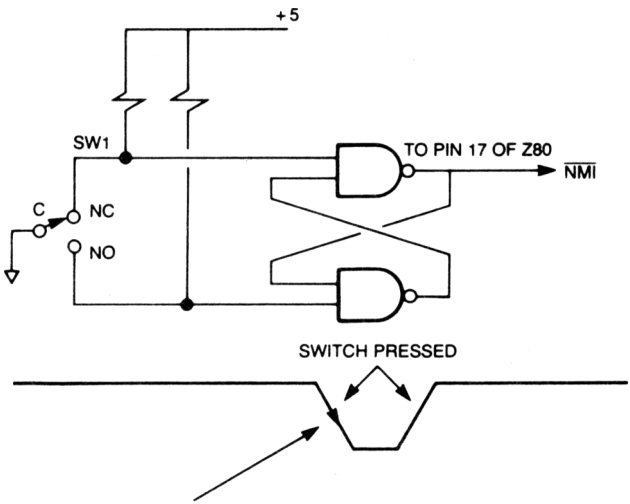


Figure 5.7: A simple method of generating an external NMI request for the Z80.

5-8: $\overline{\text{INT}}$ Input

We will now discuss how the $\overline{\text{INT}}$ input pin 16 of the Z80 operates. Much of the material presented on the $\overline{\text{NMI}}$ also applies to the $\overline{\text{INT}}$.

One electrical characteristic of the $\overline{\text{INT}}$ input to the Z80 is that it can be masked off logically. That is, there is an instruction for the Z80 labeled DI (Disable Interrupts) that when executed, will logically remove the $\overline{\text{INT}}$ input from the Z80 microprocessor. Recall that IFF1 is the internal flip-flop on the Z80 that enables or disables the $\overline{\text{INT}}$ input. It is set to a logical 0.

Although we have previously discussed IFF1, we have not detailed its use. Unlike the $\overline{\text{NMI}}$, the $\overline{\text{INT}}$ input is level-sensitive (rather than edge-sensitive). This means that the $\overline{\text{INT}}$ input must remain a logical 0 until the Z80 samples it. Sampling is done at the end of the last machine cycle of an instruction cycle. Recall that the $\overline{\text{NMI}}$ input could be removed from the Z80 before the sampling is done. We now know two main points about the Z80 $\overline{\text{INT}}$ input:

1. The $\overline{\text{INT}}$ can be electrically masked off by the Z80, using the DI instruction.

2. The $\overline{\text{INT}}$ input must remain a valid logical 0 until it is sampled by the Z80. The $\overline{\text{INT}}$ input is level-sensitive, rather than edge-sensitive, like the $\overline{\text{NMI}}$. This means that the $\overline{\text{INT}}$ input *must* be removed after the interrupt has been serviced. If the input is not removed, then another interrupt request will be granted. It is the responsibility of the system software to remove the interrupt request in order to avoid another unwanted interrupt.

There are certain peripheral devices designed for use with the Z80 that will automatically remove the interrupt request at the correct time. One example of this type of device is the Z80-PIO. The *PIO* recognizes when the Z80 is executing a RETI instruction and automatically removes the request from the system $\overline{\text{INT}}$ line. We will discuss the PIO and some of these other devices in detail in later chapters.

There is only one $\overline{\text{INT}}$ input pin on the Z80, but it operates in three modes. When using the Z80 interrupt structure you must program the mode that will be used. This is done prior to any interrupts being input to the CPU. Mode names for the Z80 interrupts are labeled mode 0, mode 1, and mode 2. To program these three modes there are three unique instructions: IM0, IM1, IM2.

When the Z80 is initially powered up, or reset, interrupt mode 0 is defaulted to. In addition, the interrupt flip-flop IFF1 is set to a logical 0, and the interrupts are disabled. Let's now discuss the three modes of interrupt operation—one at a time—and point out their differences and similarities.

5-9: Mode 1 Interrupt

Let's begin with mode 1 interrupts. We will start with this mode because it is the most like $\overline{\text{NMI}}$; and we will, therefore, be starting on familiar ground. The Z80 can be programmed into mode 1 using the IM1 instruction.

For this discussion, we will assume that the interrupts have been enabled with the EI (Enable Interrupt) instruction, and the Z80 is prepared to handle interrupt requests on the $\overline{\text{INT}}$ input line.

When the $\overline{\text{INT}}$ input goes to a logical 0 and the Z80 acknowledges the input the following will occur:

1. IFF1 will be set to a logical 0, thus disabling any further interrupt inputs from the $\overline{\text{INT}}$ line.
2. IFF2 will be set to a logical 0. Recall that for the $\overline{\text{NMI}}$, the IFF2 was a temporary storage location for the IFF1. In this interrupt mode, however, IFF2 is not used for that purpose.
3. The PC will be stored on the program stack. This will be the memory address of the instruction to be executed when the interrupt servicing is complete.
4. The Z80 will jump to location 0038H in system memory, and start executing software code.

You can see from this list that essentially the same types of operations are performed by the $\overline{\text{NMI}}$ interrupt input.

During execution of the interrupt service routine, the CPU registers must be saved if the routine is using them. This can be done by using the EX (Exchange) instruction or by pushing the used registers onto the system stack.

During execution of the service routine, the interrupt request must be logically removed from the Z80 input pin 16. The actual software required to accomplish this task varies from system to system, depending on the hardware used to assert the interrupt request. Figure 5.5 shows one way that hardware can assert the interrupt request input. In this figure the Z80 performs an output write operation to port 0FFH to remove the interrupt request from the $\overline{\text{INT}}$ input pin 16. For the Z80 to accept any further interrupts, the IFF1 flip-flop must be set to a logical 1. This is accomplished when the CPU executes the EI instruction, prior to a return from the interrupt in the service routine.

Finally, the programmer must allow the CPU to resume execution of the program that was running before the interrupt request was asserted. This is accomplished by using the RET or RETI instruction. The RET instruction will pop the two top bytes from the system stack and store them in the PC of the Z80. Recall that these two bytes are the address that the CPU would have

executed if the interrupt had not occurred. This is the same RET instruction that is executed at the end of a subroutine call.

The RETI instruction performs the same operation as the RET instruction. The major difference between these two instructions is that certain peripheral devices, such as the Z80-PIO, are designed to electrically recognize the RETI opcode. When an external device recognizes this opcode, the interrupt request is updated. (You will learn more about this feature in the chapters that follow.)

Figure 5.8 shows a sample interrupt service routine. This routine assumes that the Z80 was programmed to mode 1, and that the interrupt was input to the Z80 using the hardware described in Figure 5.5. In Figure 5.8 the OUT (OFFH),A instruction is used to clear the interrupt request before the system interrupts are again enabled with the EI instruction.

It should be noted that the interrupts are not enabled until the opcode after the EI instruction is executed. This allows the Z80 to execute the return

```
ORG     0038H

PUSH AF
PUSH HL   } SAVE ALL REGISTERS OR ONLY THE ONES
PUSH DE   } USED IN THE ROUTINE
PUSH BC

(SERVICE ROUTINE)
     SECTION


POP BC
POP DE    } RESTORE ALL REGISTERS
POP HL
POP AF

OUT (0FFH),A          CLEAR INTERRUPT
     EI               ENABLE INTERRUPTS
RET OR RETI
```

***Figure 5.8:** A sample interrupt routine that assumes the Z80 was programmed in mode 1 interrupt architecture.*

instruction at the end of the interrupt service routine—before another interrupt request is internally accepted.

In the program in Figure 5.8 the interrupts are disabled as soon as the Z80 accepts the interrupt request. This occurs when the Z80 sets the IFF1 to a logical 0. The IFF1 will remain a logical 0 until it is again set to a logical 1 by the system software. This can be done at the end of any interrupt service routine that does not allow any interrupts while an interrupt is being serviced. However, the programmer may choose to enable the interrupts during the service routine to allow other interrupts to be input. It is up to the system designer to decide if an interrupt routine should be interrupted by another request.

5-10: Mode 0 Interrupt

In this section we will discuss a second mode of operation for the interrupt structure of the Z80: mode 0. Mode 0 is enabled when the CPU first powers up, or when reset is active, or when an IM0 instruction is executed. This interrupt structure is often equated to the one on the 8080.

With the mode 0 interrupt, the CPU will input an interrupt request in the same manner as in mode 1 (described earlier). However, rather than the CPU jumping to location 0038H only, it can be made to jump to one of eight pre-selected addresses or for that matter to any memory address. This is a very nice feature. Let's examine how it works.

If we assume that the interrupt request has been input to the Z80 and is accepted, this is what occurs:

1. IFF1 and IFF2 are set to a logical 0, thus disabling any further interrupt requests.

2. The $\overline{M1}$ output line and the $\overline{IORQ}$ output line on the Z80 both go to a logical 0. This situation occurs only on the Z80 during an interrupt, and is given the special name of *interrupt acknowledge* state.

3. External hardware decodes the $\overline{M1}$ and the $\overline{IORQ}$ to equal 0 and enables a single data byte onto the Z80 data bus. The data byte enabled onto the data bus is the opcode for an RST 0–RST 7. This byte could also be a CALL instruction. We will assume here that the byte placed on the system data bus was an RST instruction.

4. The Z80 reads the byte and interprets it as an opcode.

5. The PC is pushed onto the system stack.

6. Finally, the Z80 jumps to the memory location specified by the RST instruction.

These locations include:

RST	DATA BYTE	MEMORY LOCATION (HEX)
0	C7	0000
1	CF	0008
2	D7	0010
3	DF	0018
4	E7	0020
5	EF	0028
6	F7	0030
7	FF	0038

Figure 5.9 shows a sequence of events for the hardware activity during

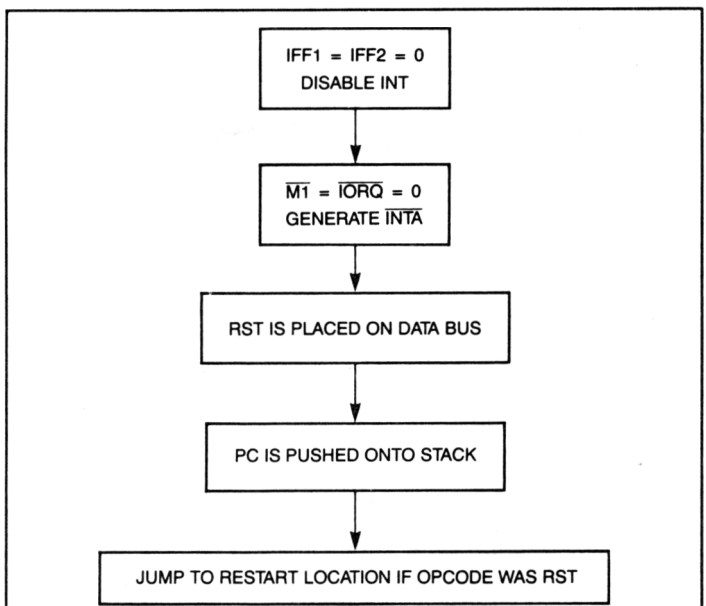


Figure 5.9: A flowchart showing the sequence of internal events that occurs when the Z80 is electrically receiving a mode 0 interrupt request.

this operation. Figure 5.10 shows one way that this particular interrupt input scheme can be realized. Here is how Figure 5.10 operates.

In this figure the $\overline{M1}$ and the $\overline{IORQ}$ are input to an OR gate. When the interrupt request is input to the Z80 and accepted, $\overline{M1}$ and $\overline{IORQ}$ will go to a logical 0, and, thus, generate an interrupt acknowledge. When this occurs, the output of the OR gate is a logical 0. In this example, logical 0 enables the tri-state buffer and places a DF hexadecimal on the system data bus. A DF is equal to an RST 3 instruction. It causes the Z80 to execute a CALL to memory address 0018H.

This example has described one way to place a data byte on the system data bus. This example has included all of the aspects of using the interrupt mode 0. It can, however, handle only one external device requesting an interrupt. In a later section of this chapter we will discuss how a multiple interrupt request can be handled by the Z80. For now it is important that you understand how the CPU services mode 0 interrupts.

We stated at the start of this section that it is acceptable to place a CALL

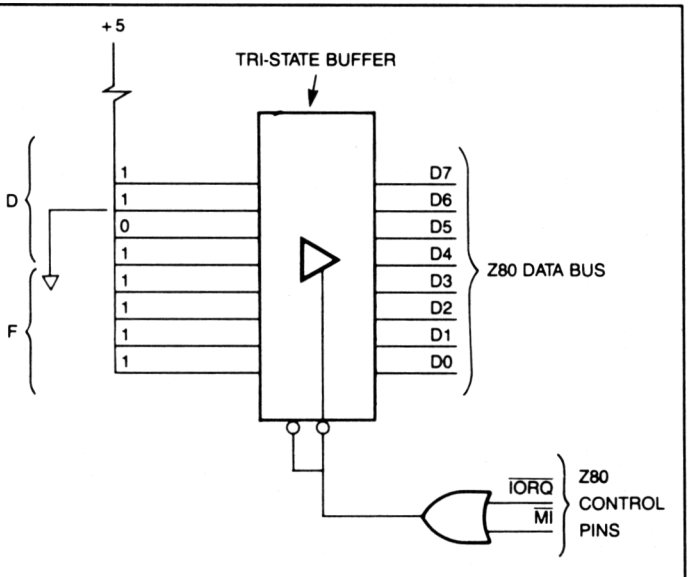


Figure 5.10: *Sample hardware for inputting an RST vector on the Z80 data bus during an interrupt acknowledge cycle.*

instruction, rather than an RST instruction, on the data bus. When the Z80 has the CALL opcode on the bus during the interrupt acknowledge time, it expects to get the next two bytes for the CALL address.

The next two machine cycles after the interrupt acknowledge are memory read cycles. During these two cycles the address for the call is read into the CPU. It is the responsibility of the external hardware to place the address on the data bus at the correct time. The flowchart in Figure 5.11 shows exactly

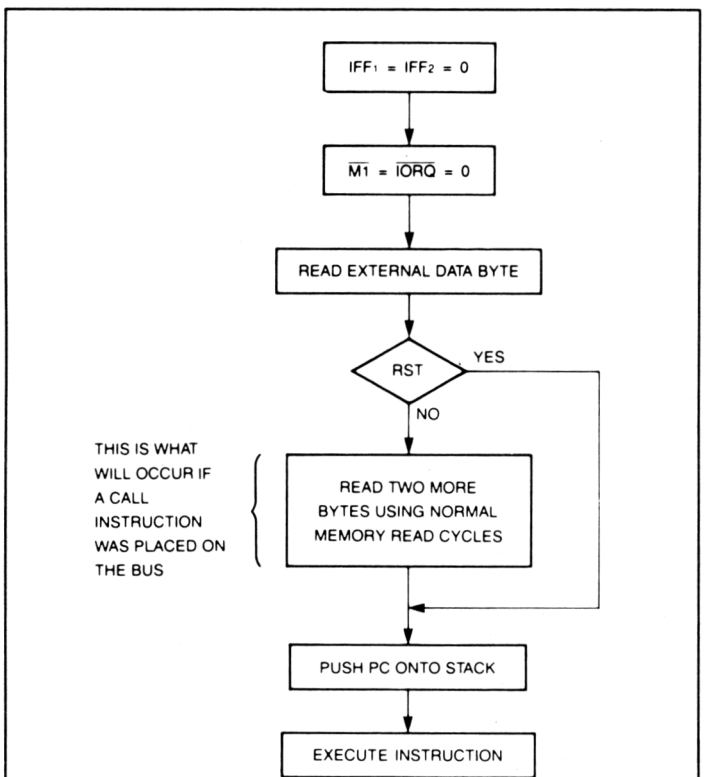


Figure 5.11: *A flowchart showing the events that occur during the time the Z80 receives an RST vector or a CALL instruction.*

what occurs on the Z80 bus when either an RST or a CALL instruction is placed on the system data bus during the interrupt acknowledge cycle.

Figure 5.11 shows that the RST instruction executes faster than the CALL instruction, but that the CALL instruction has the advantage of allowing the Z80 to jump to any address location to execute the interrupt service routine.

5-11: Mode 2 Interrupt

In this section we will discuss the third mode of Z80 interrupts, mode 2. This mode is quite powerful and straightforward to use. Mode 2 interrupts are set up by the Z80 microprocessor when it executes the IM2 instruction. These interrupts are electrically requested in exactly the same way as the mode 0 and mode 1 interrupts. Therefore, we will now concentrate on how the Z80 responds to a mode 2 interrupt request.

The general concept of the mode 2 structure is the following. When the Z80 acknowledges the interrupt request by asserting $\overline{M1}$ and $\overline{IORQ}$ at the same time, the external hardware places a data byte on the system data bus. This is the same action that occurs in a mode 0 interrupt acknowledge; however, the similarity ends there. The Z80 takes the data byte and forms a 16-bit word with an internal 8-bit register. This internal register is the I register. See Figure 5.12.

The data located at the new 16-bit address is another address. This new address is the absolute memory address of the interrupt service routine. (See Figure 5.13.) Generally, this is how the mode 2 interrupts operate. Let's examine an example.

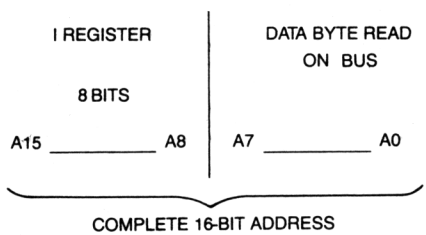


Figure 5.12: The I register and the external data byte are used to form the complete 16-bit address that will point to system memory to obtain the first byte of the interrupt vector address.

For this example, we will assume that when the mode 2 interrupt is acknowledged, the external device will place 52H on the data bus. Further, we will assume that the I register in the Z80 has been initialized to 19H, and that memory locations 1952H and 1953H contain the following:

1952H = 38H as data
1953H = 2FH as data

Let's assume that the interrupt has just occurred. The 52H from the data bus and the 19H in the I register are grouped together thus, giving a memory address of 1952H. The I register takes the upper 8 bits, and the data byte read-in takes the lower 8 bits. The Z80 takes the data stored at this address and uses it as the lower byte of a new memory address. In this example, the lower byte of the new address would be 38H. The Z80 next reads the data byte at the memory address 1952H + 1. This data is used as the upper byte of the new memory address. The total memory address formed is 2F38H. The Z80

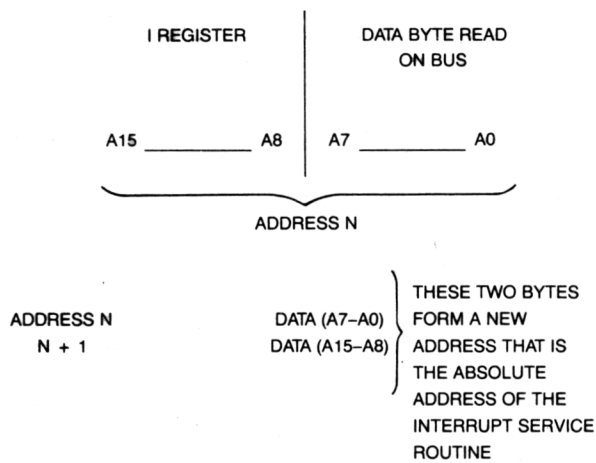


Figure 5.13: A diagram showing how the 16-bit pointer address is formed in a mode 2 interrupt scheme.

now jumps to the address and executes the interrupt service routine.

This may seem like a lot of effort, to form a single interrupt address, but let's examine what this type of architecture will do in a system environment. Suppose you had a system with several (say 10) external devices that could request an interrupt. Each device could be made to place a different byte on the data bus when the interrupt is acknowledged. By reading this single byte, the Z80 can be made to jump to any system memory address. Further, the Z80 can go to a different address for each of the 10 devices. Let's now see an example of how this can be made to occur in a system.

To start, it is necessary to know the system address that the interrupt service routine will reside at. For our purposes we will assume the following addresses:

DEVICE #	MEMORY ADDRESS
DEV 1	20F5H
DEV 2	3802H
DEV 3	1951H
DEV 4	F318H
DEV 5	E821H
DEV 6	2568H
DEV 7	1585H
DEV 8	CE80H
DEV 9	3597H
DEV 10	211EH

We must now construct a table of vectors that will be the memory addresses that point to the absolute address of the service routine. In the previous example, 2F38H was a vector pointing to the service routine of a specific hardware component.

The preceding interrupt service routines will reside at these specified locations in system memory. (*Note:* For this example, the locations have been arbitrarily chosen and serve only to illustrate the concept we are discussing.)

Next, we must decide where in system memory to place the table of address vectors. There is a maximum of 256 memory locations used for the interrupt vectors. Each vector requires two bytes; therefore, a maximum of 128 unique memory addresses or vectors may be specified using mode 2 interrupts. However, this is seldom a limitation in the system design.

For this example, we will set our vector space at locations 1100H-11FFH. This is an arbitrary choice. (It is generally determined by the system designer.) Once this space is established we must decide which data byte the external device will place on the data bus when the interrupt is acknowledged

by the Z80. For now, let's assume that the devices follow the following pattern:

DEVICE #	BYTE PLACED ON DATA BUS
DEV 1	00H
DEV 2	02H
DEV 3	04H
DEV 4	06H
DEV 5	08H
DEV 6	0AH
DEV 7	0CH
DEV 8	0EH
DEV 9	10H
DEV 10	12H

Notice that each byte will place an even number on the data bus. This is because the total vector will occupy two consecutive locations in the system memory. Once we know this information, we can fill up the vector table. In this example it would appear like this:

MEMORY ADDRESS	DATA	DEVICE VECTOR
1100H	F5H	DEV 1
1101H	20H	
1102H	62H	DEV 2
1103H	38H	
1104H	51H	DEV 3
1105H	19H	
1106H	18H	DEV 4
1107H	F3H	
1108H	21H	DEV 5
1109H	E8H	
110AH	68H	DEV 6
110BH	25H	
110CH	85H	DEV 7
110DH	15H	
110EH	80H	DEV 8
110FH	CEH	
1110H	97H	DEV 9
1111H	35H	
1112H	1EH	DEV 10
1113H	21H	

A Z80 program to set up this example appears in Figure 5.14.

5-12: Multiple Devices Requesting Interrupts

In this section we will show how more than one external device can be used to request an interrupt with the Z80. We will use this information in

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;   PROGRAM TO SETUP MODE 2 INTERRUPT FOR Z80
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
0000                    CODE 0000
;
0000    ED5E            IM 2        ;SET UP INTERRUPT MODE 2
0002    3E11            LD A,11H    ;UPPER BYTE OF VECTOR TABLE ADDDR
0004    ED47            LD I,A
0006    FB              EI          ;ENABLE INTERRUPTS
;
;
;  ALL OF THE INTERRUPT SERVICE ROUTINES WOULD
;  BE WRITTEN IN THIS SPACE OF THE LISTING
;  THESE WILL NOT BE SHOWN
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; NOW TO SET UP THE VECTOR TABLE
; THE FOLLOWING IS ONLY ONE WAY IN WHICH IT
; CAN BE ACCOMPLISHED
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
1100                    CODE 1100H      ;ADDRESS OF VECTOR TABLE
;
1100    F520            DEFW 20F5H      ;DEVICE 1
1102    0238            DEFW 3802H      ;DEVICE 2
1104    5119            DEFW 1951H      ;DEVICE 3
1106    18F3            DEFW 0F318H     ;DEVICE 4
1108    21E8            DEFW 0E821H     ;DEVICE 5
110A    6825            DEFW 2568H      ;DEVICE 6
110C    8515            DEFW 1585H      ;DEVICE 7
110E    80CE            DEFW 0CE80H     ;DEVICE 8
1110    9735            DEFW 3597H      ;DEVICE 9
1112    1E21            DEFW 211EH      ;DEVICE 10
;
;  END OF THE VECTOR TABLE
```

Figure 5.14: *A sample Z80 program to set up the microprocessor for mode 2 interrupts and to generate the vector table.*

later chapters when we discuss the Z80 peripheral devices. Figure 5.15 shows a block diagram of a possible multiple interrupt system. There are several ways for the CPU to handle this condition. In this section we will discuss the main techniques: polling, priority interrupt and daisy chain priority.

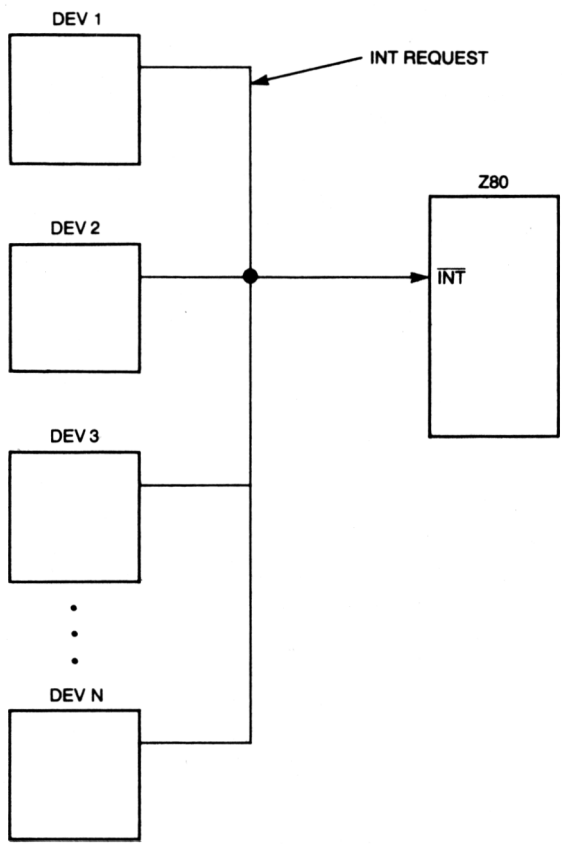


***Figure 5.15:** A block diagram of a system that can generate multiple interrupt requests to the Z80 microprocessor.*

5-13: Polling

The first technique we will describe is *polling*.

Let's assume that the Z80 gets an interrupt request. The interrupt service routine of the Z80 instructs the CPU to read a particular byte, known as the *status byte,* from each peripheral device. Generally, one bit of the status byte will logically inform the Z80 if this particular device has requested the interrupt. However, if more than one device has requested the interrupt, the Z80 must decide which device is to be serviced first. This can be done by polling.

During polling, the Z80 has the interrupt input pin asserted, making it possible for all external devices to assert the interrupt line. However, there is no way for the CPU to *directly* determine which external device asserted the interrupt request. In fact, all external devices could have their interrupt lines physically connected together due to the *open drain* (for MOS) or *open collector* (for TTL) structure of the outputs. See Figure 5.16.

The polling technique has a disadvantage in that interrupts are not handled in a rapid manner because the Z80 must poll or read a byte from each device. An advantage of this technique, however, is that the hardware is straightforward and simple to implement.

5-14: Priority Interrupts

The *priority* interrupt technique allows the Z80 to efficiently handle multiple interrupt requests. Let's see how it works.

Each external device is electrically set up to place a particular byte on the data bus when it has been interrupt acknowledged by the Z80. (This concept was discussed in detail when we examined the mode 0 and mode 2 interrupts.) In a multiple interrupt system each device cannot simply place the data byte on the bus during an interrupt acknowledge because doing this causes erroneous data to be on the data bus as multiple sources try to gain control of the bus. Therefore, the CPU needs to selectively acknowledge the interrupt requests.

To do this the hardware of the system must set up a priority scheme for the external devices. The logic of this scheme (shown in the block diagram in Figure 5.17) is that each external device inputs its interrupt request to a logic block called *priority logic.* This block electrically decides which external interrupt line is to be enabled through to the CPU, and *when* it is to be enabled. When the CPU acknowledges the interrupt, the logic block then issues the interrupt acknowledge to the external device that was allowed to interrupt the CPU. In effect, the priority logic "directs traffic" for the interrupt requests and acknowledges.

An advantage of this type of interrupt scheme is that interrupts are handled quickly and efficiently by the CPU. A disadvantage is that extra hardware is required to implement the scheme.

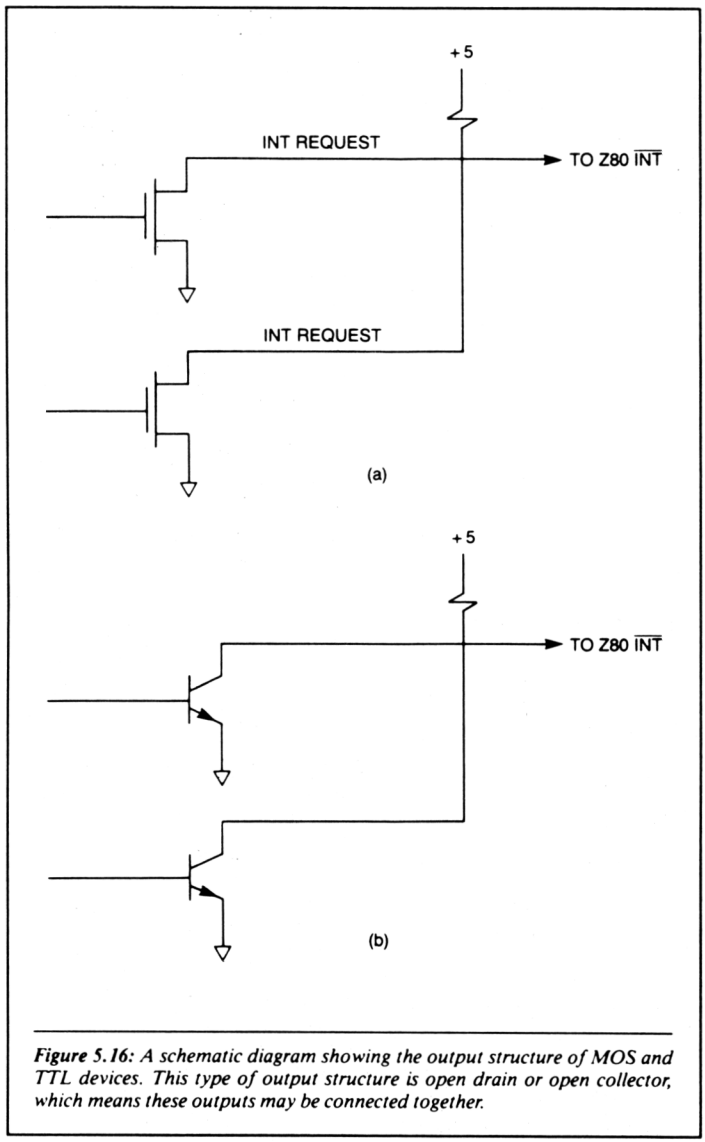


Figure 5.16: A schematic diagram showing the output structure of MOS and TTL devices. This type of output structure is open drain or open collector, which means these outputs may be connected together.

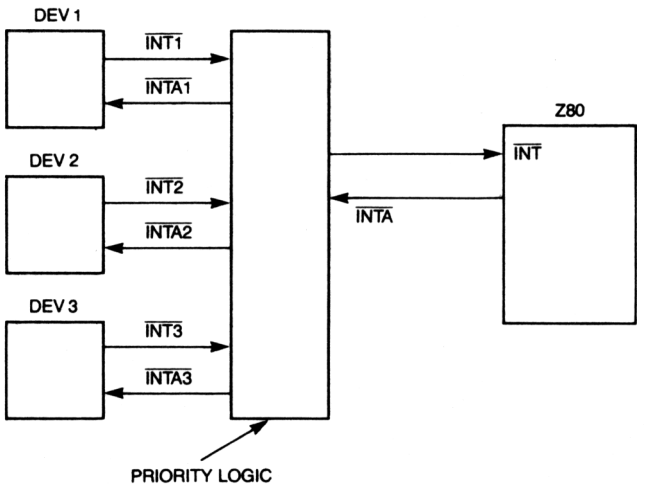


Figure 5.17: A block diagram showing one way external interrupt requests may be prioritized. This technique makes use of external logic to direct the interrupt requests and interrupt acknowledges to the requesting devices.

5-15: Daisy Chain Priority

Another variation of the priority interrupt scheme is the *daisy chain* architecture. A block diagram of a daisy chain interrupt system is shown in Figure 5.18. This approach automatically disables external devices from electrically requesting interrupts that are lower on the chain, when a higher device is requesting an interrupt (see Figure 5.18). If DEV 1 is requesting an interrupt, DEV 2-DEV 4 will be disabled. When the interrupt acknowledge is issued by the CPU, only the active device on the daisy chain will respond.

If a device on the chain is disabled and conditions exist that would cause an interrupt if the device were enabled, the request will be held off until the device is enabled—thus, no interrupts are lost in the daisy chain architecture.

Many Z80 peripheral devices are designed to be used with the daisy chain approach. These devices have built-in hardware and physical device pins that make their connection in the daisy chain straightforward and simple. In later chapters we will show how this approach is used.

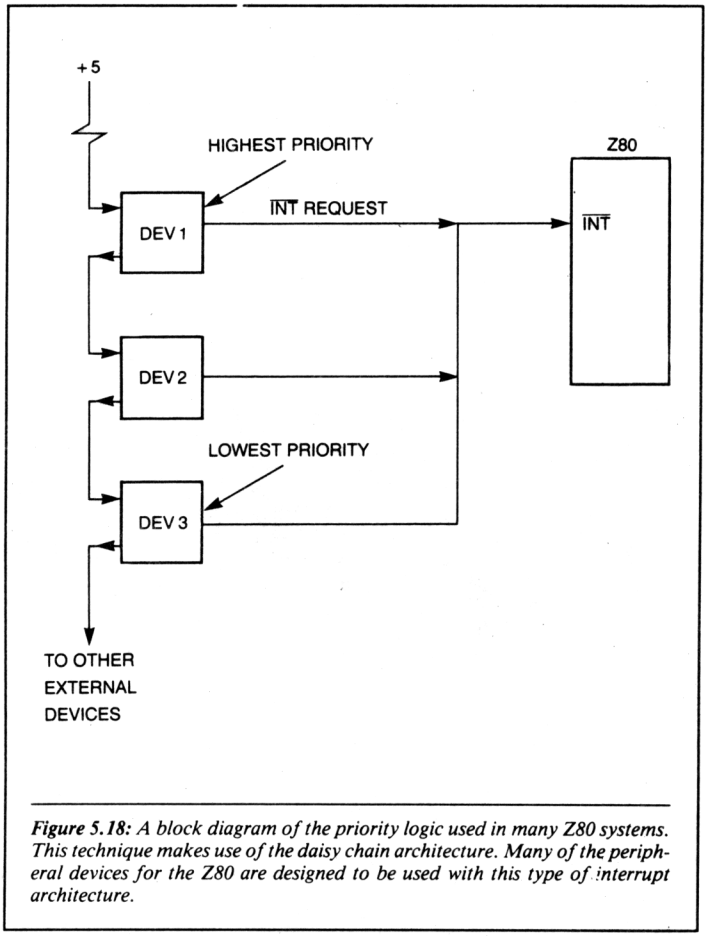
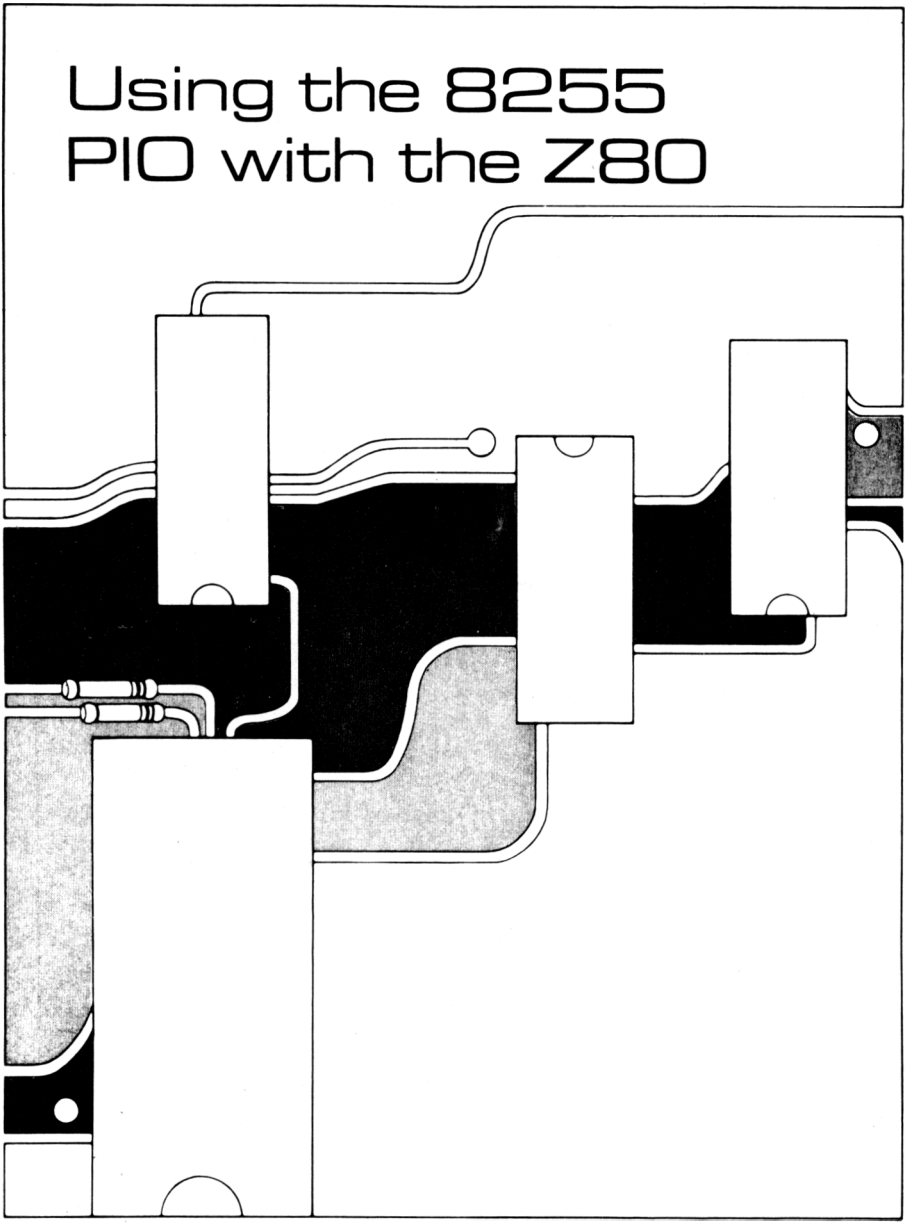


Figure 5.18: A block diagram of the priority logic used in many Z80 systems. This technique makes use of the daisy chain architecture. Many of the peripheral devices for the Z80 are designed to be used with this type of interrupt architecture.

CHAPTER SUMMARY

In this chapter we have examined various techniques for interrupting the Z80. We began with a look at the interrupt concept in general. We then discussed each of the three modes of interrupts available on the Z80: polling, priority and daisy chain.

Using the 8255 PIO with the Z80



Chapter 6

INTRODUCTION

The first large scale integration (or LSI) peripheral device we will discuss in this text is the 8255 programmable interface adapter. We will examine it to see:

1. How it operates.
2. How it connects electrically with the Z80 busses.
3. How it can be programmed to operate as a versatile I/O chip.

6-1: Overview of the 8255

The 8255 is a 40 pin, dual-in-line package (DIP), LSI device, designed to perform a variety of interface functions in a microprocessor system environment. Unlike many of the peripheral devices we will be discussing in this text, the 8255 was not originally designed to be used with the Z80 microprocessor. It was first manufactured by Intel Corporation for use with the 8080 microprocessor.

Let's begin with an explanation of how the 8255 functions prior to its connection to the Z80 busses and prior to programming. Once you are familiar with the 8255 at this basic level, you can easily explore its uses with your own and other applications.

Figure 6.1 shows a block diagram of the 8255. Let's examine the function of each block.

In this figure, we can see that there are four blocks that connect to physical lines from the PIO that can connect with an external device. These lines are labeled PA0–PA7, PB0–PB7, and PC0–PC7. The groups of signals are logically divided into three different I/O ports, labeled port A (PA), port B (PB), and port C (PC). Each port has two separate I/O blocks associated with it, and each block is connected to the 8255 internal data bus. It is via this internal data bus that information is exchanged within the 8255.

In Figure 6.1 there are two blocks, labeled group A control and group B control, that define how the three I/O ports operate. (There are several different operating modes for the 8255 and these modes must be defined by the CPU writing programming or control words to the device.) Notice that group C of the 8255 consists of two 4-bit ports. One of the groups is

8255A/8255A-5
PROGRAMMABLE PERIPHERAL INTERFACE

- **MCS-85™ Compatible 8255A-5**
- **24 Programmable I/O Pins**
- **Completely TTL Compatible**
- **Fully Compatible with Intel® Microprocessor Families**
- **Improved Timing Characteristics**
- **Direct Bit Set/Reset Capability Easing Control Application Interface**
- **40-Pin Dual In-Line Package**
- **Reduces System Package Count**
- **Improved DC Driving Capability**

The Intel® 8255A is a general purpose programmable I/O device designed for use with Intel® microprocessors. It has 24 I/O pins which may be individually programmed in 2 groups of 12 and used in 3 major modes of operation. In the first mode (MODE 0), each group of 12 I/O pins may be programmed in sets of 4 to be input or output. In MODE 1, the second mode, each group may be programmed to have 8 lines of input or output. Of the remaining 4 pins, 3 are used for handshaking and interrupt control signals. The third mode of operation (MODE 2) is a bidirectional bus mode which uses 8 lines for a bidirectional bus, and 5 lines, borrowing one from the other group, for handshaking.

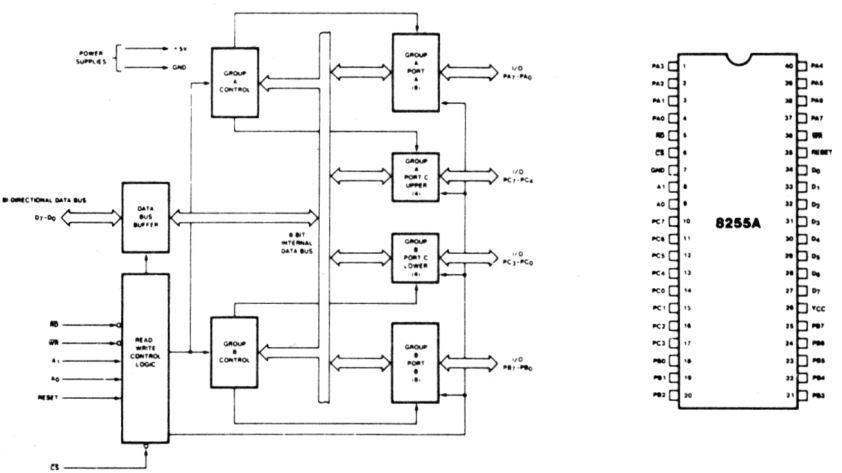


Figure 6.1: A block diagram and device pinout of the 8255 PIA.

associated with group A and the other with group B device signals. (*Note:* The reason for this division will become clear later on. For now it is only necessary to recognize the fact.)

The final logic blocks shown in Figure 6.1 are labeled data bus buffer and read/write control logic. These blocks provide the electrical interface between the Z80 microprocessor and the 8255. The data bus buffer buffers the data input and output lines to and from the CPU data bus. The read/write control logic routes the data to and from the correct internal registers with the right timing. The internal path being enabled depends on the type of operation being performed by the CPU; that is, it depends on whether the operation is an I/O read or an I/O write.

6-2: Pinout Description of the 8255

In this section we will examine each pin of the 8255 device and discuss its function. This information will be useful later on when we discuss how each pin connects with the Z80 microprocessor system busses. A pinout of the 8255 appears in Figure 6.1. Let's now examine each pin:

D0–D7 These are the data input and output lines for the device. All information written to and read from the 8255 occurs via these eight data lines.

$\overline{\text{CS}}$ (*Chip Select Input*) Whenever this input line is a logical 0, the microprocessor can electrically read and write to the 8255.

$\overline{\text{RD}}$ (*Read Input*) Whenever this input line is a logical 0 and the $\overline{\text{CS}}$ input is a logical 0, the 8255 data outputs are enabled onto the system data bus.

$\overline{\text{WR}}$ (*Write Input*) Whenever this input line is a logical 0 and the $\overline{\text{CS}}$ input is a logical 0, data is written to the 8255 from the system data bus.

A0–A1 (*Address Inputs*) The logical combination of these two input lines determines which internal register of the 8255 data is written to or read from.

RESET Whenever this input line is a logical 1, the 8255 is placed in its reset state. All peripheral ports are set to the input mode.

PA0–PA7 These signal lines are used as an 8-bit I/O port that can be connected to peripheral devices.

PB0–PB7 These signals lines are used as an 8-bit I/O port that can be connected to peripheral devices.

PC0–PC7 These signals lines are used as an 8-bit I/O port that can be connected to peripheral devices. Also, this group of signals may be

divided into two 4-bit groups and used to control the PB0–PB7 lines and the PA0–PA7 lines in certain 8255 operating modes.

Now that we are familiar with the 8255 device pins, let's use this information to connect the chip to the Z80 system busses.

6-3: Connecting the 8255 to the Z80 CPU

Let's now use the 8255 as an output port in the Z80 system. We will connect it to the Z80 CPU and assume that all communication will occur as I/O operations. The 8255 has two address inputs, labeled A0 and A1; this implies that the 8255 contains four unique I/O addresses. In normal use, the A0 and A1 inputs to the device are connected to the A0 and A1 memory address output lines from the Z80 CPU.

The $\overline{CS}$ input line on the 8255 is used to logically map the device into the system I/O space. For example, let's suppose we assign the I/O addresses equal to 10H, 11H, 12H, and 13H. We can logically and electrically accomplish this addressing by using a circuit like the one shown in Figure 6.2.

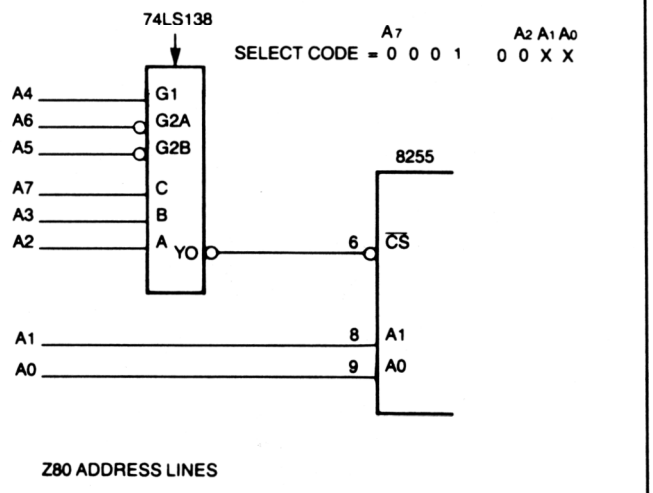


Figure 6.2: A logic diagram showing how the $\overline{CS}$ input to the 8255 can be made to respond to the address lines A2–A7.

Recall from Chapter 3 that an I/O operation requires a decoding of the lower eight address lines of the Z80, lines A7–A0. In Figure 6.2 the $\overline{CS}$ input becomes active only when the upper six address bits (A7–A2) equal 000100XX. (The lower two address bits are used to define which of the four internal registers the data is being written to or read from.) This I/O architecture is sometimes referred to as *device port I/O.* That is, the device address has four port addresses associated with it.

The next thing we do is to connect the $\overline{RD}$ and the $\overline{WR}$ input lines to the $\overline{IOR}$ and $\overline{IOW}$ control signals of the Z80 system. We do this because in our application, the $\overline{RD}$ and $\overline{WR}$ inputs to the 8255 cannot be connected directly to the $\overline{RD}$ and $\overline{WR}$ outputs from the Z80. This is due to the fact that with the Z80, these lines are also active during memory operations and if your application does not make use of memory mapped I/O, this connection will not be valid. Figure 6.3 shows one way the $\overline{IOR}$ and $\overline{IOW}$ can be connected to the 8255.

The final control signal that connects to the 8255 is the reset input. An important point to note about the reset input is that it is active in the logical 1 state. Reset input to the Z80 is active in the logical 0 state. Therefore, if it is to be used for the 8255 it is necessary to invert it after it is applied to the Z80.

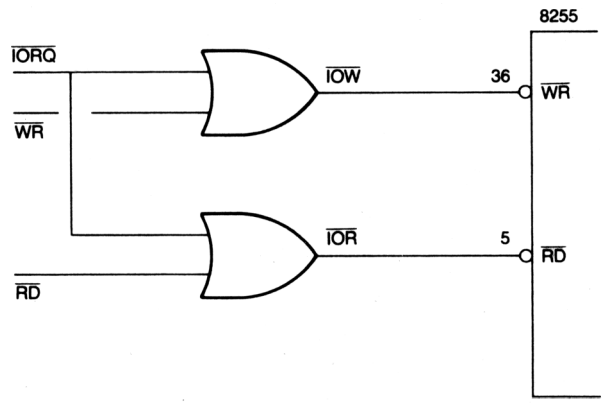


Figure 6.3: *The $\overline{WR}$ and $\overline{RD}$ inputs to the 8255 can be physically connected to the system $\overline{IOR}$ and $\overline{IOW}$ control lines. These system lines become active whenever the Z80 is communicating with an I/O device.*

Data lines D0–D7 can be connected directly to the 8255 data input lines. Figure 6.4 shows a complete connection between the Z80 and the 8255. We are assuming with this connection that the application does not require data buffering. That is, the load on the system data bus can be driven directly by the 8255 data output lines.

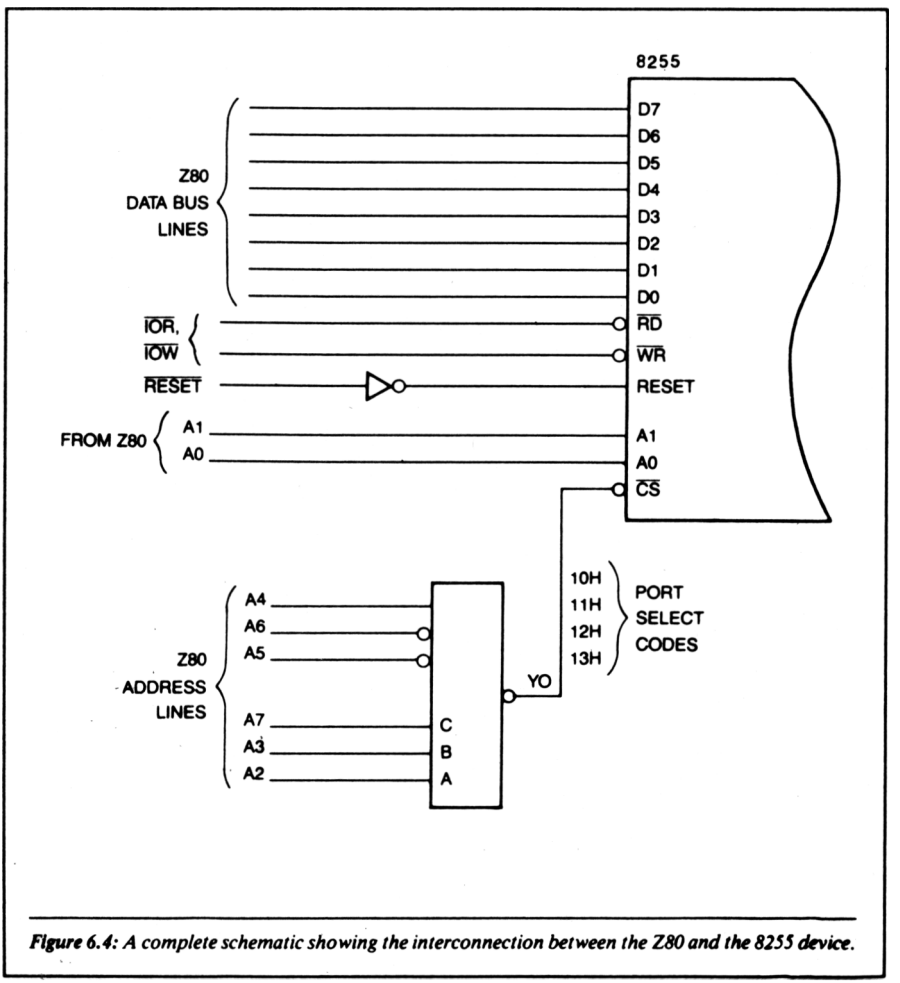


Figure 6.4: A complete schematic showing the interconnection between the Z80 and the 8255 device.

6-4: The 8255 Read and Write Registers

Now that we have the 8255 connected to the Z80, let's learn how to program the device so that it operates in the manner needed for our application. We will start by discussing the four internal registers (i.e., the four read and write registers) contained in the 8255. In our decoding example, the address of these registers is 10H, 11H, 12H, and 13H. The basic register definitions are:

DEVICE PINS				REGISTER NAME
$\overline{RD}$	$\overline{WR}$	A1	A0	
1	0	0	0	write PORT A data
0	1	0	0	read PORT A data
1	0	0	1	write PORT B data
0	1	0	1	read PORT B data
1	0	1	0	write PORT C data
0	1	1	0	read PORT C data
1	0	1	1	write control word
0	1	1	1	illegal read register

The function of registers 0–2 is defined by the word written to the control register 3. Figure 6.5 shows the bit definition of the control register. Let's now examine some of the important operating modes of the 8255 by looking at several Z80 programming examples for the device. We will now discuss the three operating modes for the 8255 device and learn how it is used in each mode.

6-5: Mode 0—Basic Register I/O

To set up mode 0 for basic register I/O, the programmer must first write a control word to the control register. This word will define how the registers are to be used in the 8255 device. The bits of the control register used for programming the standard I/O function should appear like this:

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	0	0	0	0

If we refer to Figure 6.5, we can see that:

- Bit D7 defines this word as a control word.
- Bits D6 and D5 define the mode of operation for the 8255 port A. This will be mode 0.
- Bit D4 = 0 indicates that port A is an output.
- Bit D3 = 0 sets the port C upper 4 bits as outputs.

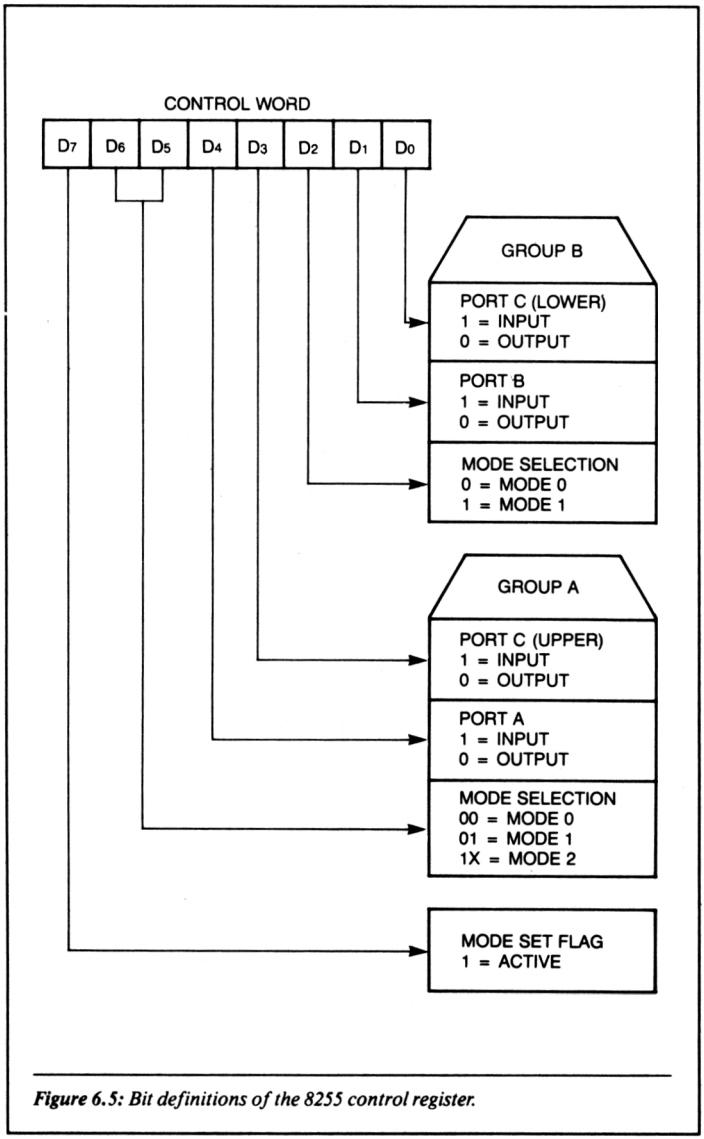


***Figure 6.5:** Bit definitions of the 8255 control register.*

- Bit D2 sets the mode for port B. A logical 0 sets mode B as an output port.
- Bit D1 = 0 sets port B up as an output port.
- Bit D0 = 0 sets the port C lower 4 bits as an output port.

This control word written to the 8255 defines all three ports—A, B, and C—as output ports. This gives the 8255 twenty-four unique output lines that can be used to interface to external devices. The Z80 instructions for setting up the 8255 in mode 0 are:

