OSBORNE/McGraw-Hill

interfacing to S-100/IEEE 696 microcomputers

Sol Libes
Mark Garetz

interfacing to S-100/IEEE 696 microcomputers

interfacing to S-100/IEEE 696 microcomputers

Sol Libes
Mark Garetz

OSBORNE/McGraw-Hill
Berkeley, California

Published by
OSBORNE/McGraw-Hill
630 Bancroft Way
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write OSBORNE/McGraw-Hill at the above address.

INTERFACING TO S-100/IEEE 696 MICROCOMPUTERS

1234567890 DODO 8987654321

ISBN 0-931988-37-3

Technical editor for this book was Curtis A. Ingraham.
Copy editor was Erfert Nielson.
A technical review was completed by Lee Felsenstein.

Cover design by Marc Miyashiro.

acknowledgements

Mark Garetz wishes to thank Richard Frank of Sorcim for midnight software advice, Bill Godbout and the engineering staff of the CompuPro Division of Bill Godbout Electronics for many design ideas, and Howard Fullmer and George Morrow for starting the S-100/IEEE 696 Standard effort.

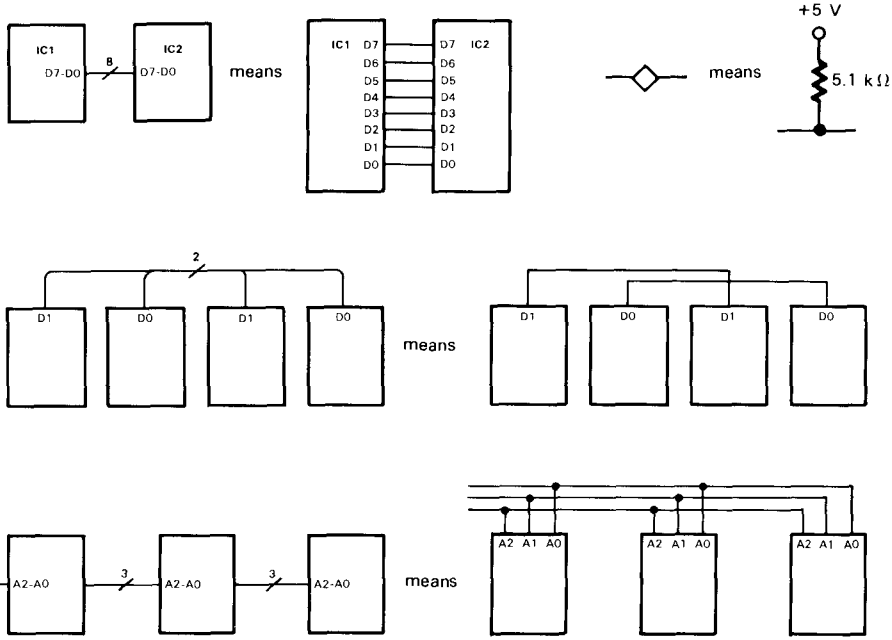
Sol Libes gratefully acknowledges the assistance of the following individuals: Dr. Alan Katz of Trenton State College; Dr. Robert Stewart, Chairman of the IEEE Computer Committee; Howard Fullmer, Chairman of the IEEE 696/S-100 Standard Committee; Lee Felsenstein of Osborne Computer Corporation; Curtis Ingraham and Denise Penrose of Osborne/McGraw-Hill.

contents

preface

SCHEMATIC SYMBOLS

The schematic symbols shown on the left in the following diagram are the ones used in this book.



BACKGROUND

For an in-depth discussion of the 8080, 8085, and Z80 microprocessors, hardware, timing, peripheral ICs, etc., it is recommended that the reader consult *An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors* and *Volume 3 — Some Real Support Devices,* by Adam Osborne (Berkeley: Osborne/McGraw-Hill, 1980).

For an in-depth discussion of the 8080A/8085 and Z80 instruction sets and programming techniques, the reader should consult *8080A/8085 Assembly Language Programming* and *Z80 Assembly Language Programming,* by Lance Leventhal (Berkeley: Osborne/McGraw-Hill, 1979).

The purpose of this volume is to build on the foundation presented in those books and show numerous examples of actual interfaces to the S-100 Bus.

A word of caution: The hardware and software examples shown are for illustrative purposes only and are not intended to necessarily be the most efficient or practical for any given application.

introduction

In early 1975 MITS Inc., an Albuquerque, New Mexico electronics manufacturer (since purchased by Pertec Computer Corporation), introduced the ALTAIR-8800 microcomputer, using the 8080 microprocessor. The unit was an immediate success and many manufacturers subsequently used the Altair Bus in their computers. These manufacturers renamed the bus the S-100 Bus, since it used 100 signal lines. Today there are over 200,000 S-100 systems in operation.

S-100 systems are so popular because of the many advantages they offer users. Here are some of those advantages:

1. *Processor independence.* There are presently manufacturers of eight different 8-bit CPU boards (8080A, 8085, Z80, 2650, 6502, 6800, 6802, and 6809) and seven different 16-bit CPU boards (9900, LSI-11-like, 8086, 8088, Z8000, 68000, and Pascal Microengine) for S-100 systems. No other computer system offers such a variety of CPU architectures.
2. *More software available.* There are several times as many languages, operating systems, and applications packages for S-100 based systems as there are for any other computer system.
3. *Greater hardware support.* There are close to 100 different manufacturers of about 400 different plug-in boards for S-100 systems. This is far greater than for any other computer system.
4. *Greater computer power.* S-100 systems offer the greatest computer power of any microcomputer system. No other microcomputer system has direct addressing of up to 16 Mbytes of memory (24 address lines) and 64 K I/O ports (16 address lines), up to 10 vectored

interrupt lines, up to 16 masters on the bus (with priority), up to 23 plug-in slots on the motherboard, up to 10 MHz data transfer rate, plug-in operator front panel, and more.

5. *Standardization.* The S-100 Bus most likely will be standardized by publication date. This assures a very high degree of compatibility among future plug-in circuit boards from different manufacturers.

The S-100 Bus thus offers a combination of hardware and software power, flexibility, and economy far greater than any other system currently available. This book is written to assist S-100 system users in expanding the power and utility of their systems.

how to use this book

1

WHAT THIS BOOK IS ABOUT

This book exists for two distinct purposes. One is to give the user of S-100 systems an understanding of how S-100 hardware systems function. This we hope to accomplish by explaining in simple terms exactly what is happening on the bus — what the signals mean and their relationships to one another. We will also present a large number of circuits and explain their operation. Similar circuits can be found in all S-100 systems.

Secondly, this book gives the builder a set of tools with which to design and build custom S-100 circuits. To do so, the builder must be able to understand all the things mentioned above. We will present a lot of "cookbook" type circuits that may be combined to form simple and complex S-100 interfaces. We will also explain why certain things need to be done in certain ways, due to the nature of the S-100 Bus.

This book does *not* attempt to show you how to build your own computer from the ground up. We assume that you already own an S-100 system (or are thinking of purchasing one) and wish to expand the system internally or externally via interfaces.

WHAT WE ASSUME YOU KNOW

This book would have to be several volumes if we had to explain everything about digital logic, computers, programming, and construction techniques. Therefore we are going to assume that you already possess certain knowledge. If

you don't possess that knowledge, then we will recommend texts that you may consult.

In order to make full use of this book, we assume —

1. you understand standard logic functions such as AND, NAND, OR, NOR, XOR and the like. As a corollary to this we assume you know how to find these functions in standard TTL family gate packages, such as 74LS00, 74LS08, etc.
2. you understand some basic electronic concepts. For example, the term "signal line" shouldn't throw you. We will also assume that you know how to read a schematic diagram.
3. you have some basic knowledge of computers and programming. We expect you to be able to follow the flow of a program. We also assume you are familiar with assembly language programming, at least to the point of knowing what a label is and being able to look up machine codes in a table of op-code mnemonics (especially if you're going to attempt hand assembly).
4. you have the technical skills to build any of the circuits in this book, if you intend to do so. If you still haven't figured out how the pins on an IC are numbered, you'd better buy all your S-100 boards from a manufacturer that has already done the designing and building for you. We will also assume that you are capable of taking two different functional blocks, presented in schematic form, and combining them to perform a more complex function.

A WORD ABOUT SOFTWARE

Very little computer hardware is useful without some software to go with it. In this book, we will give you software routines to go with the hardware. They will be written in 8080 assembler code. Many other processors are available for the S-100 bus, and not all of them execute 8080 code. Therefore we will also present a flowchart for each non-trivial program so that you may readily convert the code for your particular processor.

LOGICAL AND ELECTRICAL STATE RELATIONSHIPS

The IEEE standard uses a new notation for logical and electrical state relationships which is convenient with word processors. The overbar is no longer used to indicate an active low signal state because it is difficult for many word processors to type it. The suffix "*" is used instead to designate that an electrical signal is

active low. For example, the S-100 signal "sWO*" is the status signal that indicates a write cycle, and is active (true) when the sWO* line is low. The presence or absence of the asterisk suffix describes the electrical implementation of a signal.

The minus sign (–) prefix is used in logic equations to indicate the logical (Boolean) negation operator. It is equivalent to the use of an overbar. For example, "–sOUT" indicates that the logical signal sOUT is negated (inverted), irrespective of the electrically active state of the sOUT signal line. –sOUT would therefore indicate that the S-100 Bus is not in an sOUT state.

The "*" is pronounced "star" and "–" is pronounced "not."

THE PITFALLS OF BUILDING YOUR OWN CIRCUITS

A word or two of caution is needed here if you intend to construct some of the circuits presented in this book. Most, if not all, of the functions that we will present are obtainable on products already commercially available. Therefore, it is not always necessary to build the circuit yourself.

There are many reasons people choose to build their own circuits: money, experience, and pride, to name a few. It is more than likely you won't save much money by building your own. The components cost is often higher for a homemade project than for a commercially available one.

Then there is the problem of debugging. There is always the chance that you will make an error in construction that will cause you to spend many long and frustrating hours trying to figure out why your circuitry doesn't work, and of course there's no warranty or manufacturer to help you.

There is also the problem of time. If you have lots of free time to tinker with circuits, then time may not be a problem for you. But if you are like most of us, chances are your spouse thinks you spend too much time with the computer already. We don't want to be responsible for breaking up any marriages. (Imagine how much time it would take to wire-wrap a 32 K memory board — that's 64 chips, each with 18 pins, each of which needs a wire, which makes 1152 connections; and then there's the support circuitry, connections to the bus... Get the picture?)

If you're the kind of person who just can't be satisfied unless you can proudly say, "I built it all myself!" or if you are looking to gain some valuable hands-on experience with computer hardware, then start warming up your soldering iron.

The last reason that you may wish to build your own hardware is that you may need just one little circuit, but a manufacturer's product gives you 16 of the same or throws in a lot of extra features you don't need. Then, by all means, build just what you need.

Please don't be put off by this lecture, but having been through it ourselves, we thought we owed you the warning. Good luck!

USEFUL EQUIPMENT TO HAVE

You will need the basics: soldering iron, wire-wrap tool, pliers, screwdrivers, multi-meter, and the like. But some other equipment might be useful to have around.

There is nothing like having access to a multiple trace oscilloscope with at least 10 MHz bandwidth, but you may be able to get by with a logic probe. Having an S-100 extender board (something that elevates the board under test above the level of the rest of the boards, while still leaving it connected to the bus) will be invaluable. Mullen Computer Products makes an extender board that also has a built-in logic probe. This is probably one of the most valued possessions of the avid hardware type and is a great timesaver.

One of the most useful products on the market is a calculator called the TI Programmer. It is manufactured by Texas Instruments and can do arithmetic not only in decimal, but in hexadecimal and octal too. It's worth the price just to be able to convert a decimal number to hex with a single keystroke. Not only does it perform the four standard arithmetic functions, but it also does ANDs, ORs, XORs, shifts, and one's and two's complements. It's really a handy device to have around.

PROTOTYPING BOARDS

If you intend to construct any of the circuits in this book, you will have to do it on a product known as a "prototyping board" (proto board). There are many manufacturers of S-100 proto boards and each proto board is different. Some have etched areas for voltage regulators (some even have some bus interface circuitry), some have power and ground traces gridding the board, some have no traces at all, and some are just plain copper laminate so you can etch your own traces. Choose the board design you like best by comparing them to one another. If you have no way of doing that, Vector Electronics makes a complete range of S-100 proto boards.

the S-100 bus

2

A BRIEF HISTORY OF THE S-100 BUS

This whole thing got started in late 1974 when the Mark 8 series of computer construction articles appeared in *Radio-Electronics* magazine. It was the first time a computer had been put within reach of anyone but a large company. The Mark 8 was based on the Intel 8008 and was not an S-100 Bus computer. The response to the articles was tremendous.

Not wanting to fall behind the competition, Les Solomon, an editor of *Popular Electronics,* decided that it too should have a computer article. He suggested to Ed Roberts, then the president of a small company called MITS, that they come up with a computer kit. Ed agreed, but they decided to base the computer on Intel's new 8080 chip. Thus the MITS Altair was born. The first Altair article appeared in the January 1975 issue of *Popular Electronics.* It had a bus that used a 100-pin edge connector (it was chosen because MITS made a good surplus buy on them). It was called the Altair Bus.

Being one of the first microcomputers out, the MITS Altair had many shortcomings. A company called IMS Associates, Incorporated (IMSAI) decided that they could do a much better job than MITS had done, and thus the IMSAI 8080 was born. Luckily, IMSAI decided to "second source" the Altair and designed their computer with the same bus.

Many other companies sprang up advertising add-on boards for both computers, so the bus became known as the Altair/IMSAI Bus. More companies decided to jump on the bandwagon with bus-compatible computers, but each wanted to tack its name onto the bus name as well. The situation was getting out of hand.

Roger Mellen, one of the principals of a then small company named Cromemco, decided that a generic name was needed for the bus. His idea was to call it the "Standard 100" Bus, or S-100 for short, because it had 100 pins. The name caught on.

THE IEEE STANDARD FOR THE S-100 BUS

All the various S-100 manufacturers had adhered to the bus pinout fairly well, with only minor variations. Although the signal names were the same, the timing relationships varied from manufacturer to manufacturer. This created lots of problems for people trying to get Brand X board to work with Brand Y board, etc. Something had to be done.

A few S-100 designers decided to write up a proposed standard and submit it to the Institute of Electrical and Electronic Engineers, Inc. (IEEE) standards committee for approval. They wrote up a proposed standard and passed it around to a number of S-100 manufacturers for comments. Many were made and many were incorporated into the proposal. The IEEE agreed to the proposal for a standard and an S-100 Standard Committee was formed.

The S-100 committee made several significant additions to their draft standard for the S-100 Bus, which we will cover in detail later. But their most important accomplishment was that there now existed a *reference* that not only defined the signals present on the bus, but also specified their timing relationships to one another. A draft of the proposed standard, "Standard Specification for S-100 Bus Interface Devices," IEEE Task 696.1/D2, was published in the July 1979 issue of *Computer* magazine. The document which the IEEE ultimately approves is certain to contain changes. As this book goes to press (May 1981) the IEEE 696/S-100 standard has still not been approved. The final meeting of the standard committee will take place on June 30, 1981. This book will follow the draft standard, but incorporate changes in terminology (such as DMA to TMA) that have met with no resistance from committee members and are likely to be included in the final version of the standard.

One of the main criteria of the committee was that any change or upgrade they made to the S-100 Bus should not make obsolete any existing S-100 boards. However, they did make a few changes, and we will explain the differences between the old and the new as we encounter them.

By now the standard is becoming known by its IEEE task number — 696. The phrase "IEEE 696 compatible" may soon replace the phrase "S-100 compatible."

Please note that throughout this text we use the terms "the IEEE Standard," "the IEEE S-100 Standard," "the S-100 Standard," and "the standard." By all these names we mean the same thing: the proposed standard.

MECHANICAL DESCRIPTION

The standard S-100 circuit board is 10 inches wide, typically 5.3 inches high and 0.062 inches thick. A typical board is shown in Figure 2-1. It connects to the motherboard via a 100-pin edge connector. The pins are spaced on 0.125 inch centers. The connector is offset so that with the connector down and the component side of the board facing you, the edge connector is to the left of center, as shown. This helps to prevent the card from being inserted backwards into the bus connector.

Note that connections 1 through 50 are on the component side of the board (with pin 1 on the left) and pins 51 through 100 are on the solder side of the board (with pin 51 directly behind pin 1).

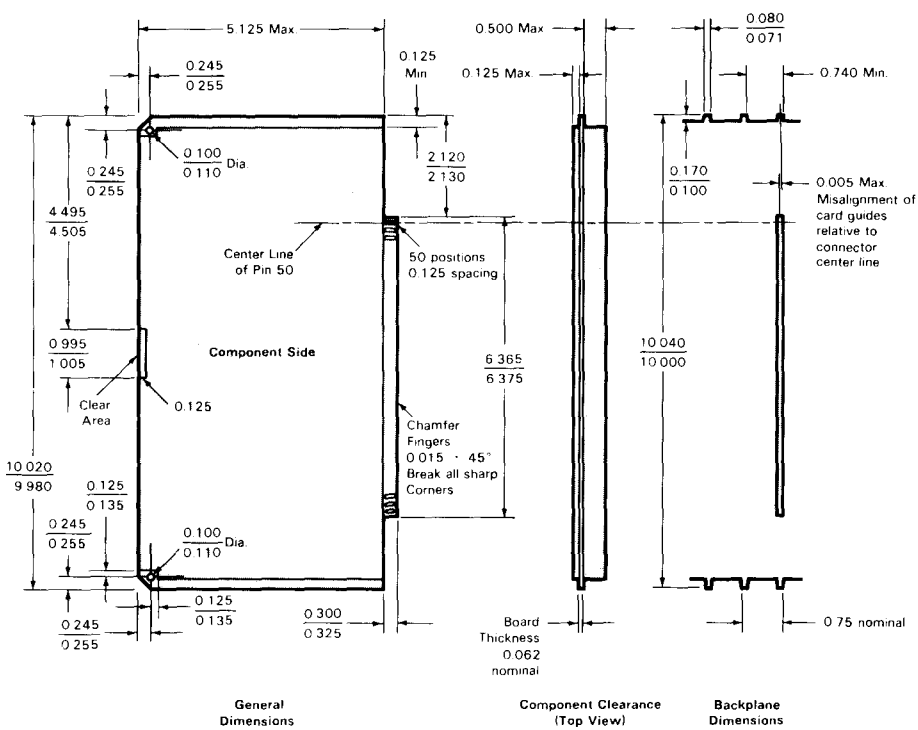


FIGURE 2-1. S-100 Plug-In Board Mechanical Specifications (Courtesy *Computer* Magazine)

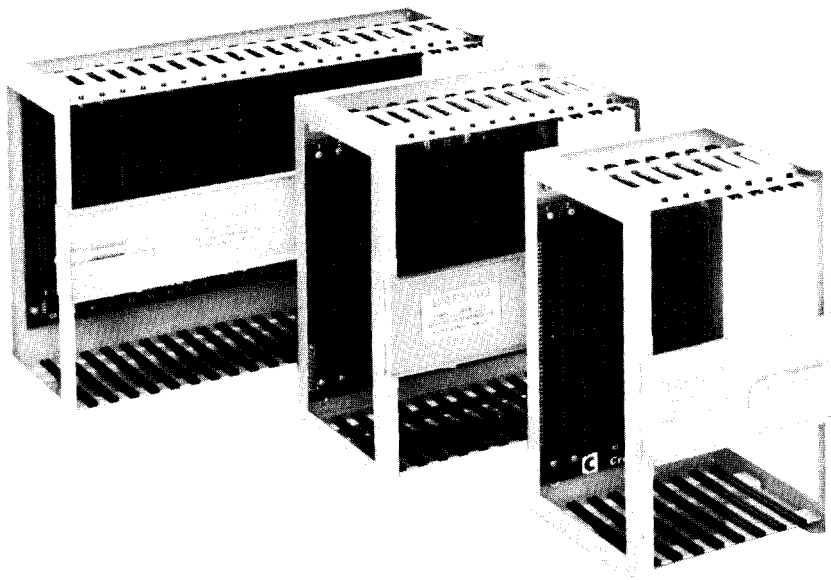


FIGURE 2-2. Typical S-100 Motherboards and Card Frames (Courtesy Cromemco)

THE MOTHERBOARD

The 100 pins of the S-100 boards are interconnected by plugging the boards into a "motherboard." In its simplest form, an S-100 motherboard is a printed circuit board consisting of 100 parallel traces connected to any number of 100-pin connectors. The connectors are soldered to the motherboard and the S-100 boards plug into the connectors. A typical S-100 motherboard is shown in Figure 2-2.

More advanced motherboards contain ground traces that are interleaved between the signal traces to minimize noise and crosstalk. Crosstalk is a signal induced on one trace by a signal in a nearby trace. Most newer motherboards also contain "termination circuitry" on all lines except power and ground, which helps to eliminate ringing. Ringing is a fluctuating signal amplitude which occurs when a signal is reflected from one end of the motherboard to the other. The termination circuitry also ensures that any line left unconnected will always be in an

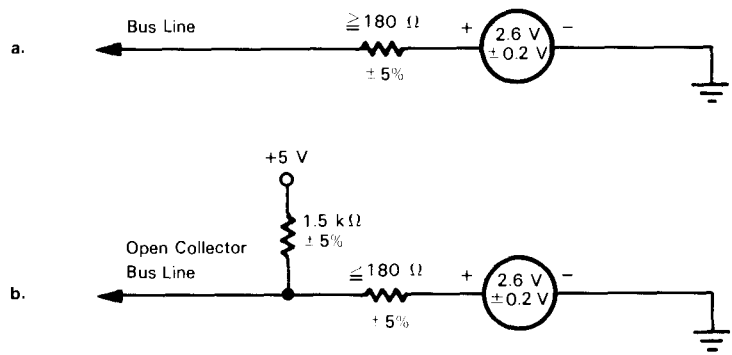


FIGURE 2-3. S-100 Bus Termination Circuits

electrically high state, rather than a random one. A block diagram of recommended termination circuitry is shown in Figure 2-3.

Note that the term "motherboard" is actually a misnomer. As used in S-100 Bus systems, the motherboard should actually be called a "backplane," but since everyone calls it a motherboard, we will too.

BUS MASTERS AND SLAVES

The IEEE Standard has divided the types of S-100 devices into two different groups: bus masters and bus slaves. Basically, a master is a device that is in control of the bus (such as a CPU board) and a slave is a device that a master controls (such as a memory board).

Bus masters can be either of two types: the permanent bus master or a temporary bus master. Every S-100 system must have a permanent bus master. The permanent bus master always provides certain signals to the bus (this will be discussed in more detail in the next chapter).

In addition, an S-100 system may have up to 16 temporary bus masters. A temporary bus master (such as a TMA controller) can request control of the bus from the permanent master. If the permanent master grants control, the temporary master can then access bus slaves. When the temporary master has completed its task, it will pass control back to the permanent master.

This whole process of transferring control is quite involved, so for now just assume your system has only a permanent bus master. We will explain the whole process in Chapter 15 under Direct Memory Access.

In most S-100 systems the permanent master is the CPU board.

The other type of S-100 device is the slave. Slaves generally accept data from and send data to the bus master. What they do with the data or what kind of data they have collected is up to your imagination. Typical bus slaves would be memory boards, serial I/O interfaces, parallel I/O interfaces, etc. In fact, most of the circuits presented in this book will be bus slaves.

S-100 SIGNAL GROUPS

Now we will begin to explain the different groups of S-100 signals. We have divided them up by function. This is intended to be a brief overview, so we will not be giving you exact specifications such as pin numbers, instantaneous maximum power requirements, etc. These will be covered in detail in the next chapter.

These signals are grouped by the standard into eight categories. In this book, our groupings of the signals will be essentially the same as that of the standard, but will have a few minor variations. This is because the standard groups the signals from a purely technical standpoint, and assumes the reader already understands their basic functions. We have grouped them differently to make them easier for you to understand.

There are eight bus groups:

Function	No. of Lines Used
1. Address	16 or 24
2. Data	16
3. Status	9
4. Control Output	5
5. Control Input	6
6. Interrupt	10
7. TMA Control	8
8. Utility	22

The Address Bus

The address bus, shown in Figure 2-4, is used by the system to determine where memory and I/O devices are located in the "address space."

The old S-100 Bus had 16 address lines for memory (A0-A15). The standard specifies eight additional memory address lines, bringing the total to 24 (A0-A23). With 16 address lines, the maximum amount of directly addressable

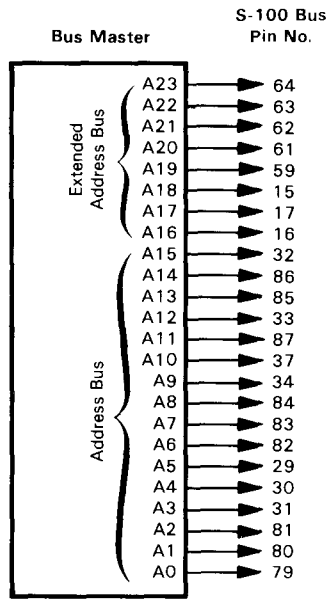


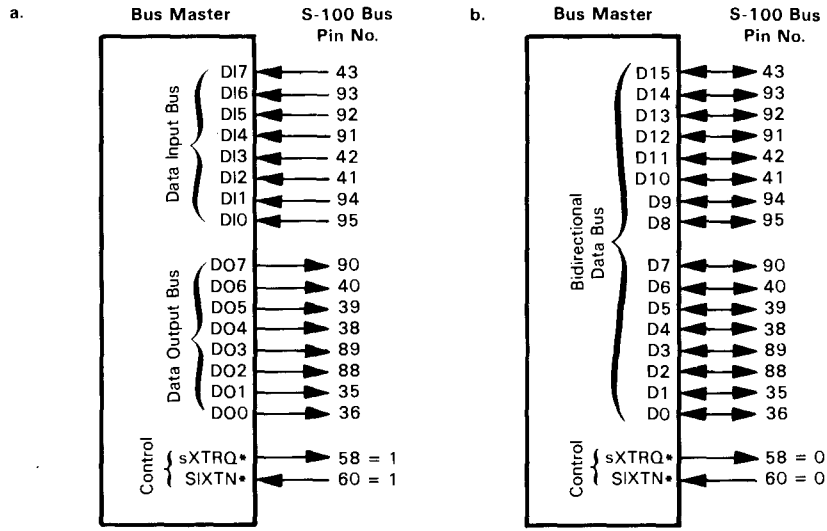
FIGURE 2-4. The S-100 Address and Extended Address Bus

memory was 65,536 bytes. Now the total is 16,777,216 bytes, or approximately 16 megabytes. Upper-case K is often used in the computer field to represent the number $2^{10} = 1024$. Thus 65,536 is equivalent to 64 K, because $65,536 = 2^{16} = 2^6 \times 2^{10} = 64$ K. Do not confuse upper-case K (2^{10}) with lower-case k, which always means 10^3.

The old S-100 Bus also used only eight lines for I/O addresses (A0-A7), and therefore could address only 256 I/O ports. The standard states that a system may utilize 16 I/O address lines (A0-A15), bringing the number of ports to 65,536. The address lines originate from the current bus master.

The Data Bus

The S-100 Bus has two 8-bit data paths, as shown in Figure 2-5. One is called the data output bus and the other is called the data input bus. The direction of data flow is always relative to the current master. Therefore, the data output bus would carry data going from the master to a slave (as while writing to memory), and the data input bus would contain data going to a master from a slave (as while reading from memory).



a. 8-bit unidirectional buses when sXTRQ* = 1 and SIXTN* = 1
b. 16-bit bidirectional bus when sXTRQ* = 0 and SIXTN* = 0

FIGURE 2-5. The S-100 Data Bus

When the S-100 Bus was first designed, only 8-bit CPUs were envisioned. But the standard now provides for 16-bit CPUs. In this case, the data input and data output buses are combined to form a 16-bit data bus that is bidirectional. That means that data can flow in either direction on the bus (the direction depends on the type of cycle being executed). Two new signals were defined to tell slaves and masters whether to use 8- or 16-bit data buses. These two new signals are SIXTN* and sXTRQ*. We will talk more about them later.

The Status Bus

There are eight status lines on the S-100 Bus. From the information contained on these lines, any device in the system may determine what kind of bus cycle the master is currently in. For example, the line sMEMR is a status signal that says the current cycle is a memory read cycle. Bus cycles will be discussed in later chapters.

A slave device can look at the address and status lines to determine if it should respond.

The mnemonics for status lines begin with a lower-case s. Since the S-100

Bus was originally used with the Intel 8080 microprocessor, many of the status and control line mnemonics come from the Intel data sheet. The status lines are:

sMEMR	Memory read
sM1	Op-code fetch
sOUT	Output
sINP	Input
sWO*	Memory or output write
sINTA	Interrupt acknowledge
sHLTA	Halt acknowledge
sXTRQ*	16-bit data transfer request

The Control Output Bus

The control output bus is a group of signals that tell the system *when* to do things. They are also known as "strobes." They differ from the previous signals (address, data, status) in that these other signals contain *what* information. For example, during a memory read cycle the address bus contains an address that specifies *what* memory board, the status bus contains information saying that it's a memory read cycle (*what* kind of cycle), but the control output bus line pDBIN tells the memory board *when* to place the data on the bus.