```
LD A,80H        SET THE CONTROL WORD
OUT (13H),A     OUTPUT THE CONTROL WORD TO THE 8255
```

Once the 8255 has been programmed with these two Z80 instructions, we can write data to any output port using Z80 OUT instructions. For example, let's suppose that we wish to write 23H to port A outputs, 41H to port B, and 73H to port C outputs. To accomplish this, we would use an instruction sequence similar to the following:

```
LD A,23H        SET UP PORT A DATA
OUT (10H),A     OUTPUT DATA TO 8255
LD A,41H        SET UP PORT B DATA
OUT (11H),A     OUTPUT DATA TO 8255
LD A,73H        SET UP PORT C DATA
OUT (12H),A     OUTPUT DATA TO 8255
```

After these instructions have been executed, the output ports A, B and C of the 8255 will be programmed to the specified data values. This is a convenient way for three individual output ports to be contained on a single chip.

It is possible to program the ports for a combination of input and output. For example, ports A and C could be programmed as output ports, and port B could be programmed as an input port. Referring to Figure 6.5, the control word to setup the 8255 in this configuration would be:

D7	D6	D5	D4	D3	D2	D1	D0
1	0	0	0	0	0	1	0

After this control word is programmed into register 3 of the 8255, the device will be logically set up, as shown in Figure 6.6. We could use an IN instruction to read the data input from port B:

```
IN A,(11H)      READ DATA FROM PORT B
```

Once the 8255 has been set up in the correct configuration it becomes an easy matter to read or write data at any device port. There are several different

combinations for setting up the 8255 in mode 0 basic I/O. Figure 6.7 shows all of these combinations, as well as the appropriate control word for programming the 8255.

6-6: A Mode 0 Example

As an example of using the Z80 with the 8255 in mode 0, let's now examine a simple keyboard interface. We will examine both the hardware and software for the keyboard. The keyboard is organized as a 4-by-4 matrix of SPST (single pole, single throw) switches. We will use the 8255 to interface the keyboard switches to the Z80. The hardware schematic for this example appears in Figure 6.8. Here is an overview of how the circuit and program operate.

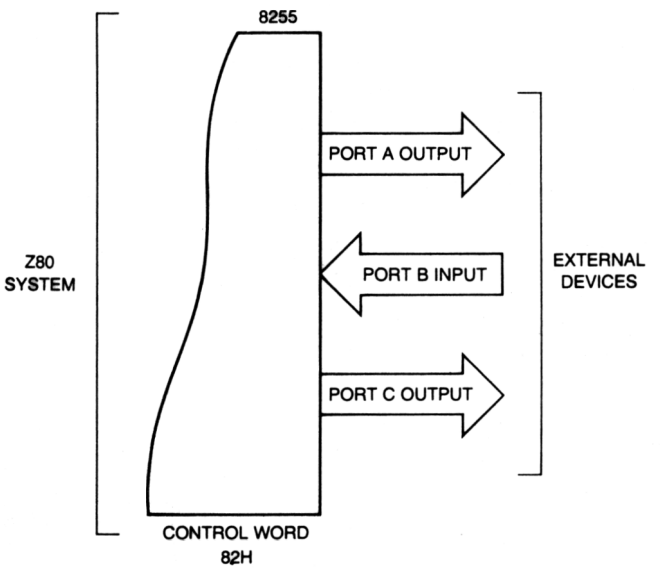


Figure 6.6: *A block diagram showing the configuration of the 8255 after the control word has been written to it.*

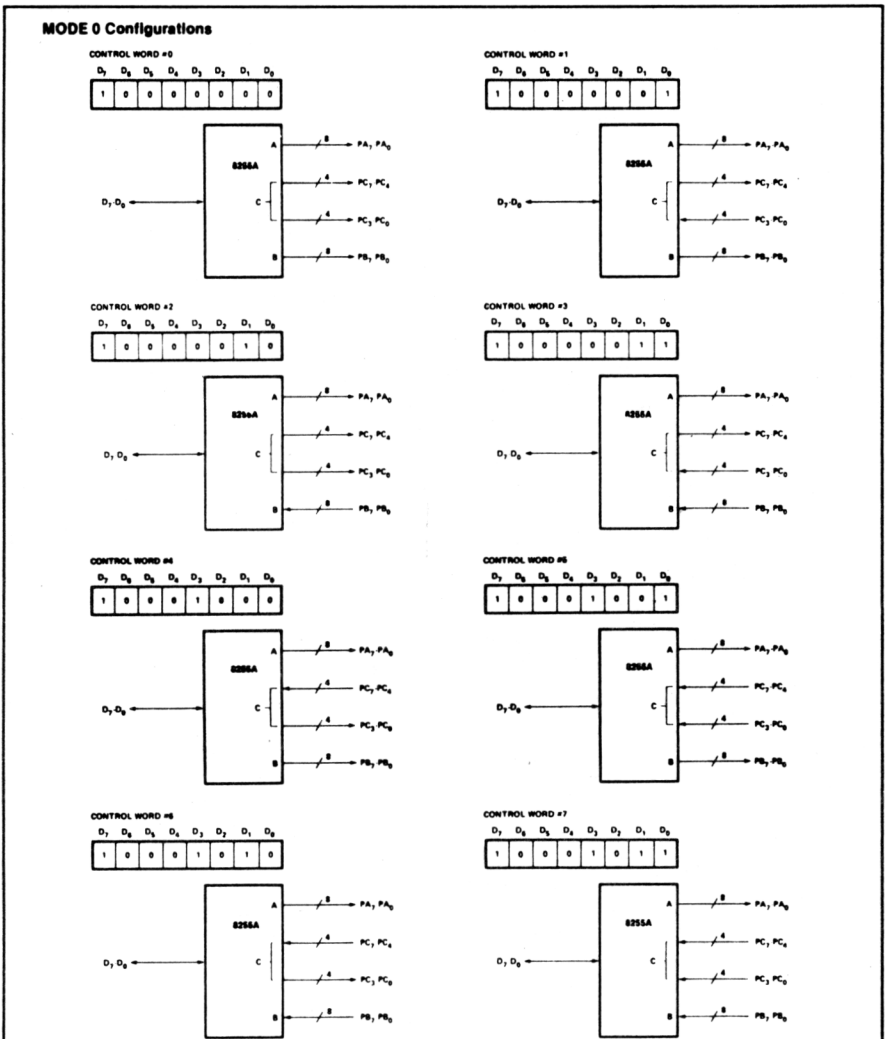


Figure 6.7: These are all the possible configurations that can be used with mode 0. (continues)

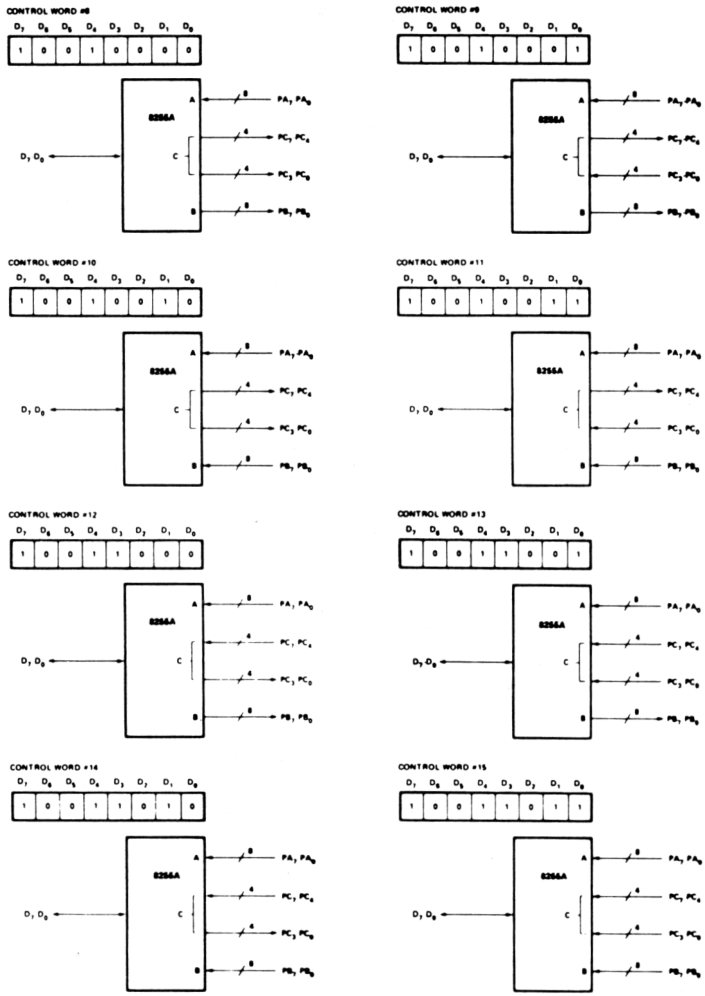


Figure 6.7: Possible configurations that can be used with mode 0.

Port B outputs are connected to the keyboard columns. Port A inputs are connected to the keyboard rows. All of the row inputs are pulled up to +5 volts via the 10K pull-up resistors. The column lines are forced to a logical 0, one at a time. After each column line has been set to a logical 0, the input port

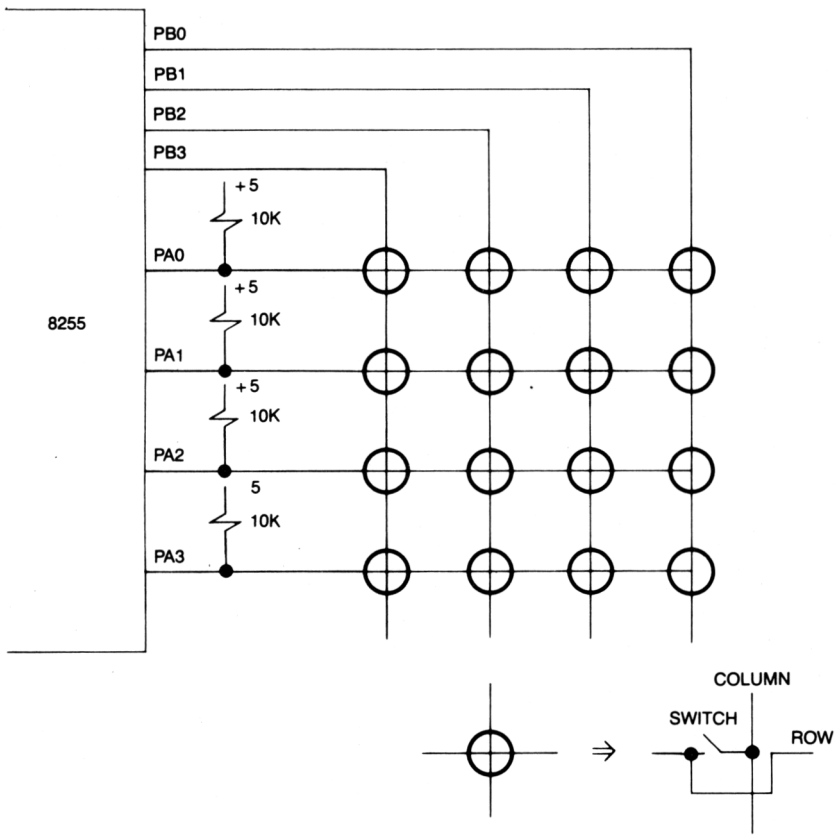


Figure 6.8: *A schematic diagram showing the interconnections for a simple 4-by-4 keyboard array interfaced to an 8255.*

A is read. If any bit read from the input port is a logical 0, then a key has been pressed. We can determine which key has been pressed by determining the bit that is a logical 0 and the column line that is active. Figure 6.9 shows a program that operates in the manner just described.

Although the previous example was for a simple 4-by-4 matrix, we can easily expand it to fit a matrix of any size. The use of the 8255 is a simple way to provide the necessary hardware for the interface.

6-7: Operating Mode 1 for the 8255

Another mode of operation of the 8255 is that of an I/O device used in a handshake environment. Ports A and B are the 8-bit data ports, while the upper bits of port C are used as handshake lines for port A, and the lower 4 bits of port C are used as handshake lines for port B.

The main idea of an I/O transfer using a handshake is this: The peripheral device electrically informs the 8255 that data is either available to be sent or ready to be taken. The handshake lines are used to provide this information. (See Figure 6.10.)

In the diagram labeled (a) in Figure 6.10, data is being sent to the 8255 from the peripheral device. Before the peripheral device writes data to the 8255, however, it must check the input buffer full flag. If this flag is true, then the data in the 8255 buffer has not been read by the Z80. That is, data has been sent to the 8255 from the external device, but the Z80 has not read the data. If the flag is false, then the external device will write data to the 8255 and the input buffer full flag will go true. When the Z80 reads the data, the input buffer full flag will go false.

In the diagram labeled (b) in Figure 6.10, the 8255 will soon be sending data to the peripheral device. Before the 8255 can send the data, however, it must first check to see if the output buffer full flag is set. This flag is a signal to the external device that the 8255 has data ready for the external device to take. When the external device strobes the data out of the 8255, the output buffer full flag goes false, thus indicating that the external device has taken the data. At this point, the Z80 can send more data to the output buffer for the external device to take.

The technique of handshaking data from one device to the next is very useful in applications where the external device is slower than the system microprocessor. With this technique, the microprocessor can put data in the output buffer and then perform other tasks. When the slow external device has taken the data, the microprocessor can send more data. Let's now go over the details of how the 8255 can be made to work in the handshake environment.

Let's assume that the 8255 is sending data to the external device from port

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;     PROGRAM FOR DETECTING A KEY PRESS ON THE HARDWARE
;     SHOWN IN FIGURE 6.8
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;   PORT B IS THE OUTPUT PORT
;   PORT A AND C ARE THE INPUT PORTS
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
0010                PORTA   EQU   10H
0011                PORTB   EQU   11H
0012                PORTC   EQU   12H
0013                CONP    EQU   13H
0099                CONWD   EQU   99H
;
1800                        CODE 1800H
;
1800    3E99                LD A,CONWD
1802    D313                OUT (CONP),A      ;SET UP THE 8255
;
1804    0E11                LD C,PORTB        ;CREG = PORTB ADD
;
1806    06FE        COL     LD B,0FEH         ;COLUMN 0 ACTIVE
1808    ED41        COL1    OUT (C),B         ;ASSERT ACTIVE COLUMN
180A    DB10                IN A,(PORTA)      ;READ THE ROWS
180C    2F                  CPL               ;COMPLIMENT INPUT WORD
180D    E60F                AND 0FH           ;MASK OFF UNUSED BITS
180F    FE00                CP 00H            ;ANY BITS = 1?
1811    C21F18              JP NZ,KEYIN       ;IF YES THEN KEY PUSHED
1814    CB00                RLC B             ;SHIFT B LEFT, NEXT COL ACTIVE
1816    78                  LD A,B
1817    FEEF                CP 0EFH           ;CHECK FOR LAST COLUMN ACTIVE
1819    C20818              JP NZ,COL1        ;NOT LAST, ASSERT NEXT COLUMN
181C    C30618              JP COL            ;LAST, ASSERT FIRST COLUMN
;
;   HANDLE A KEY PRESSED, NEED COLUMN AND ROW
;

181F    4F          KEYIN   LD C,A            ;ROW POS = 1 IN C REG
1820    78                  LD A,H            ;COLUMN POS IN A REG
1821    2F                  CPL               ;INVERT POS BYTE
1822    E60F                AND 0FH           ;UNUSED BITS = 0
1824    C9                  RET
;
;   RETURN FROM ROUTINE WITH
;   A REG = COLUMN POS, 1 IN ACTIVE COLUMN BIT
;   C REG = ROW POS, 1 IN ACTIVE ROW BIT
;
;
                            END
```

Figure 6.9: A Z80 program to scan the keyboard of Figure 6.8 and determine which key has been pressed.

A and receiving data at port B. This order could be reversed in that port A could be receiving the data and port B could be sending. Another alternative is that both ports could be receiving, or both sending. The main idea is that both port A and port B can be used to handshake data as inputs or outputs. Furthermore, each port is independent of the other. You need only modify the control word to set up the 8255 for the proper register operation.

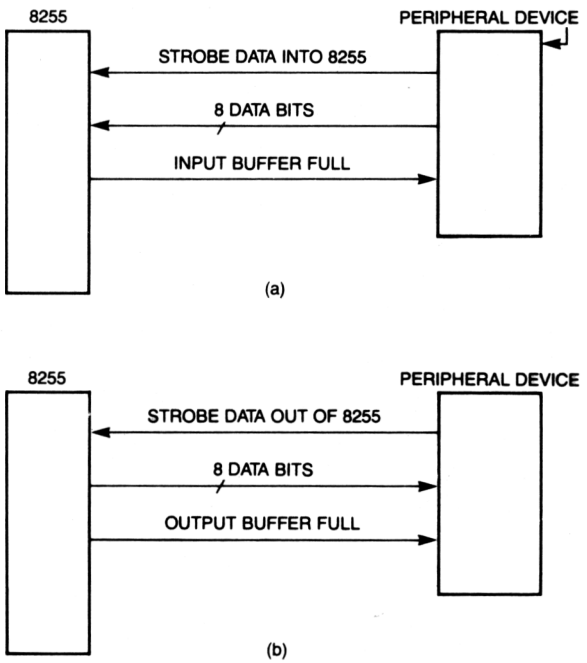


Figure 6.10: A block diagram showing how handshake lines can be used between the 8255 and an external peripheral device.

The block diagram displayed in Figure 6.11 shows the configuration we desire for this application. To place the 8255 in this configuration, the correct control word for this mode must be written to the device, prior to its use. The control word for this configuration would be 10100110 or 0A3H.

In this figure, data output is on the 8255 pins PA0–PA7; the output buffer full is on PC7; the acknowledge from the external device is on PC6; data input is on PB0–PB7; the input buffer full is on PC1; and strobe data into 8255 is on PC2. Figure 6.12 shows the port C pin definitions for mode 1 input and output.

Now that we have defined the hardware, let's examine the software that allows the 8255 to operate. For this discussion, we will assume that there will be no interrupts. The Z80 will poll the status lines of the 8255 to check to see whether data has been received for the input port or taken from the output port. Further, we will assume that the external device is electrically capable of taking the data and sending it to the 8255. It is important to note that the 8255 provides only one-half of the solution. The other half is that the external device must be made to electrically communicate in a consistent manner with

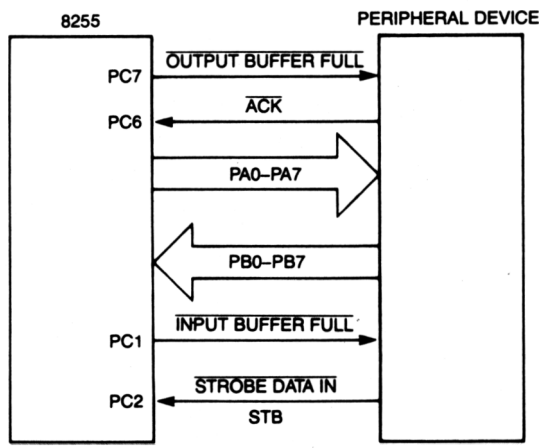


Figure 6.11: An 8255 configuration that will output data from port A lines to the peripheral device and input data to port B lines from the peripheral device.

	OUT	IN
PC0	$INTR_B$	$INTR_B$
PC1	IBF_B	$\overline{OBF}_B$
PC2	$\overline{STB}_B$	$\overline{ACK}_B$
PC3	$INTR_A$	$INTR_A$
PC4	$\overline{STB}_A$	I/O
PC5	IBF_A	I/O
PC6	I/O	$\overline{ACK}_A$
PC7	I/O	$\overline{OBF}_A$

Figure 6.12: Port C pin definitions for mode 1 input and output.

the 8255. Figures 6.13a and 6.13b show the timing for a typical transfer with the 8255 in the configuration required.

To send a character to the external device, the 8255 must first examine the output buffer full line, PC7, for a logical 1. This is done by reading the port C data, using an input instruction. If port C, bit D7, is a logical 1, then the external device has taken the data that was present in the output buffer. If bit D7 of port C is a logical 0, the Z80 should not send any more data. When bit D7 of port C goes to a logical 1, the Z80 can again write data into the port A data register. At this time the output buffer line port C, bit D6 goes to a logical 0. This indicates to the external device that data is present and can be taken. The external device responds to this line going true by strobing the ACK input line, which is port C, bit D7, as described previously. Figure 6.14 shows a flowchart for the complete output operation in the handshake mode. The Z80 mnemonics to realize this flowchart appear in Figure 6.15.

Let's now learn how the Z80 reads data from the external device via the 8255. The first operation to occur is that the Z80 checks the input buffer full line, port C, bit D1. When this line is a logical 1, it is indicating that the external device has strobed data into the 8255 via the $\overline{STB}$ input line, port C, bit D2. At this time the Z80 reads the port B data register, using the IN instruction. At this point, the input buffer line, port C, bit D1, goes to a logical 0, thus indicating to the external device that another byte may be strobed into the 8255. Figure 6.16 shows a flowchart for this type of transfer. The mnemonics to realize the flowchart appear in Figure 6.17.

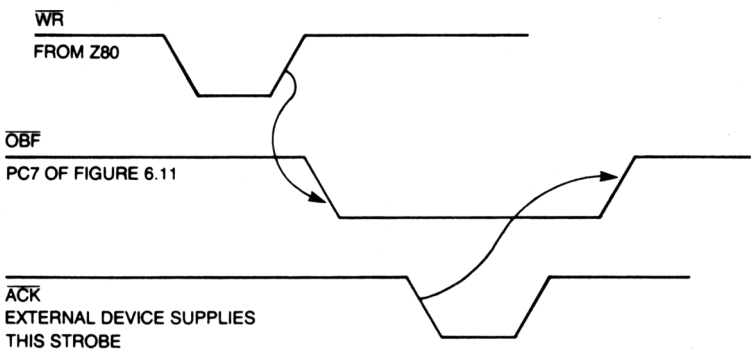


Figure 6.13a: *A timing diagram for a typical transfer on the port A lines for the 8255 in the configuration shown in Figure 6.12.*

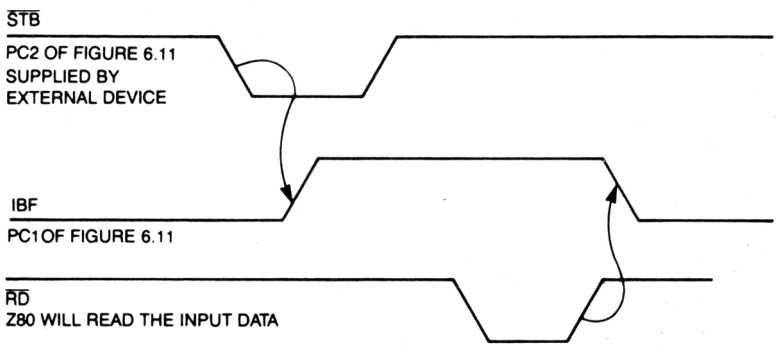


Figure 6.13b: *A timing diagram for a typical transfer on the port B lines for the 8255 configuration shown in Figure 6.12.*

6-8: Operating Mode 2 for the 8255

Another mode of operation for the 8255 is mode 2. In this mode the device can use port A as a bi-directional data port. That is, the eight lines of port A can transmit data to and from the peripheral equipment. When the 8255 is

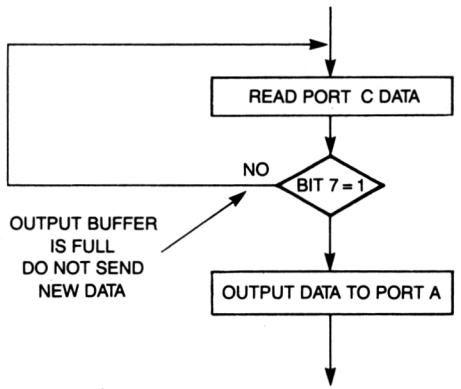


Figure 6.14: A flowchart showing the sequence of events required to transfer data to the external peripheral device using the port A lines and upper port A lines of the 8255.

```
24E8                ODATA   EQU  24E8H       ;ADDRESS OF OUTPUT DATA
;
;
1800                        CODE 1800H
;
1800  DB12           BACK    IN A,(12H)       ;READ PORT C DATA
1802  CB7F                   BIT 7,A          ;TEST BIT 7 = 1
1804  CA0018                 JP Z,BACK        ;NOT =1, KEEP READING PORT C
1807  3AE824                 LD A,(ODATA)     ;GET OUTPUT DATA IN A REG
180A  D310                   OUT (10H),A      ;OUTPUT DATA TO PORT A
180C  C9                     RET
                             END
```

Figure 6.15: A Z80 program to realize the flowchart of Figure 6.14.

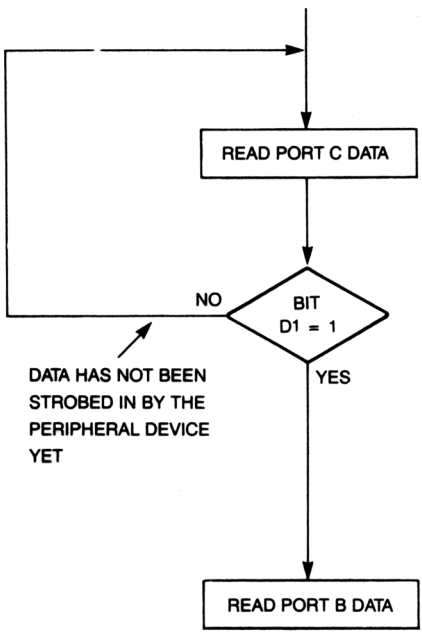


Figure 6.16: A flowchart showing the sequence of events required to transfer data into the 8255 from the external device via the port B and lower port C lines.

```
;
;
1800  DB12      BACK    IN A,(12H)       ;READ PORT C
1802  CB4F              BIT 1,A          ;TEST BIT 1= 1
1804  CA0018            JP Z,BACK        ;DATA NOT STROBED IN YET
1807  DB11              IN A,(11H)       ;DATA READY READ IT
1809  C9                RET
                        END
```

Figure 6.17: A Z80 program to realize the flowchart shown in Figure 6.16.

programmed in this mode, port A has a block diagram like the one shown in Figure 6.18.

Figure 6.18 shows an *output* and an *input* latch. The output latch holds the data written to the port by the CPU and waits for the external device enable to transfer the data onto the port A output lines. The input latch stores the data sent to the port A input lines by the external device.

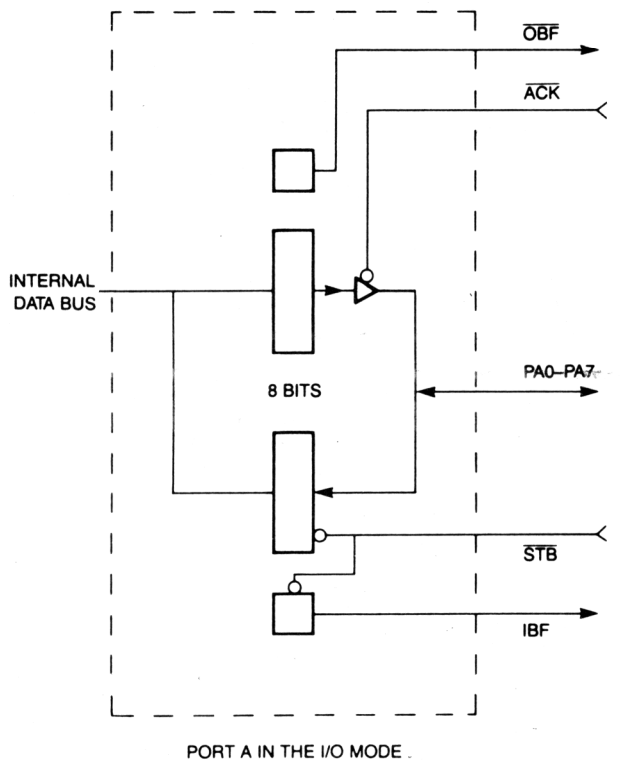


Figure 6.18: *A block diagram designed to help you understand the 8255 device when it is programmed into the mode 2 configuration.*

Let's now examine the block diagram in Figure 6.18 and see how data is transmitted to the external device. The first event to occur is that the CPU writes data into the output latch of port A. When this occurs, the output line labeled $\overline{OBF}$ (for output buffer full) is set true. This signal electrically informs the external hardware that data is available from port A. The $\overline{OBF}$ signal also indicates to the CPU that the output buffer is full and has not been read by the external device.

Next, the external device sends an $\overline{ACK}$ input to the 8255. This input enables the output latch data onto the port A data lines. It also resets the $\overline{OBF}$ line, thus indicating to the CPU that the output data in port A has been read. At this time the CPU can send more data to output port A.

Before data can be received by the 8255 from the external device, the external device must first examine the logical state of the IBF (input buffer full) output line. If this line is a logical 1, then the input buffer is full and the input data has not yet been read by the microprocessor. Let's assume that the IBF line is a logical 0, thus indicating that there is no data in the buffer.

The external device will place the data on the port A data lines and assert the $\overline{STB}$ input to the 8255. This action will strobe the data into the internal latch of port A and set the IBF line true. The CPU will monitor the logical level of the IBF line by reading the port C data. When the IBF line is true, data is available to be read. When the CPU reads the data, the IBF line will go to a logical 0. At this time the external I/O device may send more data to the 8255 in the manner just described.

To allow the CPU to examine the status of the external interface lines, port C is used to reflect the logical state of these lines. Figure 6.19 shows the bit definition for port C when the 8255 is programmed for mode 2 operation.

PORT C LINE	DEFINITION
PC0	I/O
PC1	I/O
PC2	I/O
PC3	$INTR_A$
PC4	$\overline{STB}_A$
PC5	IBF_A
PC6	$\overline{ACK}_A$
PC7	$\overline{OBF}_A$

Figure 6.19: *Port C pin definitions with the 8255 in the mode 2 configuration.*

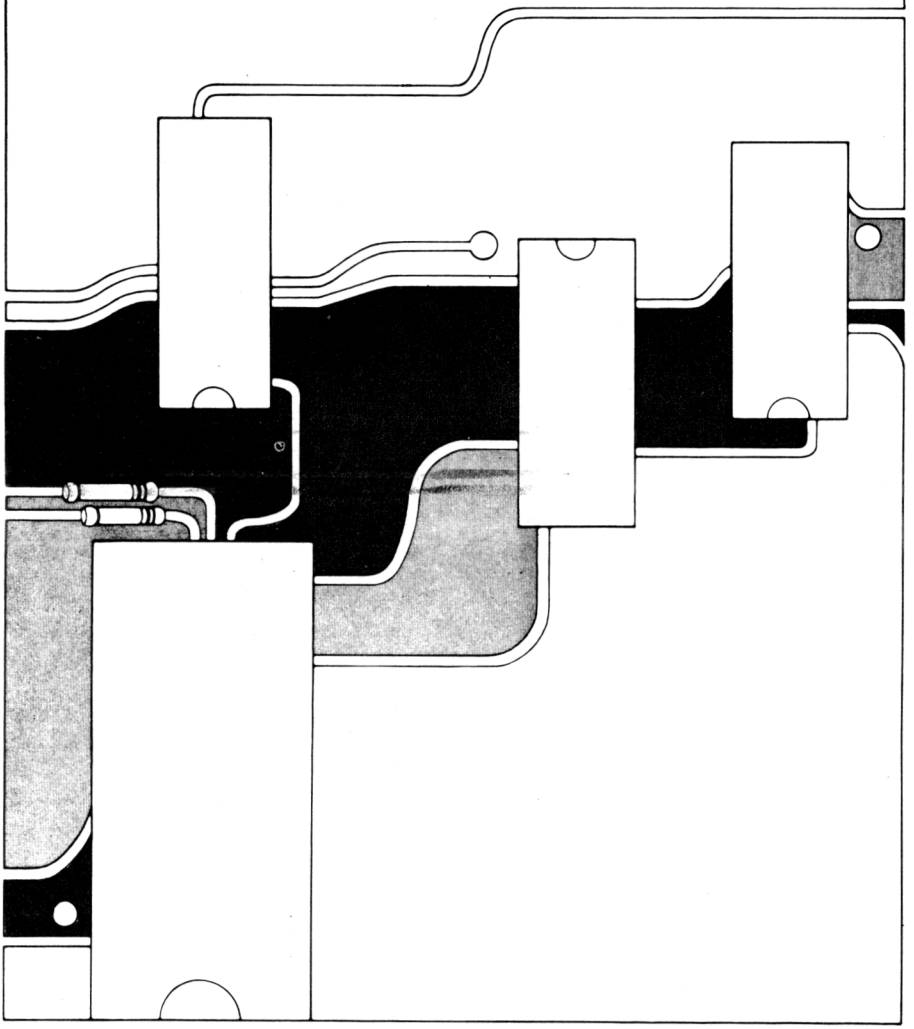
CHAPTER SUMMARY

In this chapter we have seen how the 8255 device is used with the Z80 microprocessor. We have examined the organization of the device and demonstrated one way that it can be connected to the Z80 system busses. Finally, we have discussed the software necessary to operate the 8255.

The 8255 was not originally designed for use with the Z80 CPU. However, it is an extremely versatile interface chip.

This device can simplify the hardware required to realize some system applications. Because it is inexpensive and easy to use, you may want to consider using it to interface your Z80 applications to external devices.

Using the 8253 Programmable Timer



Chapter 7

INTRODUCTION

This chapter will show you how to use the 8253 programmable timer chip with the Z80 microprocessor. We will begin by examining and describing this timer chip. We will then interface it with the Z80 microprocessor. Finally, we will write software that allows for its use in a variety of applications.

7-1: Block Diagram of the 8253 Programmable Timer

Let's begin by examining the block diagram in Figure 7.1 and exploring the internal registers and operating modes of this device. The diagram shows that the timer has three independent, programmable counters—and that they are all identical. In this chapter we will learn how to apply and use each counter.

Also shown in Figure 7.1 is the *data bus buffer.* This block contains the logic that buffers the data bus to and from the microprocessor, to the 8253 internal registers. In addition, there is the block labeled *read/write logic,* which controls the reading and writing of the counter registers. The final block—the *control word register*—contains the programmed information that is sent to the device from the system microprocessor. In effect this register defines how the chip logically operates. Figure 7.2 shows a pin configuration for the 8253 device.

7-2: The Three Counter Lines: Clock, Gate, and Out

Each counter in the block diagram in Figure 7.1 has three logical lines connected to it. Two of these lines, clock and gate, are inputs. The third, labeled out, is an output. The function of these lines changes and depends

PROGRAMMABLE INTERVAL TIMER

- **MCS-85™ Compatible 8253-5**
- **3 Independent 16-Bit Counters**
- **DC to 2 MHz**
- **Programmable Counter Modes**
- **Count Binary or BCD**
- **Single +5V Supply**
- **24-Pin Dual In-Line Package**

The Intel® 8253 is a programmable counter/timer chip designed for use as an Intel microcomputer peripheral. It uses nMOS technology with a single +5V supply and is packaged in a 24-pin plastic DIP

It is organized as 3 independent 16-bit counters, each with a count rate of up to 2 MHz. All modes of operation are software programmable

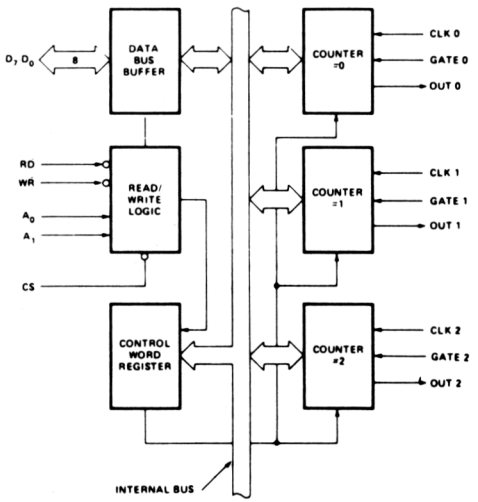


Figure 7.1: *A block diagram of an 8253 programmable interval timer.*

on how the device is initialized or programmed. Here is a general definition of the lines:

Clock This input is the clock input for the counter. The counter is 16 bits. The maximum clock frequency is 1/380 nanoseconds or 2.6 megahertz. The minimum clock frequency is DC or static operation.

Gate This input can act as a gate for the clock input line, or it can act as a start pulse, depending on the programmed mode of the counter.

Out This single output line is the signal that is the final programmed output of the device. Actual operation of the out line depends on how the device has been programmed.

7-3: 8253 Internal Registers

A list of the internal registers of the 8253 device appears in Figure 7.3. Let's first examine and discuss the *mode word register.* This register defines the overall operation of the device. Since each of the three counters is fully independent, each one can be programmed by outputting the correct data to the mode word register. We will show how this can be accomplished as our discussion proceeds. Let's first define the four internal registers shown in Figure 7.3.

Control Word Register This internal register is used to write information to, prior to using the device. This register is addressed when A0 and A1 inputs are logical 1's. The data in this register controls the operating mode and the selection of either binary or BCD (binary coded decimal)

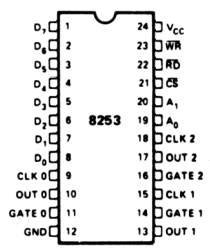


Figure 7.2: Pinout of the 8253 programmable interval timer.

counting format. This register can only be written to. The programmer cannot read information from this register.

Counter #0, #1, #2 Each counter is identical, and each consists of a 16-bit, pre-settable, down counter. Each is fully independent and can be made to count in BCD or binary. Contents of the counters can be easily read by the microprocessor. When the counter is read, the data within the counter is not disturbed. This allows the system to monitor the counter's value at any time, without disrupting the overall function of the device.

7-4: Connecting the 8253 to the Z80

Before we learn to program the 8253, let's learn how to connect it with the Z80 microprocessor.

The 8253 may be thought of as four separate I/O ports. The main selection of the I/O ports is accomplished with the CS input. When this input is logical 0, the 8253 is selected for communication with the Z80.

Address lines A0 and A1 determine which I/O ports are communicated with during the input or output cycle. This I/O architecture can be referred to as device port I/O. The device can be thought of as the 8253 and the ports associated with the device are the internal registers.

	RD	WR	A0	A1	
COUNTER 0	1	0	0	0	LOAD COUNTER 0
	0	1	0	0	READ COUNTER 0
COUNTER 1	1	0	0	1	LOAD COUNTER 1
	0	1	0	1	READ COUNTER 1
COUNTER 2	1	0	1	0	LOAD COUNTER 2
	0	1	1	0	READ COUNTER 2
MODE WORD OR	1	0	1	1	WRITE MODE WORD
CONTROL WORD	0	1	1	1	NO-OPERATION

Figure 7.3: A list of the internal 8253 registers that will program the internal counters of the device.

Decoding of the 8253 $\overline{CS}$ input is accomplished with the upper 6 bits of the lower byte address lines, namely A7–A2. As an example, let's map the device into the I/O architecture of a typical system. We will assume that the 8253 is mapped into the I/O space 30 to 33. This means that I/O ports 30H, 31H, 32H, and 33H are all associated with the 8253 (see Figure 7.4).

The $\overline{RD}$ and $\overline{WR}$ inputs to the 8253 are connected directly to the $\overline{IOR}$ and $\overline{IOW}$ system control lines. This setup is identical to the one for the 8255 device that we saw in Chapter 6. Figure 7.5 shows this connection.

Data lines on the 8253 can be connected directly to the Z80 data bus lines. (We are assuming that no data buffers are required for this application.) Figure 7.6 shows a complete connection of the 8253 device to the Z80, using

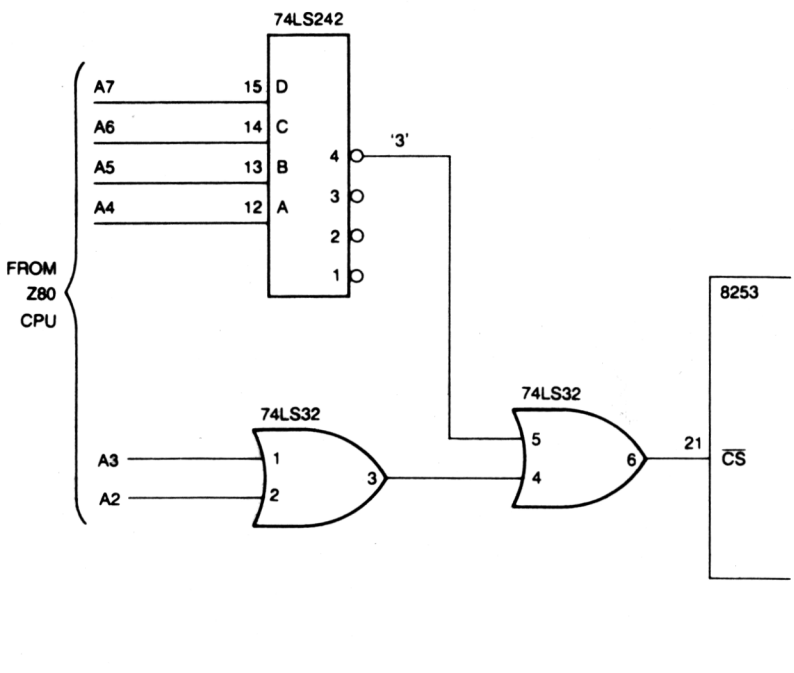


Figure 7.4: A schematic diagram showing how the $\overline{CS}$ input can be decoded from address lines A7–A2.

a non-buffered data scheme, while Figure 7.7 shows a complete connection, using a buffered data bus scheme. The data bus buffer used is the 74LS245. In Figure 7.7, whenever the 8253 is selected and the $\overline{RD}$ input is a logical 0, the 74LS245 is set to place the data from the 8253 onto the Z80 system data bus. At all other times, the 74LS245 places the data from the Z80 system data bus onto the 8255 data input pins.

Using the circuit connections shown in Figures 7.6 and 7.7, it is possible to reliably communicate between the Z80 and the 8253 device. The task now becomes one of programming the device so that it will perform in a way that your system application demands.

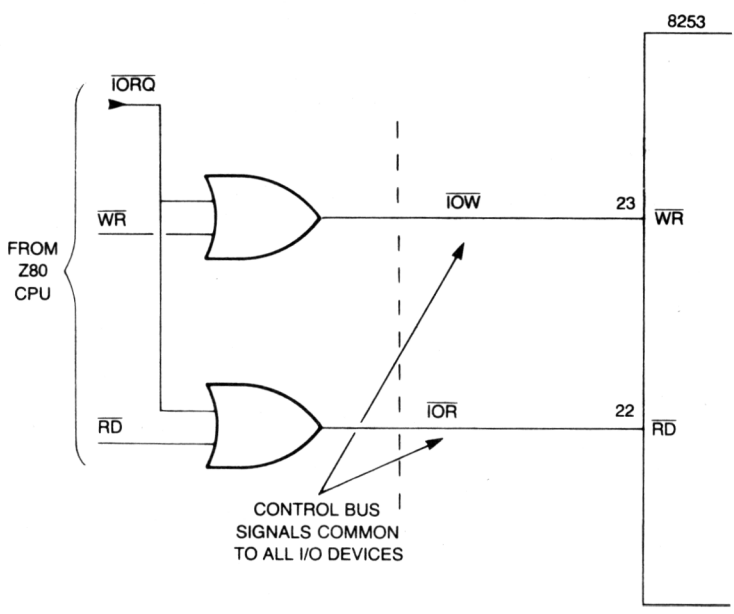


Figure 7.5: *A logic diagram indicating how the $\overline{IOR}$ and $\overline{IOW}$ system control signals are generated. Notice that this is identical to the way that signals were generated in Chapter 6.*

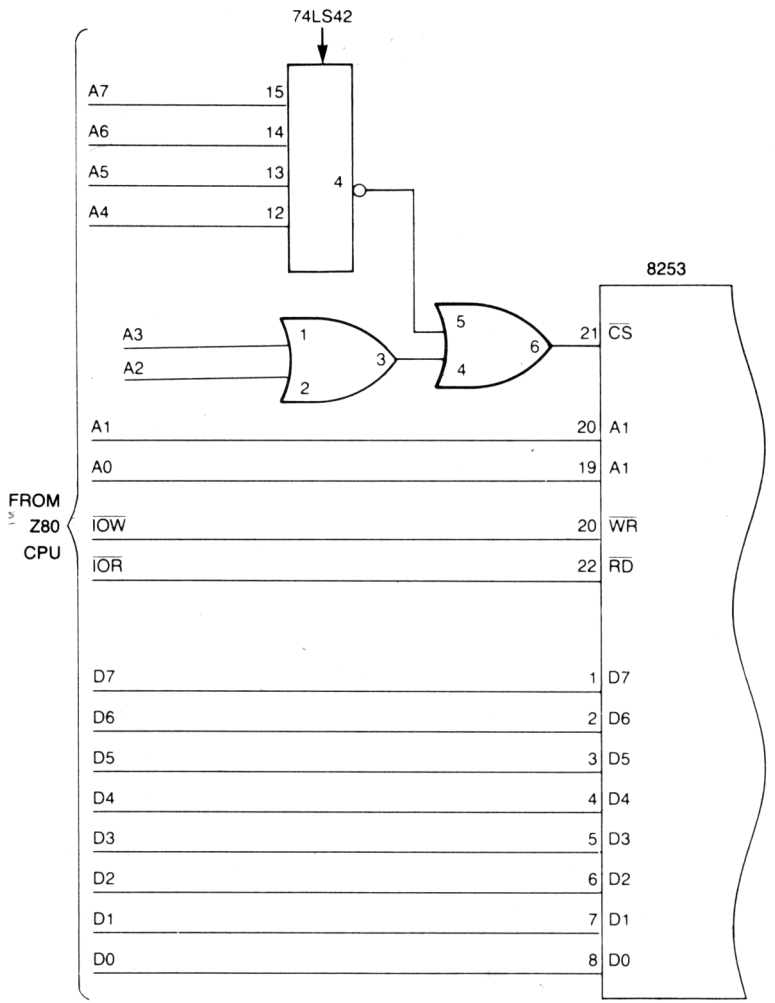


Figure 7.6: *A complete connection between the 8253 and the Z80 system busses. No data buffering is used in this figure.*

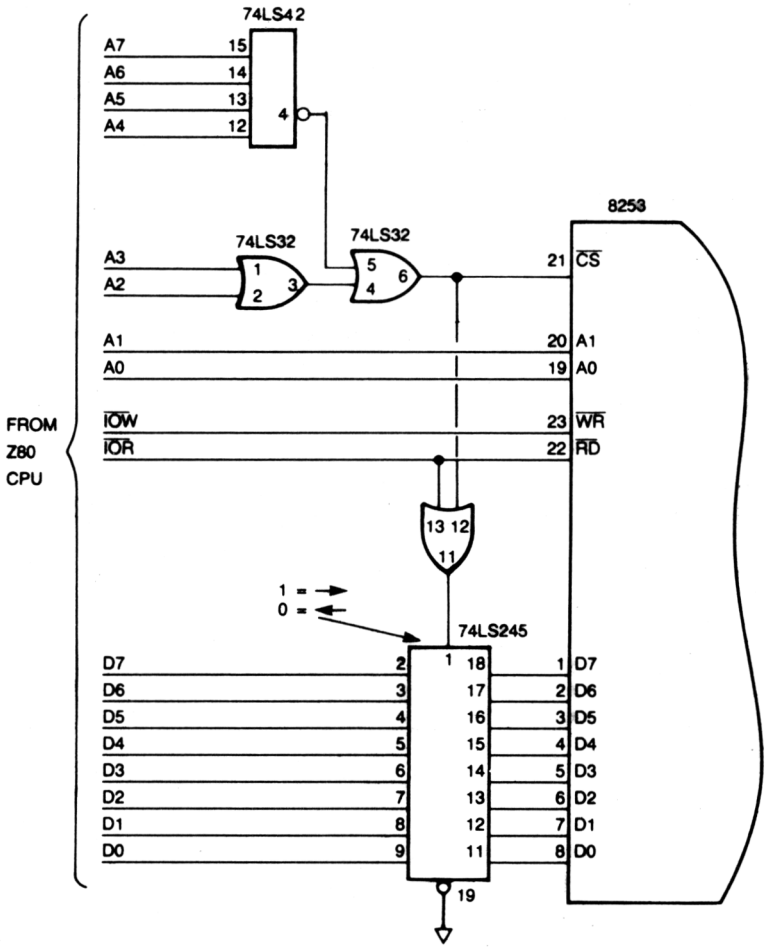


Figure 7.7: Connection of the Z80 and 8253 using data buffers.

7-5: Programming the Device (Control Word Format)

All of the operating modes for the counters are selected by writing bytes to the control register. Figure 7.8 shows the control word format. The address for the control word is A0 = 1 and A1 = 1. In this system application the control word address is 33H.

In Figure 7.8, bits D7 and D6 are labeled SC1 and SC0. These bits select the counter to be programmed. Before any counter can be programmed, it is necessary to define, using the control bits D7 and D6, which counter is being set up. It should be noted that once a counter is set up, it will remain that way until it is changed by another control word. The bits D7 and D6 are defined as follows:

D7	D6	COUNTER SELECT
0	0	0
0	1	1
1	0	2
1	1	illegal value

Bits D5 and D4 of the control word in Figure 7.8 are defined as the read/load mode for the register that is selected by bits D7 and D6. Bits D5 and D4 define how the particular counter is to have data read from or written to it by the microprocessor. These bits are defined as:

D5	D4	R/L DEFINITION
0	0	Counter value is latched. This means that the selected counter has its contents transferred into a temporary latch, which can then be read by the CPU.
0	1	Read/load least-significant byte only.
1	0	Read/load most-significant byte only.
1	1	Read/load least-significant byte first, then most-significant byte.

CONTROL BYTE D7–D0

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RL1	RL0	M2	M1	M0	BCP

Figure 7.8: Bit definition of the counter control register.

What do these bits tell the device to do? The first value, 00H, is the *counter latch mode.* It is useful for taking counter readings during the counter operation. When this mode is specified, the counter value is latched into an internal register at the time of the I/O write operation to the control register. When a read of the counter occurs, it is this latched value that is read.

If the latch mode is not used, then it is possible that the data read back may be in the process of changing while the read is occurring. (See Figure 7.9.) This could result in erroneous data being input by the CPU. To read the counter value while the counter is still in the process of counting, one must first issue a latch control word, and then issue another control word that indicates the order of the bytes to be read.

An alternative method of obtaining a stable count from the timer chip is to externally inhibit counting while the register is being read. This technique is shown in Figure 7.10. Each technique has certain disadvantages. The latching method may give the microprocessor a reading that is "old" by several cycles, depending on the speed of the count and which byte of the counter is being read. The external inhibiting function requires additional hardware. In addition, it may change the overall system operation. It is up to you to determine the best way to operate the device in a specific application.

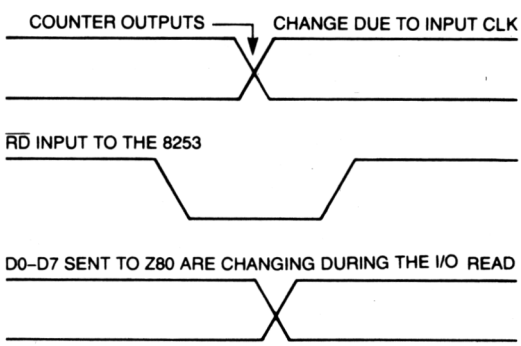


Figure 7.9: *A timing diagram showing how the counter outputs may be changing while the microprocessor is electrically reading the 8253 device. This can be avoided by latching the counter output count prior to reading the counter value.*

The next three bits of the control word in Figure 7.8 are D3, D2, and D1. These bits determine the basic mode of operation for the counter. We will now describe the mode and then follow with examples showing how to use the counter in each of the five modes. Here are the mode descriptions:

D3	D2	D1	MODE VALUE
0	0	0	mode 0: interrupt on terminal count
0	0	1	mode 1: programmable one-shot
x	1	0	mode 2: rate 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

The final bit of the control register D0 determines how the register will count—that is, if it will count in BCD or Binary. If D0 is a logical 1, the count will be in BCD; if it is a logical 0, the count will be in binary. The maximum values for the count in each count mode are 2^{16} in binary, and 10^4 in BCD.

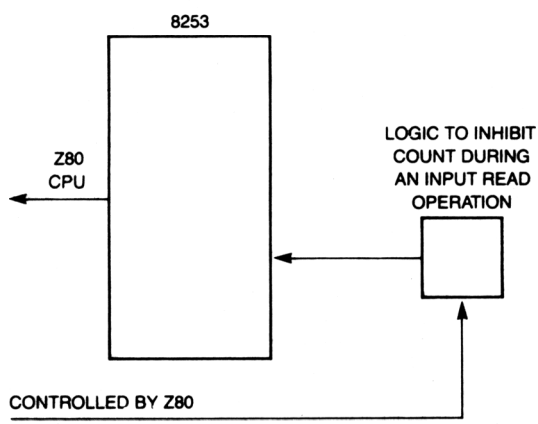


Figure 7.10: A block diagram showing how the counter output can be made stable during a CPU read operation by externally disabling the clock (count) input line.

7-6: Example of Mode 0: Interrupt on Terminal Count

Let's now learn how to use mode 0 with the 8253. We will describe the function of mode 0 and define each important pin. Finally, we will write a software program using Z80 mnemonics to set up the 8253 for use in mode 0.

Mode 0 allows the 8253 to do the following. When the internal count is equal to 0, the OUT pin of the counter will be set to a logical 1. The counter will be programmed to an initial value and then count down at a rate equal to the input clock frequency. When the count is equal to 0000, the OUT pin will be a logical 1. In this application the OUT pin could be connected to interrupt the microprocessor. The output will stay a logical 1 until the counter is reloaded with the same count, a new count, or until a mode word is written to the device.

Once the counter starts counting down, the gate input can disable the internal counting when the gate is a logical 0. To fully show how this mode operates, let's examine an example. In this example we will be counting events with counter number 0. The count will be in BCD. When the count is 125, the Z80 will be interrupted. The interrupt service routine is at location 0038H. We will assume that the Z80 is using interrupt mode 1. A block diagram of this example is shown in Figure 7.11.

To set the counter, we must program the control word. Based on the bit

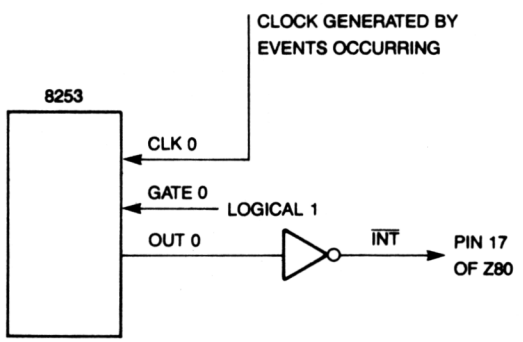


Figure 7.11: A block diagram of the system in the example. The gate input is enabled, and the counter 0 clock is input to the device. When the count reaches zero, the output will interrupt the Z80.

definitions of the control given in Figure 7.8, we will choose a control word equal to 00110001. Here is why:

Bits D7 and D6 = 00.	This defines counter 0.
Bits D5 and D4 = 11.	This sets up the register to load the least-significant byte first, and the most-significant byte next.
Control bits D3, D2 and D1 = 000.	This defines the mode as 0.
Finally, D0 = 1.	This indicates that the count mode will be BCD. The total control word written to port 33 is then 00110001.

Next, we must write the start count into the counter register. Since we are counting in BCD, the least-significant byte will be 35 decimal and the most-significant byte will be 01. These two bytes are written to port 30. Figure 7.12 shows a partial Z80 listing that will initialize the 8253 device.

As soon as the second data byte is loaded into register 0, the counting starts. When the count reaches 125, the counter register will equal 0 and the OUT pin will become a logical 1, thus interrupting the Z80. The interrupt service routine will take the appropriate action for the application. Part of the interrupt service routine reinitializes the 8253 device. Figure 7.13 shows an example of an interrupt service routine using mode 1 interrupts on the Z80. This routine will operate with the 8253 in mode 0.

```
;
;
; WE ASSUME THAT INTERRUPT ARE DISABLED
;
0000  3E31            LD A,31H     ;SET UP CONTROL WORD
0002  D333            OUT (33H),A ;OUTPUT CONTROL WORD TO 8253
0004  3E25            LD A,25H
0006  D330            OUT (30H),A ;OUTPUT LSB OF CTR 0
0008  3E01            LD A,01H
000A  D330            OUT (30H),A ;OUTPUT MSB OF CTR 0
;
;  COUNT IS IN BCD
;
;       COUNTING STARTS WHEN MSB IS OUTPUT
;
000C  FB              EI
                      END
```

Figure 7.12: A Z80 program to initialize the 8253 device.

```
;
;  WE ASSUME Z80 USING MODE 1 INTERRUPTS
;
;
;  THE COUNT HAS REACHED 125 AT THIS TIME WE
;  RELOAD THE COUNTER 0
;
0038    3E25            LD A,25H
003A    D330            OUT (30H),A     ;LOAD LSB
003C    3E01            LD A,01H
003E    D330            OUT (30H),A     ;LOAD MSB
;
;  NEW COUNT DOWN HAS STARTED
;
0040    FB              EI
0041    C9              RET
;
; PART OF THE INTERRUPT ROUTINE WOULD BE TO
; PERFORM SOME ACTION DUE TO THE INTERRUPT HAVING
; OCCURRED. IN THIS EXAMPLE WE SIMPLY RESET THE
; COUNTER AND START OVER.
;
                        END
```

Figure 7.13: A Z80 interrupt service routine using mode 1 interrupts. The 8253 is operating in mode 0.

7-7: Mode 1: Programmable One-Shot

In mode 1, the 8253 can be made to give an output pulse that is an integer number of clock pulses. The one-shot is triggered on the rising edge of the gate input. If the trigger occurs during the pulse output, the device will again be retriggered. This is shown in Figure 7.14.

Let's assume that the input clock frequency is one megahertz. We will program the 8253 for an output pulse of exactly 75 microseconds. The Z80 program to perform this setup is shown in Figure 7.15. In this figure we have assumed that counter 1 is being used as the programmable one-shot.

7-8: Mode 2: Rate Generator

In mode 2, the 8253 becomes a "divide by n" counter. The output pin of the counter goes low for one input clock period. The time between the output low-going pulses is dependent on the present count in the counter register. See Figure 7.16. If the count is loaded between output pulses, the present period will not be affected. A new period will occur during the next count sequence.

As an example, suppose that we wish to provide an output frequency on

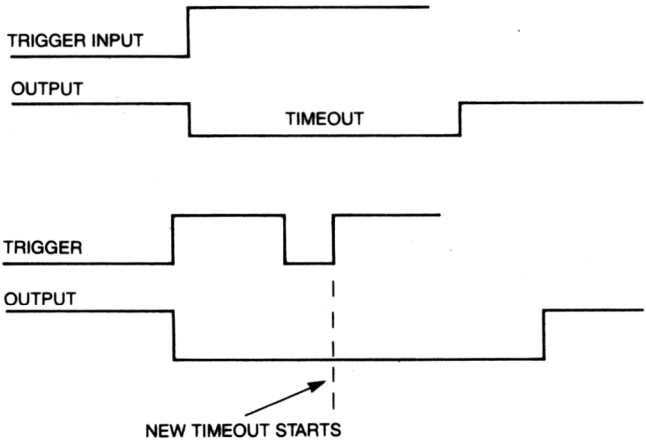


Figure 7.14: A timing diagram showing how the 8253 is used as a programmable one-shot. When the trigger input goes to a logical 1, the output will go to a logical 0 for the programmed time of the counter. If the trigger is again applied before the counter times out, the timeout count will be retriggered.

```
;
;
;
0000   3E72            LD A, 01110010B
0002   D333            OUT (33H),A        ;OUTPUT CONTROL WORD TO 8253
;
; CTR 1, RL = 3, M=1, BINARY COUNT
;
0004   3E4B            LD A, 75           ;DECIMAL 75
0006   D331            OUT (31H),A        ;OUTPUT TO CTR1 LSB
0008   3E00            LD A, 00
000A   D331            OUT (31H),A        ;OUTPUT TO CTR1 MSB
;
;
; THE DEVICE IS NOW SET UP WAITING FOR A TRIGGER
; INPUT TO THE COUNTER
;
                       END
```

Figure 7.15: A Z80 program for using the 8253 as a programmable one-shot.

counter 2 that is equal to 638 hertz. This would be a period of 1/638 = 1.567 milliseconds or 1,567 microseconds. If an input clock of 1 megahertz were applied to the clock input of counter 2, then the counter would need to be programmed to 1,567. This could be done in BCD or decimal. Figure 7.17 shows an example of a Z80 program that would program the 8253 for an operation such as this.

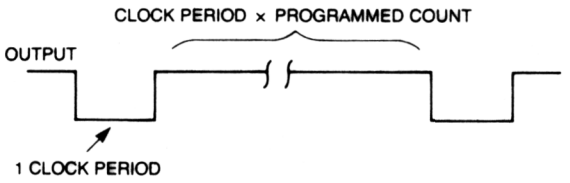


Figure 7.16: *A timing diagram showing how the output of the 8253 would appear when used in mode 2 as a rate generator. The output will go to a logical 0 for one input clock period. The time between pulses is determined by the count programmed in the device.*

```
;
;
0000   3EB5            LD A,10110101B
0002   D333            OUT (33H),A          ;OUTPUT CONTROL WORD
0004   3E67            LD A,67H
0006   D332            OUT (32H),A          ;CTR 2, LSB IN BCD
0008   3E15            LD A,15H
000A   D332            OUT (32H),A          ;CTR 2, MSB IN BCD
;
;
;  THE DEVICE IS NOW INITIALIZED AND THE OUTPUT FREQUENCY
;  WILL BE 638 HERTZ, WITH AN INPUT FREQUENCY OF 1 MHZ.
;
                       END
```

Figure 7.17: *Z80 program for using the 8253 device as a rate generator. The output frequency will be 683 hertz. The input clock frequency is 1 megahertz.*

7-9: Mode 3: Square Wave Generator

Mode 3 is similar to mode 2 except that the output will be high for half the period and low for half. If the count is odd, the output will be high for (n + 1)/2 and low for (n – 1)/2 counts. Figure 7.18 shows a Z80 program to setup counter 0 for a square wave frequency of 10k hertz, assuming that the input clock frequency is equal to 1 megahertz.

7-10: Mode 4: Software Triggered Strobe

In this mode the programmer can set up the counter to give an output timeout that will start when the count register is loaded. On the terminal count, when the counter equals zero, the output will go low for one clock period and then it will go high again. This is shown in Figure 7.19. When the mode is set, the output will go high.

7-11: An Example

As an example, let's program the device to provide a 100 millisecond, software-triggered delay at counter output 2. We will assume that the input frequency equals one megahertz, which requires an input clock count of 10^5. In addition, we wish to perform the count in BCD. The maximum BCD count in any register is equal to 10^4. Therefore, we will use two counters. The first will divide the frequency down to 1 kilohertz. The second will

```
;
;
;
0000   3E36          LD A,00110110B
0002   D333          OUT (33H),A        ;SETUP CTR0 CONTROL WORD
0004   3E64          LD A, 100
0006   D330          OUT (30H),A        ;CTR0 LSB IN BINARY
0008   3E00          LD A, 00H
000A   D330          OUT (30H),A        ;CTR0 MSB IN BINARY
;
;
; THE DEVICE IS NOW SETUP AND RUNNING WITH AN OUTPUT
; FREQUENCY OF 10KHZ WITH AN INPUT FREQUENCY OF 1MHZ.
;
                     END
```

Figure 7.18: *A Z80 program for using the 8253 device as a square wave generator. The output frequency is 10 kilohertz. The input frequency clock is 1 megahertz.*

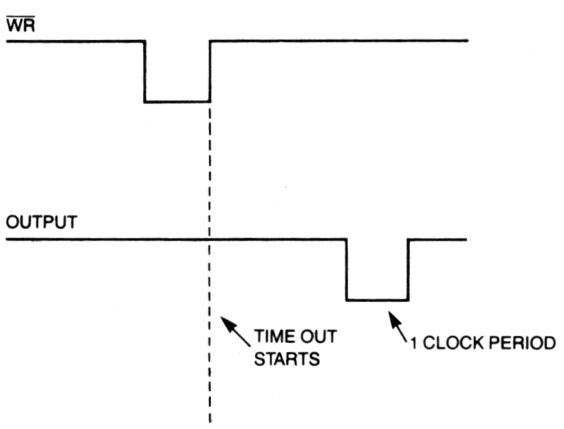


Figure 7.19: A timing diagram showing how the output of the 8253 would appear when used as a programmable software triggered strobe. When the Z80 performs an I/O write operation, the timer starts. At the end of the count the output will go low for one clock period.

provide the software-triggered strobe. This is shown in Figure 7.20. Figure 7.21 presents a Z80 program that uses the 8253 in the manner described in this example.

7-12: Mode 5: Hardware Triggered Strobe

With the counter in this mode, the rising edge of the trigger input will start the counter counting. The output will go low for one clock at the terminal count. The device is retriggerable, thus meaning that if the trigger input is taken low and then high during a count sequence, the sequence will start over (see Figure 7.22). Figure 7.23 shows a Z80 program to initialize the 8253 counter 1 to be used as a hardware triggered strobe.

7-13: Uses of the Gate Input Pin

Each mode of operation for the counter chip has a different use for the gate input pin. The table shown in Figure 7.24 summarizes the operation of this particular input pin on the device.

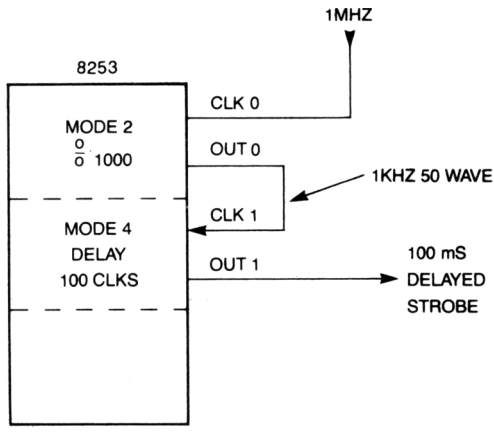


Figure 7.20: *A block diagram showing how the 8253 could be used to generate a 100 millisecond delayed software strobe.*

```
0000    3E35            LD A,00110101B
0002    D333            OUT (33H),A         ;SETUP COUNTER 0 CONTROL WORD
0004    3E00            LD A,00H
0006    D330            OUT (30H),A         ;LOAD CTR0 LSB
0008    3E10            LD A,10H
000A    D330            OUT (30H),A         ;LOAD CTR0 MSB
;
;
;  COUNTER 0 IS SETUP WITH AN OUTPUT FREQUENCY OF 1KHZ
;  SQUARE WAVE WITH AN INPUT FREQUENCY OF 1MHZ. THIS
;  OUTPUT IS CONNECTED TO CTR1 CLOCK INPUT.
;
000C    3E79            LD A,01111001B
000E    D333            OUT (33H),A         ;CTR1 CONTROL WORD
0010    3E00            LD A,00H
0012    D331            OUT (31H),A         ;CTR1 LSB
0014    3E01            LD A,01H
0016    D331            OUT (31H),A         ;CTR1 MSB
;
;  THE LAST OUT TO CTR1 NEED NOT BE DONE UNTIL THE
;  TIME OUT IS WANTED
;
                        END
```

Figure 7.21: *A Z80 program to setup the 8253 to operate a 100 millisecond software strobe. This program makes use of the block diagram given in Figure 7.20.*

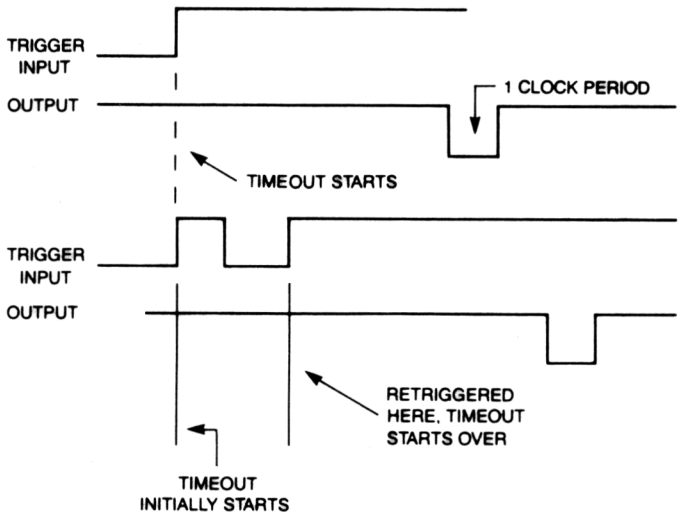


Figure 7.22: A timing diagram showing how the 8253 operates as a hardware triggered strobe. When the external trigger input goes to a logical 1, the timer will start to time out. If the external trigger occurs again, prior to the time completing a full timeout, the timer will retrigger.

```
;
;
;
;
0000   3E7A            LD A,01111010B
0002   D333            OUT (33H),A        ;CTR1 CONTROL WORD
0004   3E55            LD A,85
0006   D331            OUT (31H),A        ;OUTPUT 85 BINARY TO CTR1 LS
0008   3E00            LD A, 00H
000A   D331            OUT (31H),A        ;OUTPUT 00 TO CTR1 MSB
;
;  THE CTR IS NOW SET UP WAITING FOR AN EXTERNAL
;  TRIGGER INPUT ON TRG1.
;
                       END
```

Figure 7.23: A Z80 program to setup and use the 8253 in the hardware-triggered strobe mode as shown in the timing diagram in Figure 7.22.

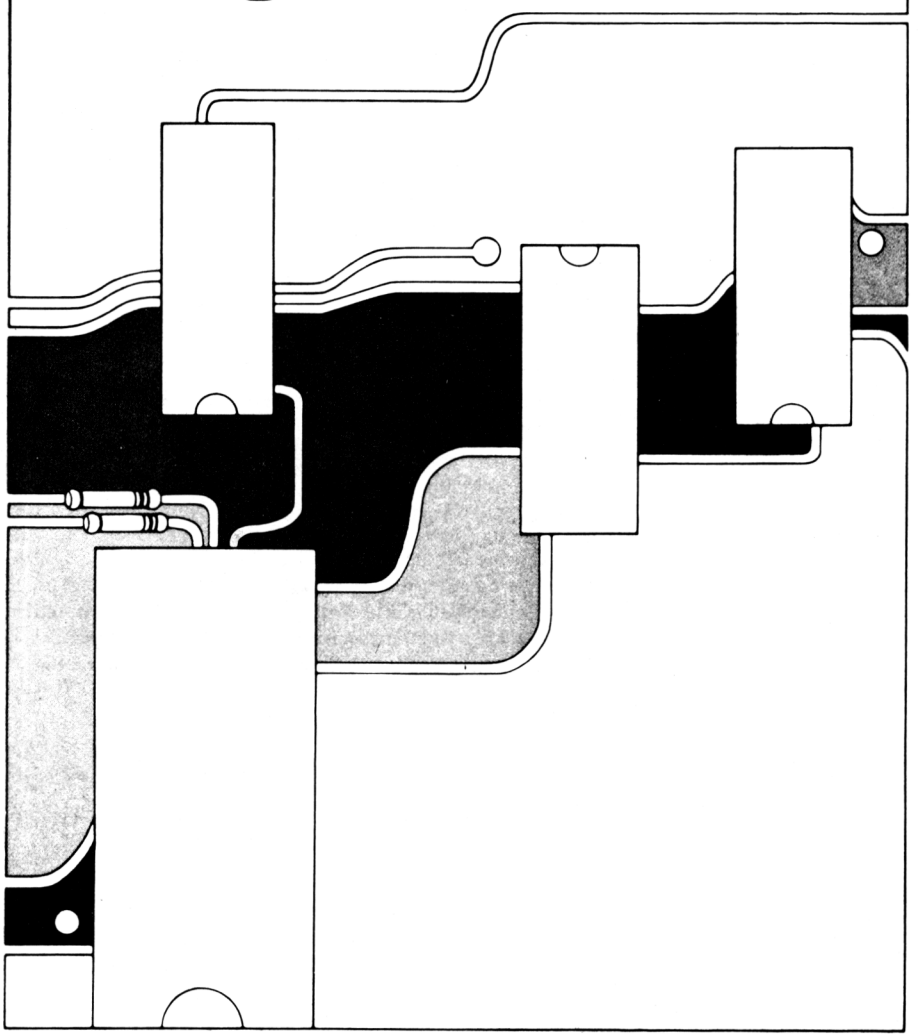
Modes \ Signal Status	Low Or Going Low	Rising	High
0	Disables counting	– –	Enables counting
1	– –	1) Initiates counting 2) Resets output after next clock	– –
2	1) Disables counting 2) Sets output immediately high	1) Reloads counter 2) Initiates counting	Enables counting
3	1) Disables counting 2) Sets output immediately high	Initiates counting	Enables counting
4	Disables counting	– –	Enables counting
5	– –	Initiates counting	– –

Figure 7.24: Table showing the different uses of the 8253 gate input pin.

CHAPTER SUMMARY

In this chapter we have examined the important points of using the 8253 programmable timer with the Z80 CPU. We have presented a block diagram of the device and a discussion of each operating mode. The 8253 can be used to serve a majority of timing functions in many system applications that use the Z80 as a system controller. The examples in this chapter have shown that using this device is not a complex task.

Using the Z80-PIO



Chapter 8

INTRODUCTION

In this chapter, we will show how the MOSTEK MK3881 (Z80-PIO) can be used and applied to the Z80 microprocessor as a programmable I/O device. Unlike the 8255, a parallel I/O discussed in Chapter 6, the MOSTEK MK3881 has been designed specifically for use with the Z80 microprocessor.

The Z80-PIO is an I/O device with several different modes of operation. In this chapter we will learn how each operating mode works and how we can use each mode with the Z80 microprocessor for most system applications.

8-1: Block Diagram of the PIO

We will begin our discussion by examining the block diagram in Figure 8.1. The diagram shows that the PIO contains two independent I/O ports: A and B. We will see later on that these ports are nearly identical in their operating characteristics.

At the output of each port are eight data lines and two control lines. Output and input lines of the port are TTL compatible. Each output can drive 2.0 milliamperes in the logical 0 state and 250 microamperes in the logical 1 state. This is enough drive capability for one standard TTL device that requires 1.6 milliamperes in the logical 0 state and 40 microamperes in the logical 1 state.

In addition, Figure 8.1 shows two other blocks: the internal control logic and the interrupt control. The *internal control logic* routes the internal data of the device to the correct internal register at the right time. The *interrupt control* allows the PIO to correctly request interrupts and handle the interrupt acknowledge from the Z80 microprocessor. Later on, we will devote an entire section of this chapter to showing how the PIO can be programmed to handle the interrupt functions.

8-2: Z80-PIO Pinout

Figure 8.2 shows a pinout of the Z80-PIO packaged in a 40-pin DIP. Power is supplied by +5 volts and ground. Let's now go over the functions of the major pins of the PIO and gain an understanding of what each pin can do. Later on we will show you how you can physically connect the PIO to the Z80 microprocessor to receive reliable communication.

Let's now examine in detail the pinout in Figure 8.2:

D7–D0 These are the data input and output lines that receive or send information to the Z80 microprocessor.

B/A Select This single input line determines if the data on the data bus is communicating with port B (logical 1) or port A (logical 0). This input line is usually connected to the A0 bit of the system address lines.

C/D Select The logical value of this input indicates to the PIO whether the data being written to the PIO device is actual data or a control word. The control words are used to program the PIO into the different operating modes.

$\overline{CE}$ *(Chip Enable Input)* Whenever this line is active logical 0, data may

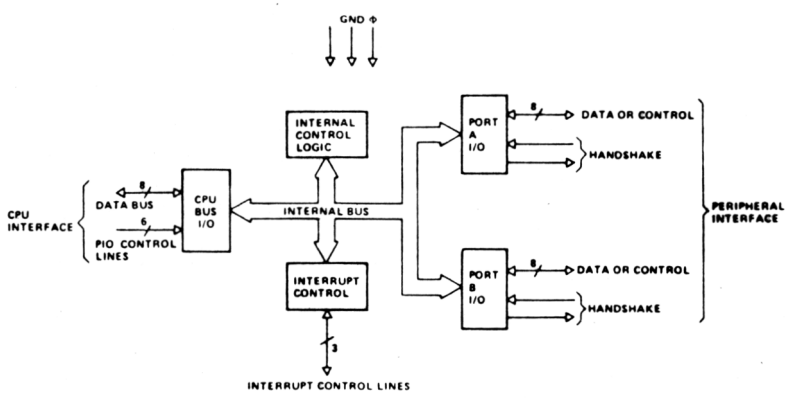


Figure 8.1: A block diagram of the Z80-PIO device.

be written to or read from the I/O device. The $\overline{CE}$ input is usually a decoded address space from the lower eight bits of the system address bus.

CLOCK A single phase clock input is used here. The PIO uses the clock to synchronize certain internal operations. (These operations will become clear as we proceed through this chapter.)

$\overline{M1}$ This is the $\overline{M1}$ output from the Z80. The PIO uses this input line to control several internal operations. When $\overline{M1}$ and the $\overline{RD}$ are both active, the Z80 is fetching an opcode. It is important for the PIO to recognize this state because the device will automatically recognize a certain opcode. This is the RETI instruction. The $\overline{M1}$ signal is also used to allow the PIO to respond to the interrupt acknowledge from the Z80. Recall that when the $\overline{IORQ}$ and the $\overline{M1}$ are active, the Z80 is acknowledging an interrupt.

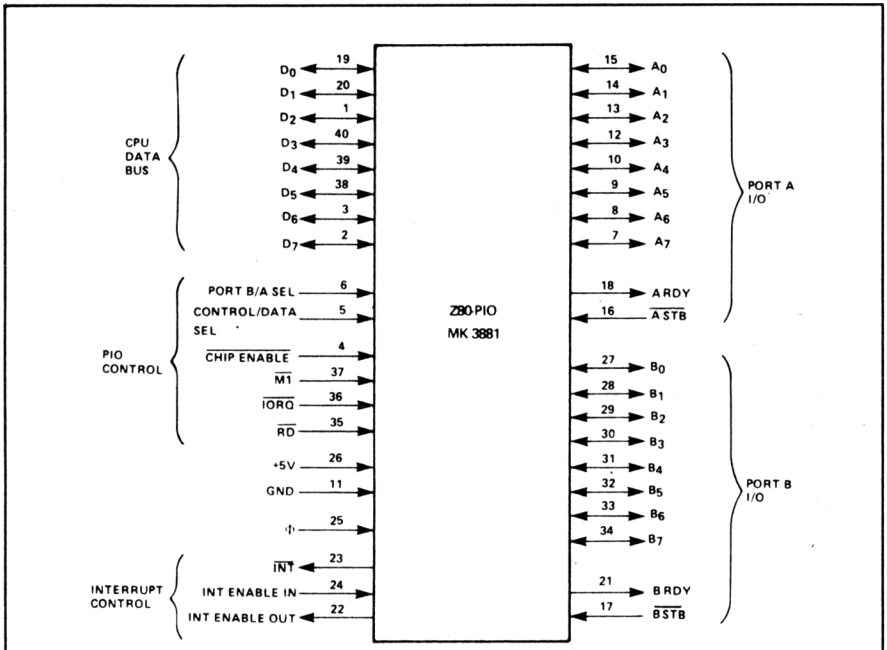


***Figure 8.2:** Pinout of the 40-pin DIP, Z80-PIO*

In addition to these two operations, the $\overline{M1}$ signal will also perform the following:

- It will synchronize the PIO interrupt logic.
- When $\overline{M1}$ occurs without an active $\overline{RD}$ or $\overline{IORQ}$, the PIO is placed in an internal reset state. This is analogous to a hardware reset on the chip. Notice that there is no input pin for external reset on the device.

$\overline{IORQ}$ This is an input to the PIO that connects directly to the $\overline{IORQ}$ output of the Z80 microprocessor. This signal is used with the B/A select, C/D select, and $\overline{CE}$ to transfer commands and data between the Z80 and the PIO. When the $\overline{IORQ}$ input and the $\overline{CE}$ input are both active (logical 0), the PIO is communicating electrically with the Z80-PIO.

Another function of the $\overline{IORQ}$ input is to inform the PIO when an interrupt acknowledge is occurring. When $\overline{IORQ}$ and $\overline{M1}$ are both logical 0, the Z80 is in the process of acknowledging an interrupt. If the PIO is programmed to respond to interrupts, and is, in fact, the interrupting device, an interrupt vector will automatically be placed onto the system data bus. In a later section of this chapter we will show how this can be accomplished.

$\overline{RD}$ This input is connected directly to the Z80 $\overline{RD}$ output line. Recall from earlier chapters on Z80 operations that the $\overline{RD}$ output will be a logical 0 whenever the microprocessor is reading data from memory or I/O. Therefore, when the $\overline{IORQ}$ input to the PIO is active, and the $\overline{RD}$ input and the $\overline{CE}$ are active, the Z80 is reading data from the PIO.

Note that there is no $\overline{WR}$ input to the PIO. If the $\overline{IORQ}$ and the $\overline{CE}$ are both active, but the $\overline{RD}$ is not active, the Z80 must be writing to the device. The write operation is a default condition and not explicitly specified with a $\overline{WR}$ input line. In effect, a $\overline{WR}$ input line would be redundant.

IEI and IEO These are interrupt enable input and output lines for the PIO. They are meant to be used in the interrupt priority daisy chain with the Z80. We will discuss them in detail in the interrupt section of this chapter.

$\overline{INT}$ This output is an open drain connection that is active whenever the PIO has been programmed for interrupts and is actively requesting an interrupt from the Z80 microprocessor. The $\overline{INT}$ will be explained further in a later section on interrupts.

A0–A7 These are the port A I/O lines. It is via these lines that the PIO communicates with an external device. A0 is the least-significant bit of this I/O bus.

$\overline{\text{ASTB}}$ This input is supplied by a peripheral device for handshaking data into the port A register. The actual operating characteristics of the input line are dependent on the mode of operation of the PIO. Here are some possible uses for this input:

- When the port is in the output-only mode, the rising edge of this strobe will indicate that the peripheral device has received the data.
- In the input-only mode, the strobe is used by the peripheral to load the data from the peripheral device into the port A register. The data is loaded into the PIO internal register when this input is active. We will show that it is not necessary to use this input line to use the PIO as an input port.
- When port A is programmed for the bi-directional mode of operation, data from the internal output register A will be placed onto the A0–A7 lines. When the line goes to a logical 1, the output data is removed, and the receipt of data by the external device is acknowledged. We will explain this in more detail in a later section.

ARDY *(Register A Ready)* The function of this output is dependent on the operating mode of the PIO. Similar to the $\overline{\text{ASTB}}$, the following are the possible functions of this pin:

- When the A register is programmed to the output mode, this line indicates to the peripheral device that data has been loaded onto the lines A0–A7. The external device connected to the port A lines will use this output as an indication that the data can be taken or read from the port.
- When the port is programmed in the input-only mode, this signal indicates when the input data has been read from the port A input register by the Z80 CPU.
- When the port is programmed in the bi-directional mode, the signal is active when the data is available in the port A output register. This signal indicates to the external device that data is present. The external device must then assert the $\overline{\text{ASTB}}$ line in order to place the output data on the A0–A7 lines.

B0–B7 These are the I/O lines that connect the port B to the external device.

$\overline{\text{BSTB}}$ *(Port B Strobe Pulse Input)* The function of this line is the same as for port $\overline{\text{ASTB}}$, with the following exception:

- When port A is programmed in the bi-directional mode, this

signal will strobe data from the peripheral device into the A register input. This implies that the B port does not have the bi-directional capability.

BRDY *(Register B Ready)* This is an output which is active logical 1. The definition of this output is identical to the ARDY signal described earlier. An exception to output is when port A is programmed in the bi-directional mode—the signal goes to a logical 1 when the port A input register has been read by the Z80 and is ready to accept new data from the peripheral device.

8-3: Connecting the Z80-PIO to the Z80

Let us now learn how to connect the Z80-PIO to the Z80 for reliable communication. Figure 8.3 shows a typical connection of the PIO to a Z80 microprocessor. Let's discuss some of the main points of this figure.

The B/A select line of the Z80-PIO is physically connected to the address output line A0 of the Z80. The C/D line is connected to the address output line A1 of the Z80. With the address lines connected in this way, the port definitions of the device are:

A1	A0	PORT SELECTED
0	0	A data
0	1	B data
1	0	A control
1	1	B control

The $\overline{CE}$ input line to the Z80-PIO is logically decoded from the system address lines A7–A2 of the Z80. The I/O addresses in Figure 8.3 are 2C, 2D, 2E, and 2F. In this figure, the interrupt lines for the Z80 and the PIO are not connected. We will connect these lines when we discuss the operation of the internal interrupts (in a later section).

8-4: Resetting the PIO

When power is first applied to the Z80-PIO, the device enters a reset state. The reset state is the following:

1. Both port mask registers are reset to inhibit all port data bits. (Because we have not yet discussed what the mask registers are, it is now important only to understand that this register is reset.)
2. Port data bus lines are set to a high impedance state and the READY handshake signals are inactive. Mode 1 (input only) on both ports is automatically selected.

3. The interrupt vector address registers are not reset. These registers are the interrupt vectors that will be placed onto the data bus when the PIO gets an interrupt acknowledge from the Z80.
4. Both port interrupt enable flip-flops are reset. This disables all interrupt response from the PIO until the device is programmed to respond.
5. Both port output registers are reset.

The preceding reset mode for the PIO occurs when power is first applied to the device. An external hardware reset may be applied to the device

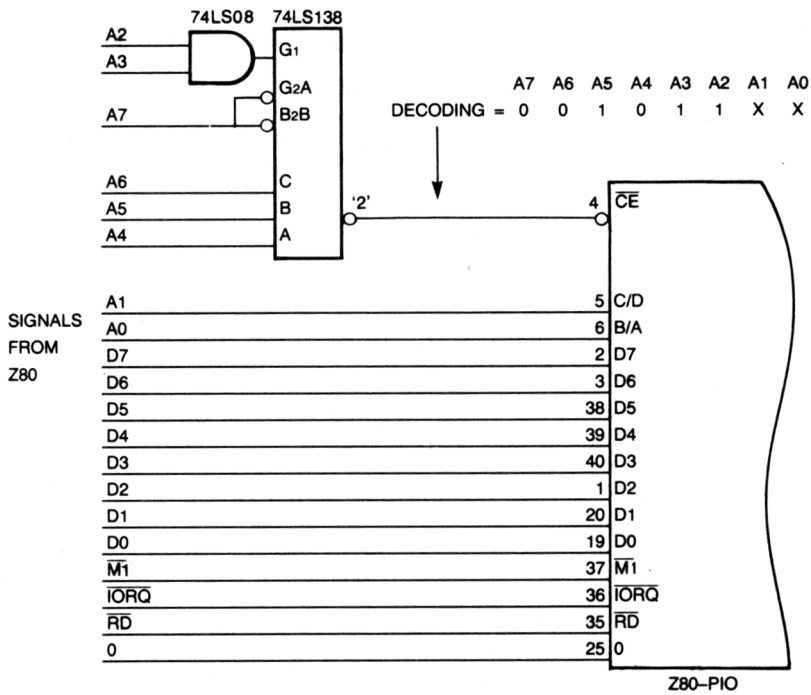


Figure 8.3: A schematic diagram showing the physical connections between the Z80 and the Z80-PIO.

by using the circuit that appears in Figure 8.4. As shown in this figure, the $\overline{M1}$ is asserted without $\overline{RD}$ or $\overline{IORQ}$ being asserted. This condition will never occur with the Z80 microprocessor during normal program execution.

8-5: Programming the PIO in Mode 0 (Output Port)

Let's now discuss the use of the PIO in mode 0. This mode allows the ports to be used as output only. Both ports can be programmed independently. In this example we will only program port A.

We will start this discussion by assuming that the device has just been reset. The first register to program in the PIO will be the control register to set the mode. The address lines will be A0 = 0 for port A, and C/D = 1 for control. Port address 2E will be the address of the port A control register. The data written to the register is shown in Figure 8.5. We can see in this figure that the lower four bits are set to a logical 1, which indicates that the mode for the port is being set. Bits D7 and D6 determine the mode. This gives the possible mode words as 0F (mode 0), 4F (mode 1), 8F (bi-directional), and CF (control mode). If bits D3-D0 are logical 1, bits D4 and D5 are ignored.

This is all that is required to use the PIO as an output port. Figure 8.6 shows a Z80 program that enables port A as an output and then writes a binary count to the output port.

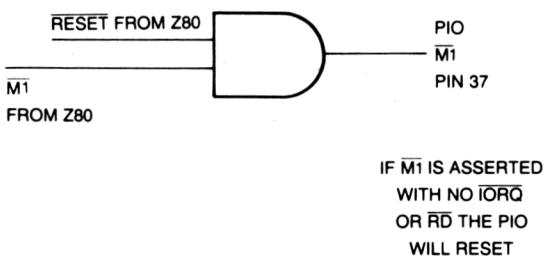


Figure 8.4: The hardware required to apply an external reset to the Z80-PIO device. The device does not have a unique input pin for reset.

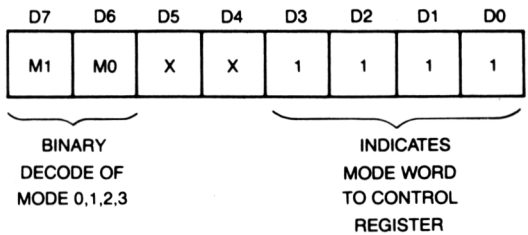


Figure 8.5: The lower four bits of the control word are set to logical 1. This indicates that the mode for the port is being set.

```
;
;
1800   3E0F            LD A,0FH
1802   D32E            OUT (2EH),A        ;OUTPUT THE CONTROL WORD
;
;  PORT A = OUTPUT, B/A = 0, C/D = 1
;
;  NOW WRITE 53H TO THE OUTPUT PORT A
;
1804   3E53            LD A,53H
1806   D32C            OUT (2CH),A        ;OUTPUT THE DATA
;
;  PROGRAM PORT B AS AN OUTPUT PORT
;
1808   3E0F            LD A,0FH
180A   D32F            OUT (2FH),A        ;OUTPUT CONTROL WORD
;
; PORT B = OUTPUT, B/A = 1, C/D = 1
;
;  PORT B MAY BE USED AS A GENERAL OUTPUT PORT
;  AT THIS TIME BY WRITING OUTPUT DATA TO
;  PORT ADDRESS 2DH.
;
```

Figure 8.6: A Z80 program that enables the Z80-PIO port A as an output and then writes bytes to the port.

8-6: Programming Mode 1

In this programming mode, the PIO will be set up as an input port. The control word that enables the PIO into this mode is 4F. This word could be written to port 2F or 2E. We will program port B as an input port using port 2F. In this mode the data from the external device will be input to the port lines B0–B7, as shown in Figure 8.7.