There are five control output signals with the mnemonics all beginning with a lower-case p. They are:

pSYNC	Indicates start of new bus cycle
pDBIN	Read strobe
pWR*	Write strobe
pHLDA	Hold acknowledge
pSTVAL*	Address and status stable on bus

All the signals are generated by the current bus master with the exception of MWRT. MWRT is generated from pWR* and sOUT, usually by a circuit on the permanent master card.

The control output bus lines originate from the current master.

The Control Input Bus

The six signals on the control input bus are generated by slave devices to tell the current master to do something — wait, for example. Two signals, RDY and XRDY, do just that. They tell the master to extend the current bus cycle.

The HOLD* signal tells the permanent master to stop what it's doing and relinquish the bus to another master. The INT* signal is an interrupt request to the master from a slave, while NMI* is a similar, but non-maskable, interrupt request.

The last control input signal is SIXTN*. This signal tells the current master that the current slave is capable of accepting 16-bit data. The control input bus signals originate from the various slave devices on the bus.

The Interrupt Bus

There are 10 lines on the S-100 Bus used to interrupt the master's current task and cause it to do another task — look at the keyboard of a terminal, for example. They are used mainly in multi-user environments or in critical real-time applications. The interrupts may originate from any slave device.

The device that wants to look at the interrupts can usually ignore them by a technique known as "masking," but there is one interrupt that can't be masked. It's called NMI* (for Non-Maskable Interrupt). NMI* is usually used to signal impending doom, such as an imminent power failure.

The TMA Control Bus

The TMA control signals control the transfer of the bus from the permanent master to a temporary master and vice versa. The timing relationships and concepts behind the operation of the TMA lines are difficult to understand unless you fully understand how the rest of the bus signals work. Therefore, we are not going to explain them until Chapter 15.

Note: TMA means Temporary Master Access. It is the process whereby an S-100 temporary master can access a slave (such as a memory board) without having to use the permanent master's program. Thus it gains direct access to the slave, as opposed to the permanent master accessing the slave via a program.

The Utility Bus

The power supply lines and all the other signals on the bus that don't really fit into any other single category are all lumped together at the end. However, these signals are very important to the operation of the system. RESET* is one of these signals, along with SLAVE CLR* and POC* (Power-On-Clear).

Also included is the system clock (Φ) signal. All the S-100 Bus timing is relative to this master clock signal. Other signals are CLOCK, a 2 MHz clock; MWRT, a memory write strobe; and PHANTOM*, a line used for bootstrapping.

S-100 computers all require three unregulated voltages. They are +8 V +16 V, and −16 V. They are all referenced to a common ground. Note that these voltages are nominal values.

All of these lines and signals will be discussed in detail in the next chapter.

POWER SUPPLY INTERFACING

DC power is distributed on the S-100 Bus system as unregulated +8 V, +16 V, and −16 V with respect to ground. Voltage regulator circuitry is generally employed on each S-100 plug-in card to provide the specific voltages required and to provide regulation of the voltages. For example, the +5 V required for the TTL logic ICs is derived from the +8 V bus lines. This is typically done using fixed voltage IC regulators. For lower current applications that do not require critical regulation a zener diode type regulator circuit may be employed. The following are considerations, guidelines, and examples for building these regulator circuits.

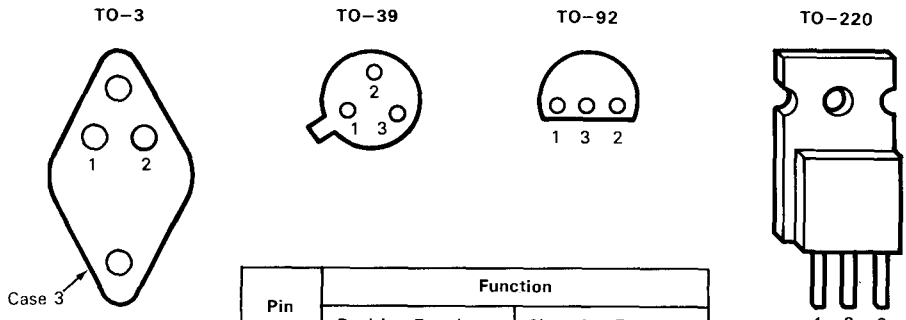
Fixed Voltage IC Regulators

The most popular families of regulator ICs currently in use are the 78XX (positive voltage) and 79XX (negative voltage) families and their LM family equivalents. The "XX" in the device number represents the characters used to denote the output voltage rating of the device. For example, the 7805 and LM340-05 are +5 V regulators.

Most of the IC regulators have internal thermal protection and current limiting circuits. Each IC regulator is specified for a given output voltage, minimum input-to-output voltage drop, and maximum output current. Table 2-1 gives the specifications of the more popular devices. For example, the 78XX series is a positive voltage regulator capable of handling up to 1 amp with a minimum voltage drop of 2 V from input to output. The 78XX series is available in either the TO-3, TO-39, TO-92, or TO-220 packages, as shown in Figure 2-6. Furthermore, the 78XX series is available in +5 V, +6 V, +8 V, +12 V, +15 V, +18 V or +24 V

TABLE 2-1. Positive and Negative Voltage IC Regulators

	Type	Output Voltages	Maximum Output Current	Case
Positive Voltage Regulators	78XXCK/LM340K-XX	5,6,8,12,15,18,24	1.5 A	TO-3
	78XXUC/LM340T-XX	5,6,8,12,15,18,24	1 A	TO-220
	78MXXHC/LM341H-XX	5,6,8,12,15,18,24	500 mA	TO-39
	78MXXUC/LM341P-XX	5,6,8,12,15,18,24	500 mA	TO-220
	78LXXHC/LM78LXXCH	5,6,12,15	100 mA	TO-39
	78LXXWS/LM78LXXCZ	5,6,12,15	100 mA	TO-92
	78LXXAHC/LM78LXXACH	5,6,12,15	100 mA	TO-39
	78LXXAWC/LM78LXXACZ	5,6,12,15	100 mA	TO-92
Negative Voltage Regulators	79XXCK/LM320K-XX	2,5,5.2,6,8,12,15,18,24	1 A	TO-3
	79XXUC/LM320T-XX	2,5,5.2,6,8,12,15,18,24	1 A	TO-220
	79MHXX	5,6,8,12,15,18,24	500 mA	TO-39
	79LWXX	5,12,15,18,24	100 mA	TO-92
	79LHXX	5,12,15,18,24	100 mA	TO-39



Pin	Function	
	Positive Regulator	Negative Regulator
1	In	Common
2	Out	Out
3	Common	In

FIGURE 2-6. Pin Diagrams for Regulator Packages

output voltage ratings. Hence, the 7805UC or LM340T-05 is a +5 V, 1 amp voltage regulator, in a TO-220 package.

For the +5 V power supply the typical practice is to use one 7805UC IC regulator circuit for each circuit drawing up to 750 mA of current. Therefore, on a plug-in card where 3 amps are drawn from the +8 V supply, four regulator circuits would be used with the loads divided equally among them (but five regulators would be a more conservative design).

A typical +5 V, 1 amp regulator circuit is shown in Figure 2-7. Capacitors C1 and C2 are Tantalum capacitors used to reduce noise. Capacitor C3 improves the transient response of the circuit under high current changes. C3 is typically selected to have 100 μF per output ampere. Therefore, in this case a 100 μF, 10 V capacitor was chosen for a 1 amp supply. These capacitors should be located near the regulator for optimum regulation.

A typical +12 V and −12 V regulator circuit is shown in Figure 2-8. Note that pins 1 and 3 of the negative voltage regulator, LM320K-12, are opposite that of the pins of the positive voltage regulator, LM340K-12.

For reliable operation it is important that the regulators remain cool. The +5 V regulators passing 1 amp of current must dissipate 3 watts each. This requires an adequate heat sink to conduct the heat away from the regulator IC. The TO-3 case has the lower thermal resistance, transferring more heat to the heat sink than the TO-220 case. Therefore, in applications where the regulator is operated at or near its maximum current rating, the TO-3 case is to be preferred over the TO-220. In addition, the use of silicon grease between the regulator and heat sink increases the heat transfer by a factor of 2, cooling the regulator further.

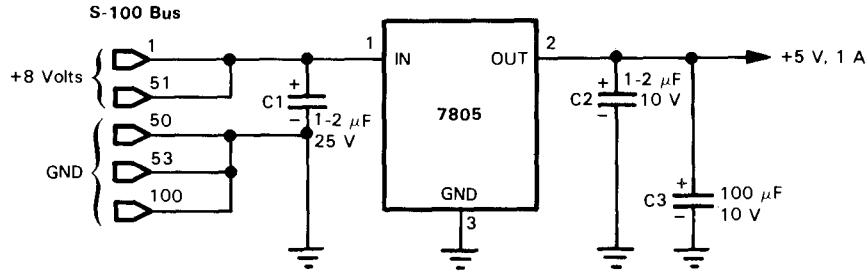


FIGURE 2-7. Typical +5 Volt, 1 A Regulator Circuit

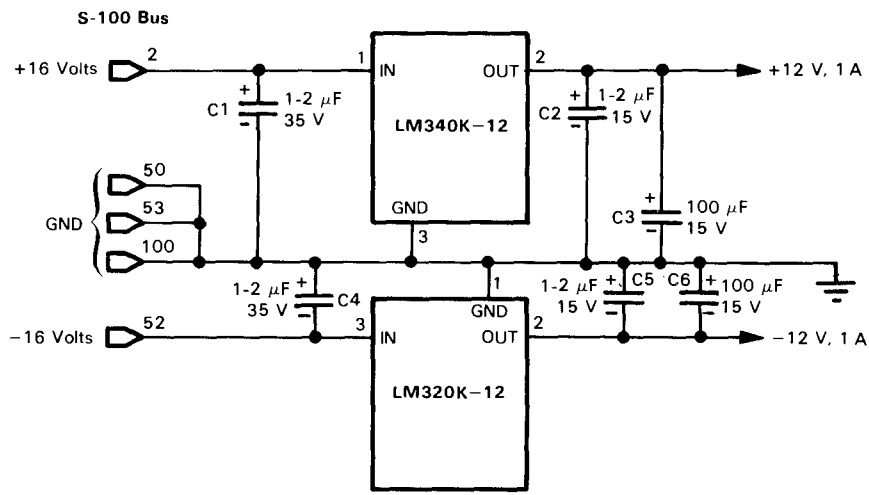


FIGURE 2-8. Typical +12 Volt and −12 Volt Regulator Circuits

Zener Diode Voltage Regulators

A simple circuit to drop the voltage of the +8, +16 or −16 V supplies to a desired lower voltage level may be accomplished with a zener diode, as shown in Figure 2-9. This circuit shows a positive voltage regulator. For a negative voltage regulator reverse the polarity of the zener diode and the capacitors.

This circuit can be used effectively in applications where the regulation of the current is not critical. The zener diode's voltage rating should equal V_{out}. The

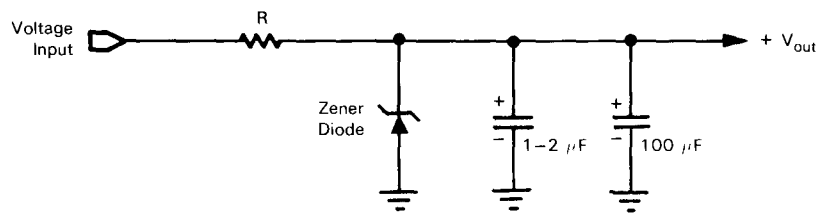


FIGURE 2-9. Typical Positive Voltage Regulator Circuit

value of R is determined as follows.

$$R = \frac{V_{in} - V_{out}}{I_z + I_{load}}$$

For example, if we were to use this circuit to provide +12 V at 10 mA, we would proceed as follows. V_{in} would be taken from the +16 V supply. We can figure that 5 mA of zener current will keep the diode operating in the zener region. Thus:

$$R = \frac{V_{in} - V_{out}}{I_z + I_{load}} = \frac{(16 - 12)\,V}{(5 + 10)\,mA} = \frac{4\,V}{15\,mA} = 267 \text{ ohms}$$

$$P_R = E \times I = 4 \times 15 \text{ mA} = 60 \text{ mW}$$

$$P_Z = E \times I = 12 \times 15 \text{ mA} = 180 \text{ mW}$$

Thus, a 270 ohm, 0.25 watt resistor and a 12 V, 0.4 watt zener diode should be selected.

Bypassing the +5 V Supply

On each S-100 card it is important to provide sufficient transient bypassing of the +5 V supply to the logic circuitry. This is usually accomplished by connecting 0.01-0.1 μF ceramic disk capacitors from the +5 V line to the ground line at regular intervals on the card. Typically one bypass capacitor is used for every two logic ICs.

the S-100 bus signals in detail

3

This chapter describes the function of every S-100 signal. For each signal we will give you the signal name, the pin number, and a description of its function. This chapter will serve as the main reference for the signals themselves. The next chapter will deal with the signals' timing relationships to one another.

As in the previous chapter, we have divided the S-100 signals into functional groups or buses.

When you have read this chapter you still will not understand how an S-100 system functions; you will need to correlate the information in this chapter with the information in the following chapter.

THE ADDRESS BUS

The address bus contains 24 signals, A0-A23. The address bus is used to select a slave device or specify a certain memory location. This bus is driven by tri-state logic and originates on the current bus master. The entire address bus can be placed in the high-impedance state by asserting a single line called ADSB*, which is not part of the address bus.

The eight address lines A16-A23 are new signals added by the standard. The eight new signals were placed on unused S-100 Bus pins. Also, a permanent bus master must drive only A0-A15, but a temporary master must drive A0-A23.

Each slave device contains a circuit called an "address decoder," so that only one memory location or I/O port responds to a given address. In pre-696 standard systems, only 16 address bits were used to address memory, allowing a max-

imum of 65,536 memory locations to be addressed. Eight-bit systems used only eight address bits for I/O addresses, allowing only 256 I/O ports.

Some newer CPU (permanent master) cards, and some older CPU (permanent master) cards with memory management capability, use all 24 address bits, allowing access to 16,777,216 memory locations. The IEEE standard also states that 16 bits (A0-A15) may now be used to address I/O ports, allowing a total of 65,536 I/O ports.

Table 3-1 lists all of the address lines and their S-100 pin numbers. A0 is the least significant bit and A23 is the most significant bit.

THE DATA BUS

There are two different data buses possible on the S-100 Bus. The buses have different meanings depending on whether an 8-bit or 16-bit data transfer is occurring. We will describe the 8-bit transfer first.

If an 8-bit transfer is occurring, the data lines are grouped as two unidirectional

TABLE 3-1. The Address Bus Signals

Address Bit	S-100 Pin Number	Comments
A0	79	Address line 0, least significant bit
A1	80	Address line 1
A2	81	Address line 2
A3	31	Address line 3
A4	30	Address line 4
A5	29	Address line 5
A6	82	Address line 6
A7	83	Address line 7
A8	84	Address line 8
A9	34	Address line 9
A10	37	Address line 10
A11	87	Address line 11
A12	33	Address line 12
A13	85	Address line 13
A14	86	Address line 14
A15	32	Address line 15
A16	16	Address line 16
A17	17	Address line 17
A18	15	Address line 18
A19	59	Address line 19
A20	61	Address line 20
A21	62	Address line 21
A22	63	Address line 22
A23	64	Address line 23, most significant bit

8-bit buses: the data output (DO) bus and the data input (DI) bus. The "input" and "output" refer to the direction of data flow, *relative to the current master.* Thus, the DI bus is used to send data to the master from a slave and the DO bus is used to send data from the master to a slave.

The DO bus consists of eight lines, DO0-DO7, with DO0 being the least significant bit. This bus is implemented in tri-state logic and originates on the current master. The DO bus can be floated by asserting the single signal DODSB*. DODSB* itself is not part of the DO bus.

The DI bus consists of eight lines, DI0-DI7, with DI0 being the least significant bit. This bus is implemented in tri-state logic and originates on slave devices. This bus should be floated (placed in the high-impedance state) until the slave determines from other S-100 signals that it is time to "turn on" the DI bus and place data onto the lines. Otherwise, all slaves would try and drive this bus at once, causing chaos. Exactly how a slave determines the correct time to drive the DI bus will be covered in the next chapter.

The foregoing describes how the two data buses work during 8-bit data transfers. But the IEEE standard provides for 16-bit transfers as well as 8-bit transfers, and the two may be intermixed in a system. Two new signals were defined to accomplish this. Although they are not part of the data buses themselves, we will describe them briefly so that you will understand the 8- and 16-bit data transfers.

The first new signal is sXTRQ* (sixteen request). This signal is asserted by a master that wants to do a 16-bit data transfer. If the slave is capable of doing the 16-bit transfer, it will assert a signal called SIXTN* (sixteen acknowledge). When the master sees SIXTN*, it will gang the DI and DO buses together into a single 16-bit bidirectional bus and perform the transfer. The direction of the transfer depends on the type of cycle.

When the two buses are ganged, DO0 becomes the least significant data bit of the even addressed byte and DI0 becomes the least significant data bit of the odd addressed byte. Not only do the functions of the data buses change, but so do the signal names. DO0 becomes DATA0, DO7 becomes DATA7, DI0 becomes DATA8, and DI7 becomes DATA15.

The exact timing of the data transfers will be described in the next chapter.

Table 3-2 shows the signal name for an 8-bit transfer, the signal name for a 16-bit transfer, and the S-100 pin number for all the data lines.

THE STATUS BUS

The information contained on the status bus is used by all devices on the bus to determine the state of the bus. The state depends on the type of cycle the current master is running. It follows that these lines are driven by the current master.

TABLE 3-2. The Data Bus Signals

8-Bit Signal Name	16-Bit Signal Name	S-100 Pin Number
DO0	DATA0	36
DO1	DATA1	35
DO2	DATA2	88
DO3	DATA3	89
DO4	DATA4	38
DO5	DATA5	39
DO6	DATA6	40
DO7	DATA7	90
DI0	DATA8	95
DI1	DATA9	94
DI2	DATA10	41
DI3	DATA11	42
DI4	DATA12	91
DI5	DATA13	92
DI6	DATA14	93
DI7	DATA15	43

Examples of bus cycles are memory read, memory write, I/O read, I/O write, etc.

Each status signal name begins with a lower-case s. The status bus is implemented in tri-state logic and may be floated by asserting the single control line SDSB* (status disable). SDSB* is not itself a status bus signal.

The status signals are listed below.

Name: sM1 (status Machine cycle 1) Active: High Pin Number: 44

Description: The name M1 comes from the old 8080 designation for an op-code fetch cycle. This status line signifies that the master is fetching an instruction from the bus. Depending on the implementation of a particular master, this line may also be active during an interrupt acknowledge cycle.

Name: sMEMR (status MEMory Read) Active: High Pin Number: 47

Description: This status line is active when the master is reading from a memory address. It will go high for all memory reads, including an op-code fetch.

Name: sINP (status INPut) Active: High Pin Number: 46

Description: This status line is active when the master is executing an input cycle and reading data from an I/O port address.

Name: sOUT (status OUTput) Active: High Pin Number: 45

Description: This status line is active when the master is executing an output cycle and writing data to an I/O port address.

Name: sWO* (status Write or Output) Active: Low Pin Number: 97

Description: This status line is active when the master is currently executing a memory write *or* an output write cycle.

Name: sINTA (status INTerrupt Acknowledge) Active: High Pin Number: 96

Description: This status line is active when the master is responding to an interrupt request and expects the interrupting device or interrupt controller to place data on the DI bus during this cycle.

Name: sHLTA (status HaLT Acknowledge) Active: High Pin Number: 48

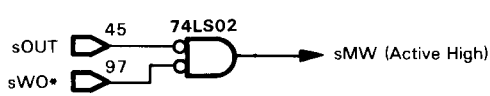
Description: This status line is active when the master enters a Halt state. An 8080, 8085, or Z80 microprocessor enters the Halt state by executing a HALT instruction. An interrupt request or reset is the only way to get out of a halted state, so this instruction is usually used to wait for an interrupt to occur. This instruction may have no equivalent in other processors. In that case, the processor would never enter the Halt state, and therefore sHLTA would never become active.

Name: sXTRQ* (status siXTeen ReQuest) Active: Low Pin Number: 58

Description: This is a new status line that is asserted by the master to request that a 16-bit data transfer occur during the current bus cycle. If this line is not asserted (if high) then an 8-bit transfer will be requested by default.

That covers all the status lines that have been assigned pin numbers, but there are some possible bus states that must be determined from a combination of status lines.

The most obvious of these missing status signals would indicate when a memory write cycle is occurring. (Remember that sWO* is active for both memory and output write cycles.) This condition can be determined by the expression sMW (status memory write) = sWO·–sOUT. A circuit for generating the sMW signal is shown in Figure 3-1. This signal should not be placed on the bus, but is intended to be used as an internal signal on a slave device.



sWO*	sOUT	sMW
L	L	H
L	H	L
H	L	L
H	H	L

FIGURE 3-1. Circuit to Generate Status Memory Write Signal

A master may have an idle state, when it is performing no action on the bus at all. The combination of status lines that indicates such an idle state is when none of the status lines is asserted true. The S-100 standard does not provide an idle state in its chart of possible bus cycles, but we have included it in Table 3-3. Table 3-3 shows the status bits as they should be asserted during the various types of bus cycles.

THE CONTROL OUTPUT BUS

The address and status buses are used to identify memory and I/O locations and the type of bus cycle currently in progress. None of this information should be used to control *when* things happen on the bus. Timing is the job of the control output bus.

The term "strobe" is commonly applied to the class of signals to which the control output bus belongs. The origin of the term strobe is unclear, but basically a strobe tells a device *when* some other set of inputs to the device is in a certain state.

For example, a write strobe would tell a slave that the data on the data output bus is now valid, and that the slave can proceed to use it. Without the strobe, the slave would not be able to tell the difference between valid and invalid data. Similarly, a read strobe would tell a slave when to place the data onto the data input bus. Without the read strobe, the slave might put data on the bus when some other device is putting data there, or at a time when the master is not ready to accept it.

TABLE 3-3. Bus Cycles and Status Signals

Bus Cycle Type	Status Signals							
	sM1	sMEMR	sINP	sOUT	sWO*	sINTA	sHLTA	sXTRQ*
Op-code Fetch	H	H	L	L	H	L	L	Y
Memory Read	L	H	L	L	H	L	L	Y
Memory Write	L	L	L	L	L	L	L	Y
I/O Read	L	L	H	L	H	L	L	Y
I/O Write	L	L	L	H	L	L	L	Y
Interrupt Acknowledge	X	L	L	L	H	H	L	Y
Halt Acknowledge	X	X	L	L	H	L	H	X
Idle	L	L	L	L	H	L	L	X

L = Low State
H = High State
X = Don't Care
Y = H for 8-bit data path operations
Y = L for 16-bit data path operations

Other strobes may inform devices that a new cycle is starting, that the address and status buses contain valid information, that the master has relinquished the bus, etc.

The control output bus is implemented in tri-state logic, and originates on the current master. The entire control output bus may be disabled by asserting a single line called CDSB* (control output disable), which is not a control output bus signal.

All control output bus lines begin with a lower-case p prefix. The p stands for "processor" and is a holdover from the pre-696 standard, before the term "master" was conceived. The control output bus lines are described below.

Name: pSYNC (processor SYNChronize) Active: High Pin Number: 76

Description: This is a strobe that indicates the start of every bus cycle. It becomes active very near the begining of every bus cycle, and remains active for approximately one cycle of Φ.

Name: pSTVAL* (processor STatus VALid) Active: Low Pin Number: 25

Description: This is a strobe that indicates that the information on the address and status buses is valid. Note that pSTVAL* is meaningful only when it occurs in conjunction with pSYNC. This is because before the IEEE standard was adapted pin 25 was used as the Φ1 clock signal. Φ1 was commonly gated with pSYNC to perform the same function as pSTVAL* in the days before pSTVAL* existed. Not wanting to make older boards obsolete, the old Φ1 signal meets the requirements of pSTVAL*.

pSTVAL* is required to have one, and only one, negative-going edge during pSYNC, but may have other negative-going edges during other parts of the bus cycle (as Φ1 did). Therefore, pSTVAL* must be gated with pSYNC in order to be useful.

Why was Φ1 replaced with pSTVAL* if you still have to gate it with pSYNC? Most of the newer processors use a single-phase clock, as opposed to the old 8080 which used a two-phase clock. It was deemed by the standard committee that it was too difficult to synthesize Φ1 with the newer processors.

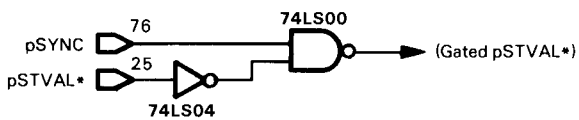
A circuit that correctly gates pSTVAL* with pSYNC is shown in Figure 3-2.

Name: pDBIN (processor Data Bus IN) Active: High Pin Number: 78

Description: This signal is the generalized read strobe for the S-100 Bus. It is asserted for memory read, I/O read, and interrupt acknowledge cycles. It is used by a slave device to turn on its data bus drivers so that the data to be read is gated onto the bus at the proper time. The master should sample the data near the end of this read strobe.

Name: pWR* (processor WRite) Active: Low Pin Number: 77

Description: This signal is the generalized write strobe for the S-100 Bus. It is



Active low; use AND gate for active high

pSYNC	pSTVAL*	Gated pSTVAL*
L	L	H
L	H	H
H	L	L
H	H	H

FIGURE 3-2. Circuit That Gates pSTVAL* with pSYNC

asserted for memory and I/O write cycles. It is used by the slave device to tell when the data output bus contains valid data.

Name: pHLDA (processor HoLD Acknowledge) Active: High Pin Number: 26
Description: This is a signal that indicates that the master has relinquished control of the S-100 Bus to another master.

There is one other strobe on the S-100 Bus that should be considered as part of the control output bus, but generally is not. The signal is MWRT (memory write) and is a strobe like pWR*, except that it occurs only during memory write cycles. The reason it is not included with the other signals is that it is not disabled by the CDSB* signal, as are all the others. In fact, it is never disabled.

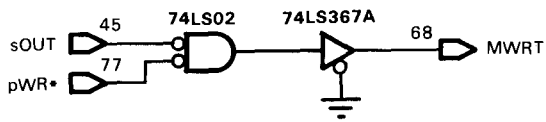
Name: MWRT (Memory WRiTe) Active: High Pin Number: 68
Description: MWRT is a memory write strobe that is not disabled along with the other control output bus signals. MWRT is derived from the following equation: MWRT=pWR·−sOUT. In other words, when pWR* is true and sOUT is false, a memory write cycle is occurring.

The standard specifies that MWRT be generated by only one device in any system. It may originate on the permanent master, a front panel, or on any other device that is permanently in the system.

Note that MWRT should be generated by the actual bus signals pWR* and sOUT. This ensures that any master will be able to write into memory which uses MWRT. A circuit that may be used to generate MWRT is shown in Figure 3-3.

THE CONTROL INPUT BUS

The control input bus consists of four lines that are asserted by bus slaves to



sOUT	pWR*	MWRT
L	L	H
L	H	L
H	L	L
H	H	L

FIGURE 3-3. Circuit to Generate MWRT

signal the master to alter its operation — for example, to wait until data is ready from a slow memory, to relinquish the bus to another master, etc.

All of the control input bus lines are implemented in open collector logic so that more than one device may drive them at one time without conflicts. The control input bus lines are described below.

Name: RDY (ReaDY) Active: High Pin Number: 72

Description: RDY is a signal that is asserted by a slave to tell the master that it is ready to complete the current bus cycle. The slave may drive RDY low to tell the master that it is not ready to complete the operation. This will cause the master to wait until RDY goes high again, in effect extending the current bus cycle.

It may be less confusing to think about RDY in the negative sense. Then RDY would become a "wait request" line telling the CPU to wait when it goes low.

RDY is used by some slaves to synchronize the master to an external event, such as a rotating floppy disk. It may also be used to "single step" the master by pulling RDY low during each cycle. This is a common technique used by front panels.

Name: XRDY (auXiliary ReaDY) Active: High Pin Number: 3

Description: This is another ready signal originally used only by front panels. Now any device may use either RDY or XRDY. XRDY functions exactly the same as RDY.

Name: HOLD* (HOLD request) Active: Low Pin Number: 74

Description: HOLD* is a signal asserted by a temporary master to request that the permanent master relinquish the bus to the temporary master. The temporary master should continue to assert HOLD* until it determines that it is either done with the bus or will not be granted access. HOLD* may be masked at any time by the permanent master.