If the port is not used in the handshake mode, the $\overline{\text{BSTB}}$ will be connected to a logical 0 (see Figure 8.8). This type of input operation simply reads the data at the input lines B0–B7. Figure 8.9 shows a Z80 program that initializes the PIO for port A as an output and for port B as an input. This program reads the data from port B and writes it to port A.

The second way to use mode 1 is via the handshake mode. In this mode the $\overline{\text{ASTB}}$, $\overline{\text{BSTB}}$ and ARDY, BRDY lines are used. The $\overline{\text{BSTB}}$ line strobes data into the input register, and the BRDY line indicates that the input register is empty, i.e., that it has been read by the Z80. Figure 8.10 shows a block diagram of how these signals are used in the handshake mode.

When the external device writes data to the PIO, the interrupt line becomes

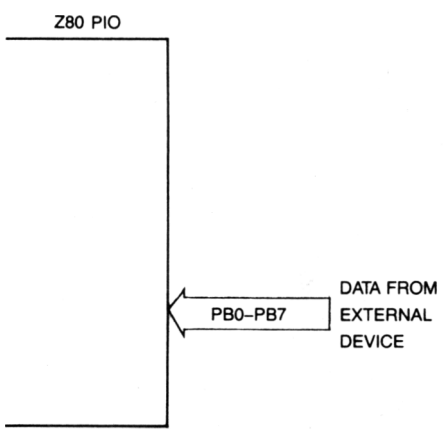


***Figure 8.7:** A block diagram showing how a peripheral device can input data to the Z80-PIO port B.*

active. As part of the interrupt service routine, the Z80 will read the data from the input port, thus setting the BRDY line to a logical 1 and indicating to the peripheral device that more data can be sent to the PIO.

If the device is used in this mode, it is necessary to perform a dummy read from the input port in order to set the BRDY line to a logical 1. This needs to be done because at reset the BRDY line is set to a logical 0.

Mode 0 may also be used in the handshake mode. In this mode the RDY line will go to a logical 1 when the Z80 has written data to the output port.

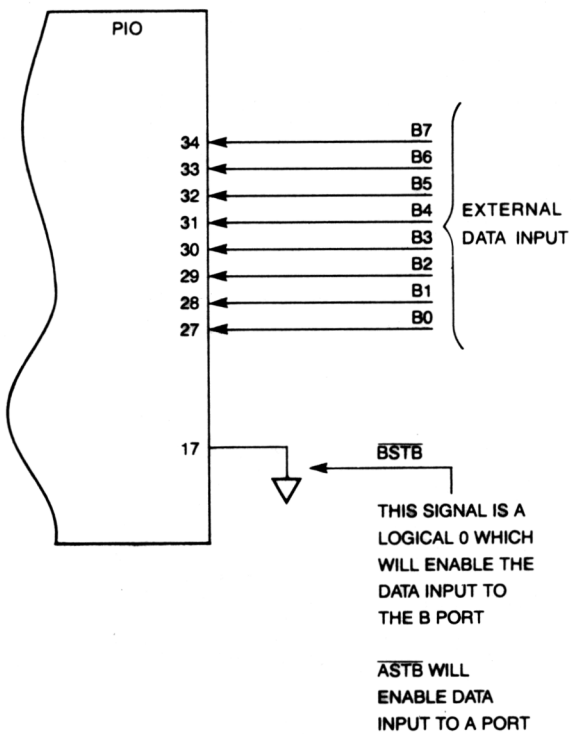


***Figure 8.8:** If the port is not used in the handshake mode, the $\overline{STB}$ input pin is connected to a logical 0.*

```
1800   3E0F            LD A,0FH          ;A REG = CONTROL WORD
1802   D32E            OUT (2EH),A       ;WRITE CONTROL WORD TO PIO
;
;  PORT A IS NOW AN OUTPUT PORT
;
1804   3E4F            LD A,4FH          ;A REG = CONTROL WORD
1806   D32F            OUT (2FH),A       ;WRITE CONTROL WORD TO PIO
;
;  PORT B IS NOW AN INPUT PORT
;
1808   DB2D            IN A,(2DH)        ;READ DATA FROM PORT B
180A   D32C            OUT (2CH),A       ;OUTPUT DATA TO PORT A
;
```

Figure 8.9: A Z80 program that will initialize the PIO for port A as an output and for port B as an input, thus allowing the program to read data from port A and write data to port B.

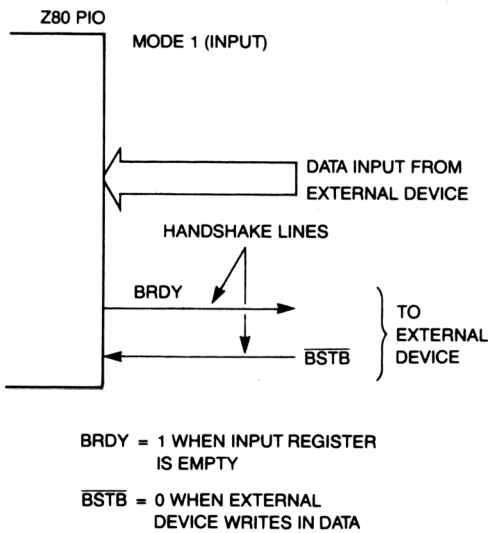


Figure 8.10: A block diagram showing how the handshake lines are used to input data to the Z80-PIO.

When the peripheral device reads this data, it will acknowledge the read operation by asserting the STB input. At this time the RDY line will go to a logical 0 and the $\overline{INT}$ output line will become asserted. Figure 8.11 shows a block diagram of how this mode is used.

8-7: Setting the Interrupt Control Word

In the preceding sections we have programmed the PIO in mode 0 and mode 1. For both of these modes we have discussed interrupts, but we have not shown how to use them. Let us now discuss how the interrupts may be programmed and used in these two mode operations.

To set the interrupt control word, it is necessary to write to the control register of a particular port. The lower four bits, D3-D0, are set to 0111. This indicates that the upper four bits will define the interrupt control word. Figure 8.12 shows the bit definitions for the interrupt control word.

If bit D7 of the interrupt control word is set to a logical 1, the interrupt enable flip-flop of the PIO is set, and the device may generate interrupts. If

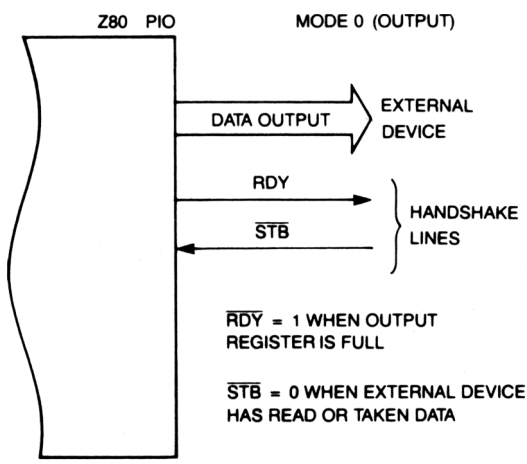


***Figure 8.11:** A block diagram showing the PIO set up to output data using the handshake lines.*

the bit is cleared or set to a logical 0, interrupts are disabled. If an interrupt condition occurs while interrupts are disabled, the PIO will generate an interrupt request when the interrupts are once again enabled. If bit D4 of the interrupt control word is set to a logical 1, any pending interrupt will be reset on the PIO.

Bits D6, D5, and D4 are used only in the mode 3 operation of the PIO. We will discuss this mode of operation later on in this chapter. For the interrupt operations of mode 0 and mode 1, we need only enable or disable the interrupt circuits of the PIO.

The interrupt enable flip-flop may be enabled or disabled by sending the following words to the port control register:

ENABLE INTERRUPTS 87H
DISABLE INTERRUPTS 07H

It is also possible to enable or disable interrupts without modifying the rest of the interrupt control word by writing an 83H to enable or an 03H to disable.

If the interrupts are enabled, the programmer must also specify the interrupt vector to be loaded onto the data bus and used by the Z80 in interrupt mode 2. The interrupt vector can be loaded into the PIO by writing

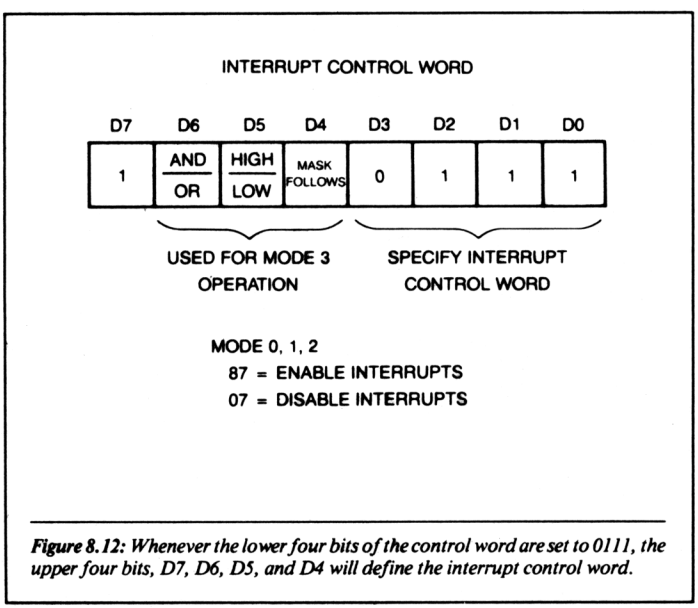


Figure 8.12: Whenever the lower four bits of the control word are set to 0111, the upper four bits, D7, D6, D5, and D4 will define the interrupt control word.

to the selected port control register the word format shown in Figure 8.13. Figure 8.14 shows a partial Z80 program that will enable port A interrupts and set up the port to be used as an output. The interrupt vector written to the PIO is equal to 76H.

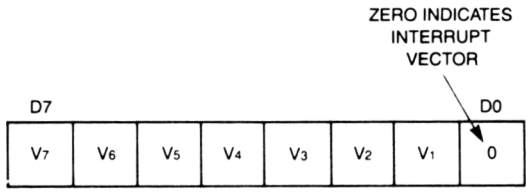


Figure 8.13: *A bit definition of the interrupt vector written to the Z80-PIO.*

```
;
1800   3E76            LD A,76H        ;INTERRUPT VECTOR = 76H
1802   D32E            OUT (2EH),A     ;OUTPUT VECTOR TO PIO
;
1804   3E0F            LD A,0FH        ;PORT A CONTROL WORD
1806   D32E            OUT (2EH),A     ;PORT A IS OUTPUT
;
;  NOW TO WRITE THE INTERRUPT CONTROL WORD
;
1808   3E87            LD A,87H
180A   D32E            OUT (2EH),A     ;ONE WAY TO ENABLE INTERRUPTS
;
; ANOTHER WAY TO ENABLE INTERRUPTS IS THE
; FOLLOWING
180C   3E83            LD A,83H
180E   D32E            OUT (2EH),A     ;THIS WILL ENABLE INTERRUPTS
;
;  ONE DOES NOT NEED TO EXECUTE BOTH TYPES OF
;  INTERRUPT ENABLES.
;
```

Figure 8.14: *A Z80 program to write the interrupt vector 76H to the Z80-PIO and to enable the interrupt system.*

8-8: Timing Review of Modes 0 and 1

Now that we have examined examples of how the PIO operates in modes 0 and 1, the timing diagram in Figure 8.15 should be easier to understand.

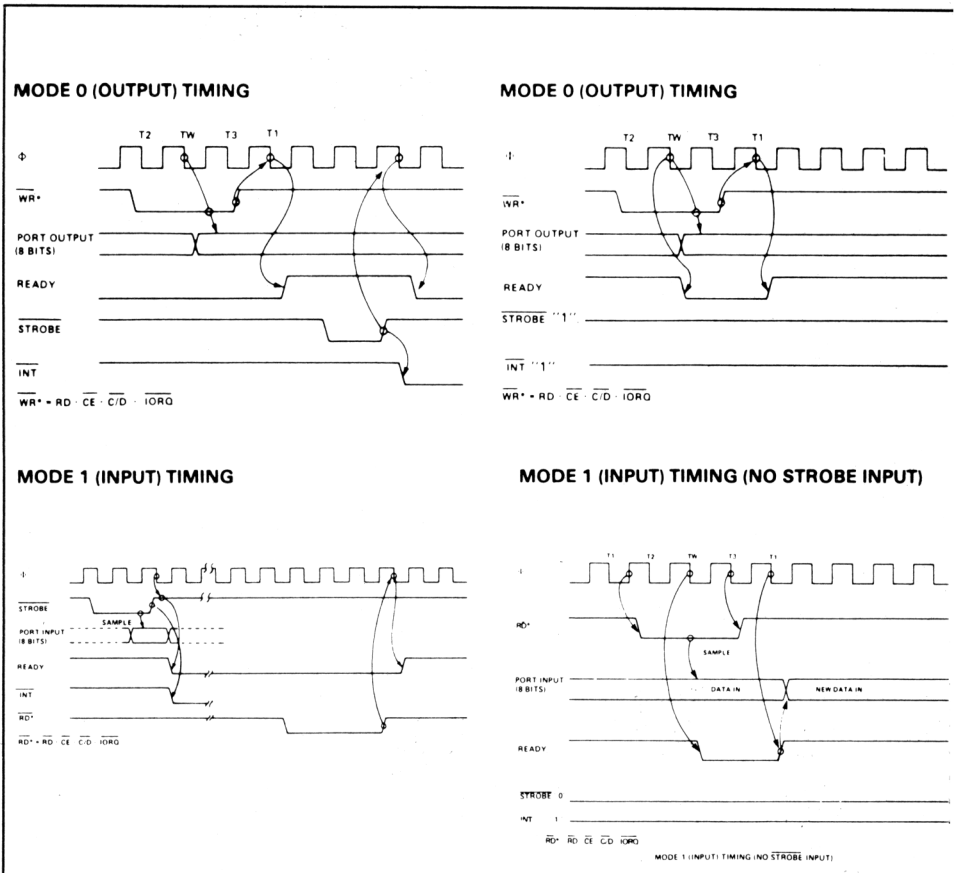


Figure 8.15: Timing diagrams of the mode 0 and mode 1 operations of the Z80-PIO.

8-9: Using the PIO in Mode 2 (The Bi-directional Mode)

The bi-directional mode of operation is a combination of modes 0 and 1. This mode uses all four handshake lines on the PIO. Because of this, the I/O mode is available only on port A. Port B must be set in the control mode. The output interrupt vector will be programmed into port A, while the input interrupt vector will be programmed into port B.

Figure 8.16 shows the timing for a typical data transfer using the PIO in the bi-directional mode. Let's now discuss some of the details of this timing diagram. The port A handshake lines are used for output control, and the port B lines are used for input control. Each handshake line operates as if the selected port were programmed in mode 0 or mode 1. Let's start with the data transfer from the PIO to the peripheral device:

1. First, the Z80 programs the PIO to be used in the bi-directional mode.

2. Next, the Z80 writes a data word to the port A output register. At this time the ARDY output line goes to a logical 1, electrically indicating to the peripheral device that data is ready to be taken from the PIO.

3. Next, the external device asserts the $\overline{\text{ASTB}}$ input to the PIO. This action places the output data on the data lines A0–A7. When the peripheral device has read the data, the $\overline{\text{ASTB}}$ line will be unasserted to the logical 1 position.

4. Next, the $\overline{\text{INT}}$ output line becomes active, thus signalling the Z80 that the output data has been taken.

5. Let's now assume that the Z80 is receiving data from an external device. The $\overline{\text{ASTB}}$ line will be a logical 1. The external device will place data on the A0–A7 lines as it asserts the $\overline{\text{BSTB}}$ input signal. When the $\overline{\text{BSTB}}$ signal goes to a logical 1 from a logical 0 (as shown in Figure 8.16), the $\overline{\text{INT}}$ output line will become asserted. This is an indication to the Z80 that the external device has sent data to the PIO and is now waiting for the Z80 to read it.

6. When the $\overline{\text{INT}}$ is asserted, the BRDY output line becomes a logical 0. This indicates to the external device that the Z80 has not read the data and that no further data should be sent until the BRDY once again goes to a logical 1. This line will go to a logical 1 when the Z80 reads data from port A.

From the previous discussion, we can see that an interrupt is requested

when the external device has taken the output data (port $\overline{\text{ASTB}}$), or when the external device has written data to the PIO (port $\overline{\text{BSTB}}$). In each case the Z80 needs to know which of these two operations has occurred. This is accomplished by loading a different interrupt vector into ports A and B.

When the external device writes data to the PIO, the B interrupt vector is placed on the system data bus. When the PIO writes data to the external device and the data has been accepted, the port A interrupt vector is placed on the data bus.

In this mode of operation there is the possibility that port B may cause an interrupt due to a mode 3 type of operation, rather than the control line operation that we desire. To keep this from happening, it is necessary to insure that port B is disabled from causing interrupts due to this type of operation. Even though we have not yet discussed the mode 3 type of interrupt, we mention this fact here because it might be a source of possible problems when using the PIO in a mode 2 application.

Figure 8.17 shows a Z80 program for setting up the PIO in a mode 2 application. Interrupt service routines are used for inputting and outputting data. We will assume that the peripheral device will assert the $\overline{\text{ASTB}}$ and the $\overline{\text{BSTB}}$ at the correct time.

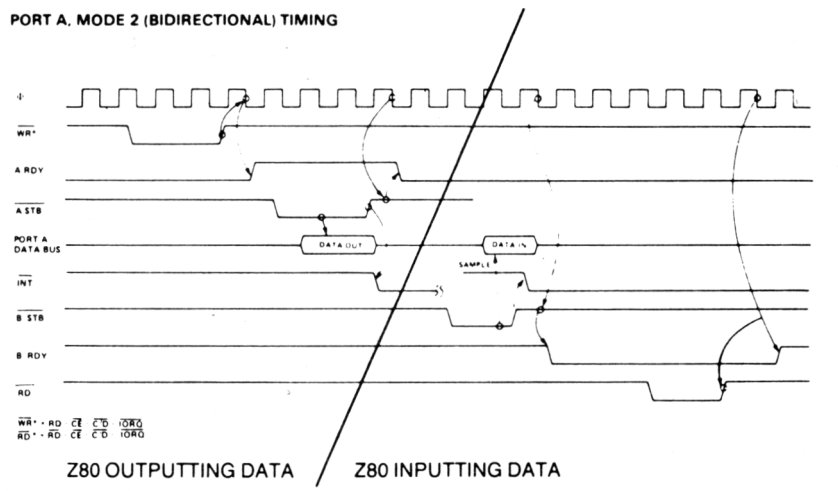


Figure 8.16: A timing diagram for a typical data transfer using the PIO in the bi-directional mode of operation.

```
;
2700            OUTDAT EQU 2700H        ;MEM LOC FOR OUTPUT DATA
2701            INDAT  EQU OUTDAT+1     ;MEM LOC FOR INPUT DATA
;
;
;
;  THIS PROGRAM WILL SET UP PORT A AS AN I/O PORT
;
1800   3E8F             LD A,8FH        ;THIS IS THE MODE WORD
1802   D32E             OUT (2EH),A     ;WRITE MODE WORD TO PA
1804   3E00             LD A,00H        ;INTERRUPT VECTOR FOR PA
1806   D32E             OUT (2EH),A     ;WRITE VECTOR TO PA
;
;
1808   3ECF             LD A,0CFH       ;MODE WORD FOR PORT B
180A   D32F             OUT (2FH),A     ;WRITE MODE WORD TO PORT B
;
;  MODE WORD SET UP PORT B IN MODE 3
;  THE BIT DEFINITIONS FOR PORT B FOLLOW
;
180C   3EFF             LD A,0FFH       ;ALL BITS ARE INPUTS
180E   D32F             OUT (2FH),A     ;WRITE TO PORT B
1810   3E17             LD A,17H        ;PORT B INT CONTROL WORD
1812   D32E             OUT (2EH),A     ;DISABLE INT, MASK FOLLOWS
1814   3EFF             LD A,0FFH       ;ALL BITS DISABLED
1816   D32F             OUT (2FH),A     ;WRITE MASK TO PORT B
;
1818   3E02             LD A,02H        ;INTERRUPT VECTOR PORT B
181A   D32F             OUT (2FH),A     ;WRITE TO PORT B
;
181C   3E3A             LD A,3AH        ;SET VECTOR TABLE UPPER BYTE
181E   ED47             LD I,A
;
;  VECTOR TABLE IS AT LOCATION 3A00 FOR PORT A
;  AND 3A02 FOR PORT B
;  THE ADDRESS OF THE SERVICE ROUTINES ARE FC81 FOR
;  PORT A, AND FD00 FOR PORT B.
;
;  PORT A INTERRUPT IS FOR OUTPUTTING DATA
;  PORT B INTERRUPT IS FOR INPUTTING DATA
;
1820   ED5E             IM 2            ;SET INTERRUPT MODE 2
1822   3E83             LD A,83H        ;ENABLE INT ON PIO
1824   D32E             OUT (2EH),A     ;ENABLE PORT A
1826   D32F             OUT (2FH),A     ;ENABLE PORT B
1828   FB               EI              ;ENABLE INT ON Z80
;
;
1829   C32918   LOOP   JP LOOP          ;WAIT FOR INTERRUPTS
```

Figure 8.17: *A Z80 program for setting up the PIO in a mode 2 application. Interrupt service routines are used for inputting and outputting data with the PIO. (continues)*

```
;
;
;     THE FOLLOWING IS FOR INITIALIZING THE INTERRUPT
; VECTORS AND THE ACTUAL SAMPLE INTERRUPT ROUTINES
;
3A00                    CODE 3A00H
3A00   81FC             DEFW  0FC81H     ;INTERRUPT VECTOR PORT A
3A02   00FD             DEFW  0FD00H     ;INTERRUPT VECTOR PORT B
;
;  NOW THE ACTUAL ROUTINES
;
FC81                    CODE 0FC81H
;
FC81   3A0027           LD A,(OUTDAT)    ;LOAD ACC WITH OUTPUT DATA
FC84   D32C             OUT (2CH),A      ;WRITE WORD TO PORT A
FC86   FB               EI               ;ENABLE INTERRUPTS
FC87   ED4D             RETI             ;RETURN FROM INTERRUPT
;
;
FD00                    CODE 0FD00H
;
FD00   D82D             IN A,(2DH)       ;READ THE DATA FROM PORT B
FD02   320127           LD (INDAT),A     ;STORE THE DATA IN MEMORY
FD05   FB               EI               ;ENABLE INTERRUPTS
FD06   ED4D             RETI             ;RETURN FROM INTERRUPTS
;
                        END
```

Figure 8.17: A Z80 program continued.

8-10: Using the PIO in Mode 3

The mode 3 operation of the PIO is intended for use in a non-handshake environment. The eight bits of each port can be assigned as individual input or output lines, in any order. For example, bits B0, B4, and B5 can be inputs, and all other port B data bits can be outputs. Here is how this is accomplished.

First, the programmer writes the mode word to the selected port control register. To select mode 3, the data written to the control register will be 0CFH. The word immediately following this mode select word will logically define how the eight bits of the port are to be used. A logical 1 in the bit position will set the data line as an input. A logical 0 will set it as an output. The following instructions will set the PIO into mode 3 and define bits B0, B3, and B6 as inputs. Lines B1, B2, B4, B5, and B7 will be outputs.

```
LD A,0CFH       LOAD CF TO A REG
OUT (2FH),A     SELECT MODE 3 FOR B REG
LD A,49H        BITS D0, D3, AND D6 = 1
OUT (2FH),A     SET B0, B3, AND B6 AS INPUTS
```

The PIO is now set up with the proper I/O bit definitions. A programmer may now send data to the output port, and read data from the input port. When data is read from the input port, the logical value of the programmed input bits will be read. The bits programmed as outputs will return the last logical value sent to the register during an OUT instruction. That is, all bits will be read during the execution of the input instruction. However, only those bits programmed as inputs will reflect valid input data.

Another feature of mode 3 operation is the interrupt capability. The PIO can force an interrupt request to occur whenever a particular logical combination exists on the port input lines. There are two (programming) parts to setting up this particular feature on the PIO. One is to specify the interrupt control word. Recall that in modes 0, 1 and 2 we specified the interrupt control word and ignored bits D6, D5, and D4. We will now make use of these bits.

Bit D6 of the interrupt control word defines how the input pattern we are looking for at the input port is to be logically formed. The two choices are ANDing and ORing. If the AND function is chosen, all of the selected bits must go to the active state before an interrupt is generated. If the OR function is chosen, an interrupt is generated if any of the selected bits goes to the active state.

The ANDing and ORing functions should not be thought of as simply Boolean functions. They should be thought of as ANDing or ORing the active state.

The active state can be a logical 1 or a logical 0. However, all of the inputs must be examined in the same active state. That is, all inputs selected must be examined for a logical 1 or a logical 0.

Bit D5 determines the active state of the selected port data lines. When D5 is a logical 1, the port data lines are monitored for a logical 1. When D5 is a logical 0, the port data lines are monitored for a logical 0.

After the ANDing and ORing of the input bits is selected and the active state is specified, it is necessary to define which of the eight bits are to be examined. If bit D4 of the interrupt control word is a logical 1, the next data sent to the command register will be the mask word. Figure 8.18 shows the mask bit word. If the programmer wishes to examine a particular bit of the port data bus, the mask bit will be a logical 0. Otherwise, the mask bit will be set to a logical 1. In other words, any bits you do not wish to logically examine, you mask off.

To fully illustrate how this mode can operate on the Z80-PIO, let's study an example. In this example, we will:

1. First, set up port B to operate in mode 3.

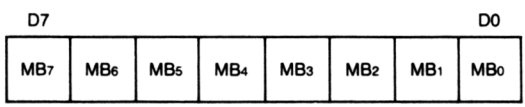
2. We will then program as inputs the port B data lines B2, B3, and B4.

3. We will then program as outputs the port B data lines B0, B1, B5, B6, and B7.
4. Next, we will monitor for an interrupt the port B data lines B3 and B4. We will mask off all other port B lines.
5. The active state of B3 and B4 will be a logical 0.
6. An interrupt will occur when both bits are active logical 0. This is the AND function of the active states.
7. The interrupt vector will be 08.

For this example, we will assume that the PIO is mapped into I/O space as defined previously—that is, with I/O ports 2C, 2D, 2E, and 2F. Figure 8.19 shows a block diagram of the external device connected to the PIO.

Here is the Z80 program to accomplish this set-up:

```
LD A,11001111B
OUT (2FH),A      SET MODE 3 IN PIO FOR PORT B
LD A,00011100B
OUT (2FH),A      SET BITS B2,B3,B4 AS INPUTS
LD A,00001000B
OUT (2FH),A      INT VECTOR = 08
LD A,11010111B
OUT (2FH),A      ENABLE INT,AND,LOW,MASK FOLLOWS
LD A,11100111B
OUT (2FH),A      BITS B3,B4 ARE NOT MASKED
```



MBx = 1 MASK OFF BIT

MBx = 0 USE BIT FOR INTERRUPT EQUATION

***Figure 8.18:** Bit definition for the PIO mask word. This byte will electrically inform the PIO which bits of the port the programmer wishes to examine.*

The preceding program will set up the PIO as described in the previous definition. Another section of the program that must be written is the interrupt service routine. (You may want to refer to Chapter 4 and the discussion on Z80 interrupts.)

A nice feature of the PIO is that it monitors the Z80 data bus, and when a RETI instruction is fetched from memory, it automatically removes the interrupt at the device. Recall that since it is normally the job of the programmer to explicitly remove the interrupt request during the interrupt service routine, this feature of the PIO relieves the programmer from having to perform this task.

8-11: Interrupt Enable and Disable

When the PIO is initially being set up, an unwanted external interrupt request may occur. This could create a problem for the system since the interrupt condition does not really exist. To eliminate the possibility of this

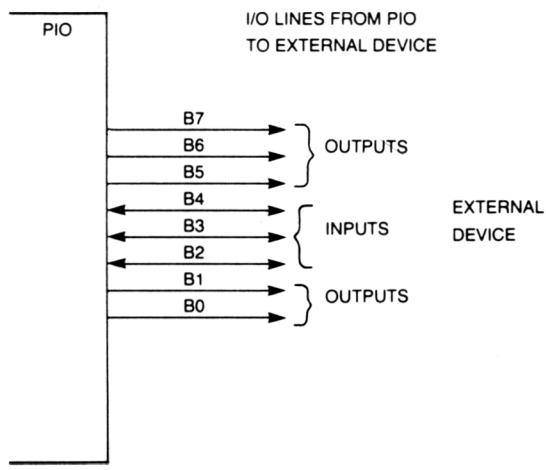


Figure 8.19: *A schematic diagram showing how the Z80-PIO is connected to an external device. Note that some of the I/O lines are inputs and some are outputs. The Z80-PIO is used in the bi-directional mode.*

happening, you should disable interrupt inputs from the Z80 while you set up the PIO. Another alternative is to simply disable the Z80 interrupts, and the program will then disable the interrupt from the PIO. You can accomplish this task with the following instructions:

```
LD A,03H
DI              DISABLE INTERRUPTS ON Z80
OUT (2EH),A     DISABLE INTERRUPTS ON PORT A OF PIO
OUT (2FH),A     DISABLE INTERRUPTS ON PORT B OF PIO
EI              ENABLE INTERRUPTS ON Z80
```

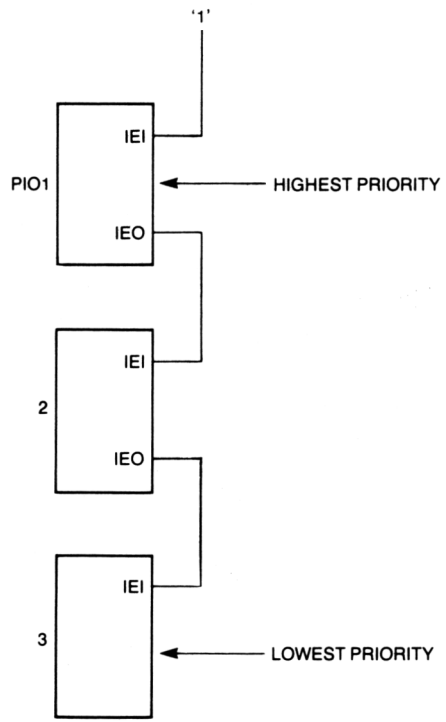


Figure 8.20: *Block diagram indicating how the interrupt priority daisy chain is used with the Z80-PIO interrupt system.*

The preceding program will allow the PIO to be disabled, and then the Z80 can have its own interrupt system re-enabled.

8-12: Priority Interrupt for the PIO

We will now discuss the three interrupt control lines of the PIO: $\overline{INT}$, IEO and IEI. The $\overline{INT}$ line is an output; it connects to the $\overline{INT}$ input line of the Z80. IEI is the interrupt enable input line. If this line is a logical 1, the PIO is electrically capable of causing an interrupt. If IEI is a logical 0, the interrupt output $\overline{INT}$ is disabled.

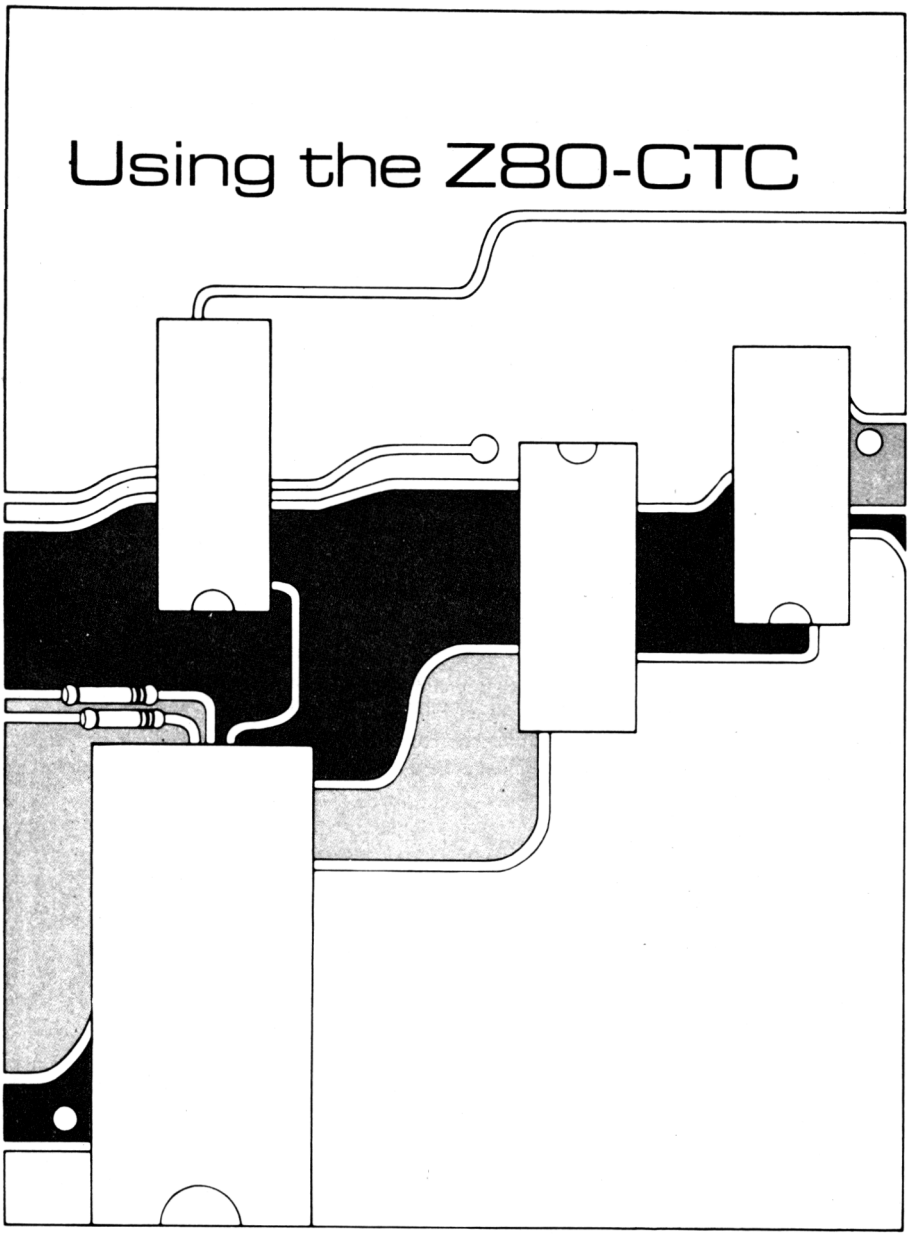
When the selected PIO device is actively requesting the interrupt, IEO output will go to a logical 0, thus indicating that this device is presently under service. By making use of these lines (IEO and IEI), it is possible to daisy chain up to four PIOs into a priority interrupt scheme, as is shown on the previous page in Figure 8.20. In this figure, the device with the highest priority has its IEI input line connected to a logical 1.

CHAPTER SUMMARY

In this chapter we have discussed the important points of the Z80-PIO. We began our discussion by examining a block diagram of the PIO and continued with a discussion of each operating mode. As each operating mode was presented, examples were given of Z80 programs to set up the device.

The hardware necessary for interfacing the PIO to the Z80 was shown and discussed. The information presented in this chapter may be applied to almost any system application.

Using the Z80-CTC



Chapter 9

INTRODUCTION

In this chapter we will discuss the Z80 counter timer chip, also known as the Z80-CTC. The CTC is an LSI chip designed specifically for use with the Z80 microprocessor. This device is extremely versatile. We will examine the special features that add to its versatility as we proceed through this chapter.

We shall begin by examining a block diagram of the device and discussing the important registers and hardware connections. By the end of this chapter you should be able to program and connect the CTC to your own Z80 system to perform the functions you desire.

9-1: Block Diagram of the CTC

The CTC derives its name from the two basic functions it performs: counting and timing. There are four independent counter-timer channels in a single 24-pin dual-in-line (DIP) package.

Figure 9.1 shows a block diagram of the CTC. Let's examine it.

We can see in this figure that the CTC has four independent, almost identical, channels called Ch0, Ch1, Ch2, and Ch3. Connected to three of the channels (0, 1, and 2) are two signal lines called zero count/time out (output) and clock trigger (input) that interface with an external system. Channel 3 has only the trigger input line. This is due to the pin limitation of the 24 pin DIP.

Also shown in Figure 9.1 is the *internal control logic.* This block insures that all data transfers on the internal device bus are timed properly.

Figure 9.1 also shows an interrupt control logic block. The interrupt structure of this device is quite powerful. Later on, we will explain in detail how the interrupt logic is used.

The final block in Figure 9.1 is the *CPU bus I/O*. This logic provides the interface between the Z80 and the CTC. There are eight data lines and six control lines connected to this block. It is via this block that the Z80 communicates electrically with the CTC, so that it can be programmed to perform in exactly the manner the programmer wishes.

9-2: A Closer Look at the Channel Block

In Figure 9.1 we can see how all of the major blocks of the CTC interact with one another. However, to program and use the CTC we must sharpen our focus. To that end, let's take a closer look at the block labeled channel 0. (Note that this examination is valid for all other channels as well, except, of course, Channel 3—as it does not have the zero count/timeout output line.)

Figure 9.2 shows a block diagram of a channel. We can see in this figure that a single channel is comprised of a channel control register, a time constant register and a down counter. Let's examine these items.

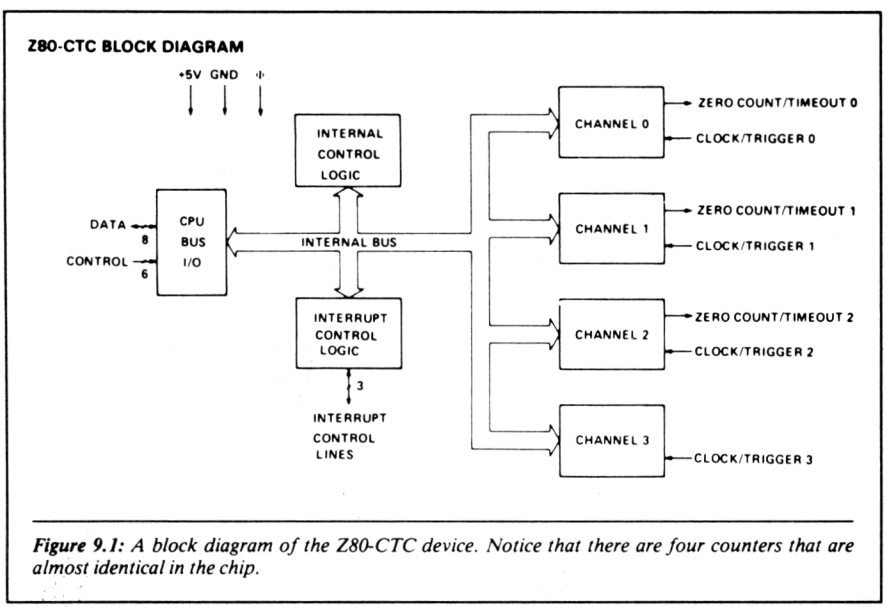


Figure 9.1: *A block diagram of the Z80-CTC device. Notice that there are four counters that are almost identical in the chip.*

The programmer writes information to the *channel control register.* In effect this register defines how the channel will operate.

Next to the channel control register is the *time constant register.* This register is 8 bits wide. It contains a binary number from 00H to FFH. The number in this register is used to set up the down counter block.

The *down counter block* is an 8-bit down counter. Its output is labeled zero count/timeout. When the counter reaches terminal count or zero, its output becomes active in a way programmed into the device.

In front of the down counter block is a block labeled *prescaler* (8 bits). This block prescales or divides the clock input to the down counter by 16 or 256, depending on how the device was programmed.

Another input to the down counter is a line labeled *external clock/timer trigger.* This line can be a direct clock input to the 8-bit down counter or it can act as an enable line for the prescaler output clock (see Figure 9.2). The function of this line depends on how the CTC was programmed to operate.

9-3: Pinout of the Z80-CTC

Before we begin programming the CTC, let us make sure that we can connect it to the Z80 microprocessor. Once we understand this connection, we can turn our efforts to writing the correct data to the CTC for proper operation. Figure 9.3 shows a pinout and signal name for this device's inputs and outputs. Let's go over each input and output line and define its function.

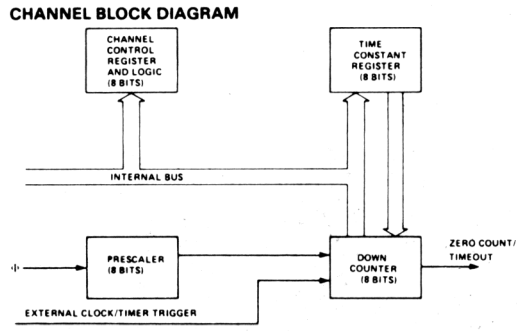


Figure 9.2: *A block diagram of a single counter channel in the overall device. This block diagram will serve for all four internal counter channels.*

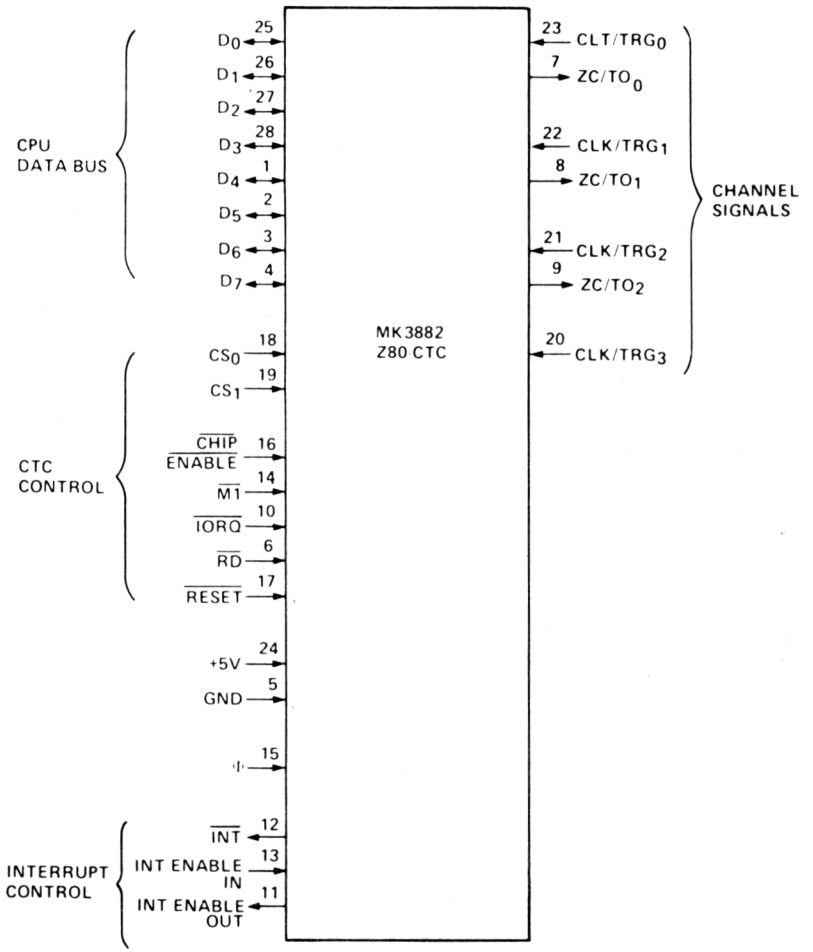


***Figure 9.3:** A pinout and signal definition for the Z80-CTC.*

9-4: CTC Signal Definitions

D7-D0 These are the data inputs and outputs from the device that connect to the Z80 data bus. D0 is the least-significant bit.

CS1, CS0 These are the two channel select lines. These two bits form the binary address of the channel that is being communicated with during an I/O read or write. Usually A0 and A1 from the Z80 CPU are connected to these inputs. The truth table for these two inputs is:

CS1	CS0	ACTIVE CHANNEL
0	0	Zero
0	1	One
1	0	Two
1	1	Three

$\overline{CE}$ This is the chip enable input, which is active low. When this line is a logical 0, the CTC is electrically capable of sending and receiving data from the Z80. This input is usually a decoded value of address lines A7-A2 of the Z80 address bus.

CLOCK This is the system clock input to the Z80 CPU. This clock is used by the CTC to syncronize certain internal data transfers.

$\overline{M1}$ The $\overline{M1}$ input line is connected to the $\overline{M1}$ output from the Z80 microprocessor. This signal is used with the $\overline{RD}$ input to determine when the Z80 is fetching an opcode from the system memory. It is also used with the $\overline{IORQ}$ input to determine when the Z80 is acknowledging an interrupt request.

$\overline{IORQ}$ This input is connected to the Z80 $\overline{IORQ}$ output line. It is an indication to the CTC that the Z80 is performing an output read or write operation.

$\overline{RD}$ The $\overline{RD}$ input connects directly to the Z80 $\overline{RD}$ output line. This line informs the CTC when the Z80 is reading data from memory or I/O. It should be noted that the CTC does not have a specific write input line. In order for the Z80 to write data to the CTC, the $\overline{CE}$ must be a logical 0, the $\overline{RD}$ a logical 1, and the $\overline{IORQ}$ a logical 0.

IEI *(Interrupt Enable Input (Active Logical 1))* When this input is a logical 1, it enables the device to output interrupt requests on the $\overline{INT}$ output line. When the IEI is a logical 0, the interrupt output is turned off on the device.

IEO *(Interrupt Enable Output (Active Logical 1))* When this line is a logical 1, it is an indication that the CTC is not servicing an interrupt

from any internal channel. This line is used in conjunction with the IEI to form a simple and very effective priority interrupt daisy chain.

$\overline{\text{INT}}$ This is the interrupt request output line that is open drain so that it may be wire ANDed with other $\overline{\text{INT}}$ output lines in the system.

$\overline{\text{RESET}}$ A logical 0 on the reset input will set the CTC into a known state. In this state, all channels are stopped from counting. All interrupt enable bits in all control registers are reset. This keeps the CTC from requesting any system interrupts. All output lines, ZC/TO and $\overline{\text{INT}}$, are set in the high impedance state. The CTC data bus output drivers are set to the high impedance state.

CLK/TRG_3-CLK/TRG_0 These are the external clock timer trigger inputs.

ZC/TO_2-ZC/TO_0 Zero count/timeout outputs are active high.

9-5: Connecting the CTC to the Z80

Now that we are familiar with the various inputs and outputs of the CTC, let's connect the CTC to the Z80 microprocessor. We will assume that there is no need for data bus buffering. (If you aren't familiar with this topic, refer to Chapter 2 and the discussion on data bus buffering, to see if your particular system application requires it.)

We will begin our discussion of interfacing to the Z80 by assuming that the interrupt capabilities of the CTC will be utilized. However, it is not necessary to use the interrupts of the CTC to take advantage of the device.

Figure 9.4 shows a complete schematic of the connections between the Z80 and the CTC. Let's examine several important points shown in this schematic.

The first connection to be made between the Z80 and the CTC is the data bus. We must connect lines D0-D7 from the Z80 to lines D0-D7 on the CTC. This connection is the physical means through which data passes between the Z80 and the CTC.

Address output lines A0 and A1 of the Z80 address bus connect to the CS0 and CS1 input pin of the CTC. The chip enable input is the decoded value from the address output lines A2-A7. In the schematic in Figure 9.4, the decoding of the CTC device is 40H, 41H, 42H, and 43H. Note that the complete decoding of the four I/O ports consists of the logical combination on the address lines A0-A7.

$\overline{\text{M1}}$, $\overline{\text{IORQ}}$ and $\overline{\text{RD}}$ are the next signals to be connected. These three signals are inputs to the CTC. They connect directly to the Z80 output pins of the same name.

Power and ground on the CTC will be + 5 and 0.0 volts. The clock input

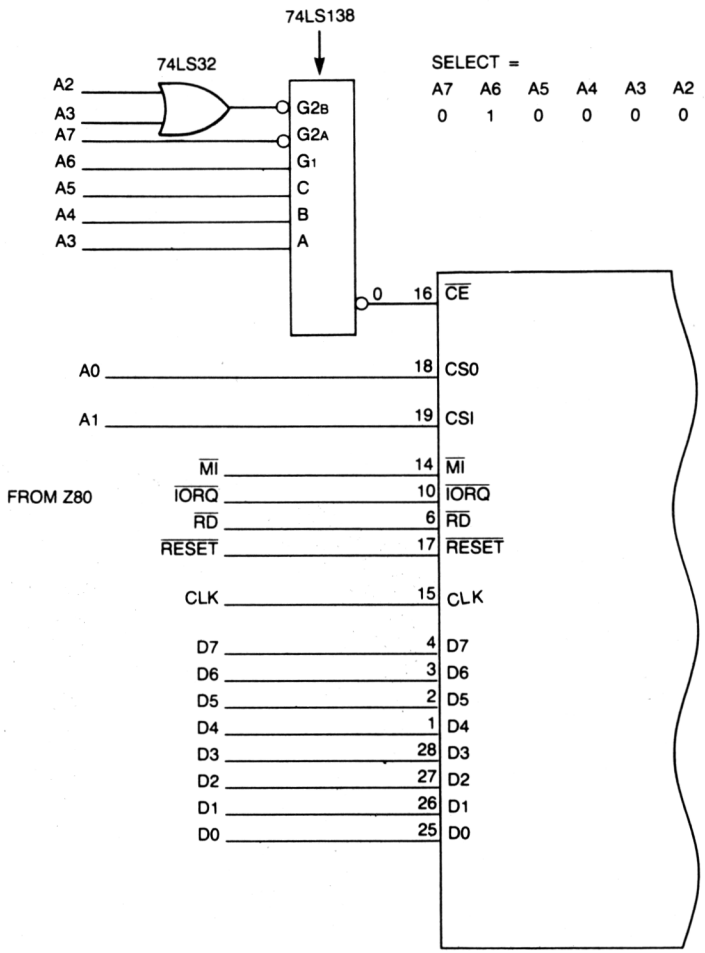


Figure 9.4: A schematic diagram showing the physical connections between the Z80-CTC and the Z80 microprocessor. Decoding for the CS input of the Z80-CTC is from address lines A7-A2.

of the CTC pin 15 is connected to the clock input pin 6 of the Z80. It should be noted that the maximum clock frequency that may be input is equal to 2.5 megahertz.