Name: SIXTN* (SIXTeeN acknowledge) Active: Low Pin Number: 60

Description: SIXTN* is a signal that is asserted by a slave device if it is capable of a 16-bit data transfer. If the master asserts sXTRQ* and SIXTN* is asserted by the addressed slave within a short period of time, then the master may proceed with a 16-bit data transfer. If SIXTN* is not asserted by the slave, then the master should perform the transfer as two 8-bit transfers. If the master is not capable of performing the 16-bit transfer as two 8-bit transfers, an error condition will result immediately, with ERROR* asserted.

Note that SIXTN* is a new S-100 signal, but it has been implemented in a way that makes it compatible with existing systems — an 8-bit memory board would never assert SIXTN*.

THE INTERRUPT BUS

The S-100 Bus contains 10 interrupt lines that may be used by slave devices to interrupt the master's current task and cause another task to be executed instead. An example is an interrupt that occurs once a second, causing the normal program flow to call a subroutine to update a memory location that keeps track of the time of day. The subroutine then returns control to the interrupted program, which continues as if nothing had happened.

There are two general interrupt request inputs to the permanent master: INT*, which is usually software maskable, and NMI*, which is non-maskable. There are eight lines called VI0* to VI7*. These lines are the vectored interrupt lines and may be assigned priorities. This means that if two or more interrupts occur simultaneously, the one with the highest priority will be serviced first. Usually VI0* has the highest priority, while VI7* has the lowest.

The VI* lines may be used to cause interrupts directly to the master, but they are usually connected to an interrupt controller that will then assert INT*. Refer to Chapter 13 for more information on interrupts.

All the interrupt lines are implemented in open collector logic and originate on slave devices. They are listed below.

Name: INT* (INTerrupt) Active: Low Pin Number: 73

Description: This is the general purpose interrupt request line for the S-100 Bus. It is usually maskable by a software instruction. When asserted by a slave, assuming the master has not masked interrupts, after completing the current cycle the master will enter an interrupt acknowledge cycle or cycles. Usually the interrupting device will send some kind of information to the master during the interrupt acknowledge cycle.

Note that INT* should be asserted as a level, meaning that INT* should remain low until the interrupting device has been serviced.

Name: NMI* (Non Maskable Interrupt) Active: Low Pin Number: 12
Description: As the name implies, NMI* is an interrupt line that may not be masked by the master. It should always respond to the NMI* line. For this reason, NMI* is usually reserved to signal some catastrophic event, such as loss of system power.

Note that an interrupt acknowledge cycle need not be generated in response to an NMI*. Also note that NMI* is asserted as a negative-going edge and not as a level.

The vectored interrupt bus consists of eight lines that are all active low. They are called "vectored" because asserting a certain line could cause the master to vector to a routine that is specifically associated with that line. The VI* lines may have a fixed priority, with VI0* usually having the highest priority. All of the lines are active low. Table 3-4 shows the pin positioning of the VI* lines.

THE TMA BUS

TMA stands for Temporary Master Access. The bus signals associated with the TMA bus are used to allow temporary masters to request and receive control of the bus from the permanent master. In the pre-696 standard days the term used to describe this was DMA, which stood for Direct Memory Access. The term DMA was really a misnomer, since the temporary master can perform any kind of bus cycle (not just memory) when it gains control of the bus. Therefore the term TMA has been substituted to better reflect the process that is occurring.

The old S-100 Bus provided for only one TMA device (temporary master) to exist in a system. The IEEE standard has made provisions for up to 16 such devices to exist in one system. To do this, the standard defines four new lines that contain a code corresponding to the highest priority temporary master that is currently requesting the bus. This allows the temporary masters to arbitrate among themselves for control of the bus. The four new lines are called TMA0*-TMA3*.

TABLE 3-4. The Vectored Interrupt Bus Signals

Vectored Interrupt	Pin Number
VI0*	4
VI1*	5
VI2*	6
VI3*	7
VI4*	8
VI5*	9
VI6*	10
VI7*	11

In addition, there are four signals that are asserted by the temporary master to disable the various output buses that are normally driven by the permanent master.

The whole process of TMA is quite involved, and you need to thoroughly understand how all the signals on the S-100 Bus work before you can understand TMA. The signal definitions in this chapter may be used as a reference in the future. The TMA process will be explained in detail in Chapter 15.

The four lines for TMA arbitration are used to resolve simultaneous requests for the bus by temporary masters. The encoded priority of all the requesters is asserted on these lines, and after settling the lines contain a number that corresponds to the highest priority requester. That temporary master is now granted access to the bus. All the lines are open collector and are active low. The following table shows their S-100 Bus pin assignments.

Signal Name	Pin Number
TMA0*	55
TMA1*	56
TMA2*	57
TMA3*	14

The other TMA bus signals are listed below.

Name: ADSB* (Address DiSaBle) Active: Low Pin Number: 22

Description: When ADSB* is asserted, all address bits (A0-A23) are floated on the permanent master.

Name: DODSB* (Data Out DiSaBle) Active: Low Pin Number: 23

Description: When DODSB* is asserted, the data output bits (DO0-DO7) are floated on the permanent master.

Name: SDSB* (Status DiSaBle) Active: Low Pin Number: 18

Description: When SDSB* is asserted, the status bus (sMEMR, sWO*, sM1, sINP, sOUT, sHLTA, sINTA, and sXTRQ*) is floated on the permanent master.

Name: CDSB* (Control output DiSaBle) Active: Low Pin Number: 19

Description: When CDSB* is asserted, the control output bus (pSYNC, pSTVAL*, pDBIN, pWR*, and pHLDA) is floated on the permanent master.

MWRT is not affected by CDSB*.

UTILITY SIGNALS

The following signals do not fit into any of the previous specific categories.

Name: Φ or System Clock Active: N/A Pin Number: 24

Description: Φ is the master system clock signal that controls timing of all bus cycles for permanent and temporary masters and slaves. It is generated by the permanent master, and there is no provision for floating the system clock signal.

Name: CLOCK Active: N/A Pin Number: 49

Description: This is a 2 MHz clock signal that does not have to be synchronous with the system clock. The frequency tolerance is + or − 0.5% and the duty cycle is between 40% and 60%. This signal is used as a timing reference for baud rate generators, real-time clocks, and interval timers.

Name: PHANTOM* Logic: Open collector Active: Low Pin Number: 67

Description: This signal is provided so that slave devices may exist in the same address space by overlaying one another. One device (the phantom device) is inactive if PHANTOM* is inactive and a normal device is active. When PHANTOM* is asserted, the phantom device becomes active and the normal device becomes inactive. PHANTOM* may originate anywhere on the bus.

Name: POC* (Power On Clear) Active: Low Pin Number: 99

Description: POC* must start out low when the system powers up, and remain low for at least 10 milliseconds after power is stable. POC* must be active only at power-on. POC* must also assert RESET* and SLAVE CLR*. A circuit for generating POC* that also asserts RESET* and SLAVE CLR* is shown in Figure 3-4.

Name: RESET* Logic: Open collector Active: Low Pin Number: 75

Description: RESET* is a signal that resets all bus masters to a known condition. Any bus slave that needs to start in a known condition relative to the master may also be reset by RESET*. RESET* is often connected to a pushbutton switch located somewhere on the machine.

Name: SLAVE CLR* Logic: Open collector Active: Low Pin Number: 54

Description: SLAVE CLR* is a signal that resets all bus slaves to a known condition. Note that SLAVE CLR* used to be called EXT CLR*, for external clear. The function is still the same, but the name was changed to be consistent with the terminology of the standard.

Name: ERROR* Logic: Open collector Active: Low Pin Number: 98

Description: This is a generalized error signal line that can be used to inform the master that some kind of error has occurred. This can be a memory parity error, an attempt to write into a protected memory location, an attempt to perform a 16-bit transfer to an 8-bit device, etc.

ERROR* should be implemented as a trap, in that it should cause address and status information on the bus to be latched so that an error handling routine can

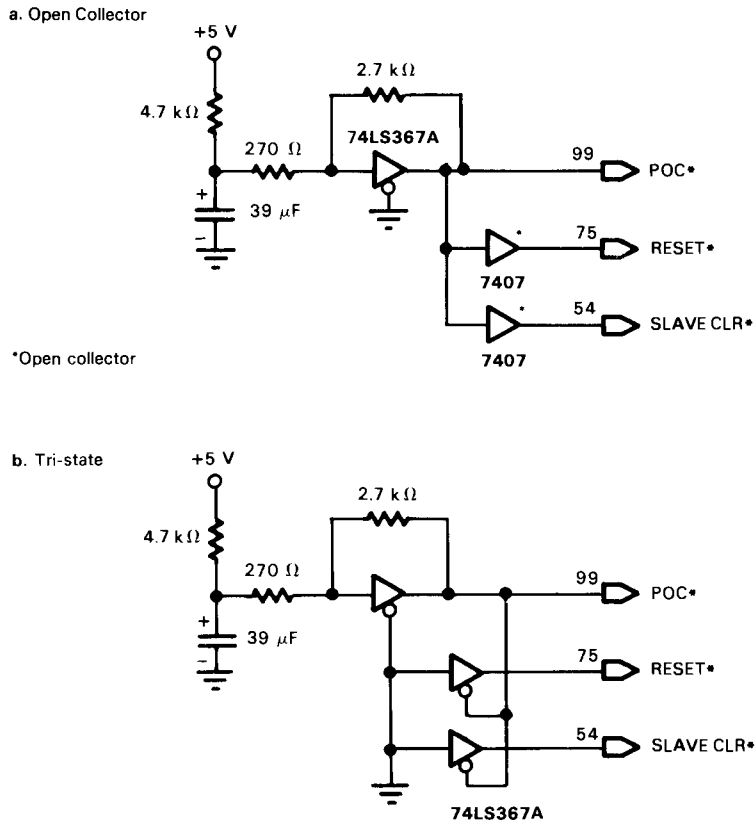


FIGURE 3-4. Two Circuits That Correctly Generate POC*

determine the source of the error and take corrective action. A circuit to implement this will be presented in Chapter 16.

Name: PWRFAIL* Logic: Open collector Active: Low Pin Number: 13

Description: This is a signal line that goes low to indicate an impending power failure. It will probably cause a non-maskable interrupt in most systems so that the permanent master can save the current machine status information.

This line goes low and remains low until power is restored and POC* is true.

POWER AND GROUND

The S-100 Bus uses three voltages: +8 volts, +16 volts, and −16 volts. These are all with respect to a common ground, and all are unregulated. Regulation of the voltages occurs on the individual S-100 cards.

Name: +8 Volts Pin Numbers: 1 and 51

Description: This is the primary logic supply for S-100 systems. It is specified as having an instantaneous minimum value of +7 V and an instantaneous maximum value of +25 V. The average maximum must be less than +11 V.

Name: +16 Volts Pin Number: 2

Description: The +16 volt line is specified as having an instantaneous minimum value of +14.5 V and an instantaneous maximum value of +35 V. The average maximum value must be less than +21.5 V.

Name: −16 Volts Pin Number: 52

Description: The −16 volt line is specified as having an instantaneous maximum value of −14.5 V and an instantaneous minimum value of −35 V. The average minimum value must be greater (more positive) than −21.5 V.

Name: GND (GrouND) Pin Numbers: 20, 50, 53, 70, and 100

Description: These are the primary ground reference for the system. All voltage levels are measured with respect to this ground.

Three of the old S-100 signals have been deleted by the standard and replaced by ground lines. These new ground lines are on S-100 Bus pins 20, 53, and 70.

NDEF LINES

There are three lines that are called NDEF, for Not to be DEfined. These are intended to be used by various manufacturers for their own use. If a manufacturer does so, it must fully document how the lines are used and provide disconnecting jumpers. One of these signals was the old front panel signal Single Step. The other two have been used for a variety of purposes.

The three NDEF lines are on S-100 Bus pins 21, 65, and 66.

RFU LINES

There are four lines on the S-100 Bus that are specified as RFU, for Reserved

for Future Use. The standard states that they may not be used for any purpose. The RFU lines are on S-100 Bus pins 27, 28, 69, and 71.

OLD S-100 SIGNALS

A few signals that were commonly used by a number of manufacturers have been eliminated from the IEEE standard. However, in keeping with the philosophy that the standard should not make older boards obsolete, most of the signals have been changed in such a way that the older boards are compatible with the standard.

We will explain some of the old signals and their replacements.

Old Name: Φ1 Replaced by: pSTVAL* Active: N/A Pin Number 25

Description: This was the opposite phase of the clock signal to Φ2 on pin 24 (which used to be called Φ2, but is now just calld Φ). It was generally considered to be non-overlapping with Φ2. Many of the newer microprocessors use a single-phase clock, and the standard committee felt that it was too difficult to "synthesize" a Φ1 whose high portion would not overlap with the high portion of Φ2, so pSTVAL* was substituted. Boards that still provide Φ1 may still meet the requirements for pSTVAL* (see the section on pSTVAL* in this and the next chapter).

Old Name: RUN Replaced by: RFU Active: High Pin Number: 71

Description: This signal was generated by front panels and was high when the machine was running and low when the front panel had stopped the machine. It was not widely used, except by some dynamic RAM boards to continue refresh operations when the front panel had stopped the machine. It was also used by some manufacturers of CPU boards to slow down the CPU speed when the front panel was in control, because the front panels were not designed to run faster than 2 MHz.

This signal line has now been designated RFU, which means it is not to be used. But older systems with front panels should still be able to drive this line without any compatibility problems, unless it is assigned a new function.

Old Name: SS (Single Step) Replaced by: NDEF Active: Low Pin Number: 21

Description: This signal was also generated by front panels. The front panel generally had a cable that connected to the CPU board. This cable carried eight data lines and was generally connected to the CPU's internal bidirectional data bus. Instructions were "jammed" over this cable to cause the front panel to operate. However, the CPU's data input bus drivers would also be turned on, causing a conflict on the internal data bus. SS would go low momentarily while the front

panel was jamming the data on the cable. SS was then used by the CPU to disable the data input bus drivers so that no conflict would exist.

This line has now been specified as NDEF, which means that any manufacturer can use it for any purpose, as long as it is documented and jumpered. Older S-100 systems can therefore still use SS on pin 21.

Old Name: SSWDSB* (Sense SWitch DiSaBle) Replaced by: GND
Active: Low Pin Number: 53

Description: This is another signal that was generated by front panels. The front panel usually had a set of switches called the sense switches connected to an I/O port. The data from the switches were sent over the data cable described above, instead of over the S-100 data input bus. As described above, another conflict would exist between the S-100 Bus and the on-board data buses. SSWDSB* would go low to float the CPU's S-100 Bus data input driver whenever an access to the sense switches was made.

This signal has been replaced by ground, which is not in keeping with the compatibilty philosophy of the standard. With an older CPU, grounding this line will keep its data input buffers permanently disabled, never allowing any data from the bus to get to the CPU. Of course, this will keep the system from operating. Some pressure has been put on the standard committee to change this line to NDEF, but they feel there is a need for a GND signal on that side of the bus. It is recommended that new masters which must work with older front panels provide an option jumper on this pin.

Old Name: UNPROT (UNPROTect) Replaced by: GND Active: High
Pin Number: 20

Description: This signal was pulsed high to cause a memory board to unprotect itself (allow data to be written into it). The source of this signal was usually a switch on the front panel. Memory protection has been abandoned by most manufacturers, so this signal was replaced by GND.

Old Name: PROT (PROTect) Replaced by: GND Active: High Pin Number: 70

Description: This signal performed the opposite function of UNPROT; it was used to protect a memory board (keep data from being written into it). The source of this signal was usually a switch on the front panel. As mentioned above, memory protection has generally been abandoned by manufacturers, so this signal has also been replaced by GND.

Old Name: PS (Protect Status) Replaced by: RFU Logic: Open collector
Active: Low Pin Number: 69

Description: This signal was generally connected to an indicator on a front panel to show that the currently addressed memory board was protected. This line went low if the board was protected and remained high if it was not protected. Since

TABLE 3-5. S-100 Bus Layout — Quick Reference

Pin	1	+8 Volts (B)		Pin	51	+8 Volts (B)	
Pin	2	+16 Volts (B)		Pin	52	−16 Volts (B)	
Pin	3	XRDY (S)	H	Pin	53	GND	
Pin	4	VI0* (S)	L	Pin	54	SLAVE CLR* (B)	L
Pin	5	VI1* (S)	L	Pin	55	DMA0* (M)	L
Pin	6	VI2* (S)	L	Pin	56	DMA1* (M)	L
Pin	7	VI3* (S)	L	Pin	57	DMA2* (M)	L
Pin	8	VI4* (S)	L	Pin	58	sXTRQ* (M)	L
Pin	9	VI5* (S)	L	Pin	59	A19	H
Pin	10	VI6* (S)	L	Pin	60	SIXTN* (S)	L
Pin	11	VI7* (S)	L	Pin	61	A20 (M)	H
Pin	12	NMI* (S)	L	Pin	62	A21 (M)	H
Pin	13	PWRFAIL* (B)	L	Pin	63	A22 (M)	H
Pin	14	DMA3* (M)	L	Pin	64	A23 (M)	H
Pin	15	A18 (M)	H	Pin	65	NDEF	
Pin	16	A16 (M)	H	Pin	66	NDEF	
Pin	17	A17 (M)	H	Pin	67	PHANTOM* (M/S)	L
Pin	18	SDSB* (M)	L	Pin	68	MWRT (B)	H
Pin	19	CDSB* (M)	L	Pin	69	RFU	
Pin	20	GND		Pin	70	GND	
Pin	21	NDEF		Pin	71	RFU	
Pin	22	ADSB* (M)	L	Pin	72	RDY (S)	H
Pin	23	DODSB* (M)	L	Pin	73	INT* (S)	L
Pin	24	Φ (B)	H	Pin	74	HOLD* (M)	L
Pin	25	pSTVAL* (M)	L	Pin	75	RESET* (B)	L
Pin	26	pHLDA (M)	H	Pin	76	pSYNC (M)	H
Pin	27	RFU		Pin	77	pWR* (M)	L
Pin	28	RFU		Pin	78	pDBIN (M)	H
Pin	29	A5 (M)	H	Pin	79	A0 (M)	H
Pin	30	A4 (M)	H	Pin	80	A1 (M)	H
Pin	31	A3 (M)	H	Pin	81	A2 (M)	H
Pin	32	A15 (M)	H	Pin	82	A6 (M)	H
Pin	33	A12 (M)	H	Pin	83	A7 (M)	H
Pin	34	A9 (M)	H	Pin	84	A8 (M)	H
Pin	35	DO1 (M)/DATA 1 (M/S)	H	Pin	85	A13 (M)	H
Pin	36	DO0 (M)/DATA 0 (M/S)	H	Pin	86	A14 (M)	H
Pin	37	A10 (M)	H	Pin	87	A11 (M)	H
Pin	38	DO4 (M)/DATA 4 (M/S)	H	Pin	88	DO2 (M)/DATA 2 (M/S)	H
Pin	39	DO5 (M)/DATA 5 (M/S)	H	Pin	89	DO3 (M)/DATA 3 (M/S)	H
Pin	40	DO6 (M)/DATA 6 (M/S)	H	Pin	90	DO7 (M)/DATA 7 (M/S)	H
Pin	41	DI2 (S)/DATA 10 (M/S)	H	Pin	91	DI4 (S)/DATA 12 (M/S)	H
Pin	42	DI3 (S)/DATA 11 (M/S)	H	Pin	92	DI5 (S)/DATA 13 (M/S)	H
Pin	43	DI7 (S)/DATA 15 (M/S)	H	Pin	93	DI6 (S)/DATA 14 (M/S)	H
Pin	44	sM1 (M)	H	Pin	94	DI1 (S)/DATA 9 (M/S)	H
Pin	45	sOUT (M)	H	Pin	95	DI0 (S)/DATA 8 (M/S)	H
Pin	46	sINP (M)	H	Pin	96	sINTA (M)	H
Pin	47	sMEMR (M)	H	Pin	97	sWO* (M)	L
Pin	48	sHLTA (M)	H	Pin	98	ERROR* (S)	L
Pin	49	CLOCK (B)		Pin	99	POC* (B)	L
Pin	50	GND		Pin	100	GND	

S Slave generated signal
M Master generated signal
M/S Master or slave generated signal
B Bus signal from power supply or front panel
* Signal is true when line is low
H Active high
L Active low

memory protection has been generally abandoned by manufacturers, this line has been designated RFU.

Old Name: SSTACK (Status STACK) Replaced by: ERROR*
Logic: Tri-state driver Active: High Pin Number: 98

Description: This was a status signal from older CPU boards to indicate that a stack operation was occurring. The 8080 is the only processor where this status is easily determined, and this line was abandoned by most manufacturers well before the standard was formulated. It has been replaced by the ERROR* signal, which is not an equivalent function, but we can't find a single S-100 board that ever used SSTACK, so there should not be any compatibility problems here.

Old Name: PINTE (Processor INTerrupts Enabled) Replaced by: RFU
Logic: Tri-state driver Active: High Pin Number: 28

Description: This was a control output signal from older CPU boards that indicated that the processor's interrupts were enabled. As with SSTACK, the 8080 was the only processor where it was easy to determine this condition. This signal was hardly ever used, so it has been replaced by an RFU line.

Table 3-5 is a numerically ordered reference chart of all the S-100 signal lines.

S-100 bus timing relationships

4

INTERFACING TO S-100 COMPUTERS

In the previous chapter we discussed all of the S-100 signals in detail, but most of those signals are meaningless unless you understand how they all interact. Interaction is what this chapter will explain. We will discuss the timing relationships between the various S-100 signals. When you have read Chapter 3 and this chapter, you will have a good idea of what's going on on the S-100 Bus.

BUS STATES AND BUS CYCLES

The System Clock signal is called "Φ." It is the master clock signal that governs all of the timing on the S-100 Bus. Most of the signal timing on the S-100 Bus is measured relative to a rising or falling edge of Φ. Φ is generated by the permanent bus master, usually a CPU board. Φ is usually the CPU clock. It is generally a square wave with a frequency between 1 and 6 megahertz.

Φ is shown in Figure 4-1. Approximately three "clock cycles" are shown. A clock cycle is defined as one complete high and low period of the clock. In the figure, one clock cycle is shown from the beginning of a rising edge to the beginning of the next rising edge (Ⓐ in the figure).

You will also notice that in Figure 4-1 there are other division marks that are one clock cycle wide (Ⓑ); these divisions are called "bus states." Each transfer of data or any other transaction on the S-100 Bus takes place in at least

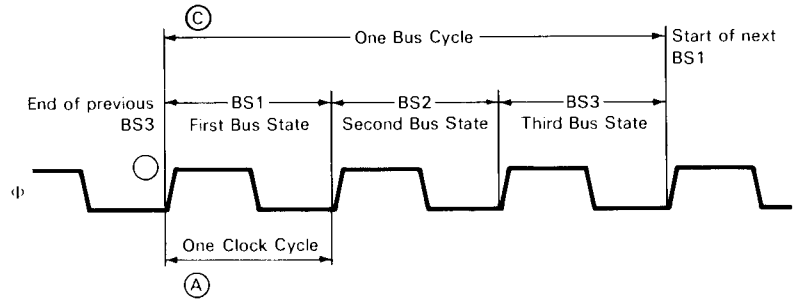


FIGURE 4-1. Bus States and Bus Cycles

three bus states, and different things should occur during the different bus states.

It is easy to confuse "clock cycle" with "bus cycle." A bus cycle (Ⓒ) may contain three or more clock cycles. Every S-100 bus cycle contains at least three clock cycles and therefore three bus states. A slave may request that the bus cycle be extended by causing a "wait state." When a bus cycle is extended, it is done by adding more bus states. This will be discussed in detail later in the chapter.

To recap, an S-100 "bus cycle" has at least three "bus states," and each bus state is one "clock cycle" wide.

THE BASIC BUS CYCLE

Figure 4-2 is a flowchart of the basic S-100 bus cycle. The bus cycle begins with Bus State 1 (BS1). As BS1 begins, the master asserts its status and address buses and pSYNC signal (pSYNC signifies the beginning of a bus cycle). After a few nanoseconds the address and status buses "settle," meaning that the information on the buses is stable and valid. The master then asserts the pSTVAL* signal, informing slaves that the status and address buses contain valid information.

The master then enters Bus State 2 (BS2). The address and status buses continue to be asserted. The master switches pSYNC false. From this point on, the cycle will have two different sequences of events depending on whether this is a read cycle or a write cycle (this is why the flowchart splits). First we will examine a read cycle (see Figure 4-3).

After pSYNC goes false, the master asserts the read strobe pDBIN. The addressed slave device uses pDBIN to turn on its data bus drivers so that the master can read the data via the data input bus. Now the master enters Bus State 3.

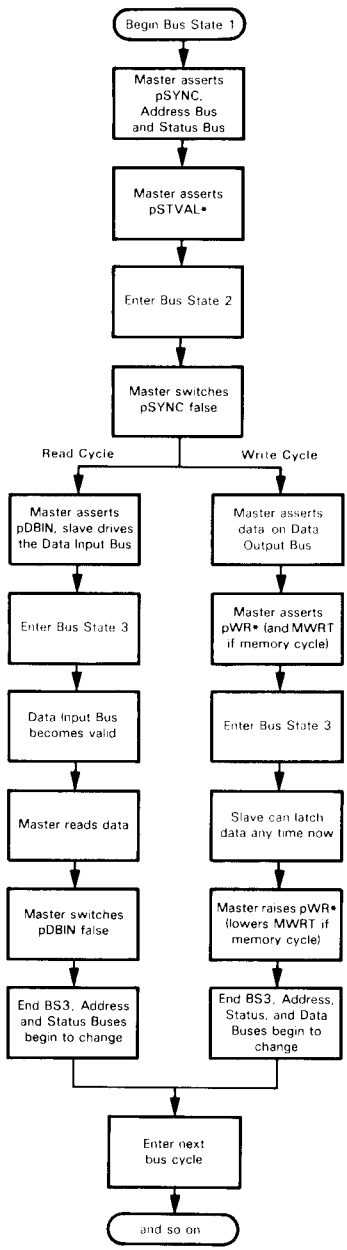


FIGURE 4-2. Basic S-100 Bus Cycle Flowchart

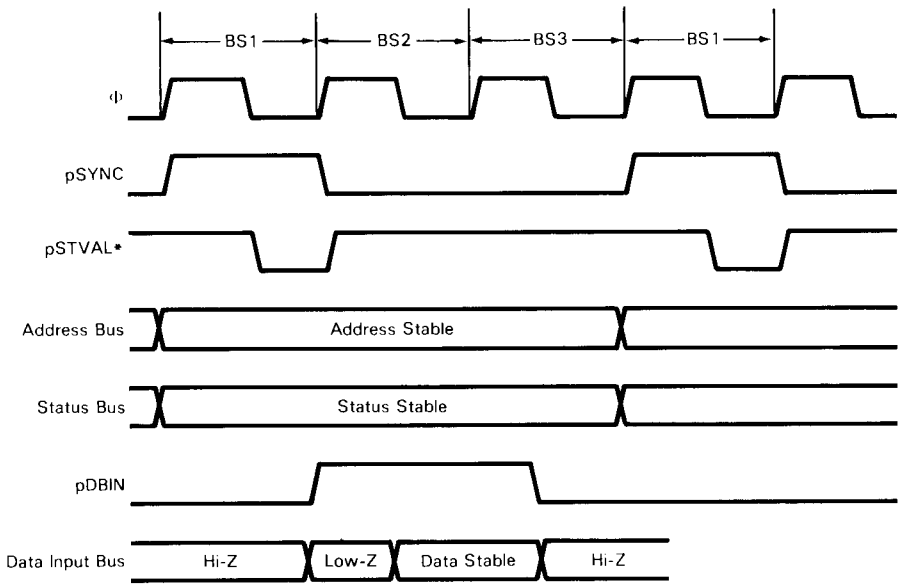


FIGURE 4-3. S-100 Bus Read Cycle

The data that the slave wishes to send to the master may not be valid yet, but will now start to become valid and stable. The master then reads the data and switches pDBIN false. The address and status buses must remain stable for a while after pDBIN switches false. BS3 now ends, ending the bus cycle. The next BS1 begins with the new status and address information for the next bus cycle.

Now we will examine what happens during a write cycle (see Figure 4-4). The first four operations are the same as a read cycle, with the master having just switched pSYNC false. The master now places the data to be written on the data output bus. After the data has settled, the master asserts the write strobe pWR* (MWRT will also be asserted if the cycle is a memory write cycle).

Now the master enters BS3. The slave device looks at the data during this time and will usually "latch" the data on either edge of the write strobe. Then the master switches pWR* false (MWRT will also switch false if it was a memory write). The data, address, and status buses must remain valid for a period of time after the trailing edge of the write strobe(s), as shown in Figure 4-4. Now BS3 ends, ending the bus cycle. The address, status, and data buses will start to be asserted for the next cycle.

These two similar sequences of events are shown in Figures 4-3 and 4-4 as "timing diagrams." They show the relationships of the signals to one another in parallel rather than the serial representation of the flowchart. The timing diagrams

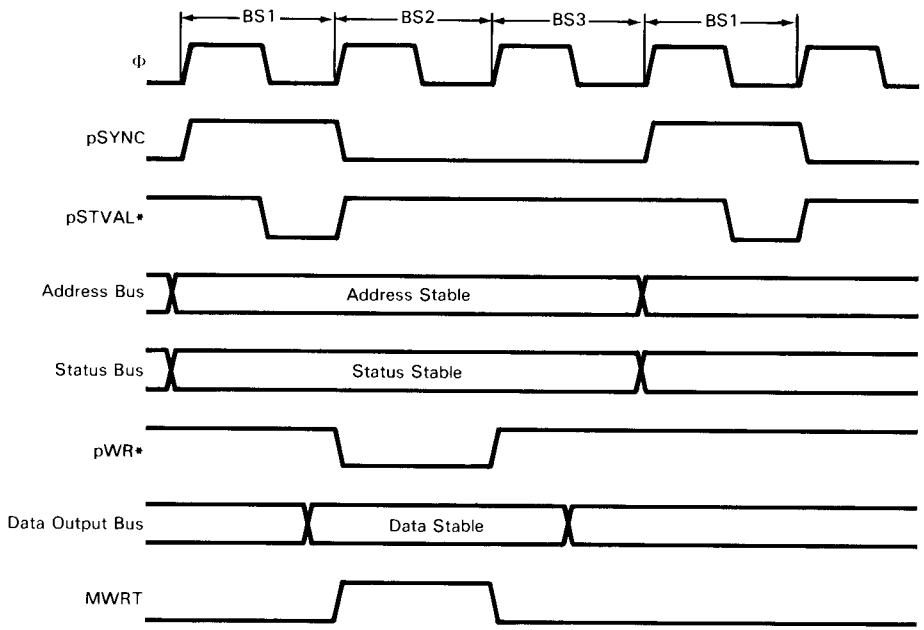


FIGURE 4-4. S-100 Bus Write Cycle

also show the relationship of Φ to the signals. Next we will discuss the read and write cycles using these timing diagrams.

THE READ CYCLE

A read cycle on the S-100 Bus is one in which the slave transfers the data to the master. The master is "reading" the data. The two types of read cycles are memory read and I/O read. The two bus cycles are virtually identical. The only difference is that during a memory read cycle the address bus may contain 16 or 24 bits of information and the status bus information would reflect a memory read, while during an I/O read cycle the address bus may contain 8 or 16 bits of information and the status bus information would reflect an I/O read.

At the beginning of BS1 the master asserts the address and status buses, as shown in Figure 4-3. After the rising edge of Φ in BS1, pSYNC will be asserted high. During this time pSTVAL* should be high. As soon as the address and status buses have settled, and pSYNC has been high for at least 20 nanoseconds, the master will assert pSTVAL* low. Note that the master may assert pSTVAL* low only once during pSYNC.

pSYNC must remain asserted until the rising edge of Φ in BS2. pSTVAL* should return high at this time as well. When pSYNC falls (just after the rising edge of Φ in BS2), the master will assert the read strobe pDBIN. Up until now the data input bus has been in a high-impedance state. When pDBIN is asserted, the addressed slave turns on its data bus drivers, causing the data input bus to be in the low-impedance state (the data on the bus may not be valid). Later (depending on the access time of the slave), the data from the slave will become stable. Most masters then sample this data at the falling edge of Φ during BS3. Very shortly thereafter, the master will drop pDBIN, causing the slave's data bus drivers to turn off, putting the data input bus in a high-impedance state. The master must continue to assert valid addresses and status for at least 50 nanoseconds after pDBIN falls. Then BS3 will end and the next bus cycle will begin.

For timing parameters of the read cycle, complete with timing marks and data showing the minimum and maximum timing variations, refer to Figure 11*a* and Table 4 in the copy of the S-100 standard in the appendix.

THE WRITE CYCLE

During the write cycle the master transfers data to an addressed slave device. The master is "writing" the data to the slave. The two types of write cycles are memory write and I/O write, and they appear virtually identical on the bus. The differences are that during a memory write cycle the address bus may contain 16 or 24 bits of information and the status bus would reflect a memory write cycle, while during a I/O write cycle the address bus may contain either 8 or 16 bits of information and the status bus would reflect an I/O write cycle. In addition, the strobe MWRT will also be asserted during a memory cycle, but not during an I/O cycle.

At the begining of BS1 the master will begin to assert the address and status buses, as shown in Figure 4-4. After the rising edge of Φ in BS1, pSYNC will be asserted high. During this time pSTVAL* should be high. As soon as the address and status buses have stabilized and pSYNC has been high for at least 20 nanoseconds, the master will assert pSTVAL* low. Note that the master may assert pSTVAL* low only once during pSYNC.

pSYNC must remain asserted until the rising edge of Φ in BS2. pSTVAL* will probably return high at this time as well. About the time that pSYNC falls (just after the rising edge of Φ in BS2), the master will start to assert the data to be written on the data output bus. After the data becomes valid, the master will assert the write strobe(s) (pWR* and MWRT for a memory write, or just pWR* for an I/O write). The slave may then sample or latch the data at any time during the write strobe. Most slaves do this on the trailing edge of the write strobe.

Sometime around the falling edge of Φ in BS3, the master will stop asserting

the write strobe(s). The master must hold the data, address, and status buses valid for at least two-tenths of a cycle after the write strobes are no longer asserted. Then the data, address, and status buses may begin to change, signifying the end of BS3 and the begining of the next bus cycle.

For timing parameters of the write cycle, complete with timing marks and data showing the minimum and maximum timing variations, refer to Figure 11*b* and Table 4 in the copy of the S-100 standard in the appendix.

THE INTERRUPT ACKNOWLEDGE CYCLE

When the master has been interrupted (by a device asserting one of the interrupt request lines), and assuming that the master's interrupts have not been masked, it will respond by executing an interrupt acknowledge cycle or cycles. During an interrupt acknowledge cycle the interrupting device is expected to put some kind of information on the data input bus that the master can use to determine what device has caused the interrupt, or where to go to find the interrupt service routine. This is usually accomplished by having the slave send a restart or call instruction during the interrupt acknowledge cycle.

The interrupt acknowledge cycle looks just like a normal read cycle, except that the status bus reflects an interrupt acknowledge status and the address bus is not defined (can contain any information, valid or not). The read strobe pDBIN will still be asserted and should be used by the interrupting device to gate the data onto the data input bus.

WHY TMA CYCLE TIMING WON'T BE EXPLAINED YET

As we have mentioned before, TMA timing is very involved, and to try and explain it now might confuse you. Therefore we will wait until Chapter 15 to explain it in detail.

THE BASIC BUS CYCLE WITH WAIT STATES

Up until now we have assumed that the slave can respond fast enough to latch the data or provide it to the bus before the master decides to take the read and write strobes away. This is an ideal condition, but there are times when a slave may have to cause the master to wait for it to respond. It does this by causing the master to insert one or more "wait states" into the basic bus cycle. The slave tells

the master to wait by pulling one of the ready lines low (saying "I'm not ready").

The timing diagram in Figure 4-5 and the flowchart in Figure 4-6 show the basic bus cycle with wait states (a read cycle is used as the example). The cycle starts out the same as a normal bus cycle. After pSYNC is asserted high, the slave will determine that it has been addressed. It should then pull either the RDY or XRDY line low. At the rising edge of Φ in BS2, the master samples the RDY and XRDY lines. If either ready line is low, the master will not enter BS3, but will instead enter BSW (Bus State Wait). The read or write strobes were asserted during BS2, and will be extended during BSW. The address and status lines also remain stable.

At the rising edge of Φ, during BSW, the master again samples the RDY and XRDY lines. If either line is still low, the master executes another BSW. It will again sample the RDY and XRDY lines at the rising edge of Φ. The process will repeat until both the RDY and XRDY lines return high. The master then enters BS3 and completes the cycle normally.

Circuits for generating wait states will be shown in the next chapter.

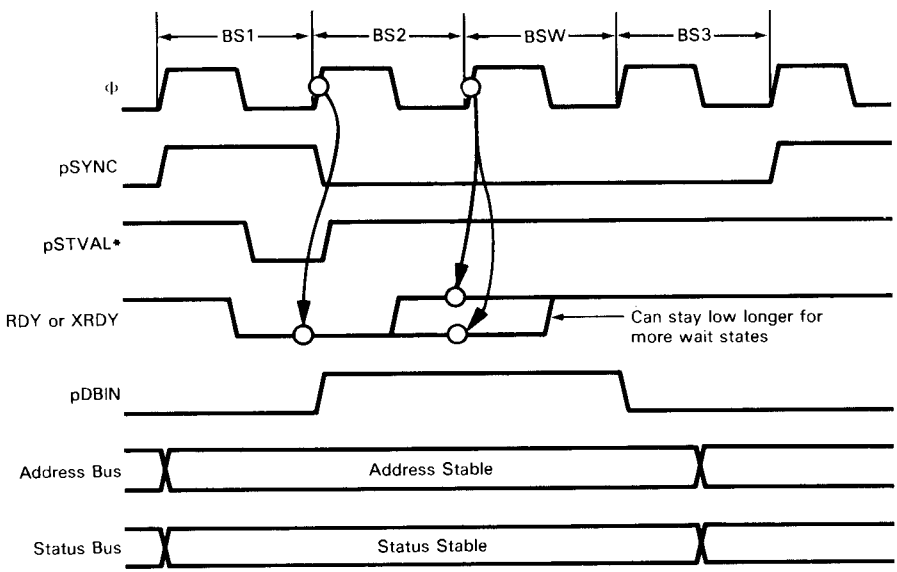


FIGURE 4-5. S-100 Bus Read Cycle with Wait States

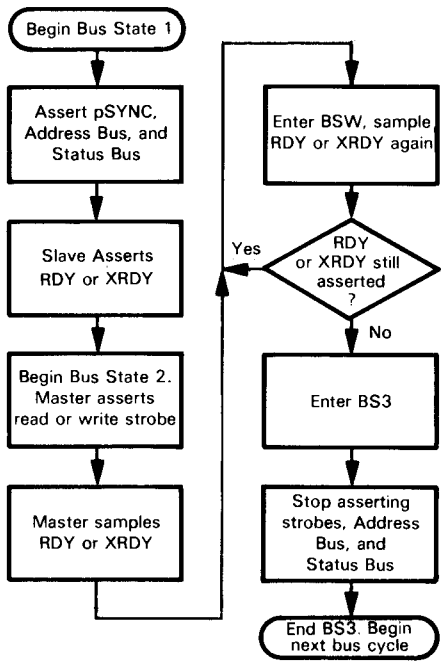


FIGURE 4-6. Bus Cycle with Wait State Flowchart

sXTRQ* AND SIXTN* TIMING: 16-BIT DATA TRANSFERS

Up until now we have assumed that all the data transfers have been only eight bits wide. The IEEE standard has provided a mechanism for transferring 16 data bits at once. Two new signals, sXTRQ* and SIXTN*, have been defined for that purpose. Refer to the timing diagram in Figure 4-7 and the flowchart in Figure 4-8.

If a 16-bit transfer is needed the status signal sXTRQ* is asserted by the master at the same time as the other status lines. This will be very near the beginning of the cycle in BS1.

If the addressed slave is capable of a 16-bit transfer, it will assert SIXTN* low. At the rising edge of Φ in BS2, the master samples the SIXTN* line. If it is low, the master will then gang the two 8-bit data buses together as one 16-bit bus and perform the transfer.

If the slave is not capable of a 16-bit transfer, it will not assert SIXTN*. When the master samples SIXTN* at the rising edge of Φ during BS2, it will find that the slave cannot perform the 16-bit transfer.

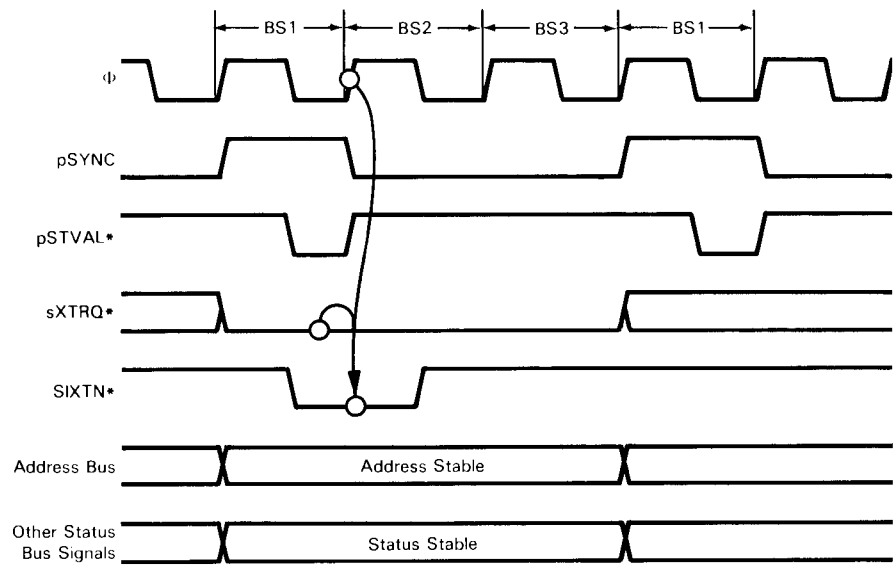


FIGURE 4-7. sXTRQ* and SIXTN* Timing.

If the master is smart enough, it can then perform the transfer as two consecutive 8-bit transfers. This is known as a "byte serial" transfer, because the information will be sent as two bytes of data, serially. If the master is not capable of a byte serial transfer, it must immediately cause an error condition, with ERROR* asserted. It will probably also cause an interrupt that causes a branch to an error handling routine. Hardware that saves the pertinent machine status during BS2 should be triggered by the ERROR* line, because the failed transfer will probably cause all kinds of havoc on the bus. A circuit for this will be presented in Chapter 16.

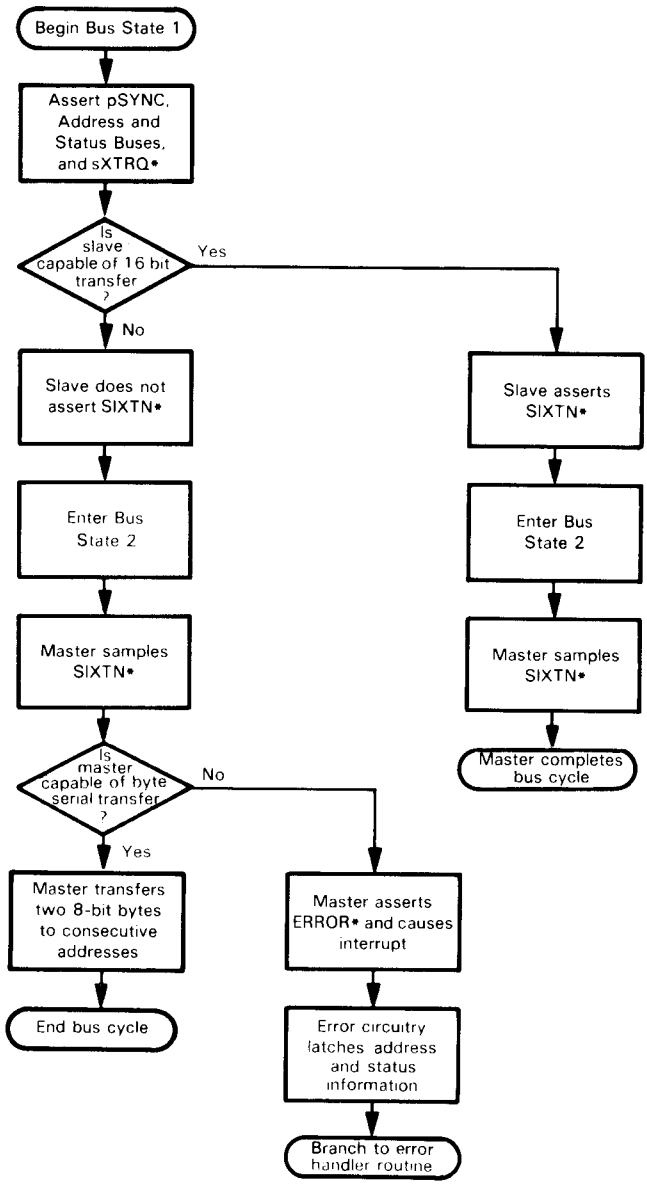


FIGURE 4-8. Sixteen-Bit Data Transfer Flowchart

decoding and buffering

5

INTERFACING TO S-100 COMPUTERS

In this chapter we will begin to explain how to connect circuits to the S-100 Bus. We are going to explain the basic circuitry that must be used to interface anything to the signals on the bus: how to determine when a slave is accessed, how to buffer the signals, what problems to look out for, etc.

We are going to show you how to decode all the relevant states that the S-100 Bus can be in, and explain why that is necessary. We will also explain what "buffering" is, and how to do it.

BUFFERING

Buffering is used to isolate an output signal on the bus from the many inputs that may be attached to it. These inputs are referred to as "loads." Every load that is placed on a signal line degrades the ability of the signal line to maintain proper logic levels. The term buffering means that the signal is isolated from the many loads that a board might require.

The term buffering is also used to describe the amplifying or strengthening of a signal which must drive many loads. ICs which drive large loads are called buffers or drivers.

Buffering is used in a third sense to describe the ability of a slave to float its output signals so that another slave's signals will not conflict. This is usually accomplished by the use of tri-state buffers or drivers. The term buffering in this instance means that one slave's outputs are buffered (isolated) from another's.

Incoming Signal Buffering

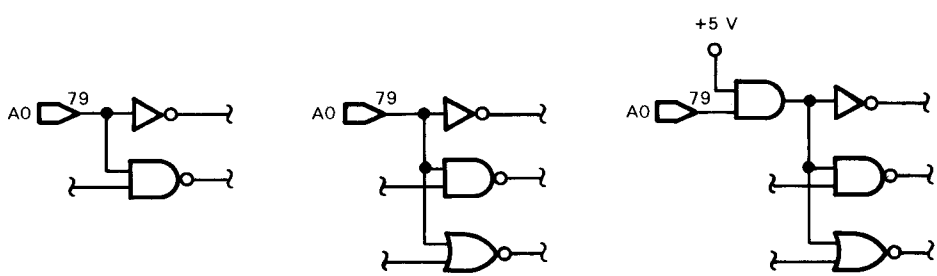
A slave that is plugged into the S-100 Bus will have signals coming into the board and signals going out from the board. A slave usually needs to input the address and status buses to determine if it is being addressed, input the incoming data bus, and input the bus strobes to determine when it may do things like drive the data bus.

The S-100 standard states that an S-100 card may not source more than 0.5 mA at +0.5 V nor sink more than 80 μA at +2.4 V on most signal lines. Hence, one should never connect any incoming S-100 signal line to more than one LS-TTL input. Two LS-TTL inputs will not meet the "worst case" specification. For example, the circuits in Figure 5-1*a* and Figure 5-1*b* do not comply with the standard. A non-inverting buffer of some type is needed between A0 and the inputs of the gates (Figure 5-1*c*). The buffer may be as simple as an unused non-inverting gate, such as an AND or OR gate, as shown in Figure 5-2*a*, or a tri-state buffer section that has its output permanently enabled, as shown in Figure 5-2*b*.

The output of a standard LS-TTL gate can drive the inputs of 20 other LS-TTL gates, or as it is commonly referred to, 20 LS-TTL loads. Another term for this is "fan-out." The fan-out would be 20 LS-TTL loads.

On the other hand, the fan-out of a tri-state driver is usually much higher (that is why it is called a driver). The 74LS367 shown in Figure 5-2*b* is capable of driving 40 LS-TTL loads. The type of buffer that you choose will depend on how many loads it has to drive.

Sometimes a slave may need the complement of an incoming signal. In this case it is acceptable to use an inverting gate or driver to do the buffering. Figure 5-3 shows some examples.



a. Not acceptable b. Not acceptable c. Acceptable

FIGURE 5-1. Meeting the "Worst-Case" Bus Loading Specification

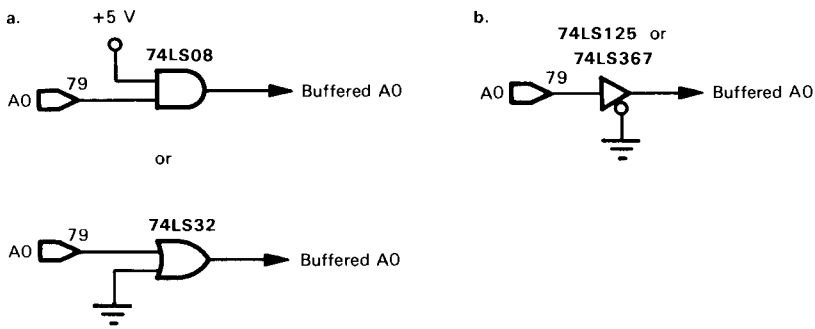


FIGURE 5-2. Non-Inverting Gates and Buffers

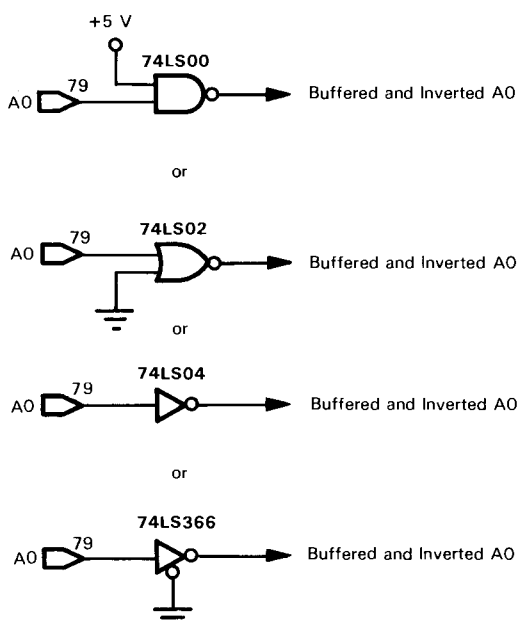


FIGURE 5-3. Inverting Gates and Buffers

Data Bus Buffering

The S-100 data input (DI) bus is implemented as a tri-state bus, where only one slave may drive the lines at a time. The strobe pDBIN is used to control when the slave may drive the bus. If the slave is being addressed and pDBIN is true, then the data output buffers on the slave may go from the high-impedance state to a low-impedance state, allowing the data out onto the bus. Slaves usually contain a signal (commonly known as "board select") that is true when the board has been addressed. (When we say addressed we mean that the address bus contains an address that the slave occupies, and that the status bus contains the proper information that distinguishes the slave's type of cycle from another type. For example, a memory status and address should not cause an I/O board to be selected.)

This board select signal (BDSEL) can be either active high or low, depending on the design of the decoder circuitry (we will discuss that later), but for purposes of this book we will always assume an active low BDSEL*. BDSEL* is used in conjunction with pDBIN to turn on a slave's data input bus drivers. BDSEL* is referred to as the "qualifier" because it qualifies a particular pDBIN pulse as being the one for this slave. A circuit that shows how to turn on a tri-state bus driver is shown in Figure 5-4. When BDSEL* is low and pDBIN is high, the output of the NAND gate will go low, causing the outputs of the 74LS244 to be enabled.

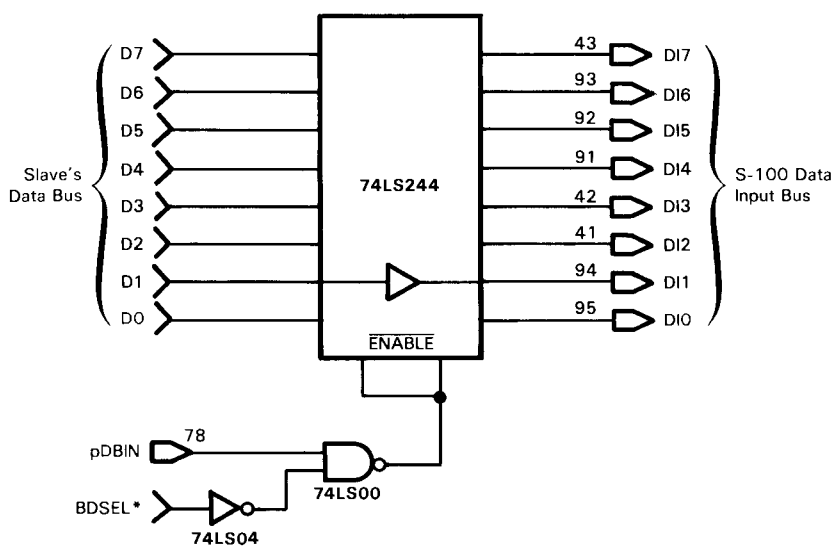


FIGURE 5-4. Enabling Tri-State Bus Drivers

The IEEE standard requires that any device that drives the bus must have an adequate fan-out. Fan-out is expressed by IC manufacturers as the amount of current sinking and sourcing capability that an output has. A standard LS-TTL gate can typically sink up to 8 mA, while some tri-state buffers can sink up to 50 mA. The standard states that the minimum amount of current sinking capability of any driven line must be at least 24 mA at +0.5 V. Drivers (except open collector drivers) must source at least 2 mA at +2.4 V. That rules out driving the bus with standard gates. Some common buffers that are capable of meeting the standard are 74LS125A, 74LS367A, 74LS244, and their inverting counterparts. Diagrams of these and other buffers are shown in Figures 5-5 through 5-8.

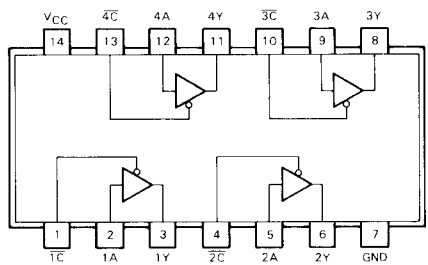


FIGURE 5-5. 74LS125A Quadruple Bus Drivers with Tri-State Outputs

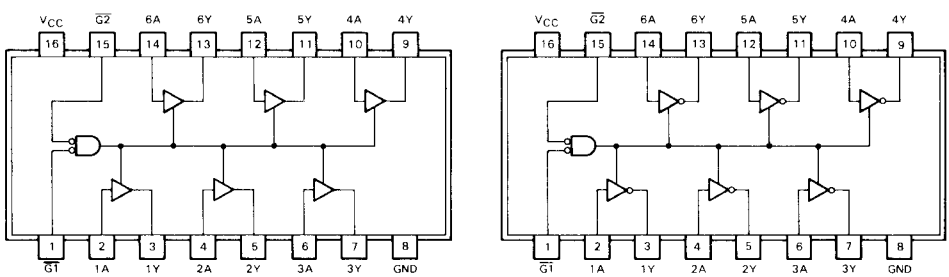


FIGURE 5-6. Hex Bus Drivers with Tri-State Outputs and Gated Enable Inputs

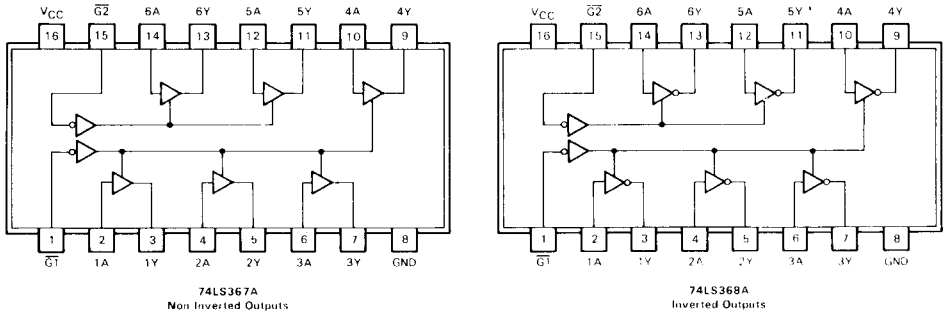


FIGURE 5-7. Hex Bus Drivers with Tri-State Outputs

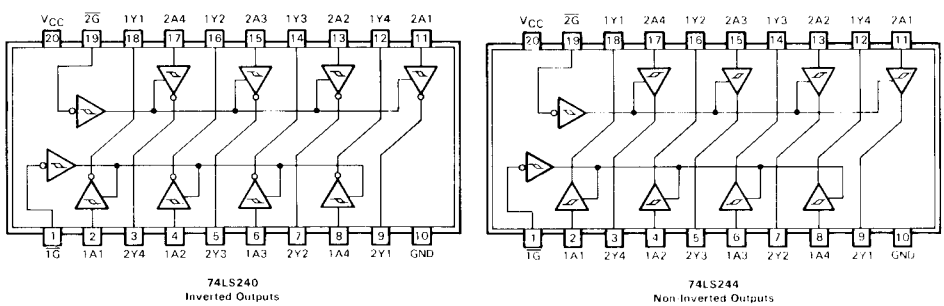


FIGURE 5-8. Octal Bus Drivers with Tri-State Outputs

OPEN COLLECTOR DRIVERS

Some of the signals on the S-100 Bus are specified as open collector lines. In a standard LS-TTL gate, the output is pulled up to a high logic level by a transistor. This is known as an active pull-up. In an open collector device, the output is not pulled up at all. The job is left to an external resistor.