Finally, the $\overline{INT}$ output line of the CTC connects to the $\overline{INT}$ input line of the Z80. Note that the output is pulled up via a 10K ohm resistor. Only one pull-up resistor is required for the single $\overline{INT}$ input line of the Z80. This is due to the AND tieing of the interrupt request inputs to the microprocessor.

If the CTC is not to be put into a priority interrupt scheme, then the IEI input will be connected to Vcc. Doing this enables the interrupt via the hardware. The IEO line may be electrically ignored (i.e., left unconnected). If the CTC is to be connected into the priority interrupt scheme, then the IEI input is connected to the IEO output of the other peripheral device. This is shown in the block diagram in Figure 9.5.

9-6: Overview of the CTC Counter Mode

Now that we have the CTC connected to the Z80 microprocessor, we can focus our attention on writing software that allows the device to perform in

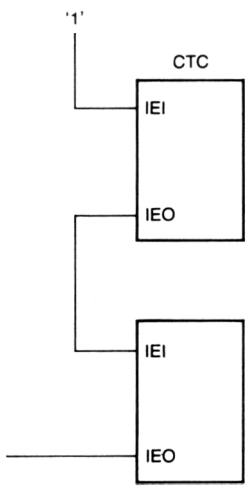


Figure 9.5: *A block diagram showing how to use the daisy chain priority scheme designed into the CTC chip.*

the way our application demands. In this section we will examine the important hardware and software details for using the CTC in the counter mode of operation. Figure 9.6 shows a block diagram of the CTC counter mode.

We can see in Figure 9.6 that the CTC is essentially set up to count external clocks. These external clocks are input on the selected channel CLK/TRG input pin. An example may be to use the CTC to count a certain number of events. When the value of the events has reached a preset limit, the microprocessor will take some action.

To use the down counter in the CTC we must program a time constant into the time constant register. This time constant must consist of a binary number with a maximum value of 0FFH. The time constant value will be the initial starting number for the down counter. When the down counter reaches 00H, the *zero count output* will become active. Later on we will show you exactly how to load the time constant register. For now it is only important that you understand what must be done in the CTC in a general sense. Later on you can concentrate on exactly how to do it.

Before the down counter can begin to perform in the counter mode, the channel control register must be made aware that this is the mode the

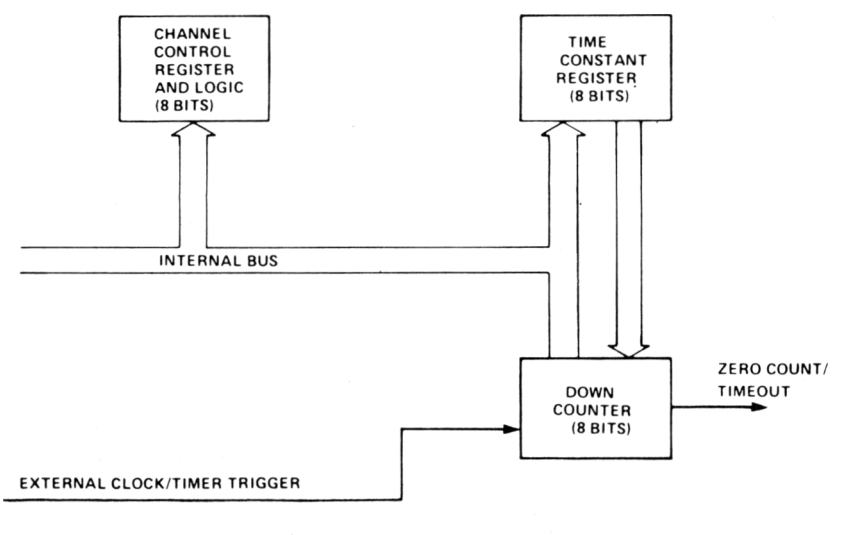


Figure 9.6: *A block diagram of the CTC used in the counter mode.*

programmer requires. To accomplish this a data word must be written to the channel control register.

After the control register has been written to, the down counter will start to count external clocks. The external clock will be sampled on the positive- or negative-going edge, as determined by the programming word sent to the control register.

An important point about the sampling of the external clock shown in Figure 9.6 is that the clock is strobed by the down counter on the positive-going edge of the system clock. This implies that you cannot input an external clock that is a frequency greater than the system clock input on pin 15 of the CTC. In fact, the specifications of the CTC will limit the upper frequency of the external clock input to one-half the system clock input frequency.

Now the counter is counting down. When the count reaches 00H, the ZC/TO output pin of the channel will go to a logical 1 for approximately one clock cycle of the system clock. If the interrupts are enabled, the CTC will generate an interrupt request to the Z80.

Another event to occur at the zero count is that the time constant register is automatically reloaded into the down counter. At that time the down count sequence is started over again. There is no interruption of the down count process. This is a nice feature because it may take several external clocks before the interrupt is serviced. The event counter will continue to operate and no event counts will be lost.

If a new time constant value is loaded into the time constant register during a down-counting sequence, the present count will continue until the zero count is reached. At that time the new time constant value will be loaded into the down counter. If the programmer wishes to terminate the count and start with a new one, then writing to the control register will accomplish this.

Both the time constant register and the control register must be written to before the CTC will begin to perform the counter function after initial power up or system reset.

At any time the data present in the down counter may be read by the Z80 to monitor system operations. This is accomplished by reading from the selected channel.

9-7: Programming the Channel Control Register

Let us now examine the programming data bits of the channel control register (discussed in the previous section). It should be noted that some of the channel control register bits will be used in the timer mode of operation that has not yet been discussed. We will examine these bits in more detail when we discuss the timer mode of operation.

Figure 9.7 shows the bits of the channel control register. The following is an explanation of each bit.

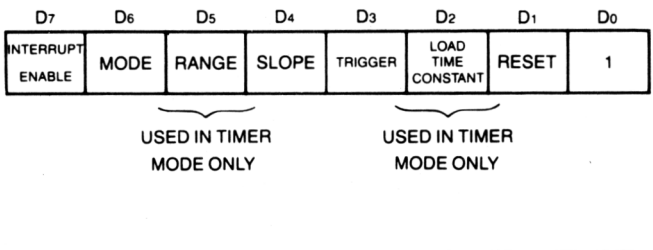


Figure 9.7: Bit definition of the channel control register for the Z80-CTC.

D0 This bit must be a logical 1 for the CTC to interpret the input data as a channel control register word.

D1 (*Channel Reset Input*) When this bit is a logical 1, the channel will stop counting or timing. None of the bits in any of the channel registers will be changed. The channel will resume normal execution when a time constant is again loaded into the time constant register.

D2 A logical 1 in this bit will electrically inform the CTC that the next channel word will be a time constant. Therefore, it is necessary to write two words to the CTC whenever a time constant is to be loaded. The first word is the control word (with this bit set) and the second is the time constant value.

D3 This bit is used in the timer mode only. If it is a logical 1, it specifies that the external trigger will start the timer running. If the bit is a logical 0, the timer will start as soon as the time constant register is loaded.

D4 This bit determines which edge of the external trigger/clock input pin is the active edge. The definition is:

Timer mode	D4 = 1	positive-edge trigger starts timing
	D4 = 0	negative-edge trigger starts timing
Counter mode	D4 = 1	positive-edge of external clock decrements
	D4 = 0	negative-edge of external clock decrements

D5 This bit is used in the timer mode only. It selects the prescaler value to be a divide by 16 or a divide by 256. A logical 1 equals a divide by 256; a logical 0 equals a divide by 16.

D6 If this bit is a logical 1, the channel is selected to be a counter. The down counter is clocked by the CLK/TRG input of the channel. If this bit is a logical 0, the channel is selected to be a timer. The down counter is clocked by the prescale clock input. The period of the pulse output is equal to (period of system clock * prescale factor (16/256) * time constant data word).

D7 When this bit is set to a logical 1, the internal interrupt structure is enabled. An interrupt vector will be written to the Z80 when the zero count is reached by the down counter. Also, if this bit is set, an interrupt vector must be written to the interrupt vector register before the channel will function correctly. If this bit is a logical 0, the internal interrupts are disabled.

9-8: Programming the Time Constant Register

The channel cannot begin operation until the time constant register has been loaded. This data word will be the next word written to the channel, provided bit D2 of the control word was set to a logical 1 on the preceding write operation.

Data in the range of 0 to 255 may be written to the time constant register. If all bits are a logical 0, then the value the down counter actually operates with is 256.

If a time constant register is loaded during a present counting sequence, the new value will not be used until the zero count of the present value is reached.

9-9: Programming the Interrupt Vector

Figure 9.8 shows the bits that may be programmed for the interrupt vector. The CTC is designed to be used with the Z80 when it is operating in a mode 2 interrupt sequence. A single interrupt vector is used for the entire chip.

As shown in Figure 9.8, bit D0 is a logical 0, which informs the CTC that this is an interrupt vector and not a control word. The vector is written to channel 0 only. Bits D1 and D2 are automatically set by the CTC when an interrupt acknowledge is issued by the Z80. The logical combination of these two bits uniquely defines a vector for a particular channel. In this way the user only needs to load a single interrupt vector for the entire chip.

The single vector has the upper 5 bits set by the user. Then, depending on which channel interrupted the processor, the CTC will place the correct D1

and D2 bits on the vector. Bit D0 will always be set to a logical 0. As an example of the four interrupt vectors, consider this. The user has written the interrupt vector 58 to channel 0.

This sets the interrupt vector bits as:

0 1 0 1 1 0 0 0

The corresponding channel interrupt vectors are then:

channel 0 = 0 1 0 1 1 0 0 0
channel 1 = 0 1 0 1 1 0 1 0
channel 2 = 0 1 0 1 1 1 0 0
channel 3 = 0 1 0 1 1 1 1 0

It is also possible for more than one channel to interrupt the CTC. There is a predetermined, internal priority on the device. Channel 0 has the highest priority; channel 3 has the lowest.

9-10: Programming the CTC for Counter Operation

Let us now study an example of using the CTC in a counter mode. We will first examine a non-interrupt driven system. We will then examine the same system, using interrupts. For this first example we will refer to the block diagram shown in Figure 9.9. We are counting the failures that occur in a system that is being tested. Each time the system fails a certain test, a pulse

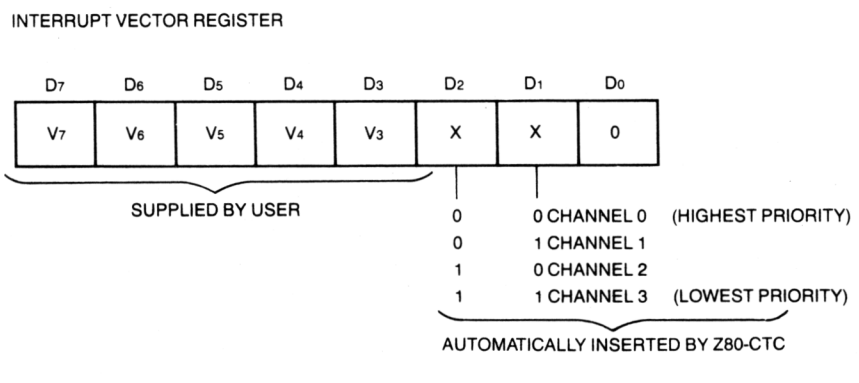


Figure 9.8: *Bit definition of the interrupt vector register contained in the Z80-CTC device.*

will be written to the CTC. The following is a complete definition:

1. The system will count the pulses. The active edge will be positive-going on the channel external clock input.
2. The CTC channel 1 will be used.
3. The CTC will be used in the polled mode. When the count reaches 46 decimal, the microprocessor will perform some action.

To start, let's define the bits for the control register. First, recall from the hardware schematic in Figure 9.4 that the I/O ports of the CTC are equal to 40H, 41H, 42H, and 43H. Since we are using channel 1, we will be writing and reading from port 41H. Before using channel 1, we must program the control register to define the counter operation. The following are the control register bits for channel 1.

bit 7 = 0	disable interrupts
bit 6 = 1	counter mode selected
bit 5 = 0	this is a don't care because it is used in the timer
bit 4 = 1	positive edge of external clock will count
bit 3 = 0	this is a don't care because this bit is used in time only
bit 2 = 1	time constant data will follow
bit 1 = 1	reset the channel if it is doing anything else. Recall that this is not a stored condition. A logical 1 generates a reset pulse to the specified channel.
bit 0 = 1	indication of a control word and not an interrupt vector

The word is written to the CTC using this Z80 instruction sequence:

```
LD  A,57H
OUT (41H),A        OUTPUT CONTROL WORD TO CTC
```

The next word that must be written to port 41H is the time constant. You may write other words to ports 40H, 42H, and 43H at this time. The CTC will expect the next word written to port 41H to be the time constant, because bit D2 was set to a logical 1 in the control word of channel 1.

In the definition of the problem, we stated that the number 46 would be the count value and that this would correspond to a hexadecimal value of 2EH. The following instructions will write the time constant to the CTC:

```
LD  A, 2EH
OUT (41H),A        WRITE THE TIME CONSTANT TO THE CTC
```

As soon as the time constant register is written to the CTC, the channel begins to operate. Here is a complete Z80 program that will monitor the count in a polled mode:

```
        LD A,57H
        OUT (41H),A      SET UP CH1 CONTROL WORD
        LD A,2EH
        OUT (41H),A      SET TIME CONSTANT = 46 DECIMAL
LOOP    IN A, (41H)      READ THE COUNT
        CP 00H           CHECK FOR FINAL COUNT
        JP NZ,LOOP       IF NOT ZERO THEN KEEP POLLING
                         AT THIS POINT THE ACTION OF THE Z80 IS
                         TAKEN INDICATING THAT THE CTC HAS
                         RECEIVED 46 FAILURE PULSES.
```

While the Z80 is performing the action to be taken, the CTC will automatically reload the time constant and begin the count again.

Let us now present the same program, but with interrupts inserted. After

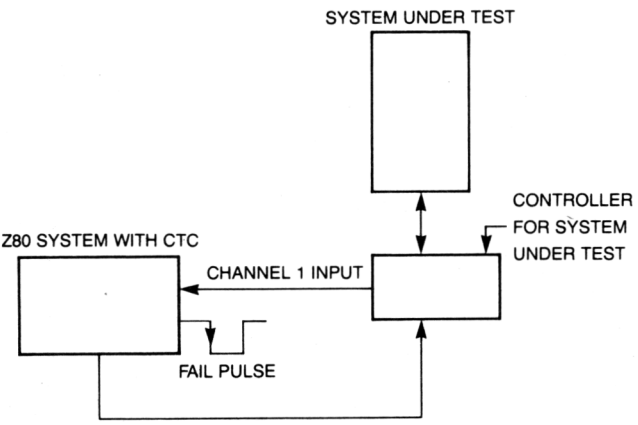


Figure 9.9: *A block diagram showing how the CTC can be used in a typical application. In this application an external system will send pulses to the CTC each time the external system fails a test.*

the Z80 has set up the CTC, it will loop, waiting for an interrupt. However, this does not have to be the case in your application. It is done here merely to illustrate a point.

The definition of the problem is essentially the same. The control register word will change to reflect the fact that interrupts are used. The Z80 program follows:

```
        DI              DISABLE INTERNAL INTERRUPTS
; FOR THE Z80
; THIS IS DONE SO THAT THE CTC DOES NOT ISSUE A FALSE
; INTERRUPT TO THE Z80 SYSTEM.
;
        IM 2            SET INTERRUPT MODE 2
        LD A,80H
        LD I,A          SET UPPER BYTE OF INTERRUPT TABLE
;
;
; IN THIS APPLICATION WE WILL ASSUME THAT THE INTERRUPT TABLE
; IS LOCATED AT MEMORY ADDRESS 8000-80FF. THE TABLE ADDRESS
; FOR THE CTC IS 8030-8037. 8030,8031 IS THE VECTOR ADDRESS FOR
; CH0 8032,8033 IS THE ADDRESS FOR CH1; 8034,8035 IS THE
; ADDRESS FOR CH2 AND 8036,8037 IS THE ADDRESS FOR CH3.
;
;
;
        LD A,0D7H CONTROL WORD FOR CHANNEL 1
;
; THE PRECEDING CONTROL WORD IS THE SAME AS FOR THE FIRST
; EXAMPLE, WITH THE EXCEPTION THAT BIT D7 IS NOW SET TO
; A LOGICAL 1
;
        OUT (41H),A     SEND THE CONTROL WORD TO CTC
        LD A,2EH        TIME CONSTANT = 46 DECIMAL
        OUT (41H),A     SEND TIME CONSTANT TO CTC
        LD A,30H        LOAD THE INTERRUPT VECTOR IN THE ACC
        OUT (41H),A     LOAD THE INTERRUPT VECTOR IN THE CTC
        EI              REENABLE INTERRUPTS
LOOP    JP LOOP         WAIT HERE FOR AN INTERRUPT
;
;
;
```

The interrupt service routine address will be loaded in the proper memory

locations 8032 and 8033. When the Z80 returns from the interrupt service routine, an RETI instruction should be used. When the CTC detects this instruction, the interrupt request is automatically removed from the $\overline{INT}$ input of the Z80.

9-11: An Example of the CTC in a Timer Mode

In this section we will show how to program the CTC for use in a typical timer mode. A block diagram of the CTC in the timer mode is shown in Figure 9.10. We will set the CTC up to be a clock output that will be a clock pulse at a frequency of 2400 hertz. This type of application could be useful for a baud rate generator or any system timer. We will assume that the input frequency of the system clock is equal to one megahertz.

This means that we must calculate the number by which to divide the input clock frequency to obtain a frequency of 2400 hertz. A block diagram of this problem is shown in Figure 9.11. The period of the 2400 hertz clock is equal to 416.7 microseconds. (For simplicity we will call it 417 microseconds.) The period of the input clock frequency is equal to one microsecond.

The prescaler on the CTC must be set to either 16 or 256. If we divide 417 by 16, we get 26.06. Thus, we will set it equal to 26. This means that the time

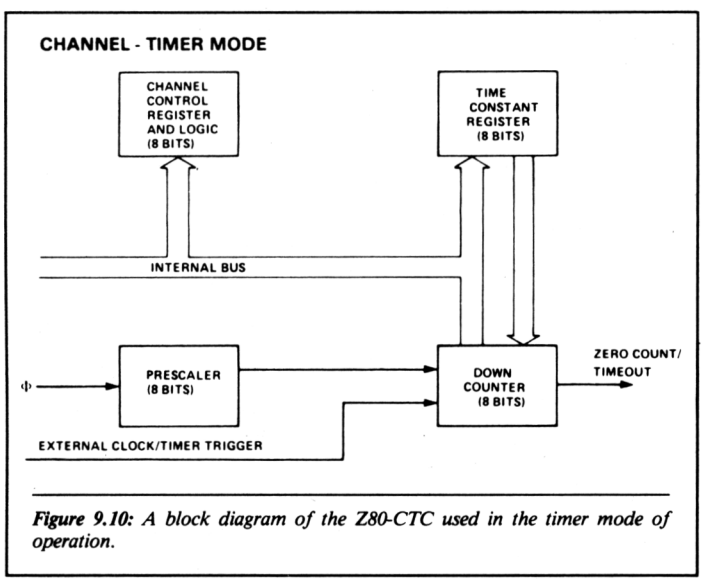


Figure 9.10: *A block diagram of the Z80-CTC used in the timer mode of operation.*

constant must equal 26. This gives us an overall divider of 16 × 26 = 416. Every 416 microseconds the output ZC/TO will go to a logical 1 and back to a logical 0. Using this figure, we can obtain a frequency very close to 2400 hertz. This is shown in Figure 9.11.

Let's now go over the setup for this example. First, we must set up the control word register. For this application we will use channel 2. The bits of the control word will be:

D7 = 0	no interrupts used in this application
D6 = 0	timer mode of operation
D5 = 0	prescale factor = 16
D4 = 0	this is a don't care because we are not using the trigger
D3 = 0	time starts when the time constant is loaded
D2 = 1	time constant word will follow this control word
D1 = 1	reset the channel if it is doing something else
D0 = 1	define this data as a control word

The time constant data will be equal to 26 decimal or 1AH. We can program the CTC in the following way. We will assume that the hardware is connected as shown in Figure 9.4.

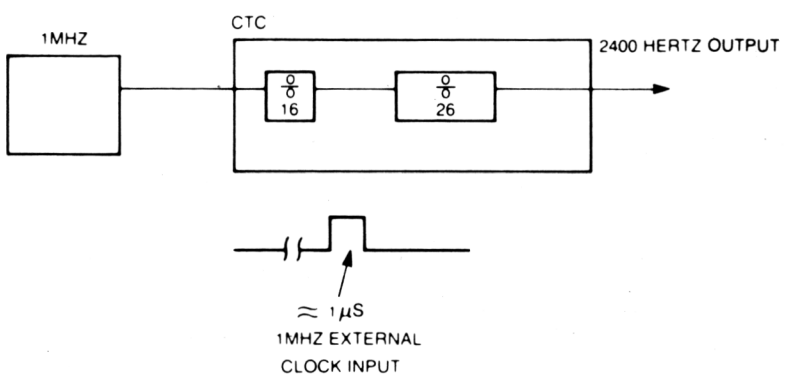


Figure 9.11: A block diagram using a CTC in a baud rate application. The output frequency will be 2400 hertz with an input frequency of 1 megahertz.

```
LD A,07H
OUT (042H),A      OUTPUT CONTROL WORD TO CH2
LD A,1AH
OUT (042H),A      SET THE TIME CONSTANT, START THE TIMER
```

The timer is now operating at channel 2 with a frequency very close to 2400 hertz.

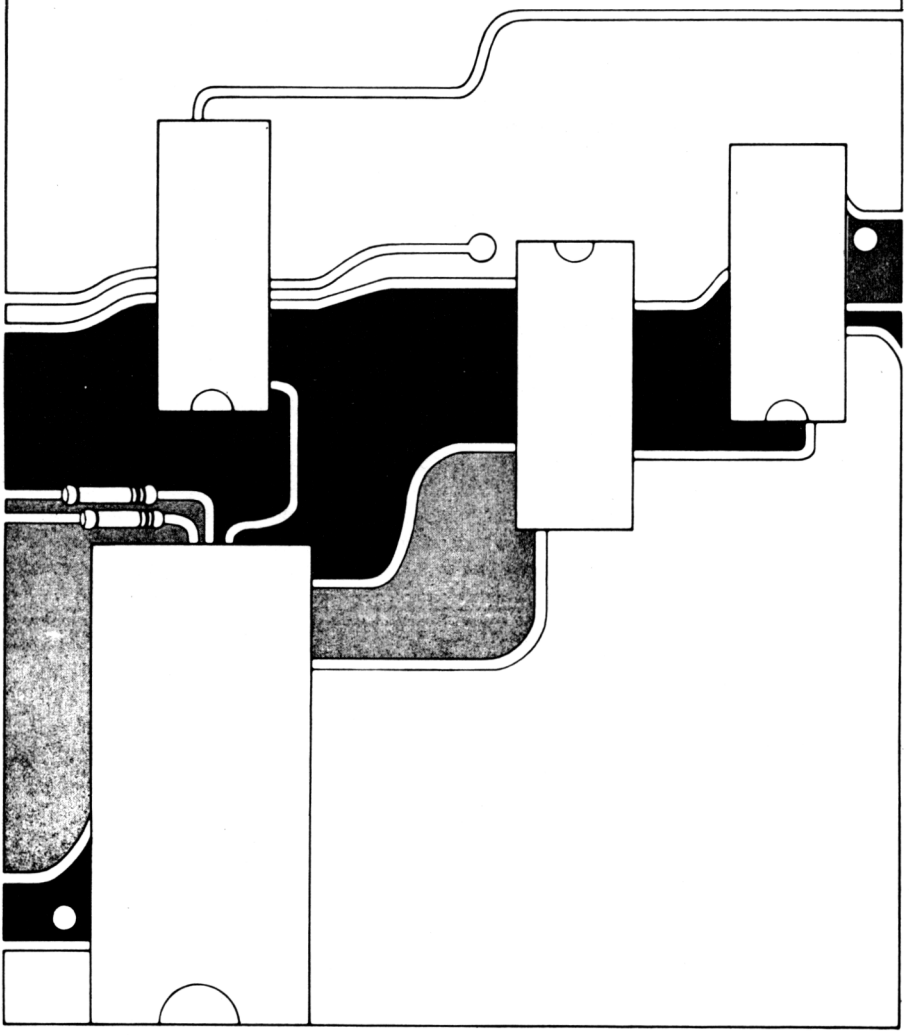
CHAPTER SUMMARY

In this chapter we have discussed the operation and application of the Z80-CTC. We began by examining a basic block diagram of the device; we then focused on the details necessary to connect the device to the Z80 microprocessor for reliable communication. Once the hardware was connected, we turned our attention to the important details of programming and using the CTC.

In this chapter, we have explored all the important internal registers. We have also presented typical Z80 software for programming the CTC. Finally, we have examined three examples of using the CTC chip. These examples were selected so that the information given would be applicable to most applications involving the CTC in a Z80 system.

The CTC is a very versatile device and can be of value in reducing the package count of many Z80 systems. As you gain experience in using this device, you will continue to discover more and more useful applications. This chapter is designed as a starting point. The information we have presented here can be used to help you connect and program the CTC in a way that will work in any system application.

Introduction to Serial Communication



Chapter 10

INTRODUCTION

In this chapter we will discuss the basic concepts of serial communication. During our discussion, we will examine an LSI device, called the 8251, which is used for the hardware implementation of a serial communication scheme.

If you are not familiar with the concept of serial communication, read this chapter carefully. It is important that you understand the material presented. By the end of this chapter you should have a firm, basic understanding of how serial communication can be realized in a microprocessor system.

10-1: What is Serial Communication?

Serial communication is the transmission of data in a bit stream, one bit at a time. All transmission occurs in a time sequential fashion.

Parallel communication, on the other hand, is the opposite of serial communication—all bits of the data transfer are received or transmitted at the same time. A good example of parallel communication is an I/O read or write operation, in which all eight data bits are transmitted (written) or received (read) at the same time. In fact, all of the data in the microprocessor communications that we have discussed so far have been transmitted and received in a parallel mode.

To further illustrate what serial communication is and how it differs from parallel communication, let's consider this example. We wish to send eight bits of data from one piece of hardware in a microprocessor system to another. We plan to send the data in two ways: parallel and serial.

Figure 10.1a shows how data appears when transmitted in a serial fashion; Figure 10.1b shows how it appears when transmitted in a parallel fashion. Notice that the parallel transmission requires 8 separate lines for the communication—one for each parallel bit to be transferred. In the serial transmission, only one physical line is required—the eight bits of data are sent over the single wire, one bit at a time.

10-2: Serial Timing

Let us now extend the simple serial example given in the previous section to illustrate some additional concepts. One of the critical points of serial

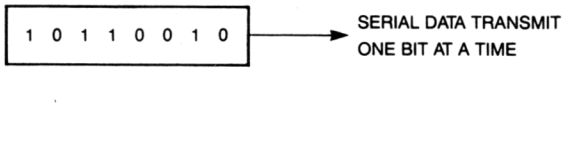


Figure 10.1a: *A serial data bit stream to be transmitted. The complete word is transmitted one bit at a time.*

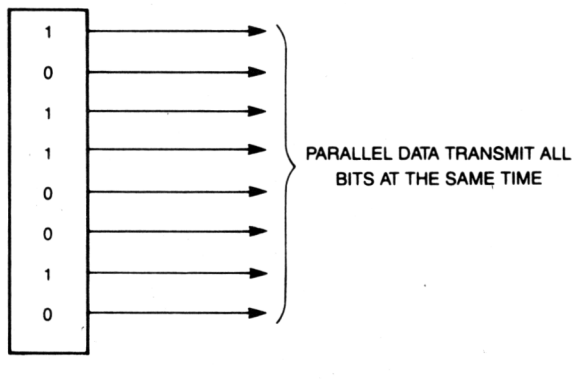


Figure 10.1b: *A parallel data byte to be transmitted. Each bit is transmitted in a parallel fashion.*

Chapter 10

INTRODUCTION

In this chapter we will discuss the basic concepts of serial communication. During our discussion, we will examine an LSI device, called the 8251, which is used for the hardware implementation of a serial communication scheme.

If you are not familiar with the concept of serial communication, read this chapter carefully. It is important that you understand the material presented. By the end of this chapter you should have a firm, basic understanding of how serial communication can be realized in a microprocessor system.

10-1: What is Serial Communication?

Serial communication is the transmission of data in a bit stream, one bit at a time. All transmission occurs in a time sequential fashion.

Parallel communication, on the other hand, is the opposite of serial communication—all bits of the data transfer are received or transmitted at the same time. A good example of parallel communication is an I/O read or write operation, in which all eight data bits are transmitted (written) or received (read) at the same time. In fact, all of the data in the microprocessor communications that we have discussed so far have been transmitted and received in a parallel mode.

To further illustrate what serial communication is and how it differs from parallel communication, let's consider this example. We wish to send eight bits of data from one piece of hardware in a microprocessor system to another. We plan to send the data in two ways: parallel and serial.

Figure 10.1a shows how data appears when transmitted in a serial fashion; Figure 10.1b shows how it appears when transmitted in a parallel fashion. Notice that the parallel transmission requires 8 separate lines for the communication—one for each parallel bit to be transferred. In the serial transmission, only one physical line is required—the eight bits of data are sent over the single wire, one bit at a time.

10-2: Serial Timing

Let us now extend the simple serial example given in the previous section to illustrate some additional concepts. One of the critical points of serial

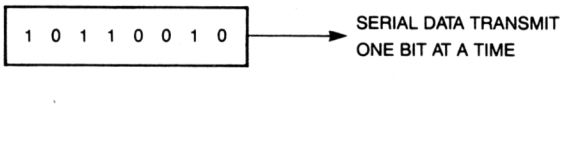


Figure 10.1a: *A serial data bit stream to be transmitted. The complete word is transmitted one bit at a time.*

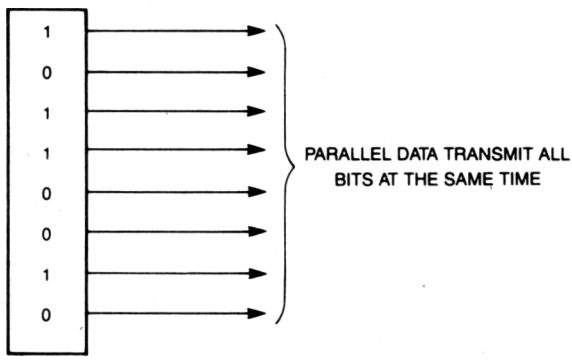


Figure 10.1b: *A parallel data byte to be transmitted. Each bit is transmitted in a parallel fashion.*

communication is the frequency of the transmitted data bit stream. This frequency is called the *baud rate*. Baud rate is defined as transmitted bits per second over a single serial line.

Typical baud rates are 110, 150, 300, 1200, 2400, 4800 and 9600. Let's suppose that we wish to transmit eight bits of data at 2400 baud. Let's take a look at what this actually means. The data to be transmitted is shown in Figure 10.2a. Figure 10.2b shows the wave form that would appear on the oscilloscope as the data is being transmitted.

Figure 10.2 also shows that the width in seconds of a single transmitted bit is equal to 1/baud rate. For this example the width of a single bit is equal to 1/2400 = .000416 seconds, or 416 microseconds. Knowing this, we can

Figure 10.2a: *Eight bits of data to be transmitted in a serial fashion.*

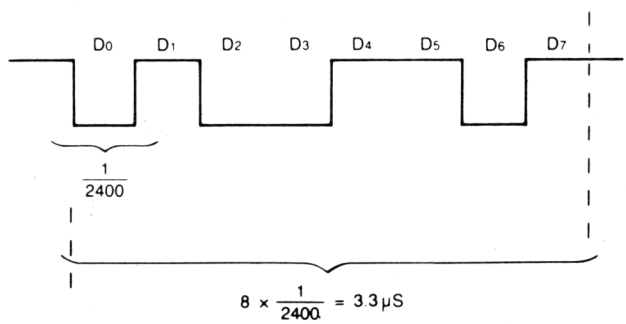


Figure 10.2b: *Wave form that would be generated if the byte were transmitted in a serial fashion; that is, bit D0, thru D1, D2, etc.*

calculate the amount of time required to transmit the entire eight bits. This time is equal to (8 × 416 μS) = 3,328 μS. The same eight bits would normally take less than 1 μS for a parallel transfer.

10-3: Converting Parallel Data to Serial Data

A major part of serial communication is the conversion of parallel data to a serial bit stream. Serial communication must take the parallel data from the microprocessor and transform it into a bit stream. The steps in the conversion process are:

1. Store the parallel 8-bit word in a shift register
2. Shift the 8 bits out of the register one bit at a time, at the correct baud rate.

These two steps are shown in the block diagram in Figure 10.3. We can see that the data to be transmitted is first generated by the microprocessor and then stored in the parallel load, 8-bit, shift register. The data is then shifted out D0 first, D7 last.

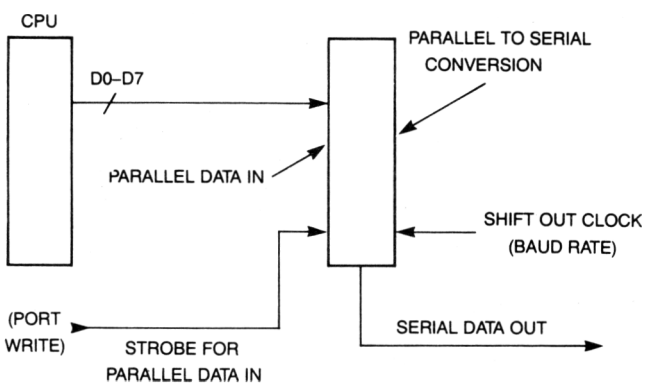


Figure 10.3: *A block diagram showing the process of converting a parallel byte into a serial bit stream to be transmitted. The microprocessor will write the data to be transmitted into a parallel to serial shift register. The data will then be shifted out at the correct rate.*

10-4: Start Bit

So far in our discussion of serial transmission we have explained baud rate and parallel to serial conversion. The data that is transmitted during a serial transmission must be capable of being received and electrically interpreted. To make this possible, another bit, called the *start bit,* is automatically added to the data bit stream.

The function of this bit is to let the receiving hardware know electrically when a new data stream is starting, and thus allow it to syncronize its clock to the input bit stream. Each data stream represents a single data character. You may want to think of each data stream as a byte of parallel data. Of course, the data does not have to be eight bits in length, but thinking of it in this way makes it easier to understand, if you are new to the subject of serial transmission.

When the data transmission output line is not sending information, it is in a state called *marking.* This is the idle state of a serial transmission line. Let us assume that the marking state of a transmission line is a logical 1. The start bit that is added to the data bit stream is of the opposite logical level to the marking logical level. In this case, the start bit will be a logical 0.

The start bit is really one bit added to the front end of the data bit stream. Whatever the baud rate of the bit stream is, the start bit will be equal to one bit width. This is shown in Figure 10.4. The receiving hardware will detect this start bit and enable the receiving section to take the new data.

10-5: Parity Bit

Another transmitted bit that can be added to a serial bit stream is the *parity bit.* This bit is inserted by the transmitter and used by the receiver. Here is an explanation of what the parity bit does.

When a word is set up to be transmitted, it contains a certain number of logical 1's. The number of logical 1's may be either odd or even. For example, the 8-bit word 54H has three 1's in it; the word 55H has four. The receiving hardware is set up to receive the data and to encounter either an even or odd number of 1's.

Suppose that the receiving hardware is set up to expect an even number of 1's in every serial bit stream. The number 55H would be alright, but the number 54H would not. Therefore, the transmitter would insert an additional 1 into the bit stream with the 54H, prior to the data being sent. This would add an additional bit to the data stream. This bit would either be a logical 1 or a logical 0, whichever makes the total number of bits in the new length bit stream an even number. Figure 10.5 shows the parity bit for the transmission of 54H and 55H.

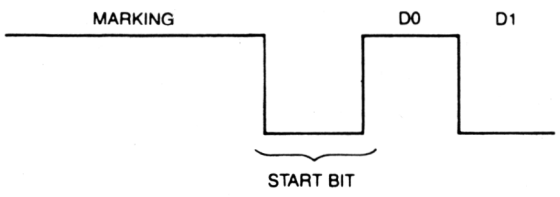


Figure 10.4a: A timing diagram showing how the start bit would appear if the first transmitted bit were a logical 1.

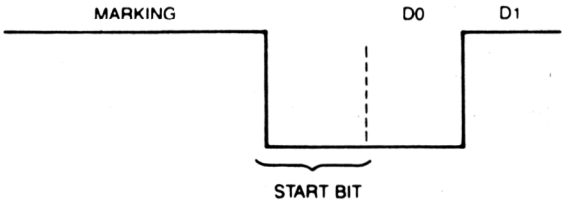


Figure 10.4b: A timing diagram showing how the start bit would appear if the first transmitted bit were a logical 0.

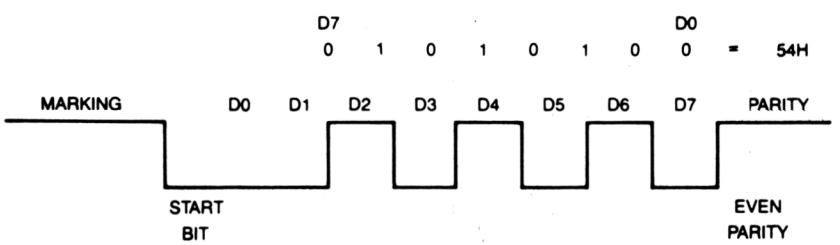


Figure 10.5a: A timing diagram showing an even parity bit for the data 54H. The parity bit is a logical 1 in this case.

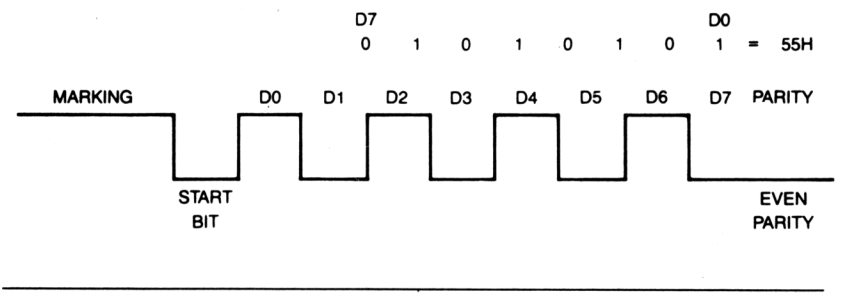


Figure 10.5b: *A timing diagram showing an even parity bit for the data 55H. Notice that the parity bit is a logical 0 for this byte.*

10-6: Stop Bit

The last bit added to the bit stream by the transmitting hardware is the *stop bit*. The receiving hardware expects to detect a stop bit at the end of the bit stream. There are different numbers of stop bits. There are 1, $1\frac{1}{2}$, or 2. Figure 10.6 shows a complete 8-bit data word with start bits, data, parity, and stop bits added. Thus a complete bit stream would consist of 12 bits, instead of the 8 that we started with. At 2400 baud, the absolute transmission time for the data shown in Figure 10.6 is equal to ($12 \times 416\mu S$) = 4.99 mS.

10-7: Review of Important Concepts of Serial Communication

In review, here is a list of the important points of serial communication:

1. Serial data is transmitted one bit at a time, the least-significant bit first.

2. Data is transmitted at a fixed rate called the baud rate. The baud rate is equal to bits per second transmitted. For example, 1200 baud would transmit data with a clock frequency of 1200 hertz.

3. Parallel data in a microprocessor system must first be converted to serial data to be communicated with, in a serial fashion.

4. Marking is a term used for the logical state of the serial transmission line when no data is present.

5. The transmitter of the serial data will add 1 data bit called a start bit at the front of the data. These bits will be of the opposite logical level to the marking state of the serial transmission line.

6. The transmitter may add a single bit, called a parity bit, to the end of the data stream. The parity bit can be inserted to generate even or odd parity. Parity is used by the receiver as a first-level error check on the transmitted data.

7. Transmitting hardware will add stop bits to the data stream after the parity bit. The stop bit will be 1, $1\frac{1}{2}$, or 2 bit widths wide. Stop bits are the same logical level as the marking level of the serial transmission line.

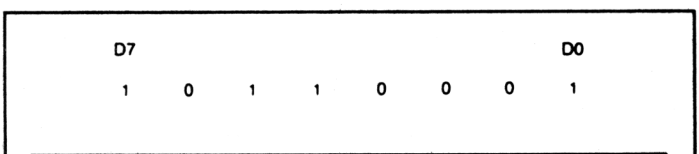


Figure 10.6a: Data to be transmitted in a serial fashion.

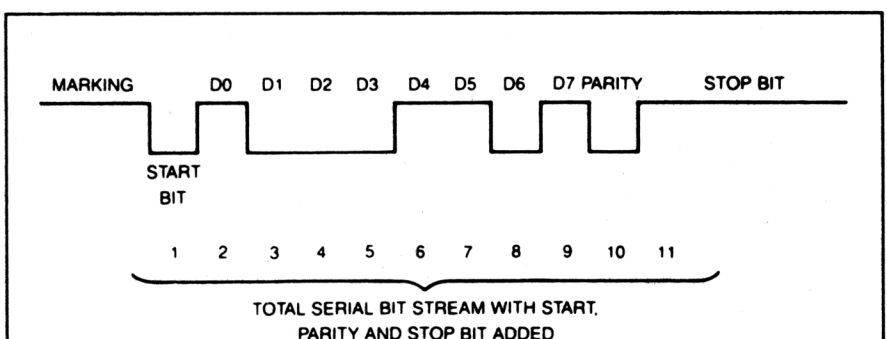


Figure 10.6b: Waveform generated by the complete transmission bit stream with start, stop and parity bits added.

10-8: Overview of the 8251

The concept of serial data transmission in a microprocessor system is often used. One of the most common uses is in computer to terminal interfacing. Since this type of data transmission *is* so common, special LSI devices have been designed to accomodate it. The 8251 is such a device. Figure 10.7 shows how the 8251 fits into the hardware architecture of a microprocessor system.

We can see in Figure 10.7 that the 8251 is treated exactly like the other special I/O devices that we have discussed (including the PIO and CTC). The Z80 communicates with the 8251 through the I/O system. The Z80 instructions set up or program the device to operate in a certain way. In this chapter, we will show you how you can connect the 8251 to a Z80 system and achieve reliable communication. First, however, let's discuss the important

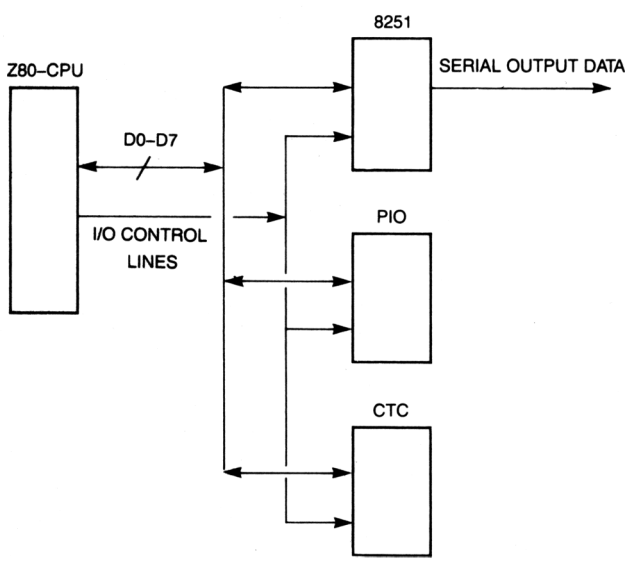


***Figure 10.7:** A block diagram showing how an 8251 fits into the architecture of the Z80 controlled system.*

software concepts involved in using the 8251. By the end of this discussion you should know how a typical serial I/O device operates. This knowledge should help you understand other serial I/O devices, like the Z80-SIO. (We will examine the Z80-SIO, in Chapter 11.)

Figure 10.8 shows a block diagram of the 8251. Let's discuss the function of each block in a serial communication scheme, starting with the *data bus buffer.* This block is the physical connection between the 8251 and the Z80 system data bus.

Next is the *read/write control logic.* This block insures that the data of the 8251 is read or written to the correct internal location with the correct timing.

The block labeled *modem control* is used to simplify the interface between the 8251 and a modem. For those readers who are not familiar with a modem, it is a device that is used to enable serial transmission over a telephone line. Figure 10.9 shows how a modem fits into a serial communication system over such a line.

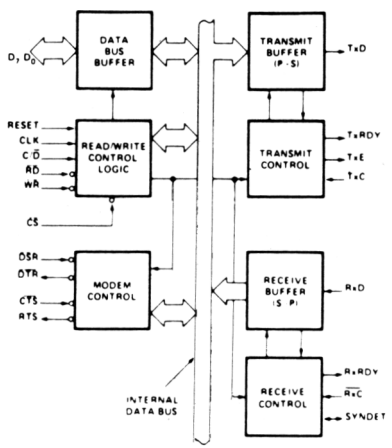


Figure 10.8: *A block diagram of the internal architecture of the 8251 Universal Synchronous Asynchronous Receiver Transmitter (USART).*

Referring to Figure 10.8, we can see that the two blocks labeled *transmit buffer (P-S)* and *transmit control* operate together to output the serial data stream. (*Note:* P-S stands for parallel to serial conversion.) Data is output from the device on the line labeled TxD. Tx stands for transmit, and D is for data. The transmit control block electrically monitors the status of the transmit buffer to determine if the buffer is empty, that is, if it is ready for another character to be transmitted.

The last two blocks shown in Figure 10.8 are *receiver buffer (S-P)* and the *receive control.* A receiver buffer will input a bit stream from another serial device and convert it into a parallel word, which can then be read by the Z80 microprocessor using an input instruction. The receiving control block will monitor the status of the receiver buffer to electrically determine when the buffer is full.

10-9: Pinout of the 8251

Figure 10.10 shows the pinout of the 8251 device. Let's now discuss the function of each pin. This will help you understand how the 8251 can be connected physically to the Z80 system busses.

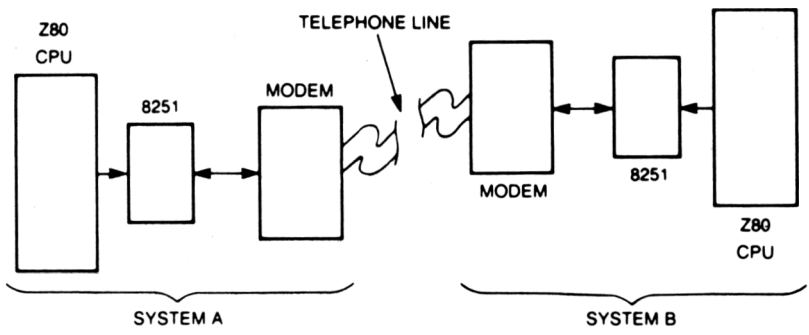


Figure 10.9: *This block diagram shows how a modem fits into the process of transmitting information from one place to another over a telephone line.*

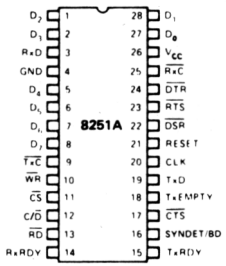


Figure 10.10: Pinout of the 8251 USART

D0–D7 These are the eight data lines that connect to the microprocessor system data bus. All information and programming words are input and output from the 8251 via these lines.

RESET This input is active logical 1. The 8251 is set to an idle state when the RESET is asserted. It then remains in this state until the appropriate programming words are sent to it via the data bus.

CLK (*Clock*) This input to the 8251 is used to syncronize internal data transfers. This is not the baud rate clock input. This clock must be 30 times faster than the baud rate of the receiver or transmitter.

WR (*Write Data, Active Logical 0*) When this input line is a logical 0, the data on the system data bus is written to the 8251 internal registers.

RD (*Read Data, Active Logical 0*) When this input line is a logical 0 the internal data from the 8251 is placed on the system data bus.

C/D (*Control/Data Input Line*) This line is used to define the internal registers of the 8251. When this line is a logical 1, data will be written or read from the control register. When it is a logical 0, data will be written or read from the data register of the device.

$\overline{CS}$ (*Chip Select Active Logical 0*) A logical 0 on this input line enables data to be written or read from the device.

MODEM CONTROL The following four pins are used to simplify the interface to a modem:

$\overline{DSR}$ Data Set Ready
$\overline{DTR}$ Data Terminal Ready
$\overline{CTS}$ Clear to Send
$\overline{RTS}$ Ready to Send

Let's examine them in detail.

$\overline{DSR}$ This input can be tested by the CPU. It is normally used to test modem conditions, such as Data Set Ready.

$\overline{DTR}$ This output can be set low by sending a certain bit pattern to the 8251. This output is normally used for Data Terminal Ready.

$\overline{CTS}$ A logical 0 on this input enables the 8251 to transmit serial data if internal conditions on the device permit it. The $\overline{CTS}$ line may also be used as a hardware handshake line.

$\overline{RTS}$ This output bit may be set low by writing the correct word to the 8251.

TxD (*Transmit Data Output*) This is the output line that serial data will be transmitted on.

RxD (*Receive Data Input*) This is the input line that data will be received on when it goes to the 8251 from the serial bit stream.

TxC (*Transmit Baud Rate Clock Input*)

RxC (*Receive Baud Rate Clock Input*)

TxRDY This output signals the CPU that the 8251 is ready to accept another character to be transmitted. This line can be used as an interrupt for the CPU.

TxEMPTY When the device has no characters to transmit, this line will go to a logical 1. It will be set automatically to a logical 0 when the CPU writes a character to the transmitter buffer.

10-10: Connecting the 8251 to the Z80 Busses

We will now learn how to connect the 8251 to the Z80 microprocessor. We will not concern ourselves with programming the device at this time. If the Z80 cannot reliably input or output data from the 8251, then proper programming of the device will be of little value, since the programming information will not electrically reach the device.

Figure 10.11 shows a complete schematic of how the 8251 connects to the Z80 system busses. The serial output connection of the device is not shown at this time. Figure 10.11 shows a data buffer between the data bus of the 8251 and the Z80 system data bus. This buffer may or may not be required in your system application. If it is not needed, then the D0–D7 data lines of the 8251 can be connected directly to the D0–D7 data lines of the Z80 data bus.

The $\overline{CS}$ input to the 8251 is the decoded address bus lines A1–A7. In this example, the port code is 7CH and 7DH. These two codes indicate that the Z80 address line A0 is connected to the C/$\overline{D}$ input line of the 8251. The $\overline{CS}$ is active when A0 is either a logical 0 or a logical 1.

$\overline{RD}$ and $\overline{WR}$ inputs are connected to the $\overline{IOR}$ and $\overline{IOW}$ control lines generated by the Z80 as part of the system control bus. The $\overline{RESET}$ input to the 8251 is active logical 1. This is the opposite logical level of the active state for the Z80 RESET input. Figure 10.11 shows that the connection between the 8251 and the Z80 system busses is similar to the connections between the busses and other peripheral devices (discussed in the previous chapters).

The baud rate clock is input to the TxC and RxC pins of the 8251. These clocks do not have to be the same frequency. The device is electrically capable of receiving and transmitting data at different baud rates. However, the baud rate clock must match the baud rate clock of the companion serial device that is involved in the communication.

Another clock input to the 8251 that must be externally supplied is input on pin 20. This clock is used for the syncronization of internal data transfers within the device. Specifications for this clock indicate that it must be at least 30 times faster than the transmit or receive clock inputs, TxC or RxC. For example, if you are transmitting or receiving at 2400 baud, the RxC and TxC input clock would be 2400 hertz and the input frequency of the clock input on pin 20 would be equal to 30 $\times$ 2400 hertz or 72 kilohertz.

10-11: The Serial Connection

Let's now connect the 8251 to the serial transmission and serial reception lines. There are various serial transmission standards available. For this example, we will choose one that is in wide general use today: the RS-232 electrical standard for serial transmission.

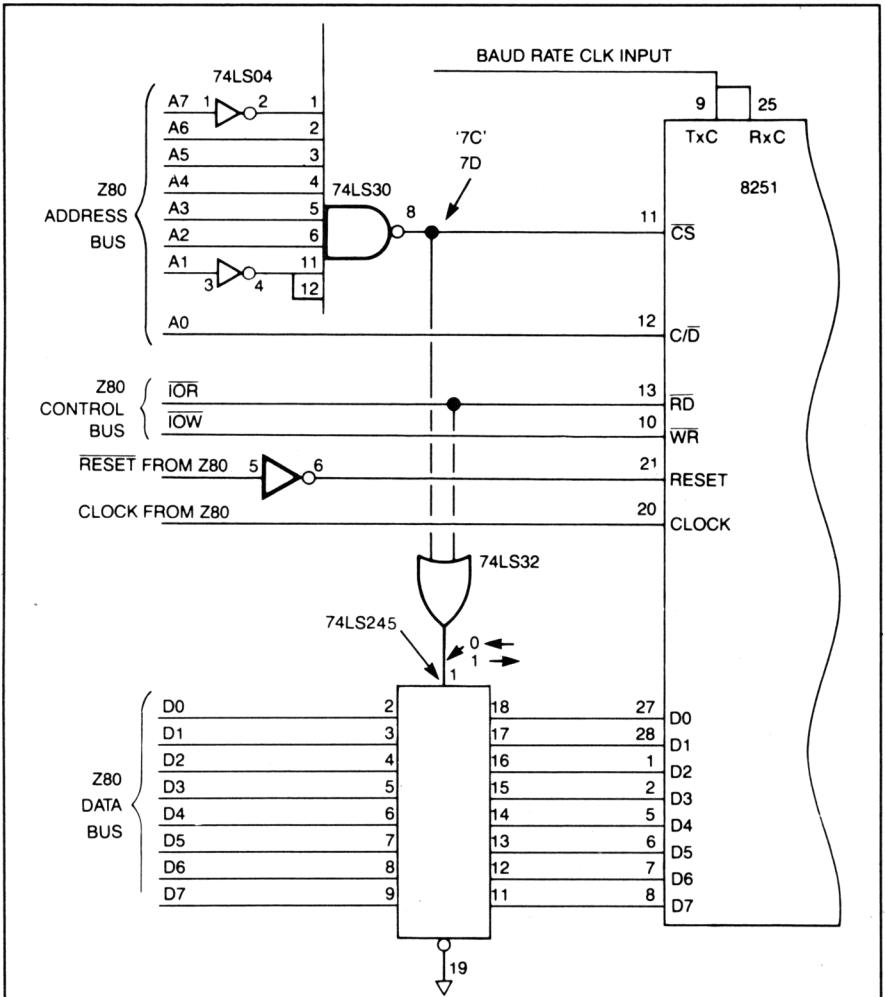


Figure 10.11: A complete schematic showing one way the 8251 can be connected to the Z80 system busses. Notice the use of the data buffer between the Z80 data bus and the 8251 data input pins.

The RS-232 uses a high voltage level pulse instead of the normal TTL levels. The voltage levels on the RS-232 transmission line are approximately ± 12 volts. Figure 10.12 shows a block diagram of a typical interface to the RS-232 bus. This figure shows that the TTL level from the 8251 must be converted to the RS-232 levels prior to transmission on the line.

At the receiving end of the RS-232 lines, the logical levels must be reconverted into a TTL level for input to the serial receiver. The conversion of TTL voltage levels to RS-232 voltage levels is so common that special integrated circuits have been designed to perform this function. An example of these special integrated circuits are the MC1488 and MC1489. These devices are shown in partial data sheet form in Figures 10.13 and 10.14.

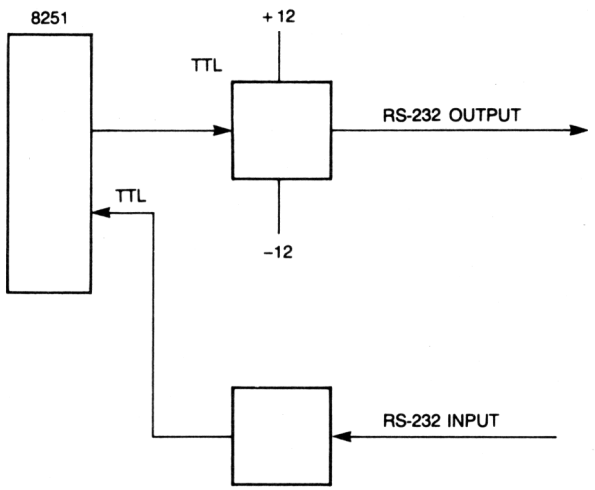


Figure 10.12: A block diagram showing how the TTL voltage levels of the 8251 USART are translated into RS-232 voltage levels. Also shown is the translator from RS-232 voltage levels into TTL levels for the USART.

MOTOROLA

MC1488

QUAD LINE DRIVER

The MC1488 is a monolithic quad line driver designed to interface data terminal equipment with data communications equipment in conformance with the specifications of EIA Standard No. RS-232C

Features

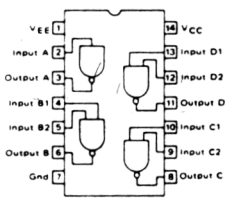
- Current Limited Output
  ±10 mA typ
- Power Off Source Impedance
  300 Ohms min
- Simple Slew Rate Control with External Capacitor
- Flexible Operating Supply Range
- Compatible with All Motorola MDTL and MTTL Logic Families

QUAD MDTL LINE DRIVER
RS-232C
SILICON MONOLITHIC
INTEGRATED CIRCUIT

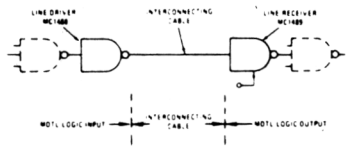
L SUFFIX
CERAMIC PACKAGE
CASE 632
TO-116

P SUFFIX
PLASTIC PACKAGE
CASE 646

PIN CONNECTIONS



TYPICAL APPLICATION



CIRCUIT SCHEMATIC
(1/4 OF CIRCUIT SHOWN)

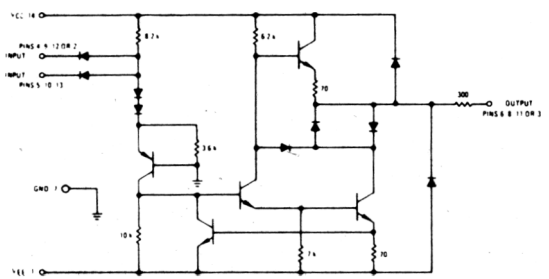


Figure 10.13: A partial data sheet of the Motorola MC1488 voltage translator from TTL to RS-232 levels.

MOTOROLA

MC1489L
MC1489AL

QUAD LINE RECEIVERS

The MC1489 monolithic quad line receivers are designed to interface data terminal equipment with data communications equipment in conformance with the specifications of EIA Standard No. RS-232C.

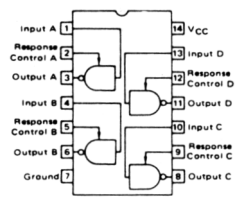
- Input Resistance – 3.0 k to 7.0 kilohms
- Input Signal Range – ±30 Volts
- Input Threshold Hysteresis Built In
- Response Control
  a) Logic Threshold Shifting
  b) Input Noise Filtering

QUAD MDTL LINE RECEIVERS RS-232C

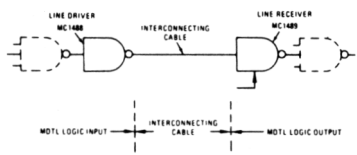
SILICON MONOLITHIC INTEGRATED CIRCUIT

L SUFFIX CERAMIC PACKAGE CASE 632 TO-116

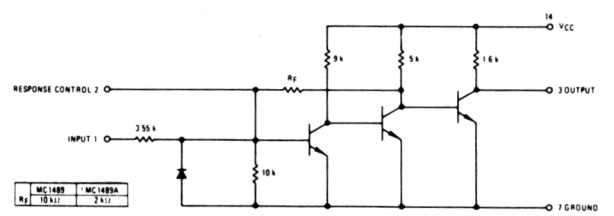
P SUFFIX PLASTIC PACKAGE CASE 646



TYPICAL APPLICATION



CIRCUIT SCHEMATIC (1/4 OF CIRCUIT SHOWN)



	MC1489	*MC1489A
R_F	10 kΩ	2 kΩ

Figure 10.14: A partial data sheet of the Motorola RS-232 to TTL voltage converter.

Figure 10.15 shows how the MC1488 and the MC1489 are used in a typical system performing RS-232 serial communication. Notice that to accomplish complete communication between one serial device and another requires only three wires, Tx (transmit), Rx (receive) and GND.

The preceding discussions of the system hardware have presented the main points of serial transmission that one should keep in mind when considering the use of serial communication. In a system that uses serial communication, the preceding hardware blocks must exist in some form. Once you know what should be there, then finding and analyzing the circuits is a much easier task.

10-12: Programming the 8251

Now that we have connected the 8251 to the Z80, let's focus on the software necessary to make the device operate in an asyncronous serial communication environment. In a general sense, the software informs the

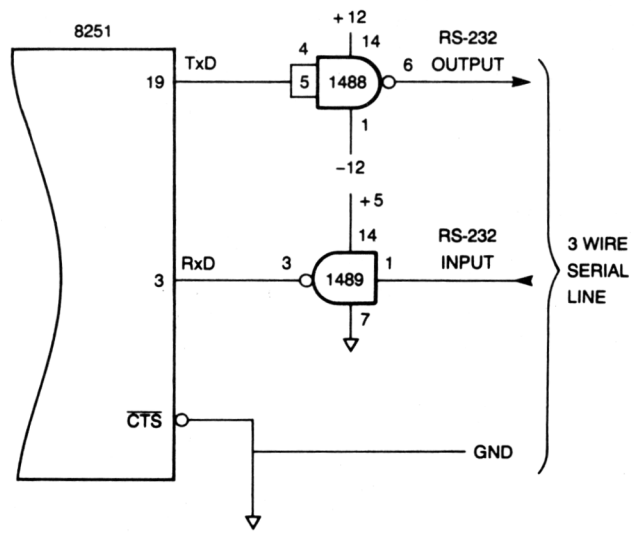


Figure 10.15: A schematic diagram showing how the MC1488 and MC1489 devices can be used in a serial communication scheme.

8251 of the parameters for the transmission. These include baud rate, number of data bits, number of stop bits, and parity information.

Once the device has been set up, the actual transmission of the data takes place. The microprocessor first determines if the transmit buffer is empty. If it is, then a character is loaded into the buffer and sent out via the serial line. This action takes place for all characters transmitted by the CPU.

For the CPU to receive a character, it must first examine a status register to determine if the receiver will read the character. It must then wait for the status to indicate when another character has been received.

The preceding description of the reception and transmission of serial characters is only a simplified version of what actually occurs. We will now explain how we can accomplish this operation using the 8251.

Prior to transmission, a set of control words must be loaded into the 8251. To accomplish this, the CPU must write to the command register immediately after the system reset has been asserted. The 8251 will electrically assume that the first I/O write to the command register is the mode instruction, and that a command word will follow immediately. There may be one or more bytes in the command word. The number of bytes depends on the byte sent for the mode instruction.

Figure 10.16 is a diagram showing the mode word bit definitions. Let's use this figure to set up a mode word for a typical transmission. In this transmission we will define our serial line to be the following:

1. The baud rate equals 2400. Our baud rate generator will have the same frequency as the transmitted data. Bits D1 and D0 of the mode word will be set to 0 and 1, respectively. It is possible to input a clock that is 16 or 64 times faster than the specified baud rate. In these cases, the 16X or the 64X baud rate factor is programmed into the 8251. In effect, the incoming Tx or Rx clock will be divided by the baud rate factor before going to the receiver or transmitter internal to the 8251.

2. There will be eight data bits transmitted for each character.

3. The parity bit will be set for even parity.

4. The transmitter will insert 2 stop bits per character.

With the preceding definition, the mode word written to the 8251 will be:

1 1 0 1 1 1 0 1

Let's break this word down and examine it closely:

Bits D7, D6 = 1 1. This sets up 2 stop bits.

Bits D5, D4 = 0 1. This sets even parity and enable parity.

Bits D3, D2 = 1 1. These bits set character length to 8 bits.

Bits D1, D0 = 0 1. The baud rate factor is now set to X1.

The Z80 instructions to write the mode word to the Z80 are:

```
LD A, 0DDH
OUT (7DH),A        OUTPUT TO THE COMMAND REGISTER
```

The next output word written to the 8251 is the command instruction. Bit definitions for this byte are shown in Figure 10.17. Based on the bit defini-

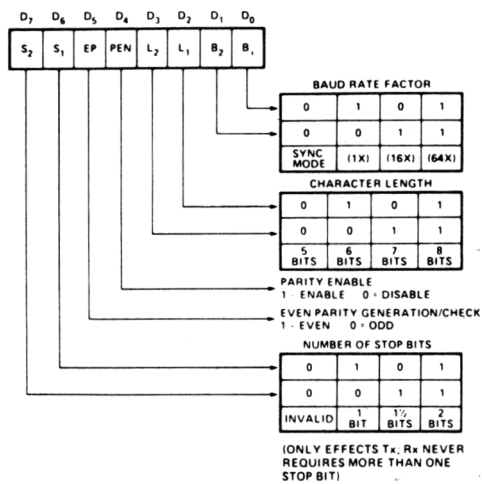


Figure 10.16: *A diagram of the mode word definitions for the 8251 device.*

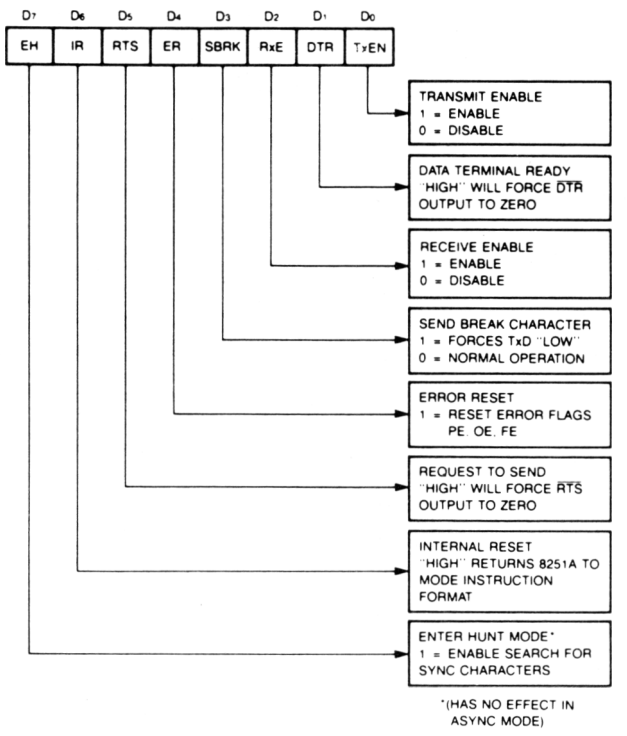


Figure 10.17: *Bit definition of the command instruction word format for the 8251 device.*

tions given in this figure the byte written will be:

0 0 0 1 0 1 0 1

This byte will enable the receiver and transmitter and reset the error registers. At this point the 8251 is ready for transmission or reception of characters.

A final register that we will now discuss is the *status register.* This register is read only. It is used so that the microprocessor can monitor the status of the serial transmission and check for any errors in transmission that the 8251 can detect. The bit definitions for the status register are shown in Figure 10.18. It is possible to read the status register with a simple input read operation, thus, insuring that the C/D input to the 8251 is a logical 1.

10-13: Framing Error

If you are not familiar with serial transmission, then an explanation of certain errors associated with this type of communication may be helpful. The first type of error we will examine is the *framing error.* A framing error occurs when a character is received in the serial input buffer and no stop bits are detected in the stream. This type of error usually results in faulty data being read.

10-14: Overrun Error

An *overrun error* occurs when a character is received in the serial input buffer and the previous character has not yet been read by the CPU. This results in the previous character being lost and the new character being written over it.

10-15: A Simple Application Program for the 8251

The following is a simple Z80 program for using the 8251. This program repeatedly sends the same character to the terminal. This program is not very useful except that it shows some important points about polled transmission. Figure 10.19 presents a flowchart for this program. The Z80 mnemonics to realize this flowchart are shown in the program in Figure 10.20.

10-16: An Expanded Application for the 8251

Let's now examine a Z80 program that allows the 8251 to echo a character sent to it by the terminal. This program shows how the 8251 operates in the

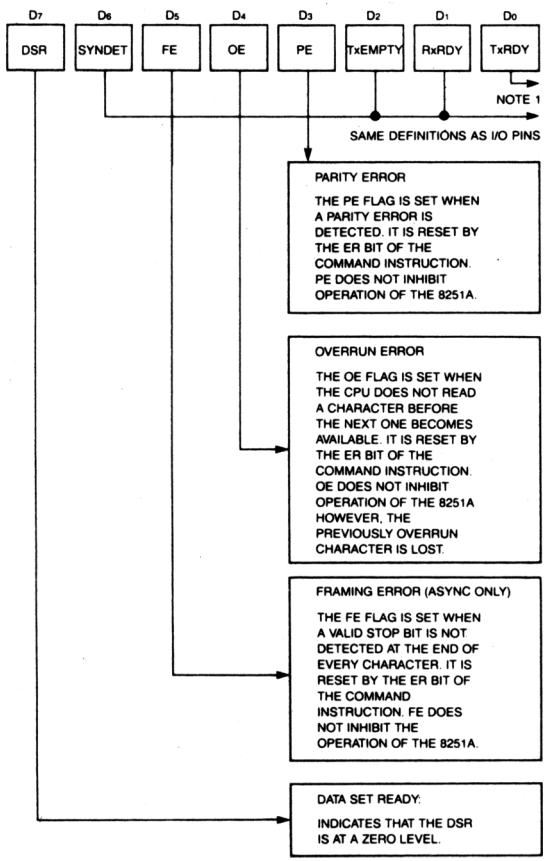


Figure 10.18: *Bit definitions of the status words for the 8251 device. The microprocessor can be informed electrically of the 8251 status by examining different bits of the status word. The status word is obtained from the 8251 by using an input instruction from the CPU.*

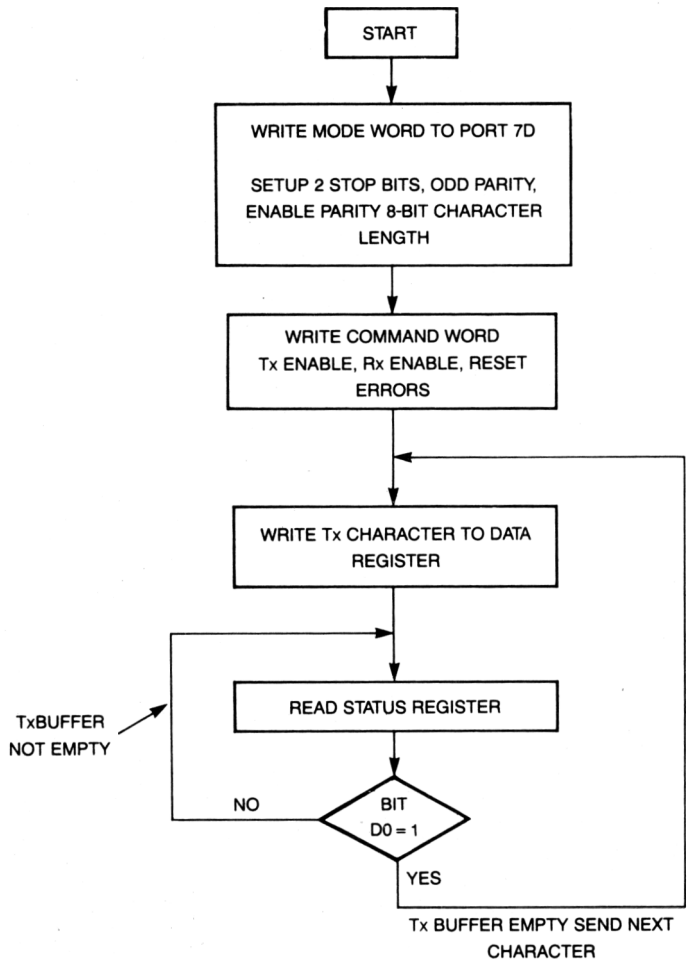


Figure 10.19: *A flowchart for the sequence of events required to transmit a single character repeatedly to another serial receiving device.*

transmission and reception of characters. Figure 10.21 shows a block diagram of the hardware interconnections. The flowchart for this program is shown in Figure 10.22.

The Z80 mnemonics for realizing the flowchart of Figure 10.22 are given in Figure 10.23.

```
1800   3EDD            LD A,0DDH        ;LOAD THE MODE WORD IN A REG
1802   D37D            OUT (7DH),A      ;OUTPUT TO THE 8251
;
; 2 STOP BITS, ODD PARITY, ENABLE PARITY, 8 BITS /CHAR
; X1 BAUD RATE MULTIPLIER
;
1804   3E15            LD A,15H         ;COMMAND WORD IN A REG
1806   D37D            OUT (7DH),A      ;OUTPUT TO 8251
;
; TX ENABLE, RX ENABLE, RESET ERRORS
;
1808   3E49     LOOP1  LD A,49H         ;ASCII "I"
180A   D37C            OUT (7CH),A      ;OUTPUT CHARACTER TO TX
180C   DB7D     LOOP2  IN A,(7DH)       ;READ STATUS REGISTER
180E   CB47            BIT 0,A          ;TEST BIT 0 = 1
1810   CA0C18          JP Z,LOOP2       ;BUFFER NOT EMPTY, KEEP POLLING
1813   C30818          JP LOOP1         ;BUFFER EMPTY, NEXT CHAR
;
```

Figure 10.20: The Z80 mnemonics to realize the flowchart given in Figure 10.19.

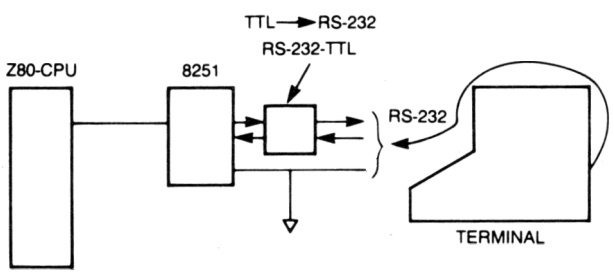


Figure 10.21: A block diagram showing how a USART fits into the communication scheme between a microprocessor system and a terminal.

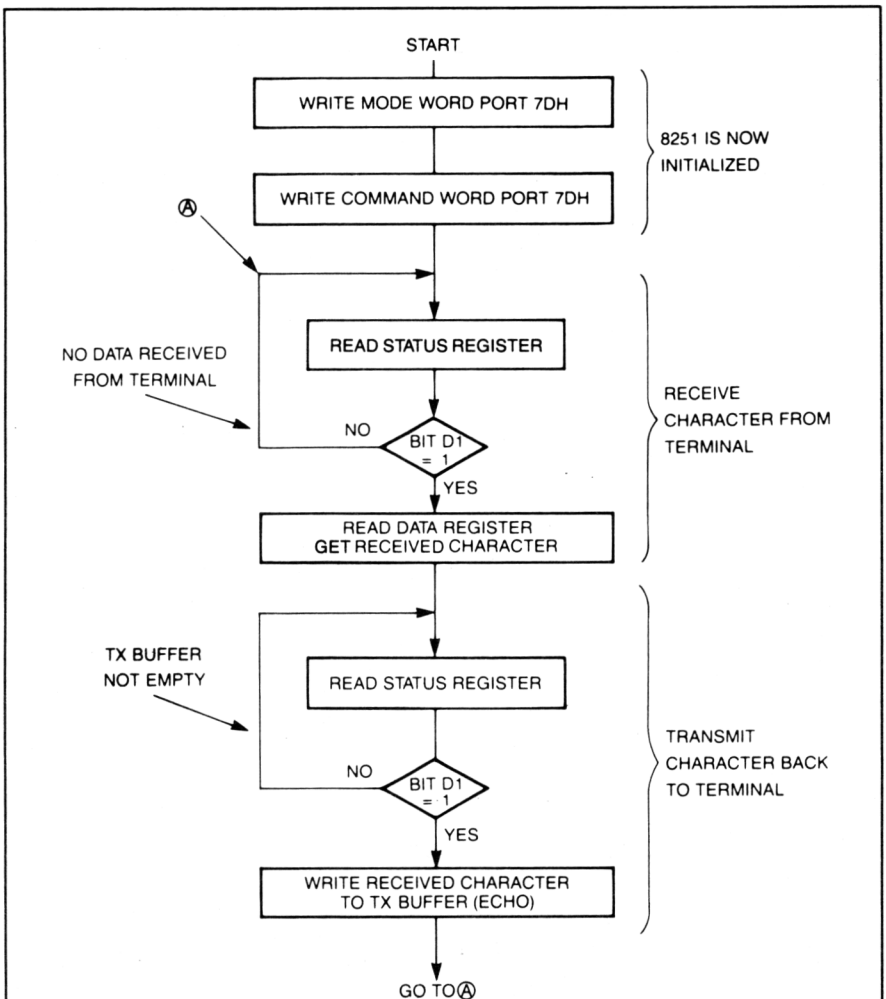


Figure 10.22: *A flowchart showing the sequence of events required to echo a character sent to the microprocessor system from the terminal and back to the terminal.*

```
;
1800   3EDD            LD A,0DDH        ;MODE WORD
1802   D37D            OUT (7DH),A      ;WRITE TO 8251
1804   3E15            LD A,15H         ;COMMAND WORD
1806   D37D            OUT (7DH),A      ;WRITE TO 8251
;
;  8251 IS NOW INITIALIZED
;
;  FIRST THE DEVICE WILL WAIT FOR A CHARACTER TO BE
;  SENT TO IT FROM THE TERMINAL
;
1808   DB7D     LOOP1  IN A,(7DH)       ;READ THE STATUS REGISTER
180A   CB4F             BIT 1,A          ;TEST BIT D1= 1
180C   CA0818           JP Z,LOOP1       ;NOT READY KEEP POLLING
;
;  WHEN WE REACH HERE A CHARACTER IS RECEIVED
;
180F   DB7C             IN A,(7CH)       ;READ THE CHARACTER
1811   47               LD B,A
;
;  WE WILL NOT ERROR CHECK THE DATA
;
;  NOW TO TRANSMIT THE DATA
;
1812   DB7D     LOOP2  IN A,(7DH)       ;READ STATUS REGISTER
1814   CB47            BIT 0,A          ;TEST D0 = 1
1816   CA1218          JP Z,LOOP2       ;XMIT NOT READY, KEEP POLLING
;
;  XMIT IS READY TO OUTPUT ANOTHER CHARACTER
;
1819   78               LD A,B
181A   D37C             OUT (7CH),A      ;CHARACTER TO 8251
181C   C30818          JP LOOP1         ;START OVER AGAIN
;
;  END OF ECHO ROUTINE
;
```

Figure 10.23: The Z80 mnemonics required to realize the flowchart given in Figure 10.22.

CHAPTER SUMMARY

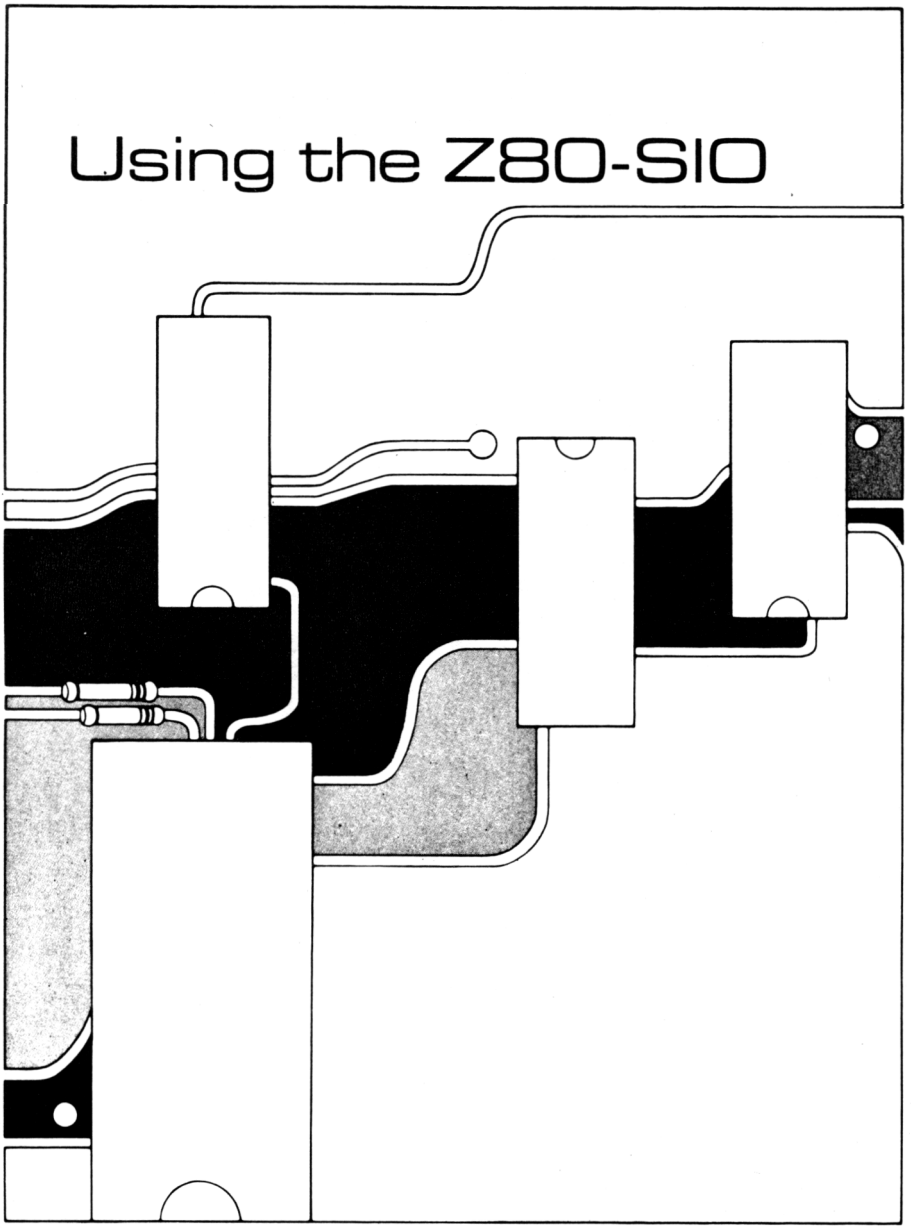
In this chapter we have examined the basics of serial communication. We have covered several important topics including baud rate, start bits, stop bits, and parity bits. We have also discussed the meaning of two basic errors sometimes encountered in serial communications: framing and overrun errors.

In the second part of this chapter we have examined a real serial communication LSI device: the 8251. We have selected this device because it is simple to

use and understand, and it is in wide general use in industry today. In addition, it solves the problem of serial communication simply and elegantly.

In Chapter 11, we will discuss the operation of another serial device: the Z80-SIO. This device is slightly more complex than the 8251. As you read this chapter, you will see that the information presented can also be applied directly to the Z80-SIO device.

Using the Z80-SIO



Chapter 11

INTRODUCTION

In this chapter we will discuss the serial input/output device, also known as the Z80-SIO. This device has many different operating modes and can be used in a variety of applications. In this chapter we will concentrate on the most popular use—as an asyncronous I/O device. Once you understand how the Z80-SIO is programmed and applied in this mode, you should be able to more easily understand how it is used in other operating modes. (*Note:* throughout this chapter we will assume that you are familiar with the basics of asyncronous communication. If you are not familiar with this topic, we recommend that you refer to Chapter 10, sections 1 through 6, for an explanation.)

11-1: Block Diagram of the Z80-SIO Device

Figure 11.1 shows a block diagram of the Z80-SIO. Let's examine this diagram and note some of the more important blocks. Let's start with the two blocks labeled channel A and channel B. They show that the Z80-SIO consists of two independent serial communications channels. Note that each channel has an associated control and status block. The channel and control blocks are connected to several input and output lines. We will discuss these lines as we proceed through this chapter.

Let's now examine the two blocks to the left of the channel blocks. These blocks, labeled channel A and channel B read/write registers, are the internal logic that allows the Z80-SIO to be programmed for the desired application.

All of the operating modes for the Z80-SIO are available under software control. Before the Z80-SIO is used, programming words must be sent to these blocks to "set up" the chip. We shall discuss the programming of these blocks in detail later on in this chapter.

Another block in Figure 11.1 is the *interrupt control logic*. The Z80-SIO has a very powerful interrupt architecture. This logic block allows the SIO to interface directly to the Z80 interrupt structure, as well as with other peripheral devices.

The block labeled *internal control logic* is used for reading and writing data on the internal data bus. This logic block correctly times all of the internal data transfers on the device.

Finally, the main data input block in Figure 11.1, labeled *CPU I/O bus*, is used for buffering, and for providing an electrical interface between the Z80 CPU and the SIO. Later on in this chapter we will learn how the SIO is actually connected to the Z80 microprocessor.

11-2: SIO Pin Definitions

Figure 11.2 shows the pinout of the Z80-SIO device. Let's now discuss each pin.

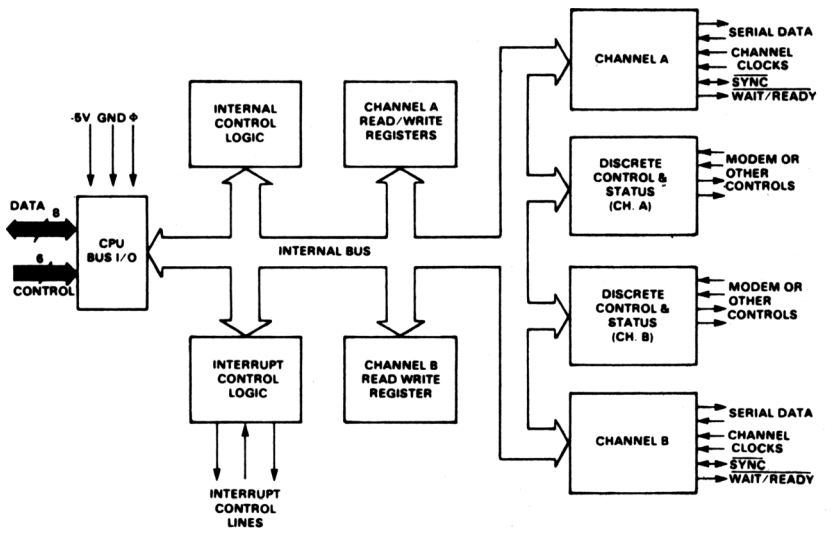


Figure 11.1: A block diagram of the Z80-SIO device.

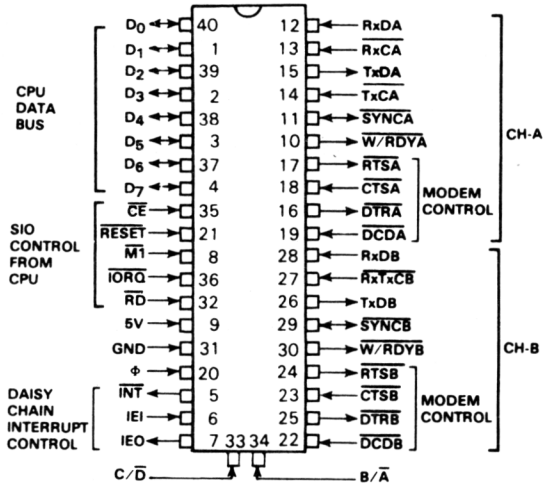


Figure 11.2: Pinout and device pin name for the Z80-SIO.

D7–D0 These are the data input and output lines that connect logically to the microprocessor system data bus. All information that passes between the SIO and the Z80 goes via these eight data lines. D7 is the MSB (most-significant bit).

B/$\overline{A}$ *(Channel B or A Select)* Whenever this line is a logical 1, channel B is selected for communication, and whenever it is a logical 0, channel A is selected. This input line is usually connected to the A0 system address output line from the Z80 CPU.

C/$\overline{D}$ *(Control or Data Select Input Line)* Whenever this input line is a logical 1, the Z80 is communicating with the control registers of channel A or B. A logical 0 on this input line electrically informs the SIO that data on D0–D7 is to be used as data, not device control information. This input line is usually connected to the system address line A1 from the Z80 CPU.

Whenever the address lines A0 and A1 are connected to the SIO pins B/$\overline{A}$ and C/$\overline{D}$, respectively, there are four unique ports associated with the physical device. These are channels A and B command or data. (*Note:* we will examine the topic of addressing further when we connect the SIO to the Z80 in the next section of this chapter.)

$\overline{CE}$ This is the chip enable input which is active logical 0. This input must be active for the Z80 to electrically communicate with any of the internal read or write registers. The input is usually decoded from the upper six bits of the lower address byte. These would be address lines A7-A2. The lower two address lines will access the unique port of the device.

CLOCK This input is the standard clock that is also input to the Z80 CPU. The clock is input at the same frequency as the CPU and will syncronize the internal communications of the SIO. The clock input has the same DC characteristics as it has for the Z80—in other words, a logical 1 must be equal to Vcc − .6 volts. Note that this is not a standard TTL logical 1 input voltage.

$\overline{M1}$ *(Machine Cycle One)* This input is connected to the $\overline{M1}$ output line from the Z80 CPU. Whenever the $\overline{M1}$ and $\overline{RD}$ inputs to the SIO are both logical 0's, the Z80 is electrically fetching an opcode from system memory. The SIO is designed to recognize the opcode for the RETI instruction. Whenever $\overline{M1}$ and $\overline{IORQ}$ inputs to the SIO are both logical 0's, the Z80 is generating an interrupt acknowledge. The SIO will automatically respond to this electrical condition, if enabled.

$\overline{IORQ}$ *(Input and Output Request)* This input line is connected to the $\overline{IORQ}$ output line from the Z80 microprocessor. It is used to electrically inform the SIO that the CPU is performing an input or output operation. It also informs the SIO when the Z80 is electrically acknowledging an interrupt request. See the description of $\overline{M1}$, for the logical conditions of this operation.

$\overline{RD}$ *(Read)* This input line is connected to the $\overline{RD}$ output line from the Z80 CPU. It is used to inform the SIO whenever the Z80 is reading data from memory or from I/O. Whenever $\overline{RD}$ and $\overline{M1}$ are a logical 0, the CPU is fetching an opcode from memory. If $\overline{RD}$, $\overline{IORQ}$ and $\overline{CE}$ are logical 0, the CPU is reading data from the SIO, using an input operation. Data will be sent to the CPU from the port selected by the C/$\overline{D}$ and B/$\overline{A}$ inputs.

Whenever the $\overline{RD}$ input is a logical 1, and $\overline{IORQ}$ and $\overline{CE}$ are a logical 0, the CPU is writing data to the SIO, using an output operation. There is no $\overline{WR}$ input for the SIO. Data will be written to the port selected by the C/$\overline{D}$ and B/$\overline{A}$ inputs.

$\overline{\text{RESET}}$ A logical 0 on this input will disable both channel A and channel B receivers and transmitters. The transmitter outputs of both channels go to the "marking" state. All modem controls are set to a logical 1. All interrupts are disabled. The SIO must be reset with a software command and all internal registers reprogrammed after an external hardware reset.

IEI *(Interrupt Enable Input)* Whenever this input line is active logical 1, the hardware interrupt system of the SIO is enabled. The IEI line will usually connect to the IEO line from another peripheral device for use in an interrupt priority daisy chain. To enable the interrupts on the SIO, input must be a logical 1. For the SIO to respond to interrupts, the software must also enable the device. We will discuss interrupts in detail in a later section of this chapter.

IEO *(Interrupt Enable Output)* This output line is active logical 1. It is used to electrically inform the peripheral devices connected in the interrupt daisy chain that no lower priority devices may cause an interrupt. If the SIO is not used in the daisy chain system environment, this output line may be electrically ignored.

$\overline{\text{INT}}$ *(Interrupt Request Output)* This output is open drain. This allows it to connect directly to the $\overline{\text{INT}}$ input pin of the Z80. Whenever the SIO is requesting an interrupt, the output will be a logical 0. Whenever it is not requesting an interrupt, the output will be open. A pull-up resistor is usually connected to the $\overline{\text{INT}}$ input pin of the CPU to pull up all open drain devices to a logical 1, whenever they are not active.

$\overline{\text{W/RDYA}}$, $\overline{\text{W/RDYB}}$ These are the WAIT/RDY lines that are associated with channels A and B. These outputs are defined under program control. If the outputs are defined as a wait function, they are open drain. If they are defined as a ready function, they are driven active high and low. A ready function is used in conjunction with DMA controllers to indicate whenever the SIO has data to send or whenever it is ready to accept data. The wait function is used whenever the Z80 is communicating with the SIO to indicate when the SIO has data ready for the CPU or when it has a space for the CPU to write data.

$\overline{\text{CTSA}}$, $\overline{\text{CTSB}}$ *(Clear to Send A,B)* These input signals are active low. They are enabled under software control as *auto enable.* When in this mode, they allow the external hardware to start the transmit operation. If these input signals are not programmed in the auto enable mode, they can be used as general-purpose inputs to the SIO.

DCDA, DCDB *(Data Carrier Detect A,B)* These input signals are active low. When active, they enable the receiver A or B.

$\overline{\text{RTSA}}$, $\overline{\text{RTSB}}$ *(Request to Send A,B)* These output signals are active low. The RTS bit can be set under program control. The output goes to a logical 1 after the transmitter is empty.

$\overline{\text{DTRA}}$, $\overline{\text{DTRB}}$ *(Data Terminal Ready A,B)* These output signals will follow the logical state of the programmed bit.

(Note: The preceding four signals, $\overline{\text{CTS}}$, $\overline{\text{DCD}}$, $\overline{\text{RTS}}$ and $\overline{\text{DTR}}$, can be used for modem control. A block diagram showing the relationship of these signals to a modem is shown in Figure 11.3.)

RxDA, RxDB *(Receive Data Inputs)* These inputs are non-inverted as the data is received.

TxDA, TxDB *(Transmit Data Outputs)* These outputs are non-inverted as the data is transmitted.

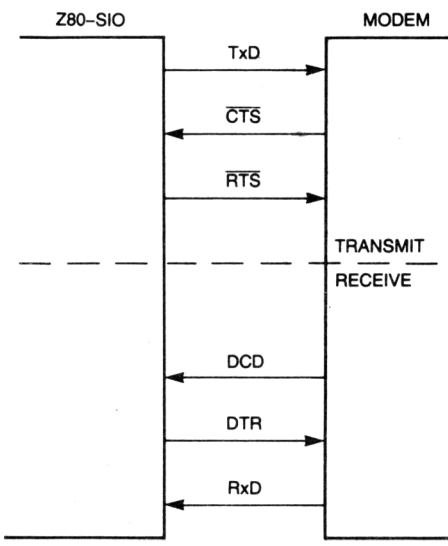


Figure 11.3: A block diagram showing the relevant signals used for modem control with the Z80-SIO.

$\overline{\text{RxCA}}$, $\overline{\text{RxCB}}$ These are the receiver clock input pins to the device. The clocks can be programmed to 1, 16, 32, or 64 times the receiver data rate.

$\overline{\text{TxCA}}$, $\overline{\text{TxCB}}$ These two inputs are the transmitter clock input pins. The clocks may be 1, 16, 32, or 64 times the transmit data rate for asyncronous operation. The multiplier for the receiver and the transmitter channel must be the same. However, the receiver and transmit data rates can be different.

For example, the transmit data rate can be 2400 baud and the receiver data rate can be 1200 baud. The multiplier for both blocks must be equal. If the multiplier is set to 16, then the clock input frequency for the transmitter would be 2400 $\times$ 16 hertz. The clock frequency for the receiver would be 1200 $\times$ 16 hertz. Both of these frequencies are different, but the rate multipliers are the same.

11-3: Connecting the SIO to the Z80 Busses

Figure 11.4 shows the typical connection of the SIO device to the Z80 microprocessor. In this diagram it is assumed that the DC loading on the data bus does not require data buffers; therefore, there are no data buffers shown.

The $\overline{\text{CE}}$ input pin 35 is decoded from the address lines A7–A2. A 74LS138 is used for the decoding. When the adddress lines A7–A2 equals 100011XX, the chip will be enabled. The XX indicates that A1 and A0 are "don't care" situations for the $\overline{\text{CE}}$ to become active. A1 and A0 are connected to the C/$\overline{\text{D}}$ and the B/$\overline{\text{A}}$ inputs, respectively.

Based on these electrical connections and decodings, the port addresses for the SIO are 8C, 8D, 8E, and 8F. These ports are defined as follows:

8C = A data
8D = B data
8E = A control
8F = B control

The $\overline{\text{INT}}$ control of the SIO is connected to the $\overline{\text{INT}}$ input pin of the Z80. IEI is pulled up to +5 volts via a 10K resistor. This is done to enable the interrupt hardware of the device. IEO is not used in this application. Figure 11.5 shows how the IEI and IEO would be connected if the SIO were used in a priority interrupt scheme with other peripheral devices.