This allows more than one open collector device to drive the same signal line. If any device is pulling the line low, the others will have no effect. Only if all device outputs are high will the line be high.

Some non-inverting and inverting open collector devices that meet the standard's drive requirements are shown in Figures 5-9 and 5-10. These figures show our convention for drawing an open collector gate. A diagonal line near the output end of the logic symbol is used in this book to identify an open collector output.

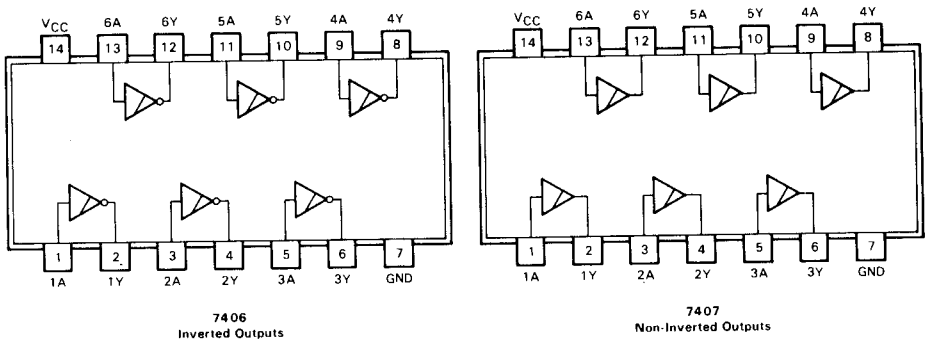


FIGURE 5-9. Hex Inverter Buffers/Drivers with Open Collector Outputs

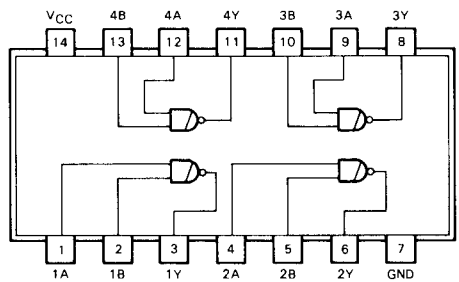


FIGURE 5-10. 74LS38 Quadruple 2-Input NAND Buffers with Open Collector Outputs

DECODING

A decoder is a circuit whose output or outputs relate to the "code" presented at its inputs. An address decoder might have three binary encoded address lines (A0-A2) as inputs and produce eight separate outputs, with an active level at each output corresponding to an individual code on the address inputs. Therefore each address would be decoded. A decoder might also have certain status lines as inputs and produce an output only when a certain cycle occurs.

The two most common types of decoders used on the S-100 Bus are address and status decoders. The following sections will show you how to build all kinds of decoders for S-100 Bus slaves. In fact, most of the remaining circuits in this book will use one or more of the following decoder circuits.

Status Decoder Circuits

There are three kinds of S-100 cycles that we will need to decode: memory cycles, I/O cycles, and interrupt acknowledge cycles. These can be further broken down into five subdivisions: memory read, memory write, input (I/O read), output (I/O write), and interrupt acknowledge.

All of the following circuits will produce a high logic level output when the type of cycle they decode is occurring. The circuits also meet the loading requirements of the standard.

Figure 5-11 shows a circuit that will decode a memory cycle. It will produce an active output for both memory reads and writes. If sMEMR is high, a memory read cycle is occurring, and this high will force the output of the OR gate high. If sWO* is low and sOUT is low, a write cycle is occurring, but not to an I/O port. Therefore the cycle must be a memory write. The two lows will also cause the output of the OR gate to go high.

A circuit is not really needed to decode a memory read cycle, since a status line already exists that indicates that a memory read is occurring. This line is sMEMR.

A portion of the circuit in Figure 5-11 is used to decode memory write status, and is shown in Figure 5-12. If sWO* is low and sOUT is low, a write cycle is occurring that is not to an I/O port, and therefore must be a memory write cycle.

A circuit that decodes an I/O cycle is shown in Figure 5-13. If either sINP or sOUT is high (meaning the cycle is either an I/O write or I/O read) then the output of the OR gate will also be high. When both inputs are low, the output of the OR gate will be low.

As with a memory read cycle, no circuitry is required to decode I/O reads or I/O writes, because the lines sINP and sOUT do this. The same is also true of interrupt acknowledge cycles because the line sINTA will go high only during an interrupt acknowledge cycle.

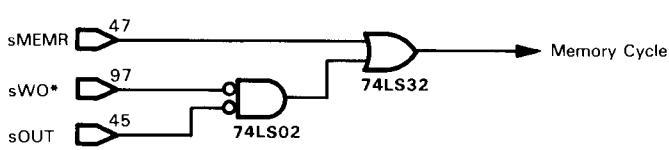


FIGURE 5-11. A Memory Cycle Decoding Circuit

There is one important thing to take note of here. The status decoders and status lines are status indicators and should not be used as strobes. This is because the status lines may be in an unknown state between cycles and cause the outputs or lines to show an invalid state (glitch). Strobes, on the other hand, are never allowed to glitch. Therefore, you should never use the "edge," or transition, of a status line or status decoder output, because false edges may occur before the lines have settled.

The signal pSTVAL* may be used to create a proper edge in conjunction with any status line or decoder. A suitable circuit is shown in Figure 5-14. When pSYNC is high and the inversion of pSTVAL* is high, the output of AND gate A will also go high. By this time, the status line (which is assumed to be active high) will have settled and will be true. The two highs at the inputs to AND gate B will cause its output to go high. Note that the output of the gate will only stay high for the duration of pSTVAL* during pSYNC; but since it is the leading edge we want, the duration does not matter. If a negative-going edge is required, an inverter may be added to the output of gate B, or a NAND gate may be substituted.

The reason that pSYNC needs to be included in the logic is that pSTVAL* may have many other edges during the cycle, but the only one that indicates valid status is the one during pSYNC.

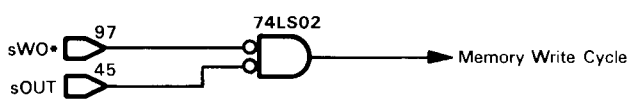


FIGURE 5-12. A Memory Write Status Decoding Circuit

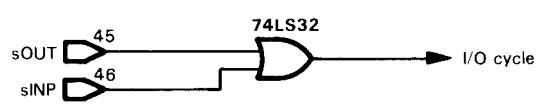


FIGURE 5-13. An I/O Cycle Decoding Circuit

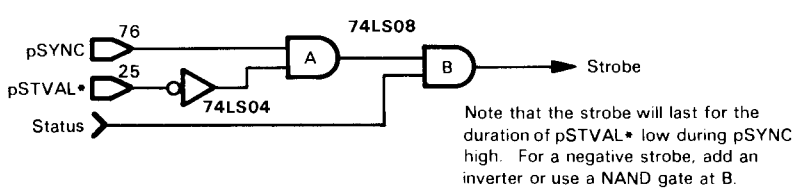


FIGURE 5-14. Circuit for Developing a Glitch-Free Strobe Signal

Address Decoder Circuits

Address decoders are used to tell a slave device that its particular memory location or I/O port is being addressed, rather than another's. The address bus contains binary coded information which needs to be decoded. It is necessary to determine two things when designing address decoders: the number of significant address inputs, and the number of locations that need to be decoded.

The number of significant address bits will vary, depending on whether the cycle is a memory or I/O reference, and whether or not extended addressing is being used. Remember that most current 8-bit masters have only 16 address lines. All 16 are used for memory addressing, but only eight are used for I/O addressing. But the standard has extended the number of address bits on the bus to 24 so that the newer masters that have up to 24 address lines may address more memory. Also, the newer masters may use 16 lines for I/O addresses.

So a memory decoder may have 16 or 24 address bits as inputs and an I/O decoder may have 8 or 16 address bits as inputs. We will show you how to design all four kinds of address decoders.

The second criterion is the number of addresses that need to be decoded. If we were making a memory board that contained 1 Kbyte of RAM, 1024 unique locations out of 65,536 would need to be decoded. However, if we used 1K memory chips, they would contain built-in decoders, so that we would simply have to present the 10 least significant address lines to the chips, which would take care of decoding the 1024 individual locations for us. The problem arises in distinguishing this 1K block from another 1K block. We would still need to decode the six most significant address bits to select which 1K memory chip block out of the possible 64 is to be used.

We would probably want to put more than one set of 1K chips on the board. If four sets were used, then we would need to decode four 1K blocks. We could do it with four separate 6-bit decoder circuits, and this would allow each 1K block to be addressed as any of the 64 possible blocks. However, most applications require that RAM reside in consecutive blocks. Now we may use two decoders, one that decodes the 4K block and one that decodes the individual 1K blocks within the 4K block. When these two circuits are combined, we call the result a decoder that decodes a 4K block into 1K segments or blocks.

Also needed is some convenient method to change the specific set of addresses decoded. This is commonly called selecting the starting address of a board. This is usually done with jumpers or DIP switches. All of our decoder examples will show switches, but you may use any method you prefer to implement the switch. DIP switches are usually chosen, because of their small size and convenience. Instead, you may elect to use full-size switches or encoded hexadecimal rotary switches, or you may simply hardwire a particular setting. The method is up to you, as long as you keep the function the same.

By decoding the proper address bits, decoders of any size and block size can be created. The following section will show memory address decoders for popular block sizes.

First we will discuss a few popular devices that are used to do decoding. The first is the 74LS266 quad exclusive NOR (XNOR) gate with open collector outputs. The connection diagram is shown in Figure 5-15. Any number of these gates may be combined to decode any number of bits. A circuit to decode four bits with a 74LS266 is shown in Figure 5-16. It works as follows. When the two inputs of a gate match (that is, when the address line matches the switch setting),

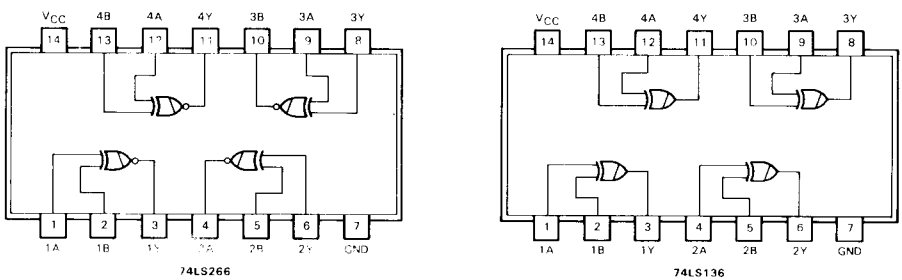


FIGURE 5-15. Quad Exclusive-NOR and -OR Gates with Open Collector Outputs

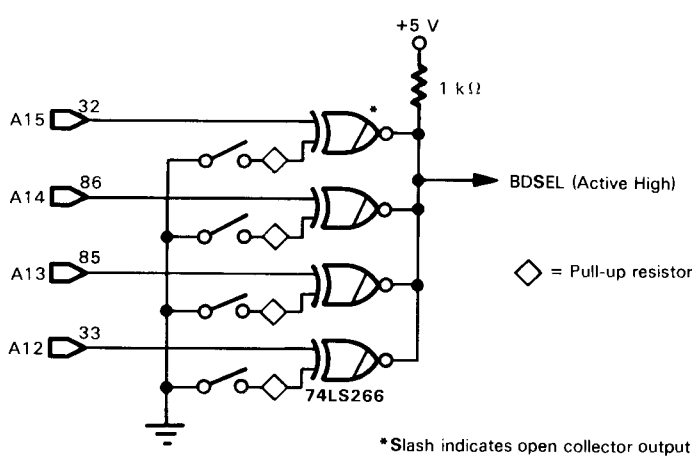


FIGURE 5-16. Circuit for Decoding Four Address Bits Using the 74LS266

the output of that gate will be high. If the two inputs don't match, the output will be low. Since all the outputs are tied together, any low gate output will win (hold the others low as well). So the output of the array (BDSEL) will go high only when all the inputs of the various gates match. This will only occur when the address lines are the same as the switch settings, so we have decoded that address. Note that a closed switch represents a binary 0 and an open switch represents a 1.

The same circuit may be built using a 74LS136 quad two-input exclusive OR (XOR) gate with open collector outputs (the connection diagram is shown in Figure 5-15). The difference is that now the output of any gate will go high only when the two inputs are opposite. We could invert the address lines to exactly duplicate the previous circuit, but why not just invert the switch function? Now a closed switch would represent a binary 1 on the address line, and an open switch would be a binary 0. In fact, this is more "logical" to the user. A switch that is off (open) is a 0 and an on (closed) switch is a 1. The circuit is shown in Figure 5-17.

The two previous circuits may be expanded simply by adding more sections and tying together the outputs. If fewer bits need to be decoded, gates can be deleted.

Notice that the circuits shown in Figures 5-16 and 5-17 contain diamond-shaped symbols in some of the signal lines. This is the symbol we will use to represent a pullup resistor. This symbol means that a resistor is to be connected between the +5 volt supply and the signal line in which the symbol appears; 5100 ohms is an appropriate value for the resistor.

Another IC that is useful in building decoders is the 74LS138, itself called a

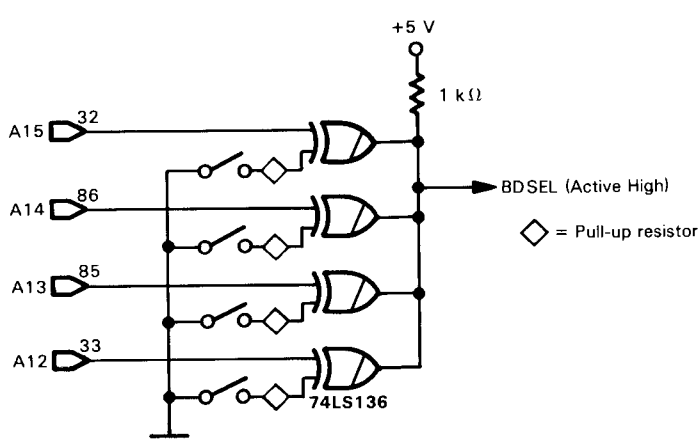
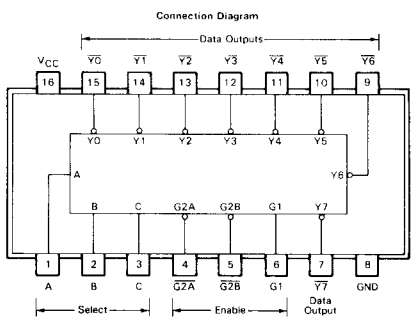


FIGURE 5-17. Circuit for Decoding Four Address Bits Using the 74LS136

decoder. It is a three- to eight-line decoder which activates one of eight outputs; the output that is activated depends on the bit pattern or code at the three binary encoded inputs. It also contains three "enable" inputs. These inputs must be at the proper logic level in order for any output to be active. A connection diagram, function diagram, and truth table for the 74LS138 are shown in Figure 5-18*a*.

Similar to the 74LS138 is the 74LS139, a dual two- to four-line decoder. It has two identical sections that decode two binary inputs to one of four possible outputs. Each section has only one enable input. A connection diagram, function diagram, and truth table for the 74LS139 are shown in Figure 5-18*b*.

With these four ICs, and assorted other gates, it is possible to build almost any kind of decoder. We will now show you some typical address decoders with various numbers of inputs and outputs. Each of the decoder schematics will have a circled letter in the upper left-hand corner. The reason for this is that you will need to use one or more of these circuits in conjunction with every other circuit in this book. Rather than redraw all the decoder circuits each time, we will show a block with the same circled letter in it. This represents the decoder circuit with the matching letter.



Truth Table

Inputs					Outputs							
Enable		Select										
G1	$\overline{G2}$*	C	B	A	$\overline{Y0}$	$\overline{Y1}$	$\overline{Y2}$	$\overline{Y3}$	$\overline{Y4}$	$\overline{Y5}$	$\overline{Y6}$	$\overline{Y7}$
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

$\overline{G2} = \overline{G2A} + \overline{G2B}$
H = high level, L = low level, X = irrelevant

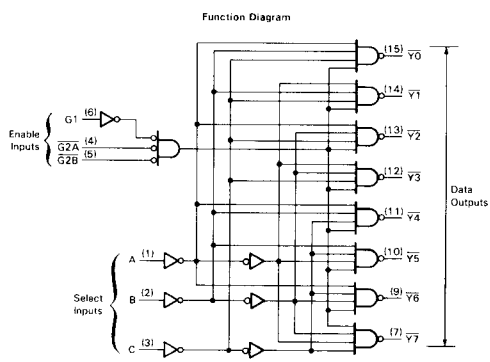
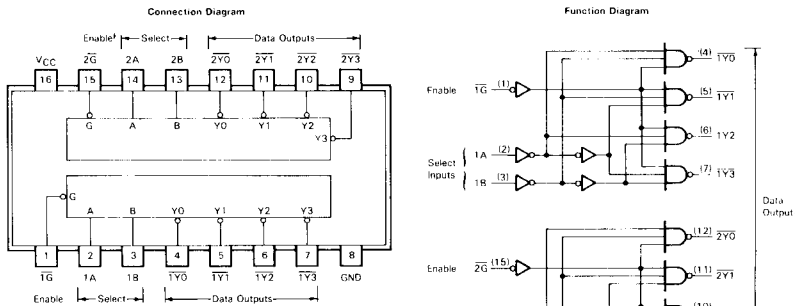


FIGURE 5-18a. 74LS138 3- to 8-Line Decoder



Truth Table
(Each Decoder/Demultiplexer)

Inputs			Outputs			
Enable	Select					
$\overline{G}$	B	A	$\overline{Y0}$	$\overline{Y1}$	$\overline{Y2}$	$\overline{Y3}$
H	X	X	H	H	H	H
L	L	L	L	H	H	H
L	L	H	H	L	H	H
L	H	L	H	H	L	H
L	H	H	H	H	H	L

H = high level, L = low level, X = irrelevant

FIGURE 5-18b. Dual 2- to 4-Line Decoder

Memory Address Decoder Circuits

The circuits shown in Figures 5-19 to 5-28 are used to decode memory addresses. All the decoders are designed primarily for 16 bits of memory addressing, but they all contain a "hook" to add in an extended address term. This hook is always a section of a 74LS136, and may be eliminated entirely if extended addressing is not required.

We will show you decoders for 1K, 2K, and 4K blocks of memory. Some will decode one block, some four blocks, and some eight blocks. These combinations can be used to construct almost any type of memory array.

You will notice a great similarity between all of the different decoder circuits; in fact, they are almost identical. About the only thing that changes is the number of address bits that are decoded — the fewer bits that are decoded, the larger the block size.

The first is a decoder that can decode one 1K block of memory. It is shown in Figure 5-19. It works exactly like the circuit in Figure 5-17; in fact, it is the same circuit with more sections. When the information on the address inputs is exactly the opposite of the setting on the switches, and if PHANTOM* is not asserted, the output BDSEL* will go low. Because the upper six address bits are being decoded, there are 64 possible addresses that may be set on the switches (2^6 = 64). Therefore the circuit will decode one of 64 possible 1K blocks.

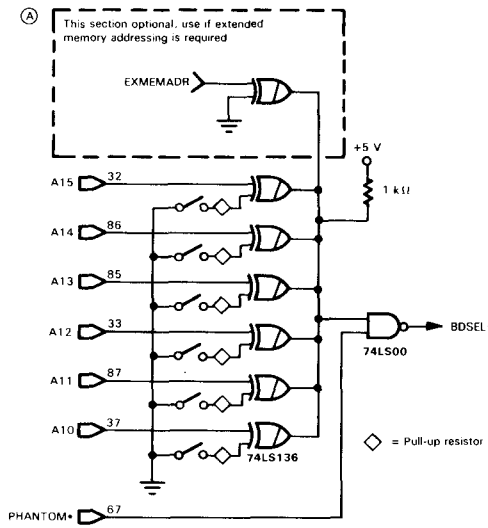


FIGURE 5-19. Memory Address Decoder for One 1K Block

The setting on the switches will determine which 1K block the decoder will decode. The switch setting is in binary.

Note that the section of the schematic in dotted lines is required only if you intend to decode the full extended address available on the S-100 Bus. The signal EXMEMADR originates on the schematic in Figure 5-28; you should combine the two schematics to make the full decoder. If you do not require the extended address, the section of the schematic in the dotted lines may be omitted entirely.

The circuit in Figure 5-20 is almost the same as the previous one, except that it has one less address bit connected to it. Therefore it will decode one 2K block, out of a possible 32, because it has five address lines going in ($2^5 = 32$). The circuit is addressable on any 2K boundary by the setting on the switches.

The circuit in Figure 5-21 is again almost the same as the previous two. This time, only four address bits are decoded. This will decode one 4K block out of a possible 16 ($2^4 = 16$). The circuit is addressable on any 16K boundary by the setting on the switches.

The circuit in Figure 5-22 is used to decode one to four 1K blocks. The 74LS136 array is used to decode the 4K block, and the 74LS139 is used to decode the individual 1K blocks within the 4K block. The signal BDSEL* (board select) will be active when any of the 1K blocks is selected, and one of the SELA*-SELD* outputs will be active to indicate which of the 1K blocks it is.

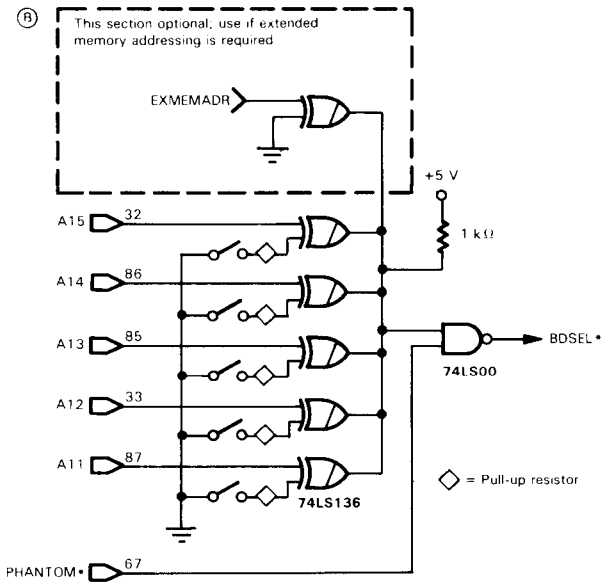


FIGURE 5-20. Memory Address Decoder for One 2K Block

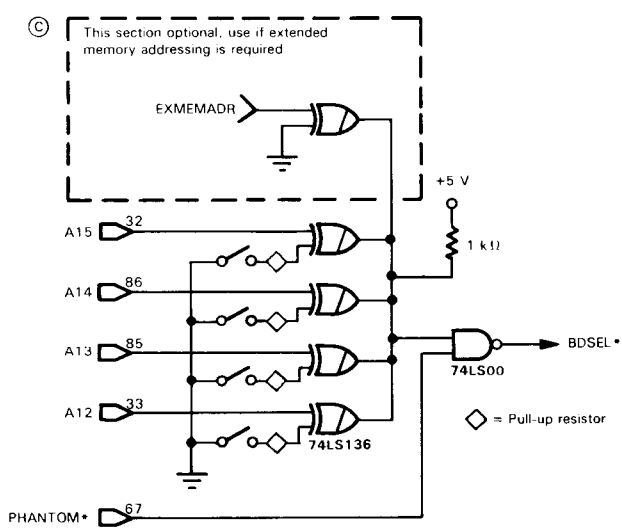


FIGURE 5-21. Memory Address Decoder for One 4K Block

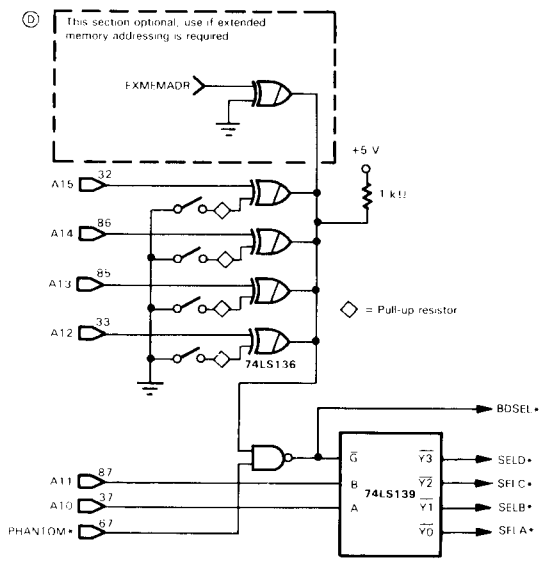


FIGURE 5-22. Memory Address Decoder for One to Four 1K Blocks

Here is how it works. The signal BDSEL* will be high if the address lines do not "match" the settings on the switches, or if PHANTOM* is asserted. This is applied to the enable (gate) input of the 74LS139. When this enable input is high, none of the outputs will be active; they will all be high.

When the address lines A12-A15 match the settings on the switches, the signal BDSEL* will go low and cause the enable input on the 74LS139 to go low. This will allow one output of the 74LS139 to go low also. The output that goes low will depend on the state of address lines A10 and A11. If they are both low, SELA* will go low; if A10 is high and A11 is low then SELB* will go low; if A10 is low and A11 is high then SELC* will go low; and if they are both high then SELD* will go low.

The circuits in Figures 5-23 and 5-24 work in exactly the same fashion, except that one decodes up to four 2K blocks and the other up to four 4K blocks. The circuit in Figure 5-23 is addressable on any 8K boundary and the circuit in Figure 5-24 is addressable on any 16K boundary.

Do you begin to see the pattern here?

The circuit in Figure 5-25 is an extension of the one in Figure 5-22, except that the 74LS139 has been replaced with a 74LS138. This provides four more 1K

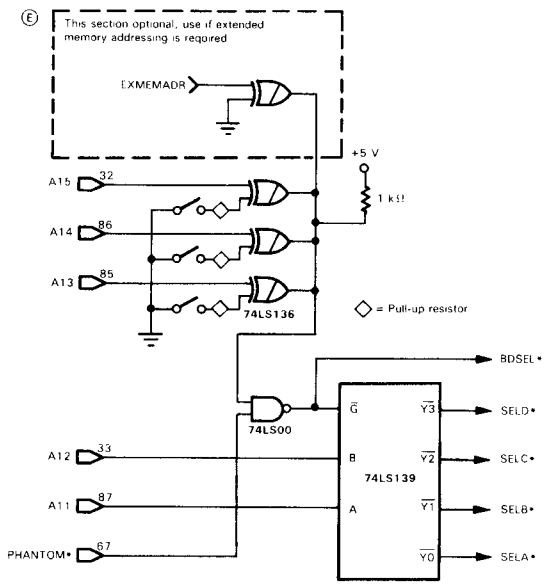


FIGURE 5-23. Memory Address Decoder for One to Four 2K Blocks

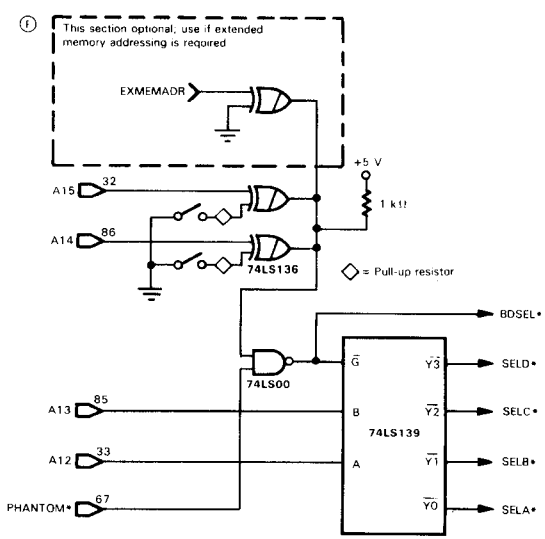


FIGURE 5-24. Memory Address Decoder for One to Four 4K Blocks

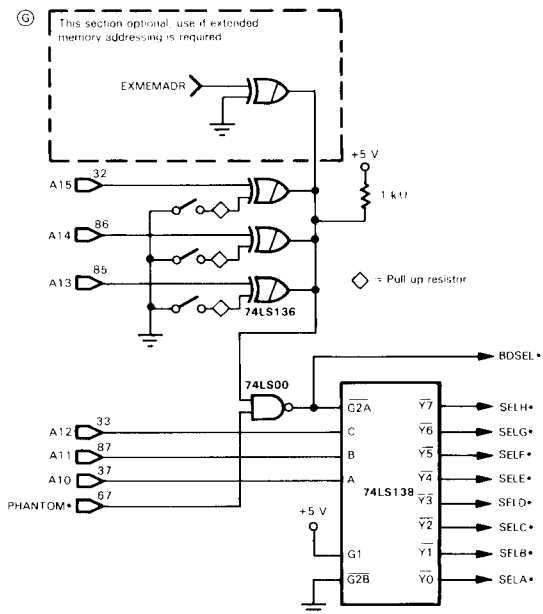


FIGURE 5-25. Memory Address Decoder for One to Eight 1K Blocks

blocks, and therefore takes an extra address bit into the 74LS138. This bit moves down from the 74LS136 array to the 74LS138, because we are now decoding a bigger block (8K), and that takes fewer address bits to decode, but the eight individual selects take one more. Notice that all the 1K address decoders input the same number of address bits, but their position in the decoder "tree" is different. Is that pattern starting to make itself evident?

The following two circuits are extensions of the previous one, but the circuit in Figure 5-26 decodes up to eight 2K blocks and the circuit in Figure 5-33 decodes up to eight 4K blocks.

The circuit in Figure 5-26 is addressable on any 16K boundary, and the circuit in Figure 5-27 is addressable on any 32K boundary (but in a 64K system there are only two).

The signal PHANTOM* has been included in the logic of all the previously described memory address decoders. PHANTOM* is a signal that is used to disable a memory block so that another block may temporarily exist in the same address

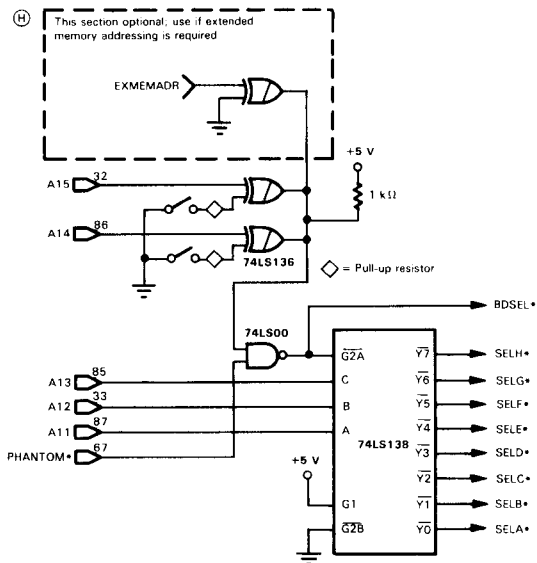


FIGURE 5-26. Memory Address Decoder for One to Eight 2K Blocks

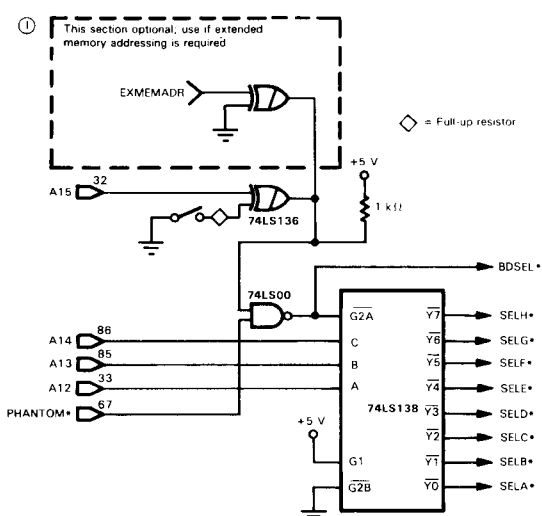


FIGURE 5-27. Memory Address Decoder for One to Eight 4K Blocks

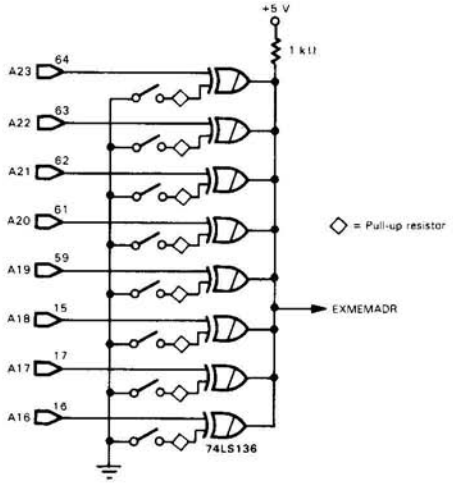


FIGURE 5-28. Extended Memory Address Decoder Circuit

space (see Chapter 3 for a more technical description of PHANTOM*). If PHANTOM* is asserted, the address decoder circuitry will be effectively disabled, causing the memory to disappear. If the address decoder must not respond to PHANTOM*, replace the 74LS00 with a 74LS04.

The circuit in Figure 5-28 may be added to any of the previous decoder circuits to give them extended addressing ability. It is the basic 74LS136 circuit, but the address inputs are the new extended address lines A16-A23. The output, EXMEMADR, is connected to the EXMEMADR input in any of the previous decoder schematics. This will allow any of the previous circuits to reside in any of the 256 possible 64K "pages."

I/O Address Decoder Circuits

The circuits shown in Figures 5-29 to 5-37 are basically the same as the memory address decoders described in the preceding section. Only a few things are different, so rather than go through all the explanations again, we will only discuss the differences.

If you look at the circuit in Figure 5-29, you will notice the familiar 74LS136 array. But notice that A7, rather than A15, is now the most significant bit (not counting the address extension, which we will discuss later). That is because only

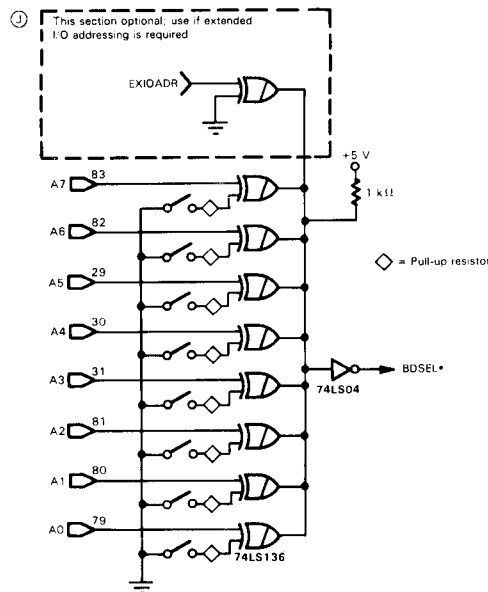


FIGURE 5-29. I/O Address Decoder for One I/O Port

256 I/O ports are addressed by most masters, and that only takes eight bits.

Now that we are dealing with I/O addresses, the block size is vastly different from that of memory addresses. Now the biggest block size is two ports (two unique addresses), whereas with memory our biggest block size was 4K (4096 unique addresses). The decoders remain the same, but the range is shifted down towards the least significant address bits.

The only new wrinkle we have introduced in this section is the addition of a decoder for two ports or two blocks of ports. (Before, we had decoders for one, four, and eight blocks. Now we have decoders for one, two, four, and eight blocks.) This is because many of the LSI I/O chips take up a block of two port addresses.

An example of this new circuit is shown in Figure 5-30. Instead of using a "decoder" IC, we have "built" a small decoder out of two NAND gates and an inverter. Here is how it works. When the address matches the switch settings, the output of the 74LS136s (BDSEL) will go high. This will cause one input to each of the NAND gates to go high. When address bit A0 is low, NAND gate B will have a high and a low at its inputs and therefore its output (IOSELB*) will be high. A0 will be inverted by the 74LS04 and therefore NAND gate A will have two highs at its inputs, so its output (IOSELA*) will go low.

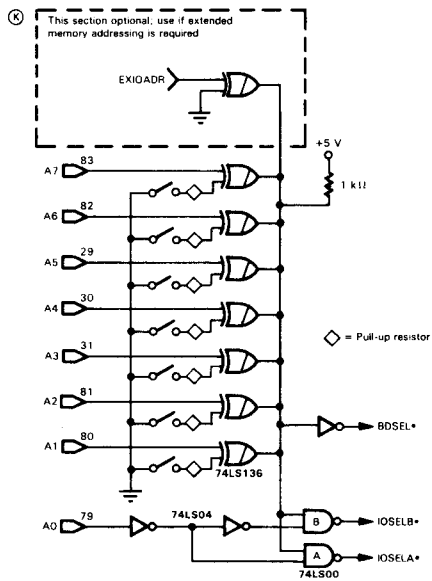


FIGURE 5-30. I/O Address Decoder for Two I/O Ports

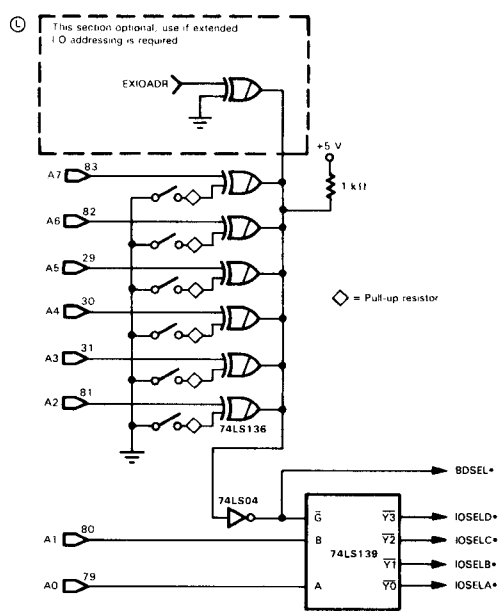


FIGURE 5-31. I/O Address Decoder for One to Four Single I/O Ports

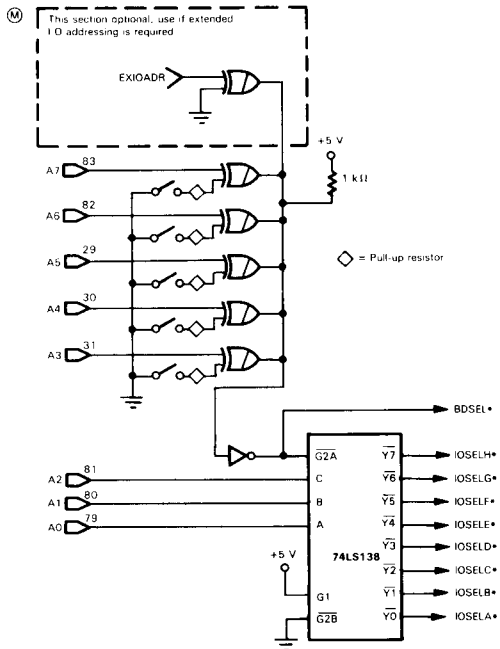


FIGURE 5-32. I/O Address Decoder for One to Eight Single I/O Ports

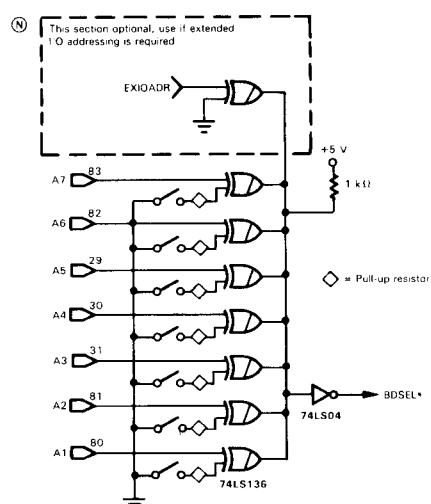


FIGURE 5-33. I/O Address Decoder for One Block of Two I/O Ports

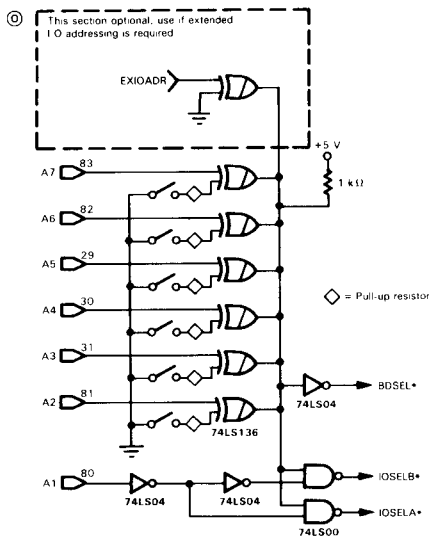


FIGURE 5-34. I/O Address Decoder for Two Blocks of Two I/O Ports

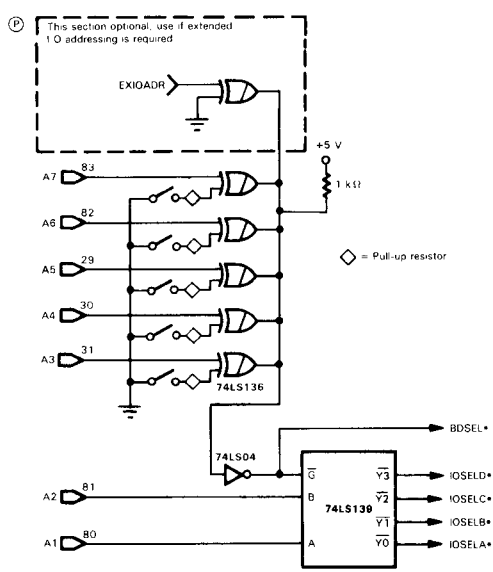


FIGURE 5-35. I/O Address Decoder for One to Four Blocks of Two I/O Ports

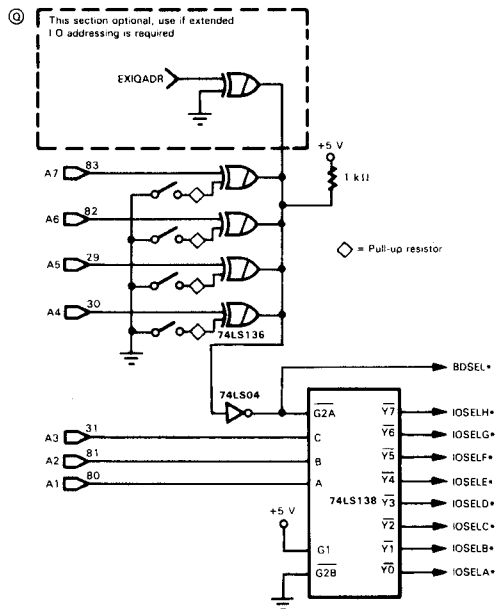


FIGURE 5-36. I/O Address Decoder for One to Eight Blocks of Two I/O Ports

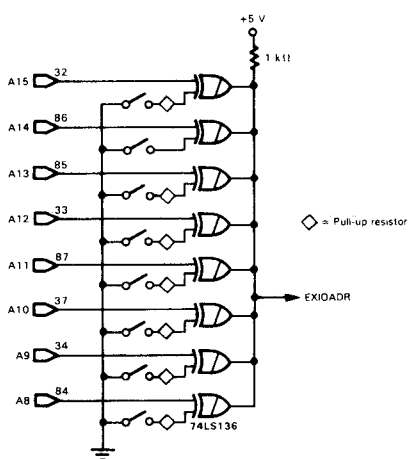


FIGURE 5-37. Extended I/O Address Decoder Circuit

When A0 is high, the whole process is reversed, causing IOSELB* to go low and IOSELA* to remain high. When the address does not match the switch settings, the output of the 74LS136s (BDSEL) will be low and therefore IOSELA* and IOSELB* will both be high (inactive).

The last difference is that the extended I/O address bits are A8-A15, rather than A16-A23 as in the memory section. The extended I/O address decoder shown in Figure 5-37 is used in conjunction with all of the other I/O address decoders, just as the extended memory address decoder was used with the memory decoders. If you want extended I/O addressing, just connect the EXIOADR output to the EXIOADR input of a decoder circuit, and if you do not want extended I/O addressing, just leave the associated circuitry out.

Figures 5-30 through 5-36 show I/O address decoders for various block sizes and numbers of blocks.

STROBE QUALIFIERS

Strobes are the signals that tell the system *when* to do something, or *when* some information on one of the buses is valid.

The strobes we are concerned with in this section are the strobes that qualify the data buses. They are pDBIN, pWR*, and MWRT. A complete description of these signals and their timing relationships to the rest of the system is contained in Chapters 3 and 4.

A "strobe qualifier" is a circuit that allows a strobe to affect another circuit only if that circuit is addressed. Otherwise the strobe is "masked" from the circuit. For example, the strobe pDBIN is used by a slave to gate its data onto the data input bus. The same strobe is used by all slaves, be they memory or I/O, and of course a normal system generally has more than one memory board and more than one I/O board. A pDBIN strobe qualifier circuit allows pDBIN to affect *only the addressed slave.* So pDBIN would have to be qualified with the status and address bus decoder outputs. A circuit that does this is called a strobe qualifier.

A strobe qualifier for memory read cycles is shown in Figure 5-38. When the signal BDSEL* is low (signifying that a memory location on the board is being addressed), sMEMR is high (signifying a memory read cycle), and pDBIN is high (the active state of the read strobe), the output of the NAND gate will go low. This will enable the outputs of the tri-state buffer, allowing the data from the addressed memory location to be gated onto the bus. When pDBIN goes low, the output of the NAND gate will return high, disabling the outputs of the buffer.

A similar circuit for I/O read cycles is shown in Figure 5-39. When BDSEL* is low (signifying that an I/O port on the board has been addressed), sINP is high (signifying an I/O read cycle), and pDBIN is high, the output of the NAND gate will

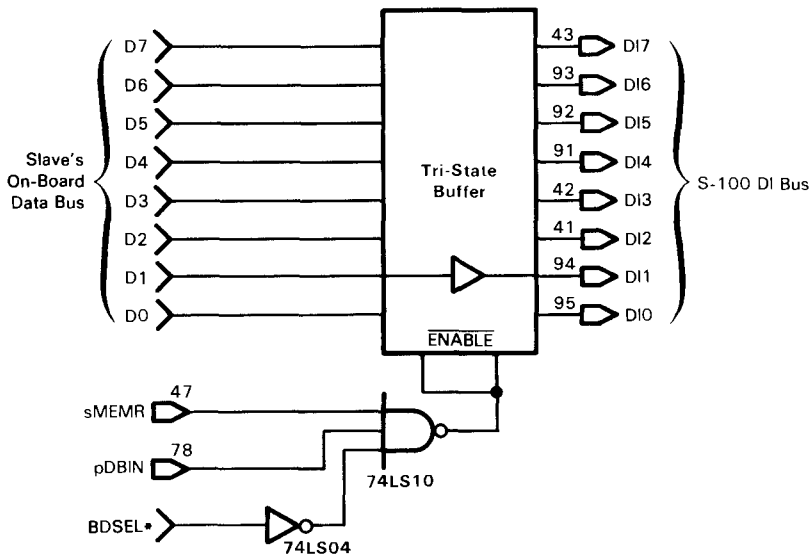


FIGURE 5-38. Memory Read Strobe Qualifier Circuit

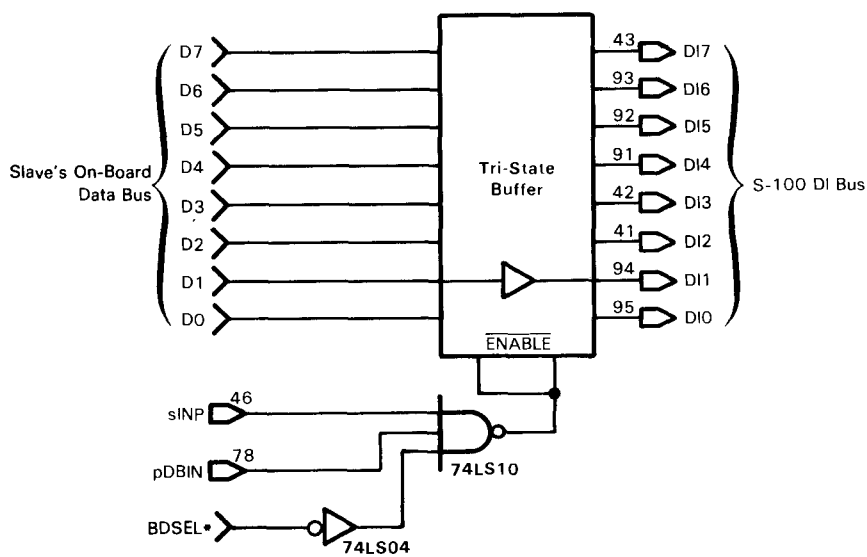


FIGURE 5-39. I/O Read Strobe Qualifier Circuit

go low, enabling the buffer. When pDBIN goes low, the output of the buffer will return to the high-impedance state.

The circuits in Figure 5-40 show three different memory write strobe qualifiers. All accomplish the same thing. Gate A has sWO* and sOUT applied to its inputs. It will produce a high at its output only when both inputs are low, signifying a write cycle that is not to an I/O port. (You may remember this as the status memory write decoder presented earlier.) When the output of gate A is high, and pWR (pWR* that is inverted by the 74LS04) is high, then the output of the NAND gate will go low.

The second circuit in Figure 5-40 inverts sOUT and pWR* and applies them to the input of a NAND gate. This is the same as the circuit above except that the sWO* term has been eliminated. This is because there are only two types of write cycles allowed on the bus: memory and I/O. If sOUT is low, the present cycle is not an I/O cycle, and if pWR* occurs, then it must be a memory write cycle.

The last circuit is the simplest of them all. It just inverts the MWRT strobe. This may be used reliably, assuming MWRT is generated correctly in your system (see Figure 3-3 for a proper MWRT generator).

None of these circuits includes the term BDSEL, as the other strobe qualifiers do. This is because all memory chips have a "chip select" signal that must be asserted before the write strobe will affect the chip. Therefore, the memory chip itself qualifies the write strobe with the BDSEL* signal.

The circuit in Figure 5-41 shows a write strobe qualifier for an output port. If sOUT is high (signifying an I/O write cycle), BDSEL* is low (signifying that this

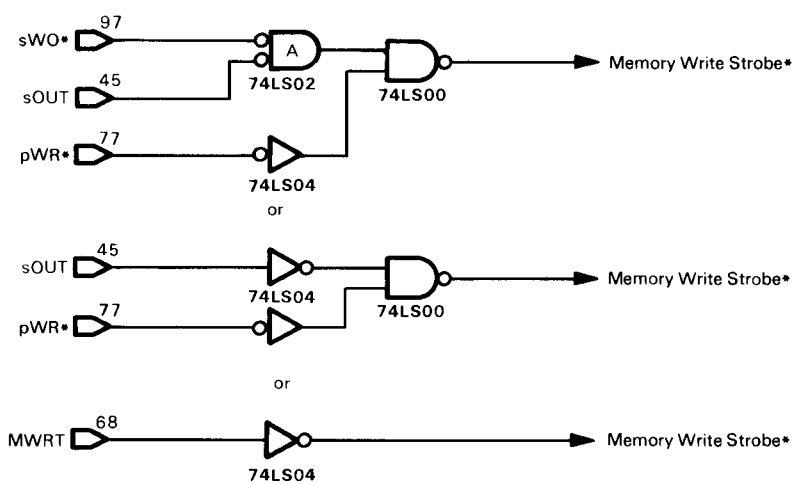


FIGURE 5-40. Memory Write Strobe Qualifier Circuits

particular port has been addressed), and the inversion of pWR* is high, the output of the NAND gate will go low. This would qualify pWR* for an I/O port access.

The circuit in Figure 5-42 shows a strobe qualifier for an interrupt acknowledge cycle. When sINTA is high (signifying an interrupt acknowledge cycle) and pDBIN is high, the output of the NAND gate will go low. This signal, INTA*, would usually be applied to the INTA* input of an interrupt controller IC and also be used to gate the "interrupt vector" onto the data input bus. Refer to Chapter 13 for more information about interrupts.

DATA BUS BUFFERING

Most slaves transfer information over the S-100 data buses. In most cases these data lines must be buffered. The incoming data bus will usually need to be buffered so that the slave does not present too great a load to the data output lines. Data lines from the slave must be buffered with sufficient drive for the data input bus. Also, the slave must enable its tri-state data input bus buffers only at the appropriate times. The following circuits show some typical data bus buffer arrangements.

The circuits in Figure 5-43 are for those slaves that have one bus for incoming

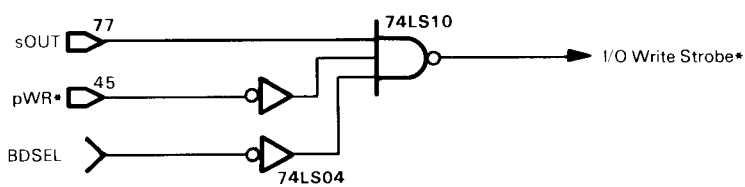


FIGURE 5-41. I/O Write Strobe Qualifier Circuit

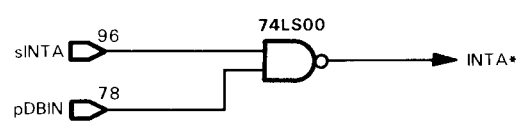


FIGURE 5-42. Interrupt Acknowledge Strobe Qualifier Circuit

data and one bus for outgoing data. These are called "unidirectional," because the data flows in one direction only. The DI and DO buses are both unidirectional.

The circuit in Figure 5-43*a* would be used for gating data out onto the DI lines during a qualified read strobe (RD*). This strobe is assumed to be active low. Any of the previous read strobe qualifier circuits will provide a proper strobe.

The circuit in Figure 5-43*b* is used for buffering incoming data from the DO bus. Since this data will be strobed into the circuitry on the slave by a proper write strobe, the outputs of the buffer are left enabled all the time by tying the active low enable inputs to ground.

The circuit in Figure 5-44 is the most common data bus buffer arrangement. Most slaves will have a bidirectional data bus on-board (there are many reasons for this; among them are that most peripheral ICs have a bidirectional data bus, and that it takes up eight fewer lines). The S-100 data bus normally uses two unidirectional buses, but this circuit turns them into an on-board bidirectional bus. Here is how it works. Assume for the moment that a qualified read strobe signal (that is active low) is not asserted. This high level will be inverted by the 74LS04 whose low output will enable the tri-state driver that is buffering the DO bus. The other buffer will be in a high-impedance state. This will allow data on the DO bus to flow into the slave's bidirectional data bus, and the slave will not be driving the S-100 DI bus. When a qualified read strobe occurs, the data on the slave's bidirectional bus will be gated onto the S-100 DI bus. This happens because the

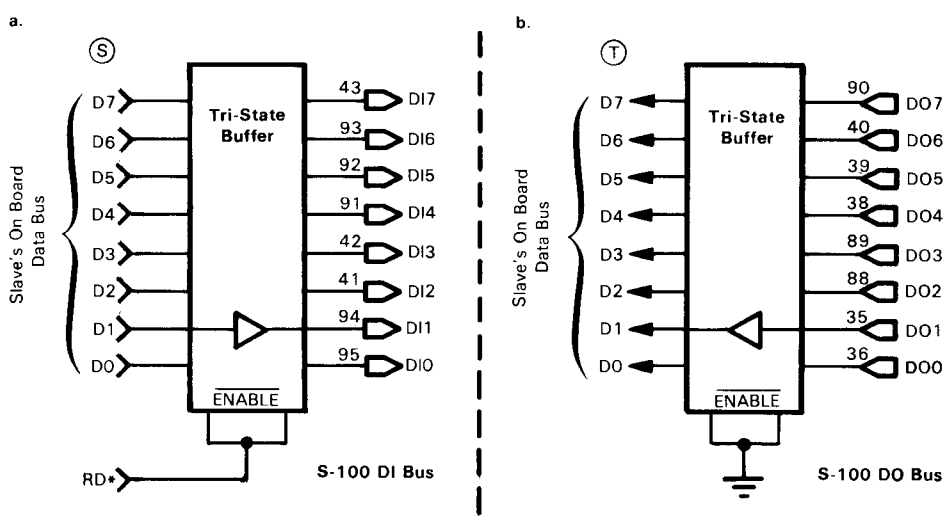


FIGURE 5-43. Data Bus Buffers for Unidirectional Buses

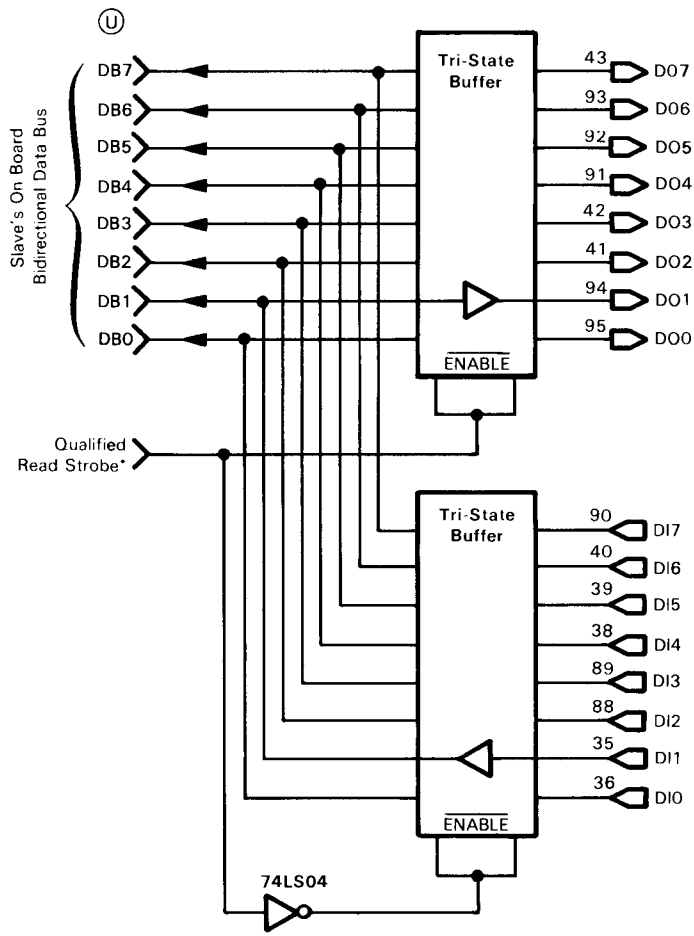


FIGURE 5-44. Bidirectional Data Bus Buffers

DI bus driver is now enabled. The DO bus buffer is now disabled so that its outputs will not conflict with the data on the slave's bidirectional data bus.