11-4: Connection of the SIO to the Serial Transmission Lines

For this connection, we will assume that the SIO will use both channel A and channel B in the serial asynchronous communication scheme. The serial link will be via the RS-232. Figure 11.7 shows the connections required to use

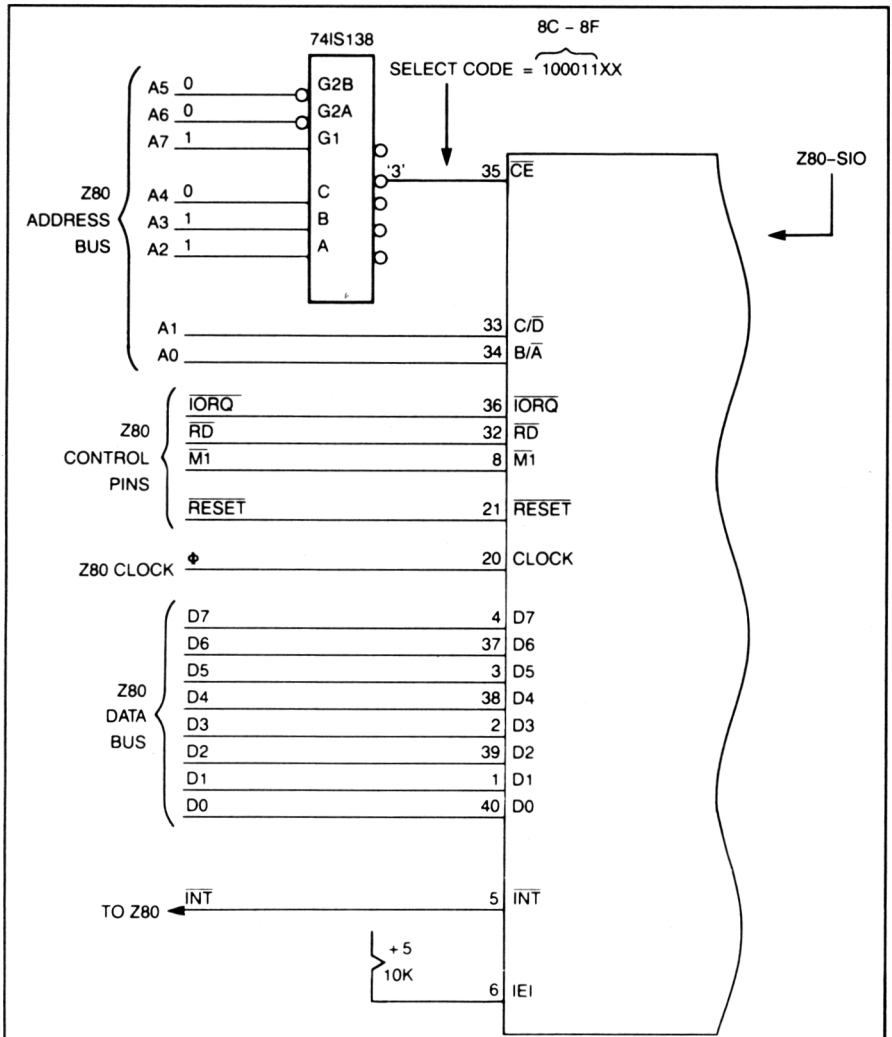


Figure 11.4: A schematic diagram showing the complete connection between the Z80 CPU and the Z80-SIO. No data bus buffering is used in this application.

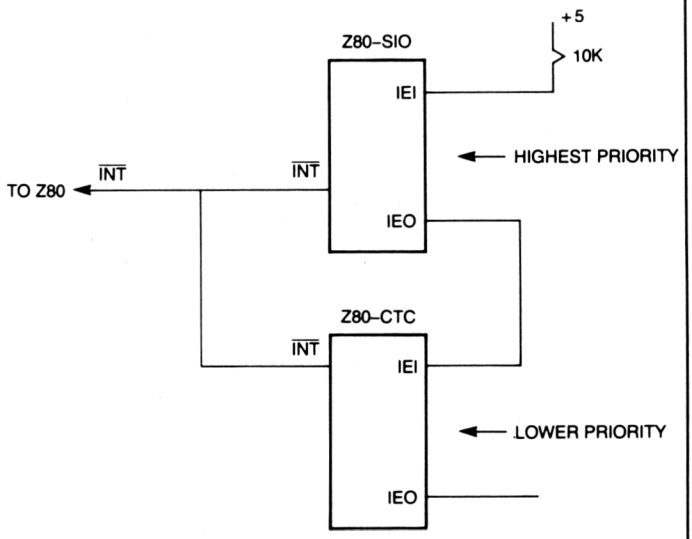


Figure 11.5: A block diagram showing how the daisy chain priority scheme is used with the Z80-SIO and other Z80 peripheral devices.

the SIO in this mode. Notice in Figure 11.6 that only the TxD, RxD, $\overline{\text{TxC}}$ and $\overline{\text{RxC}}$ for both channels need to be connected.

11-5: The SIO Registers

Once the hardware is connected as described in the previous sections, the task becomes one of programming the SIO to communicate in the manner required for your system application. In the following sections of this chapter we will make use of the SIO registers and show sample programs that can be used to initialize the SIO. These programs should help you learn more about programming the device.

The Z80-SIO contains eight internal write registers for channel B, i.e., WR0–WR7, and seven write registers for channel A, i.e., WR0–WR1 and WR3–WR7 (with no WR2). Write register WR2 contains the interrupt vector for both channels (A and B). The most significant bit of each register is D7.

Two bytes are required to write data to any register except WR0. The first byte is written to WR0. This byte is internally decoded and it points to the register to receive the next byte. This is necessary because there are no address lines to point to the internal registers. All of the basic commands can be written to WR0 in a single byte. You will understand this concept better as we proceed with our examples.

Channel B has a set of three read registers (labeled RR0 – RR2) that contain the status of the SIO. Channel A does not have the RR1 register as you will learn as we proceed in the chapter.

With this brief introduction, you should now turn to Appendix A and examine each of the write and read registers in the SIO. Appendix A presents information excerpted directly from the MOSTEK data manual, and

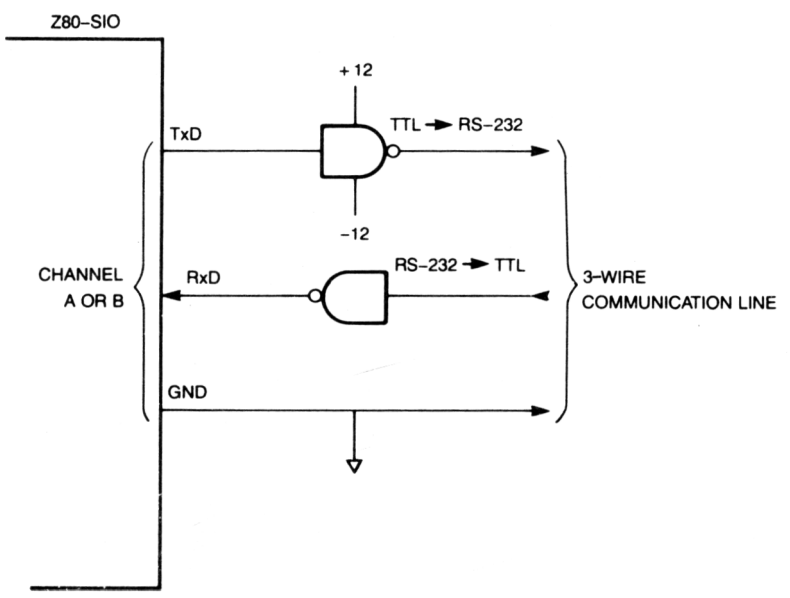


Figure 11.6: A schematic diagram indicating how voltage translators are used to allow translation between the TTL levels of the SIO and the RS-232 bus.

describes in detail each internal register of the SIO. This information is important in learning how to program the SIO device. Once you are familiar with this material, read on as we will go through a typical programming sequence, so that you can see exactly how this device can be used.

11-6: General Sequence of Initialization for the SIO

We will now present the software required to operate the SIO in a polled system environment. Our discussion will focus on several different aspects of SIO operation. In general, you will learn how to program the device and how to interface it to the SIO for serial polled operation.

Let's examine some general features of the SIO. We will begin by examining some general points regarding the registers (explained in detail in Appendix A). First of all, the SIO can receive or transmit 5, 6, 7, or 8 bits per character. The number of bits is under program control and can be different for the transmitter and receiver.

When transmitting asyncronously, the SIO will automatically insert a start bit —a logical 0—at the start of the character. It will also insert 1, $1\frac{1}{2}$, or 2 stop bits at the end of each transmitted character. The number of stop bits is under the control of the program. The stop bit is a logical 1, which is the logical level of the "marking" state for the SIO.

Prior to the insertion of the stop bits in the character, a parity bit may be inserted. (*Note:* The concept of parity was explained in Chapter 10, section 10-3.) If parity is chosen and programmed, the SIO will automatically operate with even or odd parity, whichever is desired. If parity is not chosen, then the SIO will *not* insert the extra bit to force even or odd parity.

When data is read by the CPU from the SIO, eight bits will be input. Not all of these bits, however, will have to be the received character. For example, if the received character has only six bits, then the first six bits, D0 – D5, will be the character, with D0 being the LSB (least-significant bit). The remaining bits of the serial bit stream will be stop bits and parity, if included. A stop bit will always be a logical 1. If there are some unused bits, they will be set to logical 1's or the marking level.

For example, suppose the received character is specified as 5 bits, no parity, and 1 stop bit. This means that the received character is only 5 bits + 1 stop bit. The remaining 2 bits are filled up by the SIO marking level or by logical 1's.

There are four serial communication clock inputs and one CLK input to the SIO. The single CLK input is used to time the internal data transfers and is, thus, not involved in the serial transmission. The four clocks that are involved in this communication are RxA, RxB, TxA, and TxB. The input clock frequency on these inputs may be 1, 16, 32, or 64 times the serial transmission or reception data rate.

Now that we have a basic familiarity with the SIO, let's proceed with our example of polled asyncronous communication. The following is a sequence (of register initialization) that should be followed when using the SIO in the asyncronous mode of operation:

1. *Write WR0.* This action resets the SIO. More than one byte may need to be written for the complete reset of the device. These bytes may be needed to send some of the special command bytes described in the WR0 section.

2. *Write WR1.* This action sets up the receive and transmit clock divider, stop bits and parity.

3. *Write WR3.* This action sets up the number of receive bits, auto enable, and receiver enable.

4. *Write WR5.* This action specifies the number of Tx bits, and turns on the Tx enable bit.

5. *Write WR2.* This is for channel B only. The interrupt vector is specified. A write to WR2 need only be done for an interrupt driven system.

6. *Write WR1.* This action specifies the interrupt system for the SIO. It also enables or disables the SIO interrupt structure.

Registers WR6 and WR7 are not used in the asyncronous communication mode. They are needed for syncronous communication.

To show how the preceding initialization would operate in a typical system that has an SIO, let's present an example. We will set up the device on channel A of the SIO. The serial communication parameters will be:

- x64 clock rate multiplier
- 2 stop bits
- 8 data characters
- interrupts disabled
- no parity

A Z80 program that will initialize an SIO with the preceding characteristics is shown in Figure 11.7. In this figure we will assume that the SIO is mapped into the physical I/O space locations 8CH, 8DH, 8EH, and 8FH. Figure 11.3 shows the electrical decodings of these ports.

11-7: Receiving the Serial Data

Now that the SIO is initialized, we will begin receiving data from the SIO with the CPU. In the polled mode of operation it is assumed that the Z80 will take the characters from the SIO as fast as they can be received from the serial input line. If this were not the case then we would need some type of handshake on the serial line to insure that characters are not received faster

```
;
;  THIS IS A Z80 PROGRAM FOR INITIALIZATION OF THE SIO
;  X64 CLOCK, 2 STOP BITS, 8 RX BITS, 8 TX BITS, NO INT.
;
;
;
;  SET UP I/O PORT EQUATES
;
008C              ADATA   EQU  8CH   ;SIO CHANNEL A DATA PORT
008D              BDATA   EQU  8DH   ;SIO CHANNEL B DATA PORT
008E              ACON    EQU  8EH   ;SIO CHANNEL A CONTROL PORT
008F              BCON    EQU  8FH   ;SIO CHANNEL B CONTROL PORT
;
;
1800   0E8E               LD C,ACON  ;LOAD I/O ADDRESS INTO C REG
1802   3E18               LD A,18H
1804   ED79               OUT (C),A  ;WR0, CHANNEL RESET
1806   3E04               LD A,04H   ;LOAD A WITH WR4 POINTER
1808   ED79               OUT (C),A  ;POINT TO WR4
180A   3ECC               LD A,0CCH
180C   ED79               OUT (C),A  ;WR4, X64 CLOCK,2 STOP BITS,NO PARI
180E   3E03               LD A,03H   ;LOAD A WITH WR3 POINTER
1810   ED79               OUT (C),A  ;POINT TO WR3
1812   3EC1               LD A,0C1H  ;8 REC BITS, REC ENABLE
1814   ED79               OUT (C),A
1816   3E05               LD A,05H   ;LOAD A WITH WR5 POINTER
1818   ED79               OUT (C),A  ;POINT TO WR5
181A   3E68               LD A,68H
181C   ED79               OUT (C),A  ;8 TX BITS, TX ENABLE
181E   3E01               LD A,01H   ;LOAD A WITH WR1 POINTER
1820   ED79               OUT (C),A  ;POINT TO WR1
1822   3E00               LD A,00H
1824   ED79               OUT (C),A  ;DISABLE ALL INTERRUPTS
;
;
;  AFTER THIS CODE IS EXECUTED THE SIO WILL BE INITIALIZED
;  AND READY FOR COMMUNICATION
;
                          END
```

Figure 11.7: A Z80 program to initialize the Z80-SIO for serial communication.

than they can be processed by the CPU. If this occurs, the receiver is said to have an overrun error.

The Z80-SIO has a built-in, three deep, FIFO (first-in, first-out) memory. Thus, three words can be received before an overrun problem occurs. The received data will be input to the SIO eight bits at a time, thus allowing the CPU the time to input the character and store it in memory before the next character is fully received by the SIO.

In the receive routine, we will check for overrun errors after each character. Note that if an error occurs, the CPU must reset the error flag on the SIO before it can check the next character. The error flags remain set until they are reset. A flowchart for receiving the characters is shown in Figure 11.8.

As we can see in Figure 11.8, the first operation to be performed is the read from the status register RR0. This is done to see if the receiver buffer is full. If there is even one character in the receiver FIFO, then bit D0 of RR0 will be a logical 1. If bit D0 is a logical 0, then the CPU will continually loop, waiting for a character, or it will process other information and come back to poll this register later. In Figure 11.8, the CPU will continually loop, polling the register, until a character has been received.

After a character has been received by the SIO, and bit D0 of RR0 is a logical 1, the character is read. Next RR1 is read to see if there are any errors associated with the character. Bits D6, D5 and D4 of RR1 will indicate if any errors exist. D6 indicates a framing error, D5 an overrun error, and D4 a parity error. If we get any errors, then the SIO must have the errors reset by writing to WR0.

If errors are detected, the software must report the type of error to the system console (or it will need to perform whatever action your particular application calls for).

A program to realize the flowchart for character reception in the polled mode is shown in Figure 11.9.

11-8: Transmitting a Character in a Polled Mode

In the previous section we presented a program for receiving a character in a polled mode of operation. We will now present a program for transmitting a character in the polled mode. Figure 11.10 shows a flowchart of this operation.

The first operation shown in Figure 11.10 is a read of RR0. Recall that RR0 can be read without first writing a word to the WR0 register for pointing. Bit D2 of RR0 is then tested.

If bit D2 is a logical 1, the transmit buffer is empty. If bit D2 is a logical 0, the transmit buffer is full. The flowchart must loop while waiting for the Tx buffer to become empty. When the buffer is empty, the character for transmission is written into the channel data register and the SIO will automatically start the transmission sequence. Figure 11.11 shows a Z80 program to realize the flowchart of Figure 11.10.

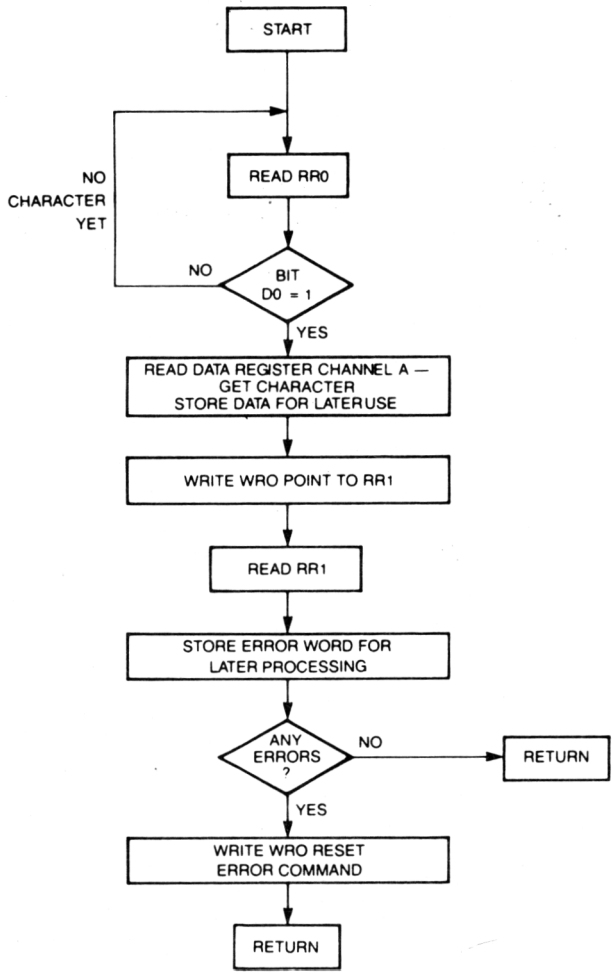


Figure 11.8: *A flowchart showing the sequence of events required to read a character from the Z80-SIO.*

```
;
;  THIS ROUTINE WILL RECEIVE A SINGLE CHARACTER
;  IN THE POLLED MODE OF OPERATION. THE
;  ROUTINE WILL RETURN WHEN A CHARACTER HAS
;  BEEN RECEIVED. THE CHARACTER IS RETURNED IN
;  MEMORY LOCATION "CHIN" AND ERRORS ARE RETURNED
;  IN MEMORY LOCATION "CHER". IF "CHER" BITS
;  D6, D5, D4 = 0 THEN THERE WAS NO ERROR
;
;  WE ASSUME THE SIO IS MAPPED TO PORTS 8C-8F AND
;  HAS BEEN INITIALIZED
;
;
;
008C            ADATA   EQU  8CH        ;SIO CHANNEL A DATA PORT
008E            ACON    EQU  8EH        ;SIO CHANNEL A CONTROL PORT
5900            CHIN    EQU  5900H      ;MEMORY LOC FOR CHARACTER INPUT
5901            CHER    EQU  CHIN+1     ;MEMORY LOC FOR RECEIVED ERRORS
;
;
2400   DB8E     LOOP    IN A,(ACON)     ;READ RR0
2402   E601             AND 01H         ;CHECK BIT D0 FOR READY
2404   CA0024           JP Z, LOOP      ;NOT READY CONTINUE TO POLL
;
;  NOW TO READ THE CHARACTER IN CHANNEL A
;
2407   DB8C             IN A,(ADATA)    ;GET THE CHARACTER
2409   320159           LD (CHER),A     ;STORE THE CHARACTER
;
;  NOW WE CHECK FOR RECEIVE ERRORS
;
240C   3E01             LD A,01H
240E   D38E             OUT (ACON),A    ;SET WR0 POINT TO RR1
2410   DB8E             IN A,(ACON)     ;GET RR1 ERROR REGISTER
2412   320159           LD (CHER),A     ;STORE ERROR
2415   E670             AND 70H         ;MASK OFF ALL BUT D6,D5,D4
2417   C8               RET Z           ;IF ZERO THEN NO ERROR
;
;  IF NO ERRORS THEN RETURN FROM ROUTINE
;  IF THERE ARE ERRORS THEN RESET BEFORE RETURN
;
2418   3E30             LD A,30H        ;SET ERROR RESET COMMAND
241A   D38E             OUT (ACON),A    ;WR0 ERROR RESET COMMAND
241C   C9               RET
;
;
;
                        END
```

Figure 11.9: A Z80 program for realizing the flowchart of Figure 11.8.

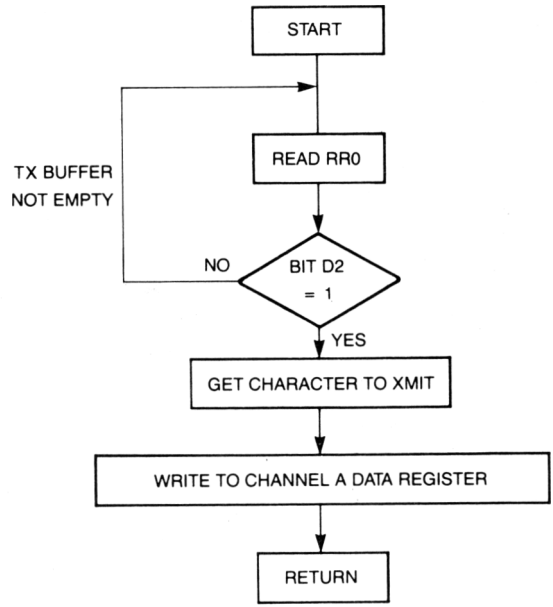


Figure 11.10: *A flowchart showing the sequence of events required to transmit a character using the Z80-SIO.*

```
;
;   THIS ROUTINE WILL BE USED TO TRANSMIT
;   A SINGLE CHARACTER USING THE SIO IN A
;   POLLED MODE OF OPERATION
;
;   THE CHARACTER TO BE TX WILL BE STORED
;   IN LOCATION "CHTX". WE ASSUME THE SIO
;   HAS BEEN INITIALIZED, AND IS MAPPED INTO
;   I/O SPACE 8C-8F.
;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Figure 11.11: *The Z80 mnemonics for realizing the flowchart in Figure 11.10. (continues)*

```
;
;
008C              ADATA   EQU 8CH         ;SIO CHANNEL A DATA PORT
008E              ACON    EQU 8EH         ;SIO CHANNEL A CONTROL PORT
5902              CHTX    EQU 5902H       ;MEM LOC FOR TX CHARACTER
;
;
2500   DB8E       LOOP1   IN A,(ACON)     ;READ RR0 CHECK FOR TX EMPTY
2502   CB57               BIT 2,A         ;TEST BIT D2
2504   CA0025             JP Z,LOOP1      ;D2=1 IF EMPTY, 0 IF NOT EMPTY
;
;  THE TX BUFFER IS NOW EMPTY GET CHARACTER
;  TO TRANSMIT
;
2507   3A0259             LD A,(CHTX)     ;LOAD CHARACTER INTO A REG
250A   D38C               OUT (ADATA),A ;OUTPUT CHARACTER TO TX
;
;  CHARACTER HAS NOW BEEN SENT TO TRANSMIT
;  RETURN FROM THE ROUTINE
;
250C   C9                 RET
                          END
```

Figure 11.11: The Z80 mnemonics for realizing the flowchart in Figure 11.10.

11-9: Interrupts for the SIO

Now that we know how characters are transmitted and received in a polled mode of operation, let's learn how the SIO can be used in an interrupt driven environment. We will begin with a review of the SIO interrupt structure. We will show examples of SIO initialization, and then transmit and receive routines in the interrupt mode. By the end of this discussion you should have a good idea of how the SIO can be used in an interrupt-driven environment.

In the Z80-SIO register, WR2 is used to store the interrupt vector. This vector is placed on the data bus in response to an interrupt acknowledge from the Z80. We will assume that the Z80 is operating in interrupt mode 2. Only channel B of the SIO has the interrupt vector register. However, this does not exclude channel A from being used in an interrupt mode.

There are eight different interrupt vectors that the SIO can automatically place on the system data bus. These eight vectors are formed from the single vector written in the initialization phase of the SIO. To allow these eight vectors to be formed, bit D2 of register WR1, called the *status affects vector,* must be set to a logical 1.

If this bit is set and the interrupts are enabled, then the type of interrupt will modify certain bits of the interrupt vector in WR2. The bits of the vector

that are modified are D3, D2, and D1. With three bits being modified, there is a possibility for eight different vectors.

The eight different vectors are divided into two groups of four. Each group is used by a different channel. Therefore, with a vector, it is possible to get the interrupt information for channel A, as well as for channel B. An appropriate question at this time is "What if both channels want to interrupt at the same time?" The interrupts are internally prioritized on the SIO. The following is a list of the possible vector bits (D3, D2, D1), listed in priority from highest to lowest.

D3	D2	D1	TYPE OF INTERRUPT	CHANNEL USED
1	1	1	special receive condition	channel A
1	1	0	received character	channel A
1	0	0	transmit buffer empty	channel A
1	0	1	external/status transition	channel A
0	1	1	special receive condition	channel B
0	1	0	received character	channel B
0	0	0	transmit buffer empty	channel B
0	0	1	external/status transition	channel B

As an example, let's suppose we have programmed the interrupt vector to 10010000. Further, let's suppose that the interrupts were enabled and an interrupt occurred. Channel A received a character. The resulting interrupt vector that would be placed on the data bus during an interrupt acknowledge sequence would be 10011100. Notice that bits D3, D2 and D1 were modified from the original vector to show the status of the interrupt (i.e., to show why the interrupt occurred).

Let us now take a closer look at what the interrupt conditions means. We will start with the special receive condition. They include parity errors, receiver overruns, framing errors, or end-of-frame (SDLC). In the previous discussion, we learned that we can test the results of these types of errors by reading RR1.

The next interrupt vector is receive character available. This mode causes an interrupt if the receiver buffer has at least one character available for the CPU to read. It should be noted that when a character is received, it is possible to have the special receive conditions active, thus indicating that an error occurred in the reception of the character. We now have two possible interrupt vectors that could occur. In this instance the special receive conditions interrupt vector would have internal priority over the character received interrupt vector.

The transmit buffer empty is next. This interrupt vector occurs when the transmit buffer is ready to transmit another character.

Finally, the external status transition interrupt vector occurs whenever there is a change on the status lines of CTS, DCD and SYNC.

11-10: Initialization of the SIO for Interrupts

The initialization sequence for the SIO when using interrupts is very similar to the initialization when not using interrupts. There are two registers that must be used, which are of little concern in a polled environment. These are WR2 (interrupt vector) and WR1 (interrupt mode). In the polled initialization it is only necessary to write a byte to WR1 to disable the interrupts.

To show how the SIO can be initialized for interrupt use, let's study an example. We will first set the SIO with the same operating characteristics that were set in the polled mode. The single difference is that the interrupt system will be enabled. We will use channel B as the communication channel. The Z80 program for the initialization is shown in Figure 11.12.

11-11: After the Initialization

Once the SIO has been initialized, it is ready to operate in an interrupt-driven environment. However, special routines must be written to handle each type of interrupt the SIO is programmed to generate. For the receiver-full interrupt to be asserted, it is only necessary that the character be read from the channel data register—this will empty the buffer and remove the source of the interrupt. The RETI instruction must then be executed to reset the hardware interrupt request.

When an interrupt is caused by the transmitter empty condition, a "Reset Tx Interrupt Pending" command is issued, and the next Tx character is set to the SIO. Figure 11.13 shows an example of two small interrupt routines that will handle the receive and transmit interrupt conditions. These routines will usually be a part of some larger service routine. They illustrate several important points about servicing the SIO interrupts.

Finally, Figure 11.14 shows an interrupt routine that can be used for servicing special receive conditions.

CHAPTER SUMMARY

In this chapter we have discussed the operation and programming of the Z80-SIO. We have examined a block diagram of the device and we have been shown how to program it. The information given in this chapter and in Appendix A should provide you with enough information to apply the SIO to your own unique system serial applications.

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  THIS PROGRAM WILL INITIALIZE THE SIO FOR AN INTERRUPT
;  DRIVEN SYSTEM. THE INTERRUPT VECTOR WILL BE 98H
;  CHANNEL B WILL BE THE ACTIVE CHANNEL. WE ASSUME THE
;  SIO IS MAPPED INTO THE I/O SPACE 8C-8F.
;
;  THE MAIN SIO PARAMETERS ARE X64 CLOCK, 2 STOP BITS,
;  8 RX BITS, 8 TX BITS, NO PARITY.
;  INT ON RX, TX, STATUS WILL AFFECT VECTOR
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;
008D              BDATA  EQU  8DH            ;CHANNEL B DATA PORT
008F              BCON   EQU  8FH            ;CHANNEL B CONTROL PORT
;
;
1800   3E18              LD A,18H
1802   D38F              OUT (BCON),A        ;WR0, CHANNEL RESET
1804   3E02              LD A,02H
1806   D38F              OUT (BCON),A        ;WR0, POINT TO WR2
1808   3E98              LD A,98H
180A   D38F              OUT (BCON),A        ;WR2, INT VECT = 98H
180C   3E04              LD A,04H
180E   D38F              OUT (BCON),A        ;WR0, POINT TO WR4
1810   3ECC              LD A,0CCH
1812   D38F              OUT (BCON),A        ;WR4, X64 CK, 2 STOP BITS, NO
1814   3E03              LD A,03H
1816   D38F              OUT (BCON),A        ;WR0, POINT TO WR3
1818   3EC1              LD A,0C1H
181A   D38F              OUT (BCON),A        ;WR3, 8 RX BITS, RX ENABLE
181C   3E05              LD A,05H
181E   D38F              OUT (BCON),A        ;WR0, POINT TO WR5
1820   3E68              LD A,68H
1822   D38F              OUT (BCON),A        ;WR5, 8 TX BITS, TX ENABLE
1824   3E01              LD A,01H
1826   D38F              OUT (BCON),A        ;WR0, POINT TO WR1
1828   3E16              LD A,16H
;
182A   D38F              OUT (BCON),A        ;WR1, INT RX, TX STATUS EFFECT
182C   C9                RET
;
;  END OF INITIALIZATION ROUTINE
;
                         END
```

Figure 11.12: *A Z80 program for initializing the Z80-SIO in an interrupt application.*

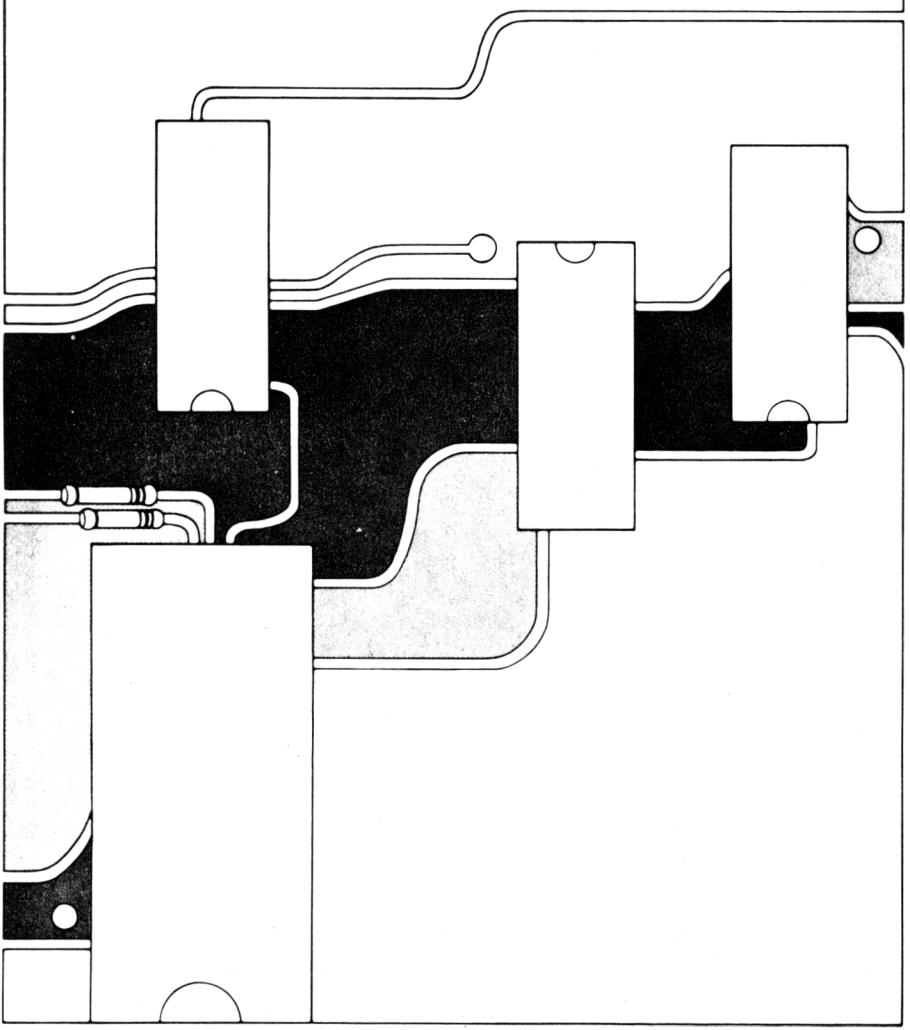
```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  THIS SERVICE ROUTINE WILL BE USED FOR
;  RECEIVING A CHARACTER
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;
008D              BDATA   EQU  8DH      ;CHANNEL B DATA REG
008F              BCON    EQU  8FH      ;CHANNEL B CONTROL REG
5900              RXCH    EQU  5900H    ;RECEIVE CHARACTER LOCATION
5901              TXCH    EQU  RXCH+1   ;TX CHARACTER LOCATION
;
1800   F5         RXINT   PUSH AF       ;SAVE REGISTERS
1801   DB8D               IN A,(BDATA) ;GET CHARACTER
1803   320059             LD (RXCH),A  ;SAVE CHARACTER
1806   F1                 POP AF       ;RESTORE REGISTERS
1807   FB                 EI           ;ENABLE INTERRUPTS
1808   ED4D               RETI         ;RETURN FROM INTERRUPT
;
;
;  THE FOLLOWING IS A SMALL ROUTINE THAT
;  HANDLES TX EMPTY INT WHEN NO MORE CHARS
;  ARE REQUIRED TO SEND.
;
180A   F5         TXINT   PUSH  AF      ;SAVE REGISTERS
180B   3E28               LD A,28H
180D   D38F               OUT (BCON),A ;WR0, RESERT TX INTERRUPTS
180F   F1                 POP  AF
1810   FB                 EI
1811   ED4D               RETI
;
; THE FOLLOWING WILL HANDLE TX EMPTY INTERRUPTS
; WHEN THERE IS ANOTHER CHARACTER TO SEND
;
1813   F5         TXINT1  PUSH  AF
1814   3A0159             LD A,(TXCH)    ;GET THE CHARACTER TO SEND
1817   D38D               OUT (BDATA),A ;SEND CHARACTER TO SIO
1819   F1                 POP  AF

181A   FB                 EI
181B   ED4D               RETI
;
;
                          END
```

Figure 11.13: Examples of two Z80 interrupt service routines for use with the Z80-SIO for transmitting and receiving a character.

```
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;  THIS IS A INTERRUPT ROUTINE FOR SERVICING
;  SPECIAL RECEIVE CONDITIONS INTERRUPT
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;
008D            BDATA  EQU  8DH        ;CHANNEL B DATA PORT
008F            BCON   EQU  8FH        ;CHANNEL B CONTROL PORT
;
1B00    DB8D           IN A,(BDATA)    ;READ INPUT CHARACTER
;
;  THE INPUT CHAR MUST BE READ TO CLEAR THE INPUT BUFFER
;  THE DATA MAY NOT BE USED DUE TO ERROR
;
1B02    3E01           LD A,01H
1B04    D38F           OUT (BCON),A    ;WR0, POINT TO RR1
;
;  WE WILL READ RR1 TO DETERMINE WHAT TYPE OF ERROR
;  CAUSED THE INT
;
1B06    DB8F           IN A,(BCON)     ;GET THE ERROR TYPE
;
;  BITS D6, D5, D4 WILL INDICATE THE TYPE OF ERROR
;
;  AT THIS POINT THE ERROR REGISTER MAY BE
;  SAVED FOR FURTHER PROCESSING. WE WILL NOT
;  DO THIS.
;
1B08    3E30           LD A,30H
1B0A    D38F           OUT (BCON),A    ;WR0, ERROR RESET COMMAND
;
;  THE PRECEEDING OUT WILL CAUSE THE ERRORS
;  TO BE RESET SO WE CAN EXAMINE THEM FOR THE
;  NEXT RECEIVED CHARACTER.
;
;  WE MUST NOW RE-ENABLE INTERRUPTS
;
1B0C    FB             EI
;
;
1B0D    ED4D           RETI
;
;  THE RETI INSTRUCTION IS NEEDED TO RESET
;  THE INTERRUPTS FOR THE SIO HARDWARE.
;
                       END
```

Figure 11.14: *A Z80 program showing an interrupt routine for servicing special receive conditions on the Z80-SIO*

Static Stimulus Testing for the Z80



Chapter 12

INTRODUCTION

In this final chapter, we will discuss a hardware debugging and troubleshooting technique, called *Static Stimulus Testing (or SST)*. Originally developed for industry, this technique offers a simple and inexpensive way to checkout the hardware of your system, without having to use software. You can use this technique to verify all the external interfaces of the Z80 system to all the LSI devices that we have discussed in this book.

The SST was originally developed for repairing and debugging microprocessor systems that fall into two main categories:

- Systems that have experienced some type of catastrophic failure and cannot execute any software (i.e., systems that once worked).
- Systems that are in a prototype mode and have never worked. These are systems that have not yet been debugged, or systems for which the software is not complete and, therefore, cannot be used in the diagnostic process. In these systems the hardware designer or troubleshooter may wish to verify proper operation of all or part of the hardware design without using the system software.

In both categories, the classical hardware troubleshooting techniques, such as Logic State Analysis and Signature Analysis, are of little help. This is because both of these techniques require that the system be capable of executing some software in order for the technique to be useful. Although new techniques are being developed for Signature Analysis that will eliminate this software dependence, there is still a large gap to be filled in the area of troubleshooting microprocessor systems.

An important question then is: "Where and how does one start to debug a system that is totally inoperative? That is, how does one debug a system that will not execute any system software?"

Static Stimulus Testing, although in its infancy, does indeed bridge this gap. With SST it is possible to debug an inoperative system in a direct and orderly fashion to a point where the software diagnostics can then be run. This can be accomplished in an efficient and straightforward manner, using simple and inexpensive instruments.

An important feature of SST is that it allows total independence from the system software. This means that a technician or engineer with little training in system software can successfully apply SST. Further, if a person is skilled in software and has little hardware training, the basics of SST can be learned quite easily.

In this text, we will give examples of using the SST technique to verify the hardware of an entire system. In these examples we will assume that the software does not exist. In other words, we will assume that we have a prototype system that has just been constructed, and that we wish to verify the hardware operation; or, that the system has malfunctioned and there is no software listing (or perhaps knowledge of the system software is minimal).

As you begin using the SST technique to verify each section of the system hardware, you will begin to develop a "feel" for what is occurring in the system hardware. In fact, using this device will give you a better understanding of how the Z80 communicates electrically in a system environment.

12-1: Overview of Static Stimulus Testing

The main concept of Static Stimulus Testing is that electrical communication within a microprocessor system is essentially static in nature. That is, there are two voltage levels, representing 1's and 0's. A microprocessor system alternates between these two basically static states. The electrical events usually take place in rapid succession, but they do not have to. The fact is, there is an upper limit to how fast a system can operate. There is, however, usually no lower limit.

In a microprocessor system, not only is communication between the microprocessor and memory and I/O, electrically static, but the signal lines of the microprocessor system are also performing a unique electrical function at any given instant in time. For example, during a memory communication, the system address lines are logically pointing to a particular memory location. This action occurs regardless of the logical state of any other system signals. Using SST, each signal line in the system can be treated as an independent logic signal.

Each system signal *always* has a point of electrical origin and a point of

electrical destination. Using SST as the source, you can force a single signal line to a desired logical condition at the point of origin. It is then possible to statically trace the electrical response of the line. (*Note:* The element of time-dependent signals is eliminated with SST.)

Using standard digital troubleshooting techniques and SST you can debug the hardware of an entire microprocessor system. Regardless of the complexity of the system hardware, once the dynamic situation has been transformed into a static one, problem areas can be found much easier.

An added advantage of SST is that it will work in systems where special LSI devices are used, such as PIO's or SIO's . This is due to the fact that it operates on the premise that all microprocessor communication within the system is static.

You may disagree with this premise, saying that dynamic RAMs are not static devices. And while this is essentially true, it must be qualified. The only part of the dynamic RAM system that is dynamic is the storage cell of the memory. All addressing inputs, $\overline{RAS}$, $\overline{CAS}$, and MUX, can be thought of as static operations. Thus, it is true that the memory cell itself will not operate in the static mode, but you can use the static approach to debug all of the peripheral signals, as well as the hardware of the memory system.

To illustrate the main point of SST, let's study an example. In this example we will check the address inputs to the system ROM. (*Note:* in this text, it is not possible to give detailed procedures for using SST. The main objective is to show you how the concept of SST can be applied to a real problem.)

Figure 12.1 shows a block diagram of the ROM in a microprocessor system. This figure also shows the point of origin and termination for the address inputs of the ROM. We will use this example to verify proper hardware operation of these address lines using SST. To accomplish this objective, we will follow five steps:

1. First, we will remove the microprocessor from the system and install a cable that is connected to the SST switch panel as shown in Figure 12.2.

2. Let's assume that we have a switch for every address line, A0–A15. Each switch can force one address line to either a logical 1 or a logical 0. These 16 switches may be set by the operator to any of the 64K different possible logical combinations.

3. The combination that is set on the address switches can be left indefinitely (static). We can now verify the inputs to the address buffers in a Z80 system by using static DC measurement techniques. We can determine if all points along these lines show the same (proper) logical condition that was set on the switches. In addition, the outputs of the

address buffers should be equal to the input lines A0–A15 that were set using the address switches on the SST.

4. Next, we examine the logical voltage levels of the address lines at the ROM address inputs. To do this, we can use an oscilloscope, a logic probe, a DVM, or any DC measurement control.

5. All address decoding logic may also be verified at this time in a static fashion.

In this general example we have examined only the address lines. The SST makes it possible to examine these lines separately, because it creates an independence of system signals. Further, note that we have not mentioned anything about absolute time. This is due to the fact that with SST the emphasis is on the sequence of electrical events, and we can take as long as necessary to trace a particular signal line from its origin to its destination.

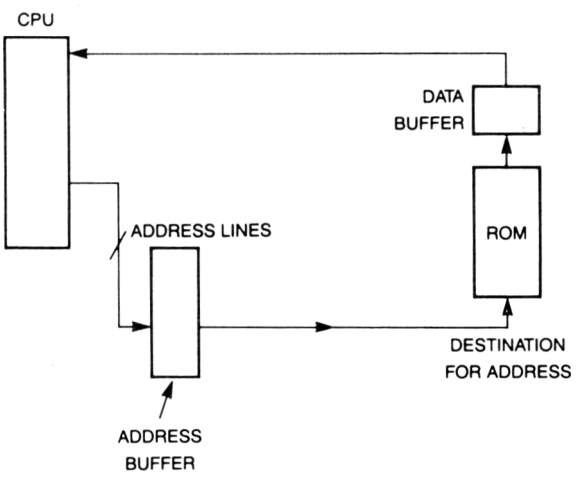


Figure 12.1: *A block diagram showing how the ROM in a system is physically connected to the CPU.*

12-2: Hardware for the Static Stimulus Tester

The SST is a very simple hardware device. It is well-suited for educational purposes as well as for industry. The idea behind the instrument operation is that "the operator has static control of the logic level of any system signal line that the microprocessor would normally control." For the Z80 these signal lines are:

A0–A15	$\overline{\text{IORQ}}$
D0–D7	$\overline{\text{RFSH}}$
$\overline{\text{RD}}$	$\overline{\text{HALT}}$
$\overline{\text{WR}}$	$\overline{\text{MI}}$
$\overline{\text{MREQ}}$	$\overline{\text{BUSAK}}$

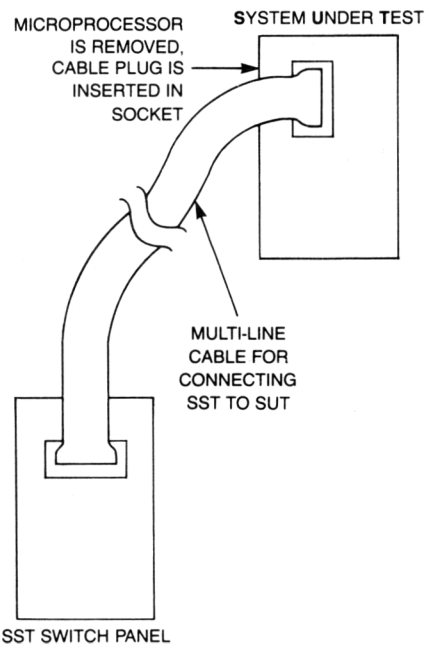


Figure 12.2: *A block diagram showing how the SST switch panel connects to the microprocessor system under test. Notice that the system microprocessor is removed and the SST cable is installed in the vacated socket.*

The remaining Z80 device pins are inputs that we need not be concerned with at this time.

As a final note, it is important to remember that use of the SST will break all feedback loops in the system.

Let us now examine the design of the SST hardware.

12-3: Address and Data Output Lines for the SST

Figure 12.3 shows a block diagram of the hardware required to realize the address and data stimulus for the SST. In this figure, the logical value of the address output is determined by the physical position of the DIP switches. The output of each switch is logically inverted and buffered. The buffers

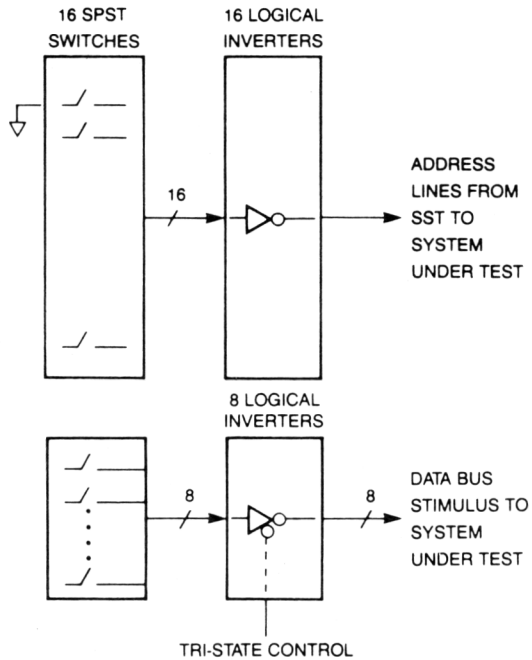


Figure 12.3: A block diagram showing how the address and data lines are stimulated using SPST switches.

used for data stimulus are capable of tri-state control. (We will discuss the reason for this later on in this chapter.) For now, it is only necessary to know that the data buffers can be tri-stated.

The outputs of the buffers labeled D0–D7 are used for the generation of the system data lines. When the SST is used for troubleshooting the Z80 system, the DIP switches are used for data output stimulus in the same way that they are used by the Z80. Figures 12.4 and 12.5 show the actual hardware schematic for realization of the address and data stimulus section of the SST.

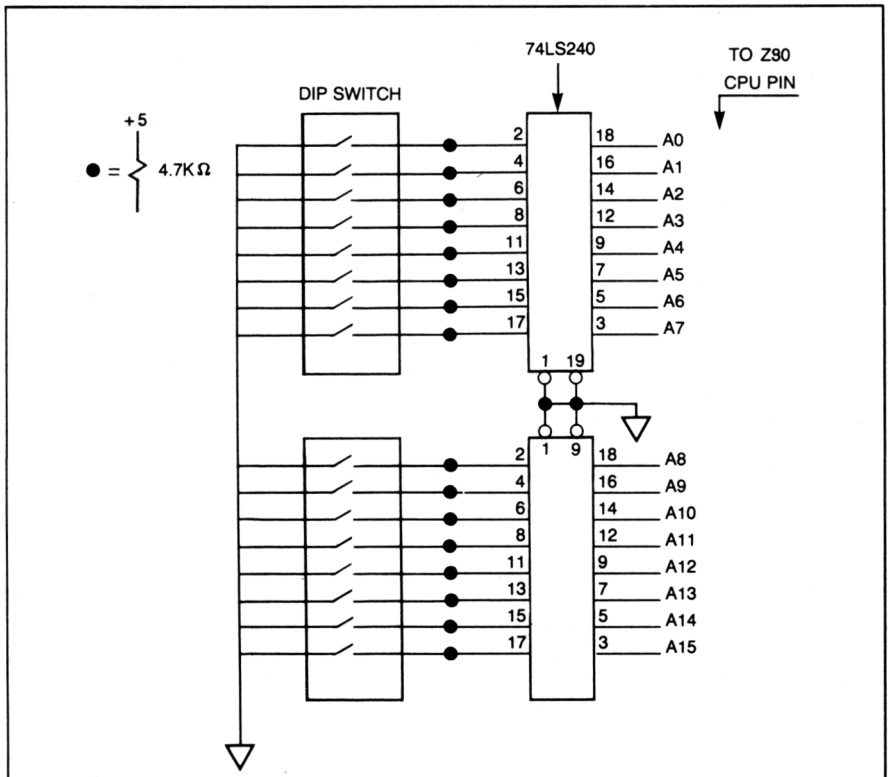


Figure 12.4: *A schematic diagram showing how the address outputs are set to a logical 0 or a logical 1.*

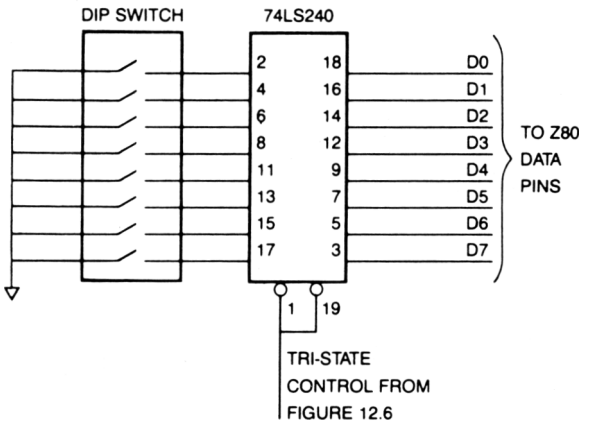


Figure 12.5: A schematic diagram showing how the data lines are set to a logical 1 or a logical 0.