Do not use the write strobe to enable the DO bus driver, because you want the data on the slave's bus to be stable before and after the strobe occurs. If the write strobe were used to gate the data onto the board, the data would be changing at the edges of the strobes, which is forbidden.

WAIT STATE GENERATORS

In the previous chapter we discussed the S-100 Bus timing with and without wait states. During this discussion it might be helpful to refer to Figures 4-5 and 4-6.

Wait states are used by slaves that are not fast enough to read or write the data from the master during a normal bus cycle. Such slaves will cause one or more wait states to allow the slave to catch up. In older systems, wait states were almost never needed, but with current CPU chips running faster and faster, the I/O chips are having a hard time keeping up. Memory devices are getting faster, but the LSI peripheral chips are still relatively slow. EPROMs are also typically slower than other forms of memory, and therefore wait states are generally needed.

The following circuits show three different wait state generators. The first generates one wait state, the second generates two, and the last will allow the generation of zero to eight wait states, the number being selected by a switch.

The circuit in Figure 5-45 will generate one wait state whenever the particular slave is accessed (BDSEL* is low). On the negative-going edge of Φ after pSYNC goes high, the inverter's (74LS04) output will go high, clocking the high level at the D input of the 74LS74 to the non-inverting output Q. The inverting output $\overline{Q}$ will go low, and the 7406 will invert that again, causing the RDY line to go high. This is the normal state of the RDY line and no wait state will be generated. Think of the RDY line as the wait request* input to the master.

When BDSEL* is low (signifying that this slave has been accessed) and pSYNC is high (signifying the start of a bus cycle) the output of the NAND gate

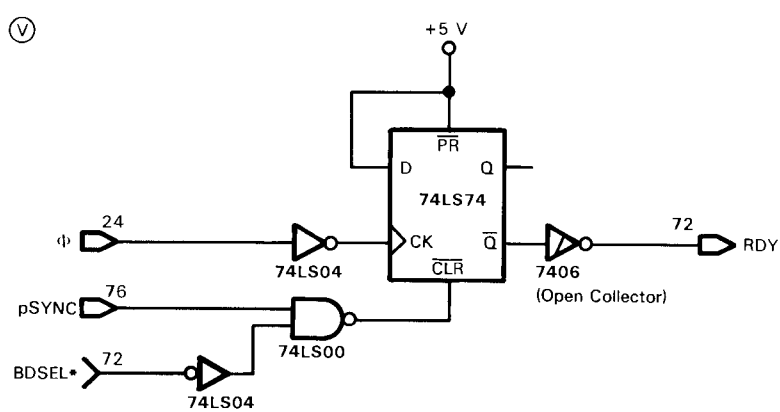


FIGURE 5-45. Circuit to Generate One Wait State

will go low, setting the non-inverting output of the flip-flop high. This is inverted by the 7406, which pulls the RDY line low, requesting a wait state.

At the next falling edge of Φ, the clear input of the flip-flop will remain low because pSYNC and BDSEL* will still be active. This overrides the clock input to the flip-flop, so nothing will happen, and RDY stays low. The master samples the RDY line at the next rising edge of Φ and enters a wait state. Shortly thereafter, pSYNC will go low, causing the clear input of the flip-flop to go high. The next negative-going edge of Φ will then clock the high level at the D input through, causing the RDY line to go high again. The CPU will sample the RDY line at the next rising edge of Φ, find it high, and end the wait state.

The circuit in Figure 5-46 adds a second flip-flop to the previous circuit. Both flip-flops are cleared by the same signal, starting the first wait state. When the clear inputs return high, the next negative-going edge of Φ will clock a low level into the B flip-flop which will leave the RDY line low, causing a second wait state to be entered. The same edge of Φ will clock a high into the D input of flip-flop A, causing the Q output of A to go high. Therefore, at the next falling edge of Φ this high will be clocked through flip-flop B, causing RDY to go high, ending the wait state. Thus two wait states have been generated. This circuit is basically a two-stage parallel load shift register.

The circuit in Figure 5-47 expands this circuit with shift register IC, a 74LS165. This IC has eight flip-flops (A through H) which are loaded in parallel by bringing the SHIFT/$\overline{\text{LOAD}}$ input low. Clocking of the register is inhibited when this input is low, and the data at the input to flip-flop H will immediately appear at the output, in this case the inverting output ($\overline{\text{QH}}$). If switch position 1 is closed, a low

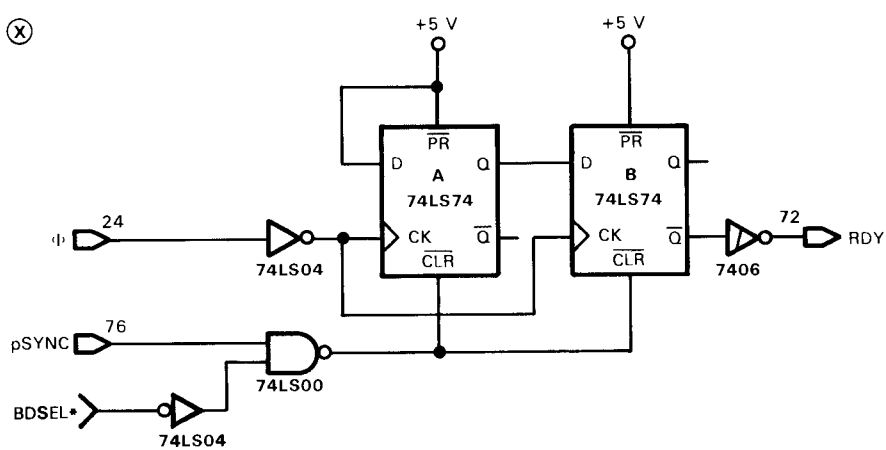


FIGURE 5-46. Circuit to Generate Two Wait States

will be presented to the data input of flip-flop H, and when SHIFT/$\overline{\text{LOAD}}$ goes low, a high will be present at the $\overline{\text{QH}}$ output. This high will be inverted by the 7406, which pulls the RDY line low, starting a wait state.

Assume that all the other switch positions are open. High levels will be presented to all the other flip-flop D inputs. When SHIFT/$\overline{\text{LOAD}}$ goes high (when pSYNC goes low) the register can be clocked. The high that was loaded into the G input will now be clocked into flip-flop H and will appear at the $\overline{\text{QH}}$ output as a low. This will end the wait state. Subsequent clock cycles will continue to clock highs down the register because that was what was loaded into it with the SHIFT/$\overline{\text{LOAD}}$ pulse. If switches 1 and 2 were closed, two wait states would be generated. If the first three switches were closed, three wait states would be generated, etc.

If you do not want any wait states, leave all the switches open. If you want one wait state, close switch position 1. If four wait states are desired, then close switches 1 through 4. If eight wait states are desired, close all the switches. Eight wait states should be enough for any device.

HOW TO APPLY THIS CHAPTER TO THE REST OF THE BOOK

This section is very important. All (or most) of the decoding and buffering circuits presented in this chapter are intended to be used in conjunction with the rest of the circuits in this book. You may have noticed a circled letter at the upper left-hand corner of the schematics in this chapter. If while looking at the circuit diagrams that follow, you see a box with a circled letter in it, that means use the circuit in this chapter with the corresponding letter. The signals that emanate from the box will correspond to the signals with the same names in the schematics in this chapter. Signals going into the boxes should also match up.

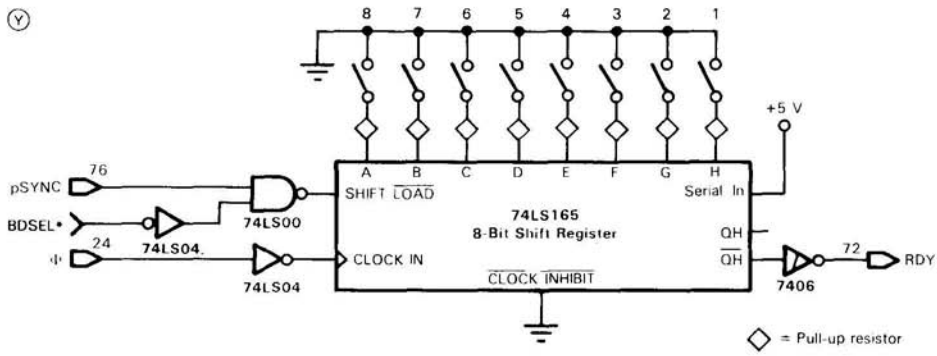


FIGURE 5-47. Circuit to Generate 0 to 8 Wait States

This method is used so that we do not have to continually redraw these circuits, which are contained on almost every S-100 slave device. It also converts part of the schematic into a block diagram, hopefully making the schematic more readable, because only the relevant signals are shown in detail. Figure 5-48 shows how a typical schematic will look later on, and Figure 5-49 shows the boxes of Figure 5-48 converted to the actual schematic they represent. The example shows a 2716 EPROM circuit for the S-100 Bus, with one wait state.

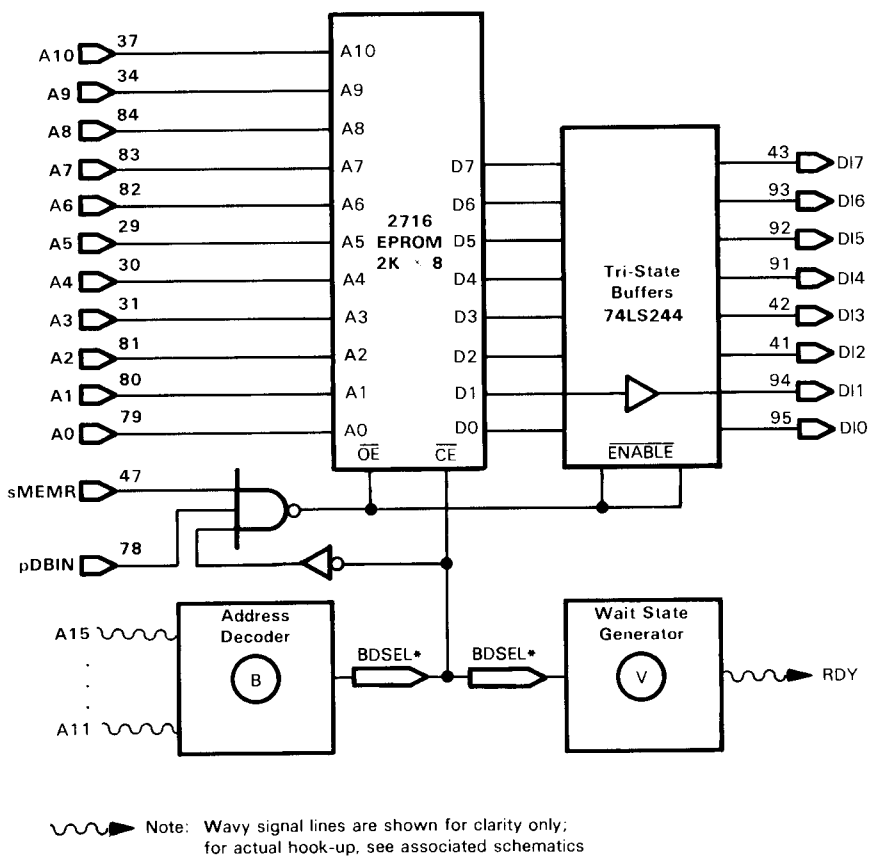


FIGURE 5-48. Typical Schematic in Rest of Book

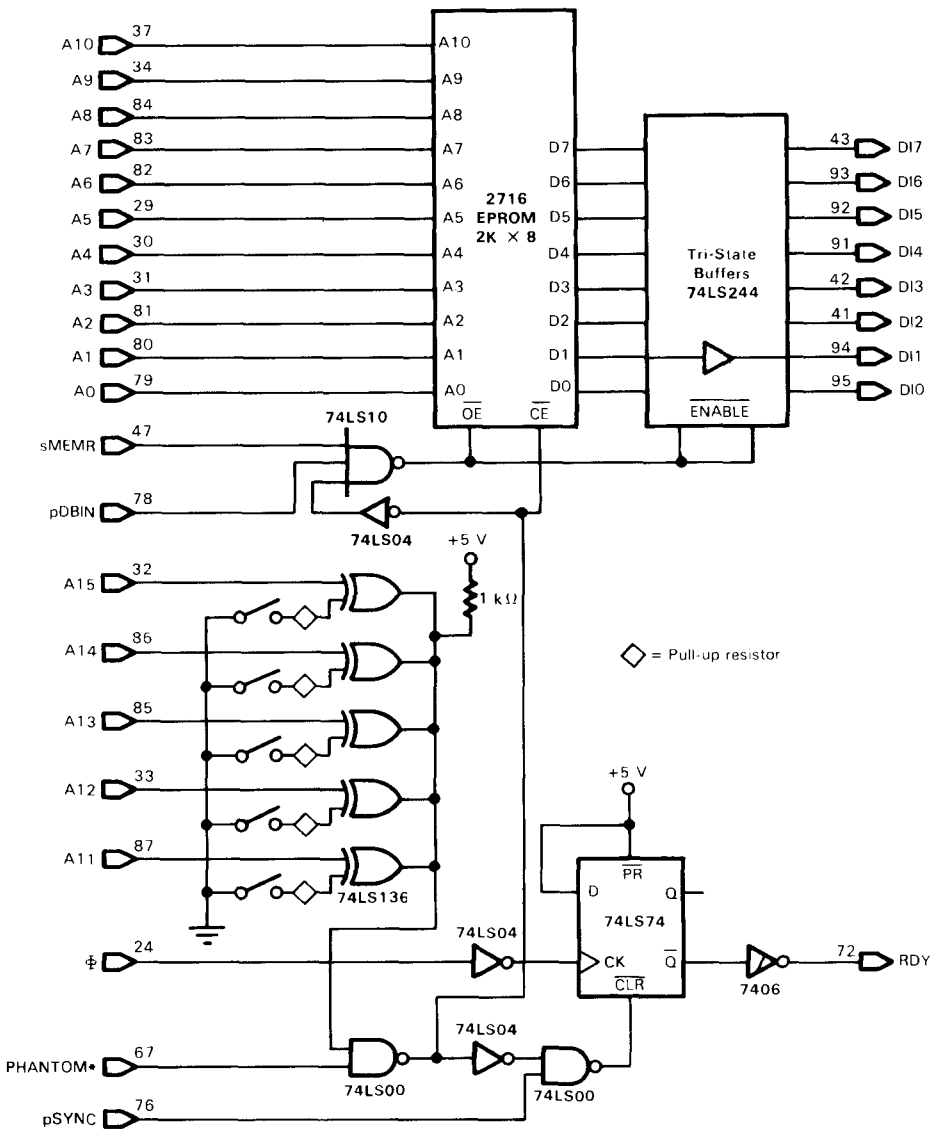


FIGURE 5-49. "Expanded" Schematic of Figure 5-48

memory interfacing

6

This chapter describes how to interface to both Random Access Memories (RAMs) and Erasable/Programmable Read-Only Memories (EPROMs). Using the information presented in this chapter, and the decoder and buffer circuits in Chapter 5, you should be able to build a memory system of any size and configuration.

There are two types of memories in common use today: RAMs and ROMs (which encompass PROMs and EPROMs). RAMs are called "random access" because the earliest semiconductor memories were serial access shift registers. With a serial memory you have to read all the bits in order until you get to the desired data. As the name implies, random access memory allows direct access to any bit (random, in this case, means in no particular order).

Although ROMs are also "random access," the term RAM has become used to denote a memory that may be easily read from and written to (read/write memory). A Read-Only Memory is programmed at the factory and its contents may never be changed. EPROMs may be programmed by the user, but usually require special hardware, and can only be erased by an intense ultraviolet light.

There are two types of RAMs in common use today: static and dynamic. When data is written into a static RAM, it will remain unchanged until it is rewritten or until power is lost. The data is therefore "static." A dynamic RAM's data must be refreshed periodically or it will be lost. This refresh cycle must be interleaved with the normal system accesses; this can cause some complicated problems and requires highly sophisticated logic designs. Therefore, we will only show you how to interface static memories to the S-100 Bus.

SOME COMMON RAMS AND THEIR ARRAYS

There are two common sizes of RAM ICs currently in use. They are the 1K (1024-bit) device and the 4K (4096-bit) device. The 4K RAM is available in two common organizations. The organization of a RAM IC refers to how the bits are arranged by address. A 4K-bit IC may have 4096 unique addresses, each containing one bit, or 1024 unique addresses each containing four bits. The former is referred to as a 4K by 1 organization, and the latter is referred to as a 1K by 4 organization.

When the S-100 Bus first appeared, a "large" memory IC contained 256 bits, and 1024-bit ICs were just becoming available. Consequently, a "large" memory board contained 8 Kbytes. The S-100 standard provides for addressing up to 16 *Mbytes!* 32 Kbyte static memory boards are commonplace today.

Nevertheless, having a small amount of memory might be of use to you, so we will describe the 21L02 1K by 1 memory IC. A logic symbol with pin connections for the 21L02 is shown in Figure 6-1. A typical 1K × 8 "array" of eight 21L02s is shown in Figure 6-2. Note that all the address and control lines are bused together, but the data lines remain separate.

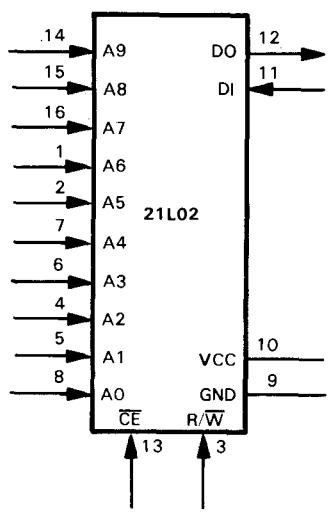


FIGURE 6-1. 21L02 Logic Symbol with Pin Connections

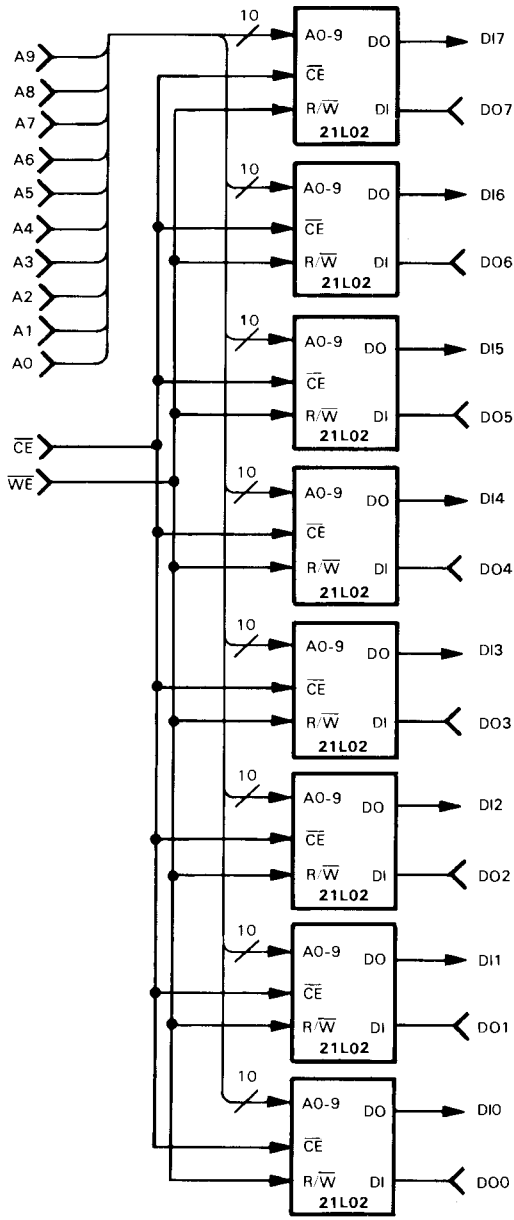


FIGURE 6-2. 1K × 8 RAM Using Eight 21L02s
(Interface to S-100 Bus is shown in Figure 6-8)

This IC is made by many different manufacturers, and each manufacturer has its own part number for the same device. The following table lists some of the larger memory suppliers' numbers for the basic 21L02.

Manufacturer	Part Number*
Intel	2102AL-X
Fairchild	2102LX
Advanced Micro Devices (AMD)	91L02-X
National Semiconductor	2102AL-X

* The X in the part number will be a digit or letter that will vary depending on the speed of the part.

The most popular type of RAM chip in use today is the 4K (4096-bit) device, which comes in two organizations: 2114 and 4044/5257/2147. The 2114 is 1K by 4, and the 4044 is 4K by 1. Surprisingly, a 2114 is called a 2114 by almost all of the manufacturers, but the 4044 type part has many different pin compatible versions. For example, the 2147 is a super high speed version of the 4044.

The 2114 logic symbol with pin connections is shown in Figure 6-3. This IC has four bits of data per address and thus only two ICs are required to provide a full byte of data, as opposed to eight ICs for the 1-bit wide parts. This is quite useful where only a small amount of memory (1 or 2 Kbytes) is needed. Two 2114s

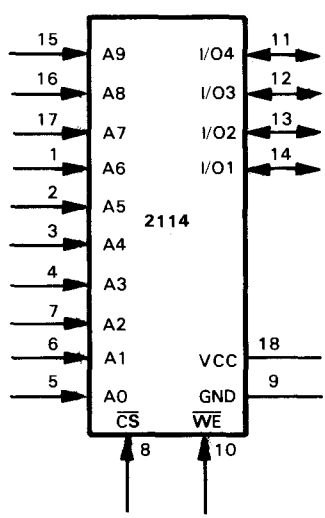


FIGURE 6-3. 2114 Logic Symbol with Pin Connections

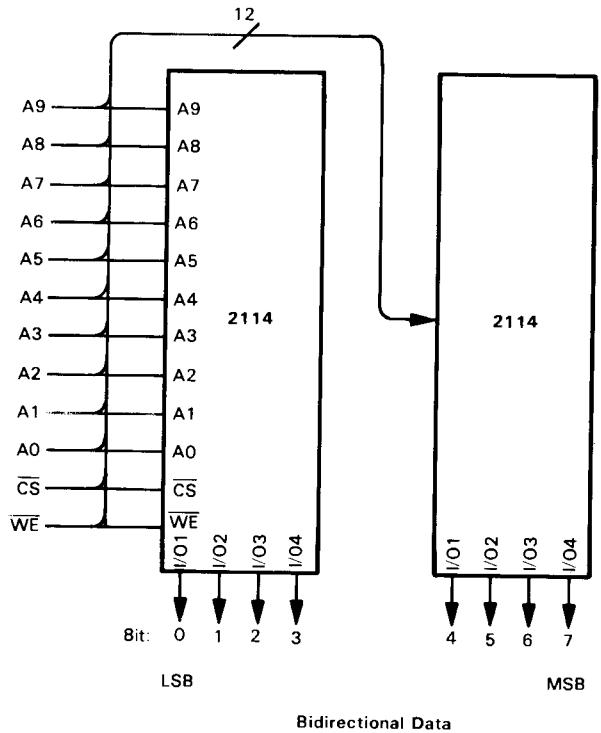


FIGURE 6-4. Two 2114 RAMs Connected in a Typical Array (1K × 8)

are shown connected in Figure 6-4. Note that this is exactly the same amount of RAM as shown in Figure 6-2, but now only two chips are used, instead of eight (think of how many less wire-wrap connections that is).

Where larger amounts of memory are required (more than 16K × 8), the 4K by 1 ICs such as the 4044 are preferred. This is because it takes less decoding circuitry — eight 4K blocks for a 32K × 8 memory as opposed to 32 1K blocks if 2114s were used. A logic symbol with pin connections for the 4044 RAM is shown in Figure 6-5. This part has many designations, depending on the manufacturer, and has many higher performance, pin compatible upgrades. The following table shows the "garden variety" parts, and then the higher performance parts.

Manufacturer	Part Number	Comments
Intel	2141L-X	
Texas Instruments	40L44-X	
AMD	90L44-X	
National Semiconductor	5257-XL	
Higher Performance Upgrades		
Intel	2147-X	High speed and current*
Intersil	7147	High speed, CMOS for extremely low current.
Hitachi	6147	As 7147 above.

* National, TI, Motorola, NEC, and many others also make this part with similar part numbers. Also note that even though this part has much higher current requirements than a standard 4044 type, in a system using several rows of parts the total current consumption will be much less. This is because the part "powers down" when not selected, and in a larger system only one row of parts will be selected at any one time.

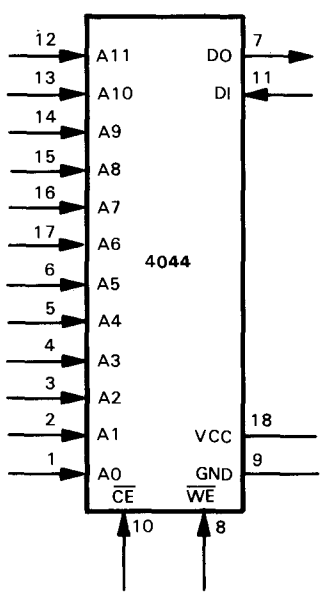


FIGURE 6-5. 4044 Logic Symbol with Pin Connections

Figure 6-6 shows eight of these parts connected together in a standard 4K × 8 array. Note that all the address and control lines are bused together, and all the data lines are separate. Note the pull-up resistor on the $\overline{CE}$ line. This is not normally required, but when using a "power-down" version of these chips (the 2147, for example), it ensures that the chip draw as little current as possible. The closer $\overline{CE}$ is to +5 volts, the less power is consumed.

The following circuits show some typical RAM arrangements using the decoders presented in Chapter 5. There are circuits for the three types of RAMs we have discussed in a few different sizes. Once you understand the approach, you should be able to design a memory of any organization.

A block diagram of a typical S-100 memory board is shown in Figure 6-7. The low-order address lines are buffered and applied directly to the memory array. The ICs have built-in decoders that decode the address bits into unique RAM locations on the chip. The higher-order address bits are connected to a decoder circuit that selects the board and provides the individual enables for each row of RAMs. The data lines are buffered, and the buffers are controlled by the qualified read and write strobes. An optional wait state generator is provided if slow memories (or fast masters) are used.

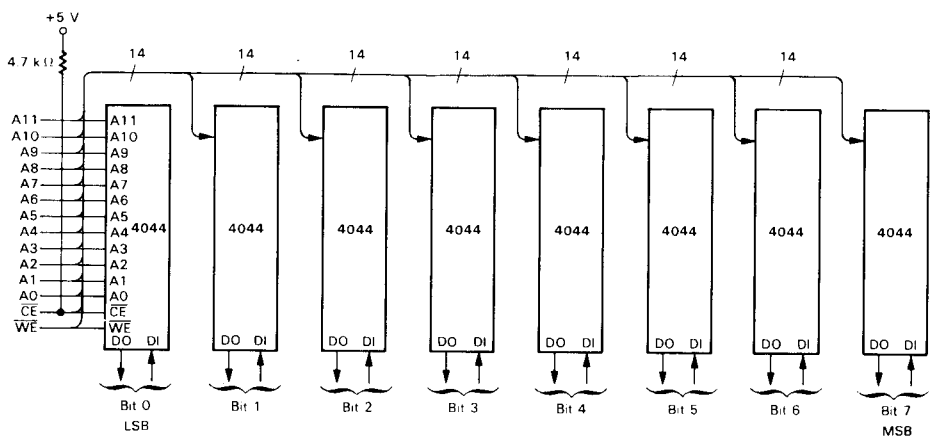


FIGURE 6-6. Eight 4044-Type RAMs in a Typical Array (4K × 8)

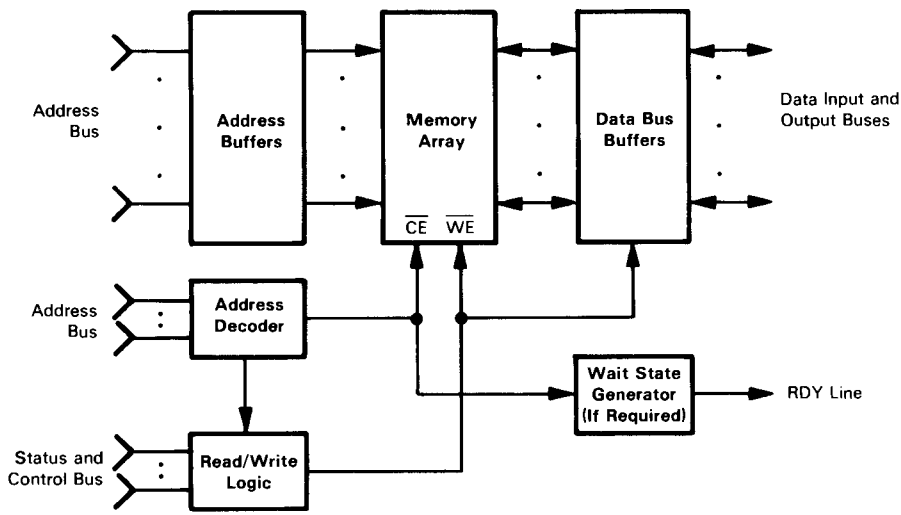


FIGURE 6-7. Block Diagram of Typical S-100 Memory Board

Figures 6-8 through 6-10 show some typical S-100 memory interfaces; two are for 1K bytes and one is for 16K bytes. Feel free to build the memory circuits we have shown you, but if you need a memory board larger than a few Kbytes, we strongly recommend that you purchase the memory board from an S-100 manufacturer.