12-4: $\overline{\text{M1}}$, $\overline{\text{MREQ}}$, $\overline{\text{IORQ}}$, $\overline{\text{RD}}$, $\overline{\text{WR}}$, $\overline{\text{RFSH}}$, $\overline{\text{HALT}}$, $\overline{\text{BUSAK}}$

We will now learn to generate the following control signals on the SST:

$\overline{\text{M1}}$
$\overline{\text{MREQ}}$
$\overline{\text{IORQ}}$
$\overline{\text{RD}}$
$\overline{\text{WR}}$
$\overline{\text{RFSH}}$
$\overline{\text{HALT}}$
$\overline{\text{BUSAK}}$

We are discussing all of these signals in the same section because the hardware required for each is identical.

All the signals listed above are control bits generated by the Z80 and output to the system. Most are valid at the start of a memory cycle and will remain valid for an entire cycle. To generate these signals, SPDT toggle switches are used for the stimulus. Outputs of the switches are input to switch debounce circuits. Debounce circuit outputs are used as the final outputs to the system under test. Figure 12.6 shows the schematic for generation of these signals.

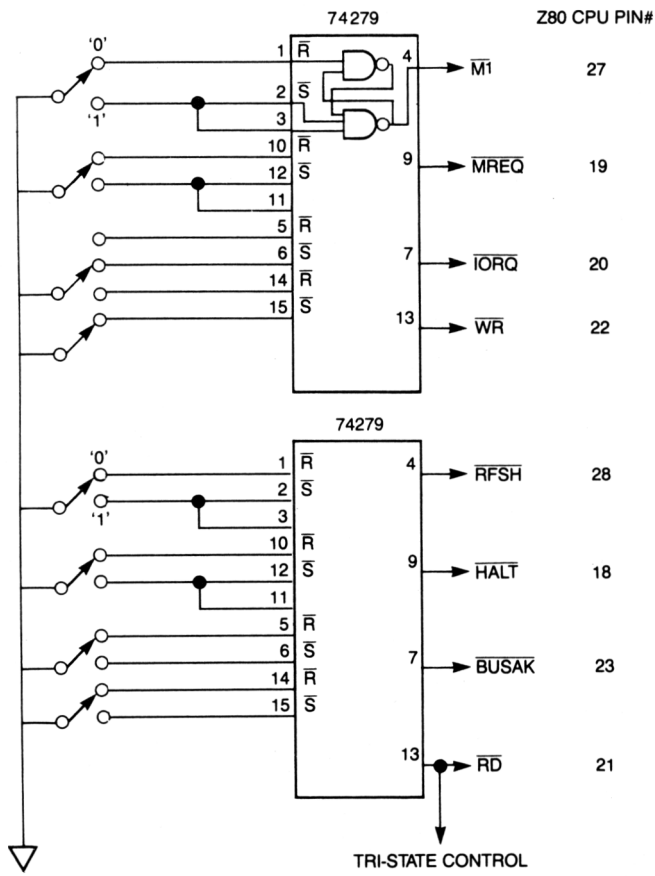


Figure 12.6: Schematic diagram of the hardware necessary to debounce all major control signals generated by the Z80. The 74LS279 device has four debounce, set, and reset latches internal to it. The switches used are SPDT toggle switches.

The schematic in Figure 12.6 makes use of the 74279 IC. This device contains four SET (S)–RESET (R) latches. These latches are used as the debounce circuit for the switches.

12-5: LED Display for the Data Bus

We have now discussed all the circuits of the SST hardware except for the circuit that allows us to visually examine the system data bus. This visual examination can be accomplished by using eight LEDs to display the logical condition of each line of the system data bus. Figure 12.7 shows how this display is realized. Here, the microprocessor (or SST) data bus is connected to the inputs of the 74LS240 inverters. The outputs of these devices drive the LEDs. A logical 0 on the inputs of the 74LS240 will turn the LEDs off. A logical 1 will turn them on.

The LEDs reflect the logical condition of the signal lines at the microprocessor device pins D0–D7. This makes it possible to actually see the logical value of the data being input or output by the microprocessor. If two

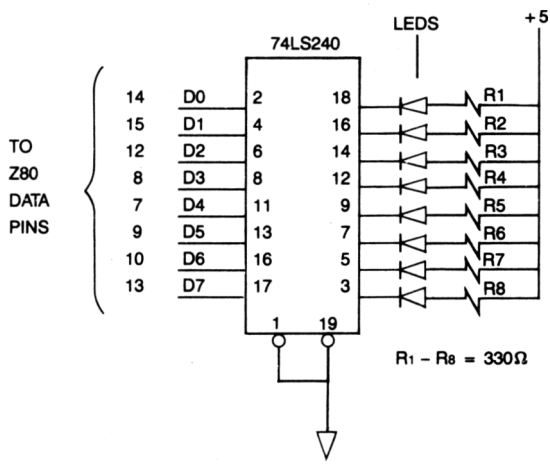


Figure 12.7: A schematic diagram showing how the LEDs on the SST will give an indication of the logical level of the system data bus.

data lines are sorted, or if the data is not reaching the microprocessor input pins, it will quickly show up on the LED display.

During a write operation you can easily see what data the Z80 or SST is outputting to the system hardware. Notice that this display shows data directly at the CPU device pins. This is helpful when checking the system data bus lines.

CHAPTER SUMMARY

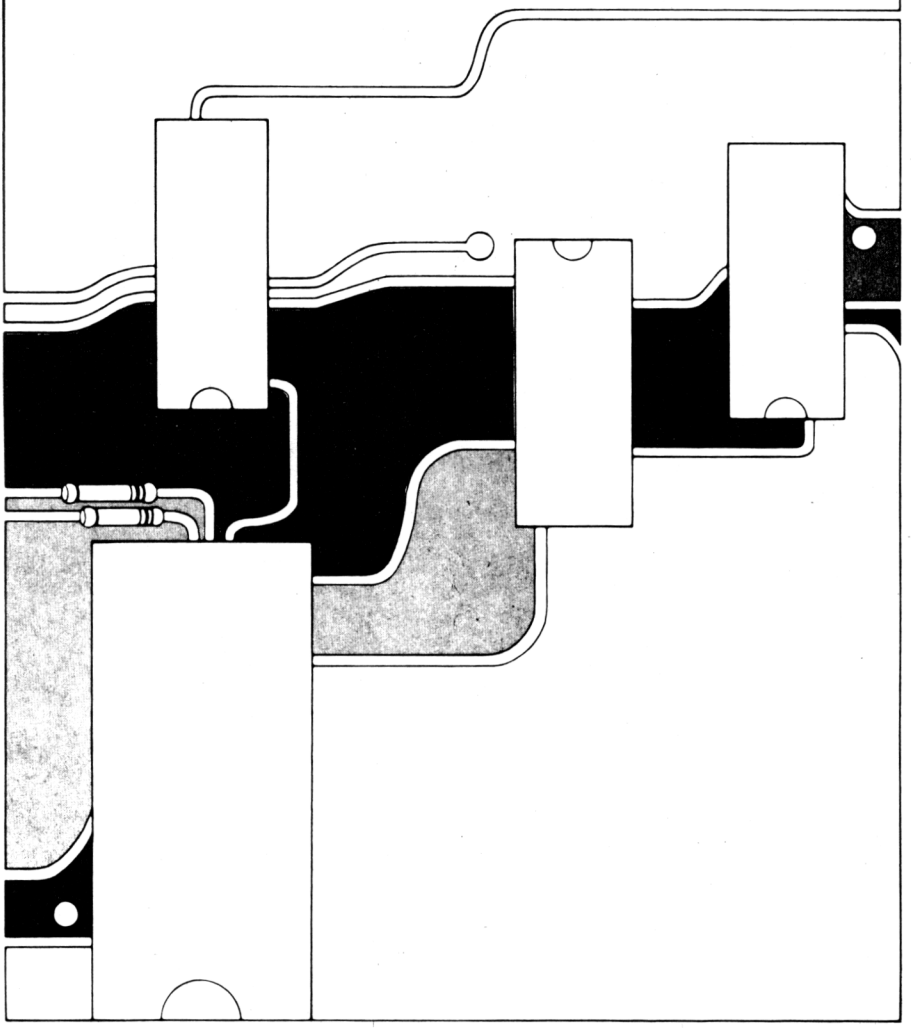
In this chapter we have examined the hardware required to use the Static Stimulus Testing (SST) technique on the Z80. This technique is a powerful tool for debugging a microprocessor system. It can be used to debug prototype systems, as well as to debug malfunctioning production systems. The SST can be used by anyone. It does not require a highly skilled microprocessor system designer, software person, or hardware person to achieve excellent results.

As a final note, if you do not have a Static Stimulus Tester, and desire to acquire one, you have two options. You may construct one from the schematics given in this chapter or you may purchase one from a manufacturer.[1]

---

[1] Two versions of the Z80 SST are available at Creative Microprocessor Systems, Inc., P.O. Box 1538, Los Gatos, CA. 95030.

Z80-SIO Internal Register Descriptions



Appendix

WRITE REGISTERS

The Z80-SIO contains eight registers (WR0-WR7) in each channel that are programmed separately by the system program to configure the functional personality of the channels. With the exception of WR0, programming the write registers requires two bytes. The first byte contains three bits (D_0-D_2) that point to the selected register; the second byte is the actual control word that is written into the register to configure the Z80-SIO.

Note that the programmer has complete freedom, after pointing to the selected register, of either reading to test the read register or writing to initialize the write register. By designing software to initialize the Z80-SIO in a modular and structured fashion, the programmer can use powerful block I/O instructions.

WR0 is a special case in that all the basic commands (CMD_0-CMD_2) can be accessed with a single byte. Reset (internal or external) initializes the pointer bits (D_0-D_2) to point to WR0.

The basic commands (CMD_0-CMD_2) and the CRC controls (CRC_0, CRC_1) are contained in the first byte of any write register access. This maintains maximum flexibility and system control. Each channel contains the following control registers. These registers are addressed as commands (not data).

WRITE REGISTER 0

WR0 is the command register; however, it is also used for CRC reset codes and to point to the other registers.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
CRC Reset Code 1	CRC Reset Code 0	CMD 2	CMD 1	CMD 0	PTR 2	PTR 1	PTR 0

Pointer Bits (D_0-D_2). Bits D_0-D_2 are pointer bits that determine which other write register the next byte is to be written into or which read register the next byte is to be read from. The first byte written into each channel after a reset (either by a Reset command or by the external reset input) goes into WR0. Following a read or write to any register (except WR0), the pointer will point to WR0.

Command Bits (D_3-D_5). Three bits, D_3-D_5, are encoded to issue the seven basic Z80-SIO commands.

COMMAND	CMD_2	CMD_1	CMD_0	
0	0	0	0	Null Command (no effect)
1	0	0	1	Send Abort (SDLC Mode)
2	0	1	0	Reset External/Status Interrupts
3	0	1	1	Channel Reset
4	1	0	0	Enable Interrupt on next Rx Character
5	1	0	1	Reset Transmitter Interrupt Pending
6	1	1	0	Error Reset (latches)
7	1	1	1	Return from Interrupt (Channel A)

Command 0 (Null). The Null command has no effect. Its normal use is to cause the Z80-SIO to do nothing while the pointers are set for the following byte.

Command 1 (Send Abort). This command is used only with the SDLC mode to generate a sequence of eight to thirteen 1's.

Command 2 (Reset External/Status Interrupts). After an External/Status interrupt (a change on a modem line or a break condition, for example), the status bits of RR0 are latched. This command re-enables them and allows interrupts to occur again. Latching the status bits captures short pulses until the CPU has time to read the change.

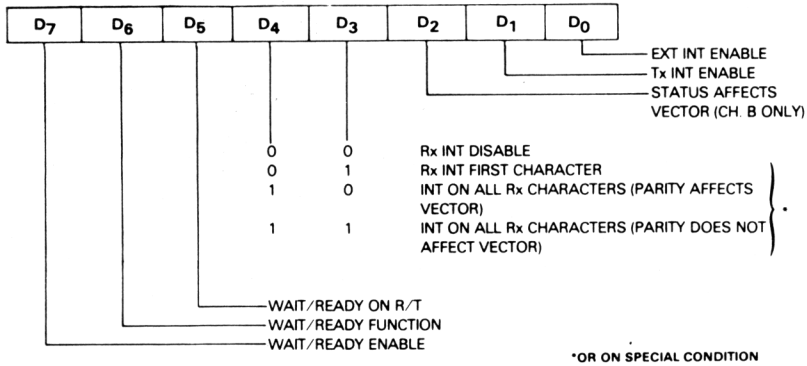
Command 3 (Channel Reset). This command performs the same function as an External Reset, but only on a single channel. Channel A Reset also resets the interrupt prioritization logic. All control registers for the channel must be rewritten after a Channel Reset command.

WRITE REGISTER BIT FUNCTIONS

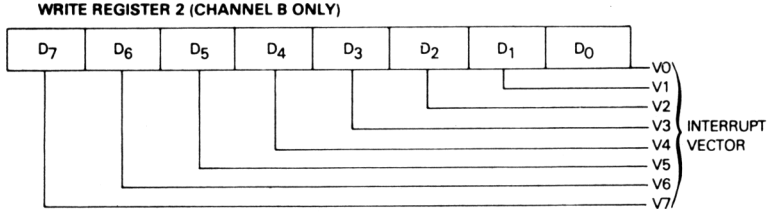
WRITE REGISTER 0

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
					0	0	0	REGISTER 0
					0	0	1	REGISTER 1
					0	1	0	REGISTER 2
					0	1	1	REGISTER 3
					1	0	0	REGISTER 4
					1	0	1	REGISTER 5
					1	1	0	REGISTER 6
					1	1	1	REGISTER 7
		0	0	0	NULL CODE			
		0	0	1	SEND ABORT (SDLC)			
		0	1	0	RESET EXT/STATUS INTERRUPTS			
		0	1	1	CHANNEL RESET			
		1	0	0	ENABLE INT ON NEXT Rx CHARACTER			
		1	0	1	RESET Tx INT PENDING			
		1	1	0	ERROR RESET			
		1	1	1	RETURN FROM INT (CH-A ONLY)			
0	0	NULL CODE						
0	1	RESET Rx CRC CHECKER						
1	0	RESET Tx CRC GENERATOR						
1	1	RESET Tx UNDERRUN/EOM LATCH						

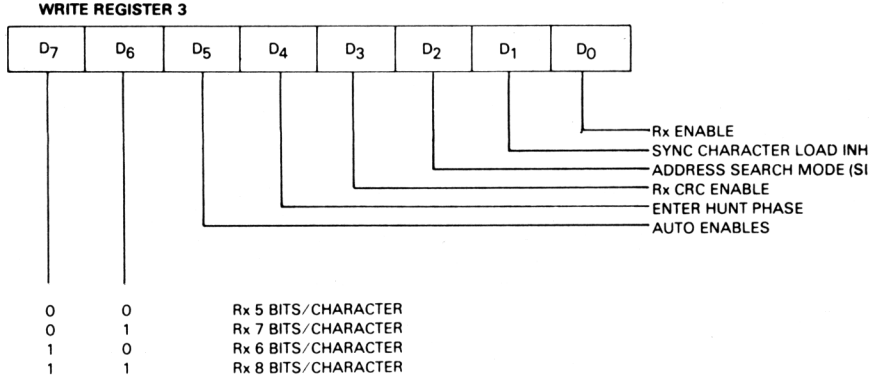
WRITE REGISTER 1



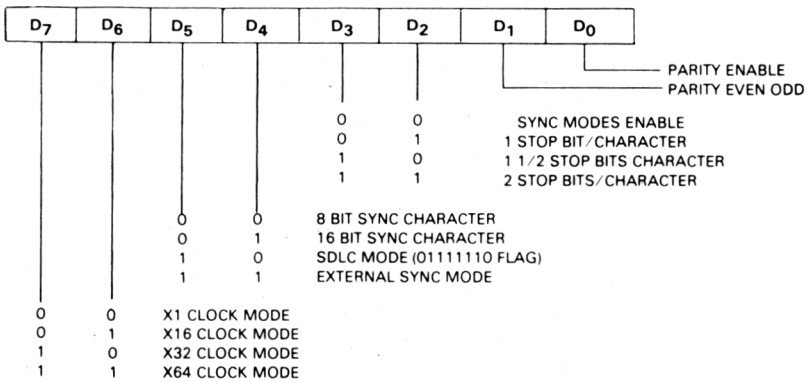
WRITE REGISTER 2 (CHANNEL B ONLY)



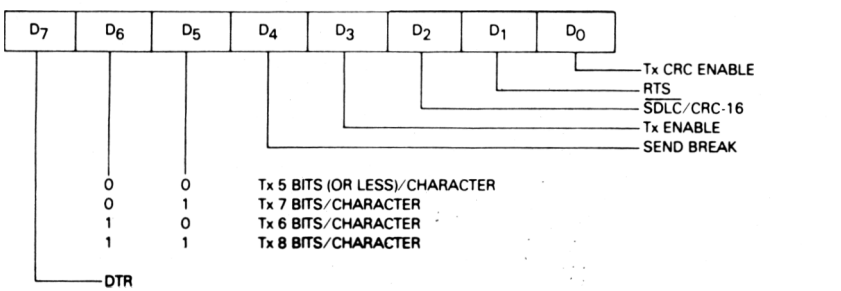
WRITE REGISTER 3



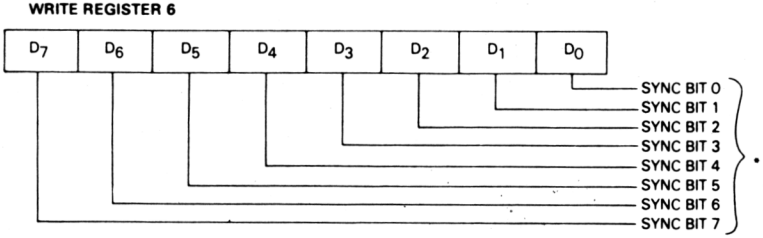
WRITE REGISTER 4



WRITE REGISTER 5

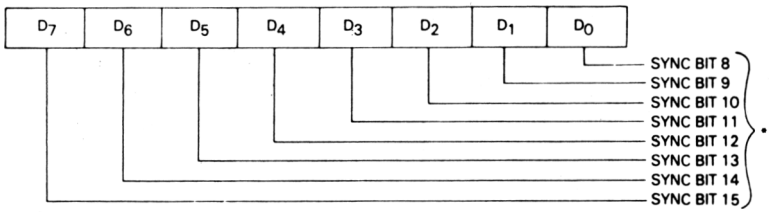


WRITE REGISTER 6



*ALSO SDLC ADDRESS FIELD

WRITE REGISTER 7



*FOR SDLC, IT MUST BE PROGRAMMED TO "01111110" FOR FLAG RECOGNITION

After a Channel Reset, four extra system clock cycles should be allowed for Z80-SIO reset time before any additional commands or controls are written into that channel. This can normally be the time used by the CPU to fetch the next op code.

Command 4 (Enable Interrupt On Next Character). If the Interrupt On First Receive Character mode is selected, this command reactivates that mode after each complete message is received to prepare the Z80-SIO for the next message.

Command 5 (Reset Transmitter Interrupt Pending). The transmitter interrupts when the transmit buffer becomes empty if the Transmit Interrupt Enable mode is selected. In those cases where there are no more characters to be sent (at the end of message, for example), issuing this command prevents further transmitter interrupts until after the next character has been loaded into the transmit buffer or until CRC has been completely sent.

Command 6 (Error Reset). This command resets the error latches. Parity and Overrun errors are latched in RR1 until they are reset with this command. With this scheme, parity errors occurring in block transfers can be examined at the end of the block.

Command 7 (Return From Interrupt). This command must be issued in Channel A and is interpreted by the Z80-SIO in exactly the same way it would interpret a RETI command on the data bus. It resets the interrupt-under-service latch of the highest-priority internal device under service and thus allows lower priority devices to interrupt via the daisy chain. This command allows use of the internal daisy chain even in systems with no external daisy chain or RETI command.

CRC Reset Codes 0 and 1 (D_6 and D_7). Together, these bits select one of the three following reset commands:

CRC Reset Code 1	CRC Reset Code 0	
0	0	Null Code (no effect)
0	1	Reset Receive CRC Checker
1	0	Reset Transmit CRC Generator
1	1	Reset Tx Underrun/End Of Message Latch

The Reset Transmit CRC Generator command normally initializes the CRC generator to all 0's. If the SDLC mode is selected, this command initializes the CRC generator to all 1's. The Receive CRC checker is also initialized to all 1's for the SDLC mode.

WRITE REGISTER 1

WR1 contains the control bits for the various interrupt and Wait/Ready modes.

D_7 Wait/Ready Enable	D_6 Wait Or Ready Function	D_5 Wait/Ready On Receive/Transmit	D_4 Receive Interrupt Mode 1
D_3 Receive Interrupt Mode 0	D_2 Status Affects Vector	D_1 Transmit Interrupt Enable	D_0 External Interrupts Enable

External/Status Interrupt Enable (D_0). The External/Status Interrupt Enable allows interrupts to occur as a result of transitions on the $\overline{DCD}$, $\overline{CTS}$ or $\overline{SYNC}$ inputs, as a result of a Break/Abort detection and termination, or at the beginning of CRC or sync character transmission when the Transmit Underrun/EOM latch becomes set.

Transmitter Interrupt Enable (D_1). If enabled, the interrupts occur whenever the transmitter buffer becomes empty.

Status Affects Vector (D_2). This bit is active in Channel B only. If this bit is not set, the fixed vector programmed in WR2 is returned from an interrupt acknowledge sequence. If this bit is set, the vector returned from an interrupt acknowledge is variable according to the following interrupt conditions:

	V_3	V_2	V_1	
Ch B	0	0	0	Ch B Transmit Buffer Empty
	0	0	1	Ch B External/Status Change
	0	1	0	Ch B Receive Character Available
	0	1	1	Ch B Special Receive Condition*
Ch A	1	0	0	Ch A Transmit Buffer Empty
	1	0	1	Ch A External/Status Change
	1	1	0	Ch A Receive Character Available
	1	1	1	Ch A Special Receive Condition*

*Special Receive Conditions: Parity Error, Rx Overrun Error, Framing Error, End Of Frame (SDLC).

Receive Interrupt Modes 0 and 1 (D_3 and D_4). Together, these two bits specify the various character-available conditions. In Receive Interrupt modes 1, 2 and 3, a Special Receive Condition can cause an interrupt and modify the interrupt vector.

D_4 Receive Interrupt Mode 1	D_3 Receive Interrupt Mode 0	
0	0	0. Receive Interrupts Disabled
0	1	1. Receive Interrupt On First Character Only
1	0	2. Interrupt On All Receive Characters—parity error is a Special Receive condition
1	1	3. Interrupt On All Receive Characters—parity error is not a Special Receive condition

Wait/Ready Function Selection (D_5-D_7). The Wait and Ready functions are selected by controlling D_5, D_6 and D_7. Wait/Ready function is enabled by setting Wait/Ready Enable (WR1, D_7) to 1. The Ready Function is selected by setting D_6 (Wait/Ready function) to 1. If this bit is 1, the $\overline{\text{WAIT}}/\overline{\text{READY}}$ output switches from High to Low when the Z80-SIO is ready to transfer data. The Wait function is selected by setting D_6 to 0. If this bit is 0, the $\overline{\text{WAIT}}/\overline{\text{READY}}$ output is in the open-drain state and goes Low when active.

Both the Wait and Ready functions can be used in either the Transmit or Receive modes, but not both simultaneously. If D_5 (Wait/Ready or Receive/Transmit) is set to 1, the Wait/Ready function responds to the condition of the receive buffer (empty or full). If D_5 is set to 0, the Wait/Ready function responds to the condition of the transmit buffer (empty or full).

The logic states of the $\overline{\text{WAIT}}/\overline{\text{READY}}$ output when active or inactive depend on the combination of modes selected. Following is a summary of these combinations:

	And $D_6 = 1$	If $D_7 = 0$	And $D_6 = 0$
	$\overline{\text{READY}}$ is High		$\overline{\text{WAIT}}$ is floating
	And $D_5 = 0$	If $D_7 = 1$	And $D_5 = 1$
$\overline{\text{READY}}$	Is High when transmit buffer is full.	$\overline{\text{READY}}$	Is High when receive buffer is empty.
$\overline{\text{WAIT}}$	Is Low when transmit buffer is full and an SIO data port is selected.	$\overline{\text{WAIT}}$	Is Low when receive buffer is empty and an SIO data port is selected.
$\overline{\text{READY}}$	Is Low when transmit buffer is empty.	$\overline{\text{READY}}$	Is Low when receive buffer is full.
$\overline{\text{WAIT}}$	Is floating when transmit buffer is empty.	$\overline{\text{WAIT}}$	Is Floating when receive buffer is full.

The $\overline{\text{WAIT}}$ output High-to-Low transition occurs when the delay time t_{DIC}(WR) after the I/O request. The Low-to-High transition occurs with the delay $t_{DH\Phi}$(WR) from the falling edge of Φ. The $\overline{\text{READY}}$ output High-to-Low transition occurs with the delay $t_{DL\Phi}$(WR) from the rising edge of Φ. The $\overline{\text{READY}}$ output Low-to-High transition occurs with the delay t_{DIC}(WR) after $\overline{\text{IORQ}}$ falls.

The Ready function can occur any time the Z80-SIO is not selected. When the $\overline{\text{READY}}$ output becomes active (Low), the DMA controller issues $\overline{\text{IORQ}}$ and the corresponding B/$\overline{\text{A}}$ and C/$\overline{\text{D}}$ inputs to the Z80-SIO to transfer data. The $\overline{\text{READY}}$ output becomes inactive as soon as $\overline{\text{IORQ}}$ and $\overline{\text{CS}}$ become active. Since the Ready function can occur internally in the Z80-SIO whether it is addressed or not, the $\overline{\text{READY}}$ output becomes inactive when any CPU data or command transfer takes place. This does not cause problems because the DMA controller is not enabled when the CPU transfer takes place.

The Wait function—on the other hand—is active only if the CPU attempts to read Z80-SIO data that has not yet been received, which occurs frequently when block transfer instructions are used. The Wait function can also become active (under program control) if the CPU tries to write data while the transmit buffer is still full. The fact that the $\overline{\text{WAIT}}$ output for either channel can become active when the opposite channel is addressed (because the Z80-SIO is addressed) does not affect operation of software loops or block move instructions.

WRITE REGISTER 2

WR2 is the interrupt vector register; it exists in Channel B only. V_4-V_7 and V_0 are always returned exactly as written; V_1-V_3 are returned as written if the Status Affects Vector (WR1, D_2) control bit is 0. If this bit is 1, they are modified as explained in the previous section.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
V_7	V_6	V_5	V_4	V_3	V_2	V_1	V_0

WRITE REGISTER 3

WR3 contains receiver logic control bits and parameters.

D_7	D_6	D_5	D_4
Receiver Bits/ Char 1	Receiver Bits/ Char 0	Auto Enables	Enter Hunt Phase
D_3	D_2	D_1	D_0
Receiver CRC Enable	Address Search Mode	Sync Char Load Inhibit	Receiver Enable

Receiver Enable (D_0). A 1 programmed into this bit allows receive operations to begin. This bit should be set only after all other receive parameters are set and receiver is completely initialized.

Sync Character Load Inhibit (D_1). Sync characters preceding the message (leading sync characters) are not loaded into the receive buffers if this option is selected. Because CRC calculations are not stopped by sync character stripping, this feature should be enabled only at the beginning of the message.

Address Search Mode (D_2). If SDLC is selected, setting this mode causes messages with addresses not matching the programmed address in WR6 or the global (11111111) address to be rejected. In other words, no receive interrupts can occur in the Address Search mode unless there is an address match.

Receiver CRC Enable (D_3). If this bit is set, CRC calculation starts (or restarts) at the beginning of the last character transferred from the receive shift register to the buffer stack, regardless of the number of characters in the stack. See "SDLC Receive CRC Checking" (SDLC Receive section) and "CRC Error Checking" (Synchronous Receive section) for details regarding when this bit should be set.

Enter Hunt Phase (D_4). The Z80-SIO automatically enters the Hunt phase after a reset; however, it can be re-entered if character synchronization is lost for any reason (Synchronous mode) or if the contents of an incoming message are not needed (SDLC mode). The Hunt phase is re-entered by writing a 1 into bit D_4. This sets the Sync/Hunt bit (D_4) in RR0.

Auto Enables (D_5). If this mode is selected, $\overline{DCD}$ and $\overline{CTS}$ become the receiver and transmitter enables, respectively. If this bit is not set, $\overline{DCD}$ and $\overline{CTS}$ are simply inputs to their corresponding status bits in RR0.

Receiver Bits/Character 1 and 0 (D_7 and D_6). Together, these bits determine the number of serial receive bits assembled to form a character. Both bits may be changed during the time that a character is being assembled, but they must be changed before the number of bits currently programmed is reached.

D_7	D_6	Bits/Character
0	0	5
0	1	7
1	0	6
1	1	8

WRITE REGISTER 4

WR4 contains the control bits that affect both the receiver and transmitter. In the transmit and receive initialization routine, these bits should be set before issuing WR1, WR3, WR5, WR6, and WR7.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Clock Rate 1	Clock Rate 0	Sync Modes 1	Sync Modes 0	Stop Bits 1	Stop Bits 0	Parity Even/$\overline{\text{Odd}}$	Parity

Parity (D_0). If this bit is set, an additional bit position (in addition to those specified in the bits/character control) is added to transmitted data and is expected in receive data. In the Receive mode, the parity bit received is transferred to the CPU as part of the character, unless 8 bits/character is selected.

Parity Even $\overline{\text{Odd}}$ (D_1). If parity is specified, this bit determines whether it is sent and checked as even or odd (1 = even).

Stop Bits 0 and 1 (D_2 and D_3). These bits determine the number of stop bits added to each asynchronous character sent. The receiver always checks for one stop bit. A special mode (00) signifies that a synchronous mode is to be selected.

D_3 Stop Bits 1	D_2 Stop Bits 0	
0	0	Sync modes
0	1	1 stop bit per character
1	0	1½ stop bits per character
1	1	2 stop bits per character

Sync Modes 0 and 1 (D_4 and D_5). These bits select the various options for character synchronization.

Sync Mode 1	Sync Mode 0	
0	0	8-bit programmed sync
0	1	16-bit programmed sync
1	0	SDLC mode (01111110 flag pattern)
1	1	External Sync mode

Clock Rate 0 and 1 (D_6 and D_7). These bits specify the multiplier between the clock ($\overline{\text{TxC}}$ and $\overline{\text{RxC}}$) and data rates. For synchronous modes, the x1 clock rate must be specified. Any rate may be specified for asynchronous modes, however, the same rate must be used for both the receiver and transmitter. The system clock in all modes must be at least 5 times the data rate. If the x1 clock rate is selected, bit synchronization must be accomplished externally.

Clock Rate 1	Clock Rate 0	
0	0	Data Rate x1 Clock Rate
0	1	Data Rate x16 Clock Rate
1	0	Data Rate x32 Clock Rate
1	1	Data Rate x64 Clock Rate

WRITE REGISTER 5

WR5 contains control bits that affect the operation of transmitter, with the exception of D2, which affects the transmitter and receiver.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
DTR	Tx Bits/ Char 1	Tx Bits/ Char 0	Send Break	Tx Enable	CRC-16/ SDLC	RTS	Tx CRC Enable

Transmit CRC Enable (D_0). This bit determines if CRC is calculated on a particular transmit character. If it is set at the time the character is loaded from the transmit buffer into the transmit shift register, CRC is calculated on the character. CRC is not automatically sent unless this bit is set when the Transmit Underrun condition exists.

Request To Send (D_1). This is the control bit for the $\overline{RTS}$ pin. When the $\overline{RTS}$ bit is set, the $\overline{RTS}$ pin goes Low; when reset, $\overline{RTS}$ goes High. In the Asynchronous mode, $\overline{RTS}$ goes High only after all the bits of the character are transmitted and the transmitter buffer is empty. In Synchronous modes, the pin directly follows the state of the bit.

CRC-16/$\overline{SDLC}$ (D_2). This bit selects the CRC polynomial used by both the transmitter and receiver. When set, the CRC-16 polynomial ($X^{16} + X^{15} + X^2 + 1$) is used; when reset, the SDLC polynomial ($X^{16} + X^{12} + X^5 + 1$) is used. If the SDLC mode is selected, the CRC generator and checker are preset to all 1's and a special check sequence is used. The SDLC CRC polynomial must be selected when the SDLC mode is selected. If the SDLC mode is not selected, the CRC generator and checker are present to all 0's (for both polynomials).

Transmit Enable (D_3). Data is not transmitted until this bit is set and the Transmit Data output is held marking. Data or sync characters in the process of being transmitted are completely sent if this bit is reset after transmission has started. If the transmitter is disabled during the transmission of a CRC character, sync or flag characters are sent instead of CRC.

Send Break (D_4). When set, this bit immediately forces the Transmit Data output to the spacing condition, regardless of any data being transmitted. When reset, TxD returns to marking.

Transmit Bits/Character 0 and 1 (D_5 and D_6). Together, D_6 and D_5 control the number of bits in each byte transferred to the transmit buffer.

D_6 Transmit Bits/ Character 1	D_5 Transmit Bits/ Character 0	Bits/Character
0	0	Five or less
0	1	7
1	0	6
1	1	8

Bits to be sent must be right justified, least-significant bits first. The Five Or Less mode allows transmission of one to five bits per character; however, the CPU should format the data character as shown in the following table.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
1	1	1	1	0	0	0	D	Sends one data bit
1	1	1	0	0	0	D	D	Sends two data bits
1	1	0	0	0	D	D	D	Sends three data bits
1	0	0	0	D	D	D	D	Sends four data bits
0	0	0	D	D	D	D	D	Sends five data bits

Data Terminal Ready (D_7). This is the control bit for the $\overline{DTR}$ pin. When set, $\overline{DTR}$ is active (Low); when reset, $\overline{DTR}$ is inactive (High).

WRITE REGISTER 6

This register is programmed to contain the transmit sync character in the Monosync mode, the first eight bits of a 16-bit sync character in the Bisync mode or a transmit sync character in the External Sync mode. In the SDLC mode, it is programmed to contain the secondary address field used to compare against the address field of the SDLC frame.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Sync 7	Sync 6	Sync 5	Sync 4	Sync 3	Sync 2	Sync 1	Sync 0

WRITE REGISTER 7

This register is programmed to contain the receive sync character in the Monosync mode, a second byte (last eight bits) of a 16-bit sync character in the Bisync mode and a flag character (01111110) in the SDLC mode. WR7 is not used in the External Sync mode.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Sync 15	Sync 14	Sync 13	Sync 12	Sync 11	Sync 10	Sync 9	Sync 8

READ REGISTERS INTRODUCTION

The Z80-SIO contains three registers, RR0-RR2 (Figure 7.1), that can be read to obtain the status information for each channel (except for RR2-Channel B only). The status information includes error conditions, interrupt vector and standard communications-interface signals.

To read the contents of a selected read register other than RR0, the system program must first write the pointer byte to WR0 in exactly the same way as a write register operation. Then, by executing an input instruction, the contents of the addressed read register can be read by the CPU.

The status bits of RR0 and RR1 are carefully grouped to simplify status monitoring. For example, when the interrupt vector indicates that a Special Receive Condition interrupt has occurred, all the appropriate error bits can be read from a single register (RR1).

READ REGISTER 0

This register contains the status of the receive and transmit buffers, the $\overline{DCD}$, $\overline{CTS}$ and $\overline{SYNC}$ inputs, the Transmit Underrun/EOM latch; and the Break/Abort latch.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Break Abort	Transmit Underrun/ EOM	$\overline{CTS}$	Sync/ Hunt	$\overline{DCD}$	Transmit Buffer Empty	Interrupt Pending (Ch. A only)	Receive Character Available

Receive Character Available (D_0). This bit is set when at least one character is available in the receive buffer; it is reset when the receive FIFO is completely empty.

Interrupt Pending (D_1). Any interrupting condition in the Z80-SIO causes this bit to be set; however, it is readable only in Channel A. This bit is mainly used in applications that do not have vectored interrupts available. During the interrupt service routine in these applications, this bit indicates if any interrupt conditions are present in all Z80-SIO. This eliminates the need for analyzing all the bits of RR0 in both Channels A and B. Bit D_1 is reset when all the interrupting conditions are satisfied. This bit is always 0 in Channel B.

Transmit Buffer Empty (D_2). This bit is set whenever the transmit buffer becomes empty, except when a CRC character is being sent in a synchronous or SDLC mode. The bit is reset when a character is loaded into the transmit buffer. This bit is in the set condition after a reset.

Data Carrier Detect (D_3). The DCD bit shows the inverted state of the $\overline{DCD}$ input at the time of the last change of any of the five External/Status bits (DCD, $\overline{CTS}$, Sync/Hunt, Break/Abort or Transmit Underrun/EOM). Any transition of the $\overline{DCD}$ input causes the DCD bit to be latched and causes an External/Status interrupt. To read the current state of the DCD bit, this bit must be read immediately following a Reset External/Status Interrupt command.

Sync/Hunt (D_4). Since this bit is controlled differently in the Asynchronous, Synchronous and SDLC modes, its operation is somewhat more complex than that of the other bits and, therefore, requires more explanation.

In Asynchronous modes, the operation of this bit is similar to the DCD status bit, except that Sync/Hunt shows the state of the $\overline{SYNC}$ input. Any High-to-Low transition on the $\overline{SYNC}$ pin sets this bit and causes an External/Status interrupt (if enabled). The Reset External/Status Interrupt command is issued to clear the interrupt. A Low-to-High transition clears this bit and sets the External/Status interrupt. When the External/Status interrupt is set by the change in state of any other input or condition, this bit shows the inverted state of $\overline{SYNC}$ pin at the time of the change. This bit must be read immediately following a Reset External/Status Interrupt command to read the current state of the $\overline{SYNC}$ input.

In the External Sync mode, the Sync/Hunt bit operates in a fashion similar to the Asynchronous mode, except the Enter Hunt Mode control bit enables the external sync detection logic. When the External

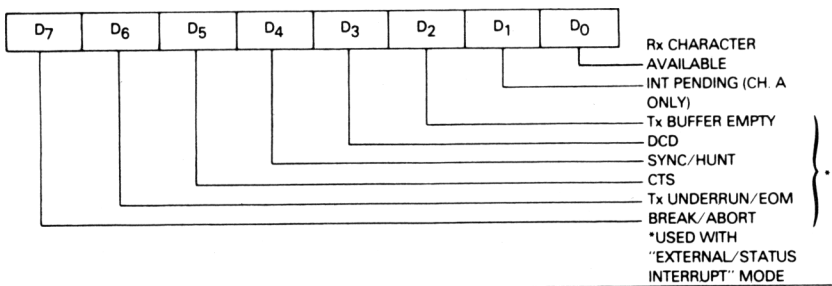
Sync Mode and Enter Hunt Mode bits are set (for example, when the receiver is enabled following a reset), the $\overline{SYNC}$ input must be held High by the external logic until external character synchronization is achieved. A High at the $\overline{SYNC}$ input holds the Sync/Hunt status bit in the reset condition.

When external synchronization is achieved, $\overline{SYNC}$ must be driven Low on the second rising edge of $\overline{RxC}$ on which the last bit of the sync character was received. In other words, after the sync pattern is detected, the external logic must wait for two full Receive clock cycles to activate the $\overline{SYNC}$ input. Once $\overline{SYNC}$ is forced Low, it is a good practice to keep it Low until the CPU informs the external sync logic that synchronization has been lost or a new message is about to start. Refer to Figure 8.6 for timing details. The High-to-Low transition of the $\overline{SYNC}$ input sets the Sync/Hunt bit, which—in turn—sets the External/Status interrupt. The CPU must clear the interrupt by issuing the Reset External/Status Interrupt command.

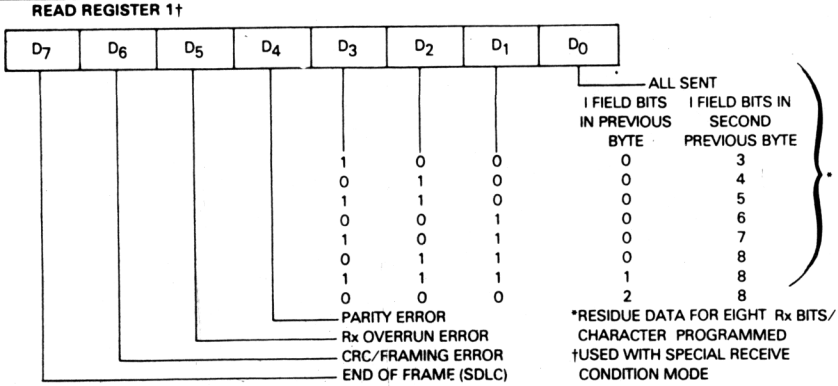
When the $\overline{SYNC}$ input goes High again, another External/Status interrupt is generated that must also be cleared. The Enter Hunt Mode control bit is set whenever character synchronization is lost or the end of message is detected. In this case, the Z80-SIO again looks for a High-to-Low transition on the $\overline{SYNC}$ input and the operation repeats as explained previously. This implies the CPU should also inform the external logic that character synchronization has been lost and that the Z80-SIO is waiting for $\overline{SYNC}$ to become active.

READ REGISTER BIT FUNCTIONS

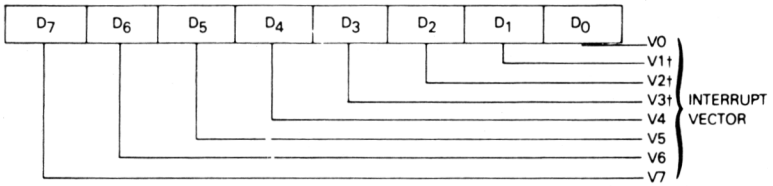
READ REGISTER 0



READ REGISTER 1†



READ REGISTER 2



†VARIABLE IF "STATUS AFFECTS VECTOR" IS PROGRAMMED

In the Monosync and Bisync Receive modes, the Sync/Hunt status bit is initially set to 1 by the Enter Hunt Mode bit. The Sync/Hunt bit is reset when the Z80-SIO establishes character synchronization. The High-to-Low transition of the Sync/Hunt bit causes an External/Status interrupt that must be cleared by the CPU issuing the Reset External/Status Interrupt command. This enables the Z80-SIO to detect the next transition of other External/Status bits.

When the CPU detects the end of message of that character and synchronization is lost, it sets the Enter Hunt Mode control bit, which—in turn—sets the Sync/Hunt bit to 1. The Low-to-High transition of the Sync/Hunt bit sets the External/Status interrupt, which must also be cleared by the Reset External/Status Interrupt command. Note that the SYNC pin acts as an output in this mode and goes Low every time a sync pattern is detected in the data stream.

In the SDLC mode, the Sync/Hunt bit is initially set by the Enter Hunt mode bit or when the receiver is disabled. In any case, it is reset to 0 when the opening flag of the first frame is detected by the Z80-SIO. The External/Status interrupt is also generated and should be handled as discussed previously.

Unlike the Monosync and Bisync modes, once the Sync/Hunt bit is reset in the SDLC mode, it does not need to be set when the end of message is detected. The Z80-SIO automatically maintains synchronization. The only way the Sync/Hunt bit can be set again is by the Enter Hunt Mode bit or by disabling the receiver.

Clear to Send (D_5). This bit is similar to the DCD bit, except that it shows the inverted state of the $\overline{CTS}$ pin.

Transmit Underrun/End of Message (D_6). This bit is in a set condition following a reset (internal or external). The only command that can reset this bit is the Reset Transmit Underrun/EOM Latch command (WR0, D_6 and D_7). When the Transmit Underrun condition occurs, this bit is set, its becoming set causes the External/Status interrupt, which must be reset by issuing the Reset External/Status Interrupt command bits (WR0). This status bit plays an important role in conjunction with other control bits in controlling a transmit operation. Refer to "Bisync Transmit Underrun" and "SDLC Transmit Underrun" for additional details.

Break/Abort (D_7). In the Asynchronous Receive mode, this bit is set when a Break sequence (null character plus framing error) is detected in the data stream. The External/Status interrupt, if enabled, is set when Break is detected. The interrupt service routine must issue the Reset External/Status Interrupt command (WR0, CMD_2) to the break detection logic so the Break sequence termination can be recognized.

The Break/Abort bit is reset when the termination of the Break sequence is detected in the incoming data stream. The termination of the Break sequence also causes the External/Status interrupt to be set. The Reset External/Status Interrupt command must be issued to enable the break detection logic to look for the next Break sequence. A single extraneous null character is present in the receiver after the termination of a break, it should be read and discarded.

In the SDLC Receive mode, this status bit is set by the detection of an Abort sequence (seven or more 1's). The External/Status Interrupt is handled the same way as in the case of a Break. The Break/Abort bit is not used in the Synchronous Receive mode.

READ REGISTER 1

This register contains the Special Receive condition status bits and Residue codes for the I-field in the SDLC Receive Mode.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
End of Frame (SDLC)	CRC/ Framing Error	Receiver Overrun Error	Parity Error	Residue Code 2	Residue Code 1	Residue Code 0	All Sent

All Sent (D_0). In Asynchronous modes, this bit is set when all the characters have completely cleared the transmitter. Transitions of this bit do not cause interrupts. The bit is always set in Synchronous modes.

Residue Codes 0, 1, and 2 (D_1-D_3). In those cases of the SDLC receive mode where the I-field is not an integral multiple of the character length, these three bits indicate the length of the I-field. These codes are meaningful only for the transfer in which the End Of Frame bit is set (SDLC). For a receive character length of eight bits per character, the codes signify the following:

Residue Code 2	Residue Code 1	Residue Code 0	I-Field Bits In Previous Byte	I-Field Bits In Second Previous Byte
1	0	0	0	3
0	1	0	0	4
1	1	0	0	5
0	0	1	0	6
1	0	1	0	7
0	1	1	0	8
1	1	1	1	8
0	0	0	2	8
I-Field bits are right-justified in all cases				

If a receive character length different from eight bits is used for the I-field, a table similar to the previous one may be constructed for each different character length. For no residue (that is, the last character boundary coincides with the boundary of the I-field and CRC field), the Residue codes are:

Bits per Character	Residue Code 2	Residue Code 1	Residue Code 0
8 Bits per Character	0	1	1
7 Bits per Character	0	0	0
6 Bits per Character	0	1	0
5 Bits per Character	0	0	1

Parity Error (D_4). When parity is enabled, this bit is set for those characters whose parity does not match the programmed sense (even/odd). The bit is latched, so once an error occurs, it remains set until the Error Reset command (WR0) is given.

Receive Overrun Error (D_5). This bit indicates that more than three characters have been received without a read from the CPU. Only the character that has been written over is flagged with this error, but when this character is read, the error condition is latched until reset by the Error Reset command. If Status Affects Vector is enabled, the character that has been overrun interrupts with a Special Receive Condition vector.

CRC/Framing Error (D_6). If a Framing Error occurs (asynchronous modes), this bit is set (and not latched) for the receive character in which the Framing error occurred. Detection of a Framing Error adds an additional one-half of a bit time to the character time so the Framing Error is not interpreted as a new start bit. In Synchronous and SDLC modes, this bit indicates the result of comparing the CRC checker to the appropriate check value. This bit is reset by issuing an Error Reset command. The bit is

not latched, so it is always updated when the next character is received. When used for CRC error and status in Synchronous modes, it is usually set since most bit combinations result in a non-zero CRC, except for a correctly completed message.

End of Frame (D_7). This bit is used only with the SDLC mode and indicates that a valid ending flag has been received and that the CRC Error and Residue codes are also valid. This bit can be reset by issuing the Error Reset command. It is also updated by the first character of the following frame.

READ REGISTER 2 (Ch. B Only)

This register contains the interrupt vector written into WR2 if the Status Affects Vector control bit is not set. If the control bit is set, it contains the modified vector shown in the Status Affects Vector paragraph of the Write Register 1 section. When this register is read, the vector returned is modified by the highest priority interrupting condition at the time of the read. If no interrupts are pending, the vector is modified with V_3=0, V_2=1, and V_1=1. This register may be read only through Channel B.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
V_7	V_6	V_5	V_4	V_3	V_2	V_1	V_0
				Variable if Status Affects Vector is enabled			

Index

Z80 APPLICATIONS

Now you can develop your own applications using the Z80 microprocessor. This book offers easy reading and clear illustrations providing you with the necessary instructions for using your Z80 to control peripheral LSI devices.

Numerous diagrams and examples provide information on the use of:

- ROM and Static RAM with the Z80
- Input and Output devices
- Dynamic RAMs
- Interrupts
- peripheral devices including Z80-SIO, PIO, and CTC

Read this book and see how easily you can design your own system to suit your own specific needs.

About the Author

James W. Coffron is a computer systems engineer specializing in the design and testing of microprocessor-based systems. He has taught seminars in both academic and industrial settings, and is the author of several books about microprocessors. He holds an MSEE degree from Santa Clara University.

ISBN 0-89588-094-6

James W. Coffron

Z80 APPLICATIONS