Figures 6-11 and 6-12 show logic symbols and pin connections for the 2708 and 2716 EPROMs, the two most popular in use today. The 2708 requires three supply voltages and the 2716 requires only one. (The first 2716 produced by Texas Instruments required three supply voltages, but they now make a single supply version called the 2516. We recommend that only the single supply part be used.) The address bus buffer shown in Figure 6-13 is required to provide a high-level input voltage of at least 3.0 volts to the 2708.

Figures 6-13 through 6-16 are schematics for various sizes of EPROM memory connected to the S-100 Bus. These schematics include a wait state generator (you may use the one you prefer, depending on how many wait states are required) because EPROMs are typically slower devices than RAMs.

Higher density EPROMs are under constant development. At the time this book was written the 2732 EPROM was becoming widely available. The cost of a 2732 was still quite high compared to that of two 2716s, but since that price will probably drop rapidly, we have included a logic and connection diagram for the 2732 in Figure 6-17. It is a 4K by 8 device and requires a single power supply.

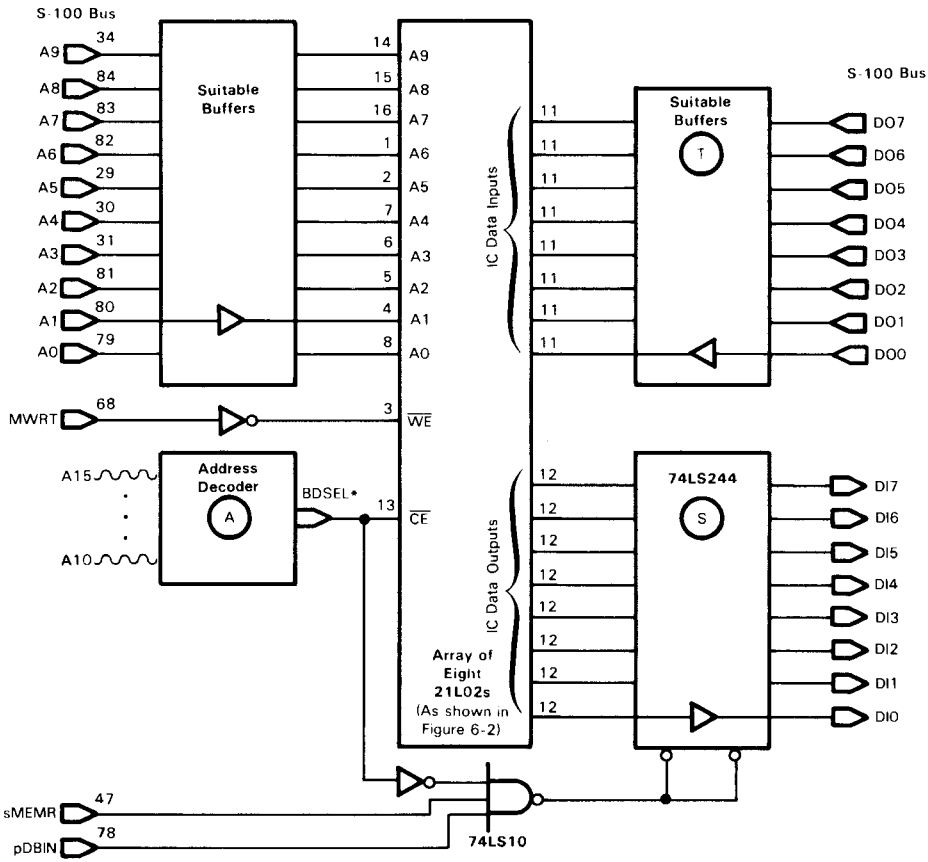


FIGURE 6-8. 1K × 8 of 21L02-Type Memory Interfaced to the S-100 Bus

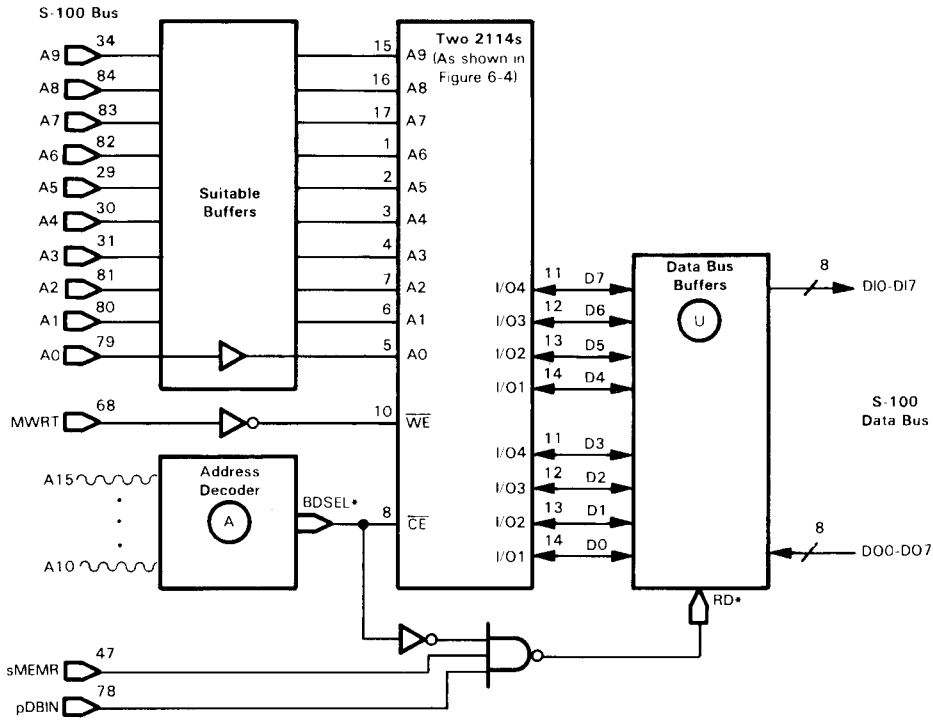


FIGURE 6-9. 1K × 8 of 2114-Type-Memory Interfaced to the S-100 Bus

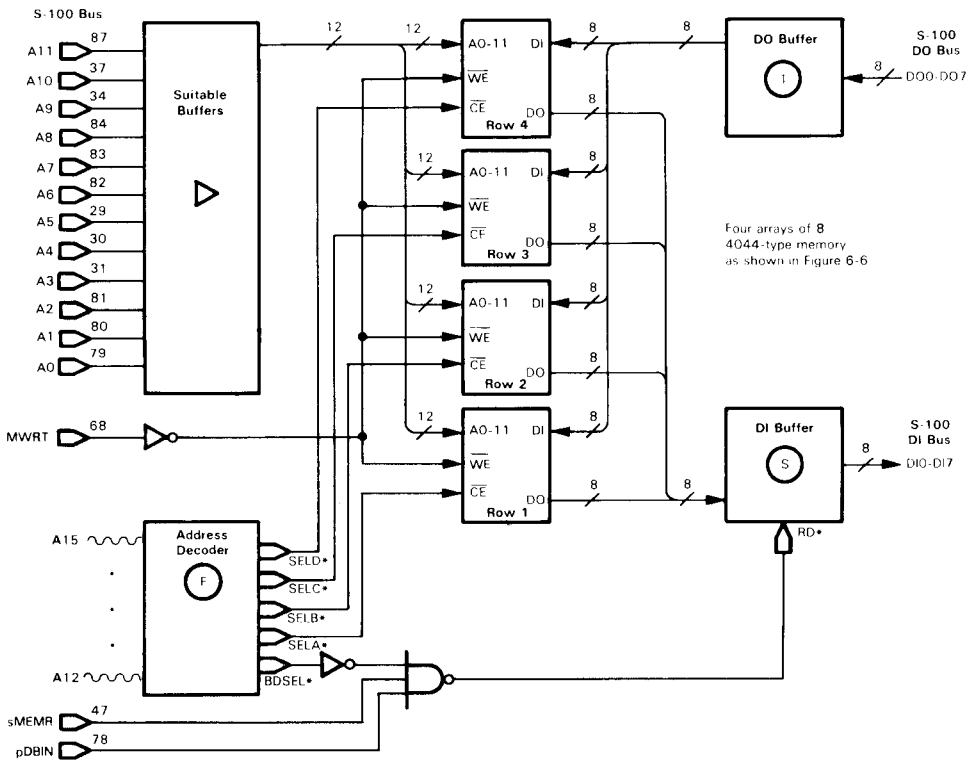


FIGURE 6-10. 16K × 8 of 4044-Type Memory Interfaced to the S-100 Bus

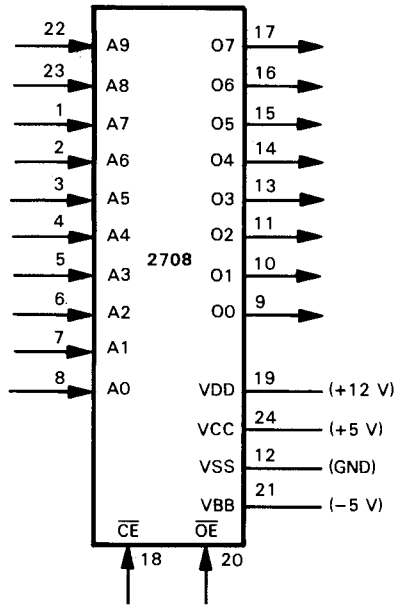


FIGURE 6-11. 2708 Logic Symbol with Pin Connections

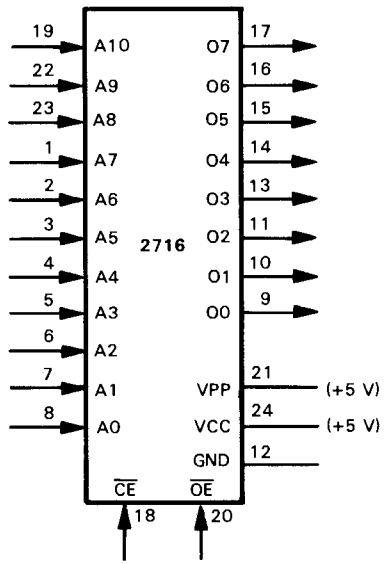


FIGURE 6-12. 2716 Logic Symbol with Pin Connections

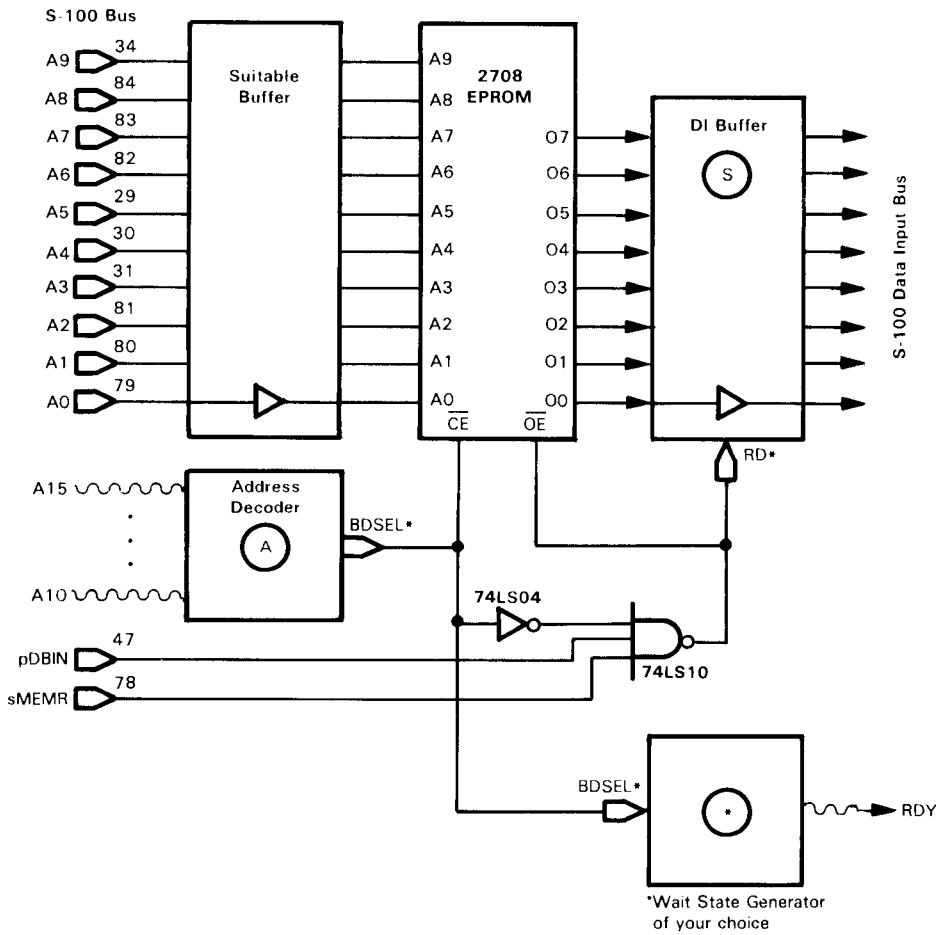


FIGURE 6-13. 1K × 8 of 2708-Type EPROM Interfaced to the S-100 Bus

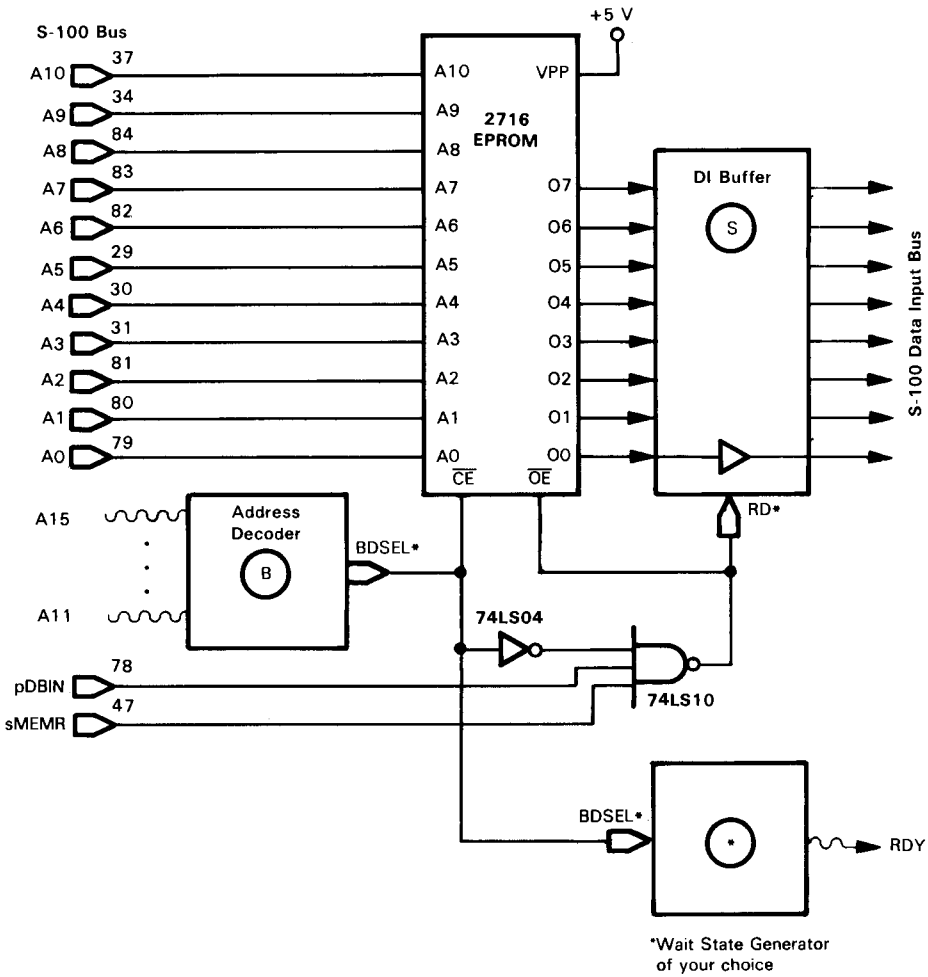


FIGURE 6-14. 2K × 8 of 2716-Type Memory Interfaced to the S-100 Bus

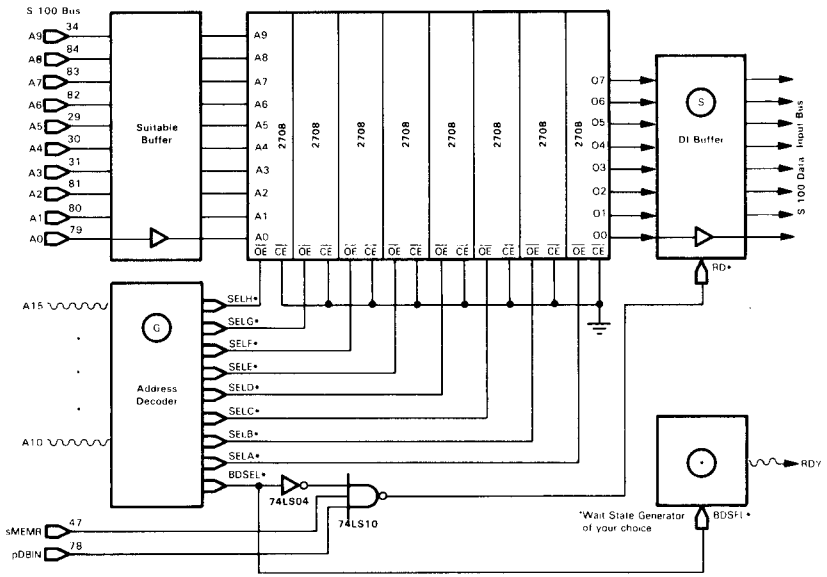


FIGURE 6-15. 8K × 8 of 2708-Type Memory Interfaced to the S-100 Bus

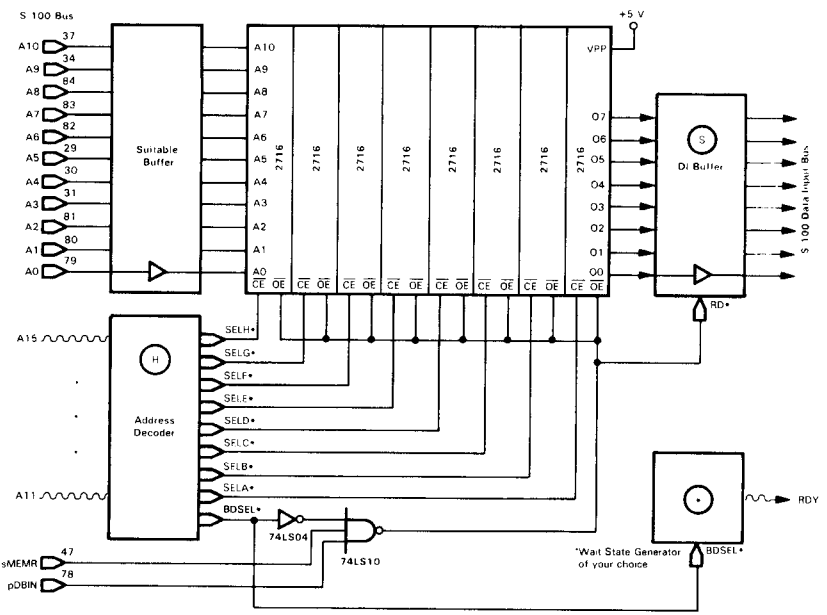


FIGURE 6-16. 16K × 8 of 2716-Type Memory Interfaced to the S-100 Bus

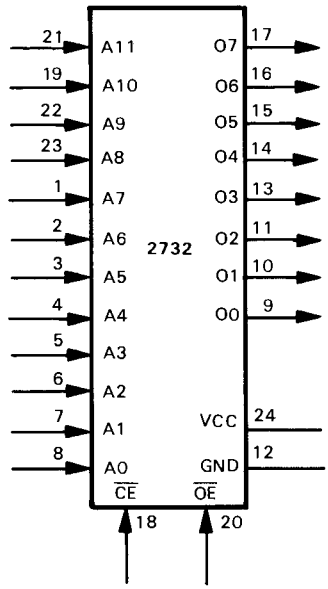


FIGURE 6-17. 2732 Logic Symbol with Pin Connections

BANK SELECT

The S-100 standard provides for 24 address lines, allowing 16 Mbytes of memory to be addressed. Before these lines were established by the standard, many manufacturers used a scheme known as "bank select" to allow the use of more than 64 Kbytes of memory.

Bank select works like this: an entire memory board, or a portion of it, may be turned on and off by writing a command to an output port (see the next chapter for a description of output ports). By writing a particular bit pattern to the port, the software would control which "bank" of memory would be selected, and which banks would be deselected. Only one bank of memory may be turned on in the same address space at any one time.

Figure 6-18 shows a schematic for implementing a bank select scheme. Like the extended memory address decoder shown in Chapter 5, it may be added to any of the memory address decoders shown in that chapter. Here is how it works: at the end of the write strobe (pWR*) the output of the 74LS30 will be clocked into the 74LS74 (D type flip-flop). If any of the Bank Select switches are closed and any of the associated data bits (after inversion) are low, then the output of the 74LS30 will be high. This will cause the output of the 74LS74 to be high and will

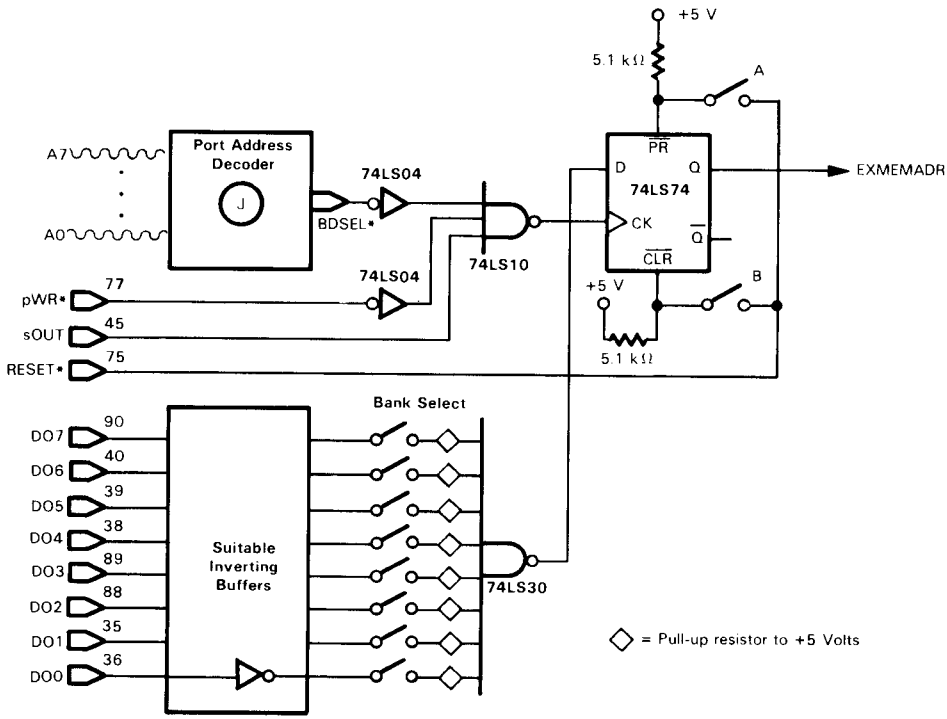


FIGURE 6-18. Bank Select Circuitry

turn on the memory board via EXMEMADR. If none of the data bit/switch pairs matches, then the output of the 74LS30 will be low, and the memory board will be turned off. If more than one switch is closed, the board will be turned on if any of the data bits match. This allows a board to reside in more than one bank. Up to eight banks are possible with this scheme.

The circuit is also able to determine whether or not the board is enabled or disabled at power-up and when a RESET occurs. The RESET* line can be connected to the PRESET or CLEAR input to the flip-flop by closing either switch A or B. If you want the board to be enabled at power-up and at RESET, turn switch A on and switch B off. If you want the board to be disabled at power-up and at RESET, turn switch B on and switch A off. Never turn switches A and B on at the same time. Leaving both switches off will make the board come up in a random state. The reason that RESET* may be used instead of RESET*·POC* is that POC* must assert RESET*. Older S-100 systems that do not meet the standard may not do this, so you may want to use a gate to AND POC* and RESET*.

I/O ports, an introduction

7

This chapter describes how an S-100 computer "talks" to anything outside of its own small world, and how these external things may talk to the computer. This is done via an interfacing circuit referred to as an I/O (Input/Output) port. I/O ports are slave devices on the S-100 Bus.

THE CONCEPT OF A PORT

An I/O port is called a port because that is where information enters and leaves the computer, just as cargo enters and leaves a country via a port. A port can be an input port (the master reads data from the port), an output port (the master writes data to the port), or an input/output port (the master can either read data from or write data to the port). An input/output port is referred to as a "bidirectional port."

Many ports can exist in a computer system. The number is limited only by the number of address bits that are reserved for port addresses. The S-100 Bus allows up to 65,536 unique I/O ports to exist. Although a subset of the system address lines is used, these port addresses do not conflict with the memory address space because the status bus reflects an I/O cycle instead of a memory cycle. Eight-bit microprocessors generally use an 8-bit I/O address and hence can individually address any one of 256 input and 256 output ports. Sixteen-bit microprocessors generally use larger port address words and hence can address more I/O ports. The S-100 Standard provides a 16-bit I/O address, yielding 65,536 port addresses.

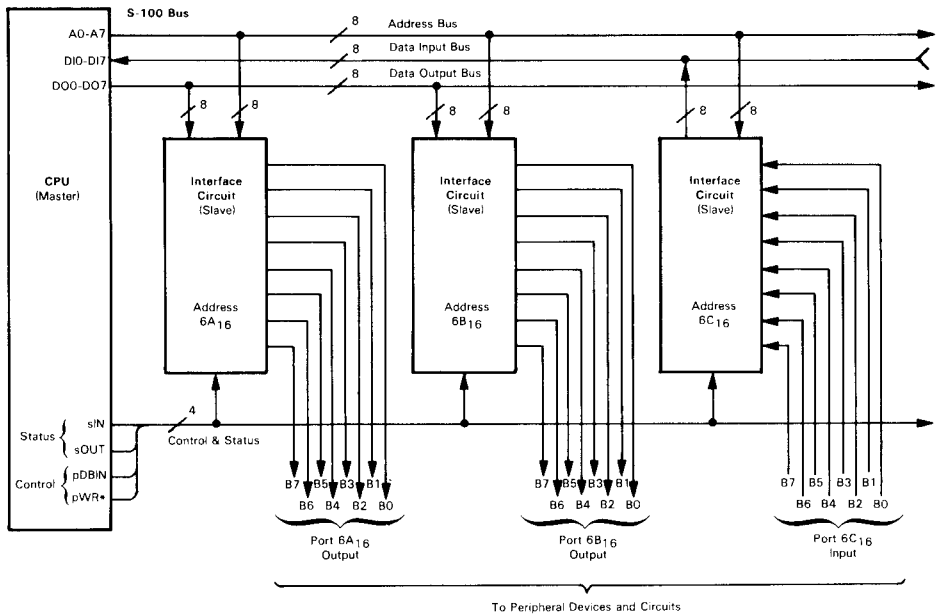


FIGURE 7-1. Block Diagram of CPU With Two Output Ports ($6A_{16}$ & $6B_{16}$) and One Input Port ($6C_{16}$)

Figure 7-1 illustrates the basic scheme for creating input and output port interfaces. The examples show three ports. Each port has an address. Each port is merely an 8-bit interface to a peripheral device or circuit. The ports are selected by an address — in this example, the hexadecimal values 6A, 6B, and 6C. The 8080, 8085, 8086, Z80, and Z8000 have specific instructions (IN and OUT) for transmitting data to or from a port. For example, when an OUT 6A instruction is executed by an 8080, the master places $6A_{16}$ on the lower eight bits of the address bus and the data in the accumulator is placed on the data output bus. The out-to-a-port control signals are placed on the S-100 status and control bus by the bus master.

The output port interface circuit, shown in Figure 7-2, typically consists of an 8-bit latch, an address decoder, and an I/O write qualifier circuit (shown in Chapter 5). When an OUT instruction (e.g., OUT 6A) is executed, the addressed interface circuit latches the data from the data output bus and presents it to the peripheral device or circuit. The input port interface circuit, as shown in Figure 7-3, typically consists of eight tri-state gates or an 8-bit latch, an address decoder, and an I/O read qualifier circuit (refer to Chapter 5). When addressed with an IN instruction (e.g., IN 6C), the input port gates the data onto the data input bus for the master.

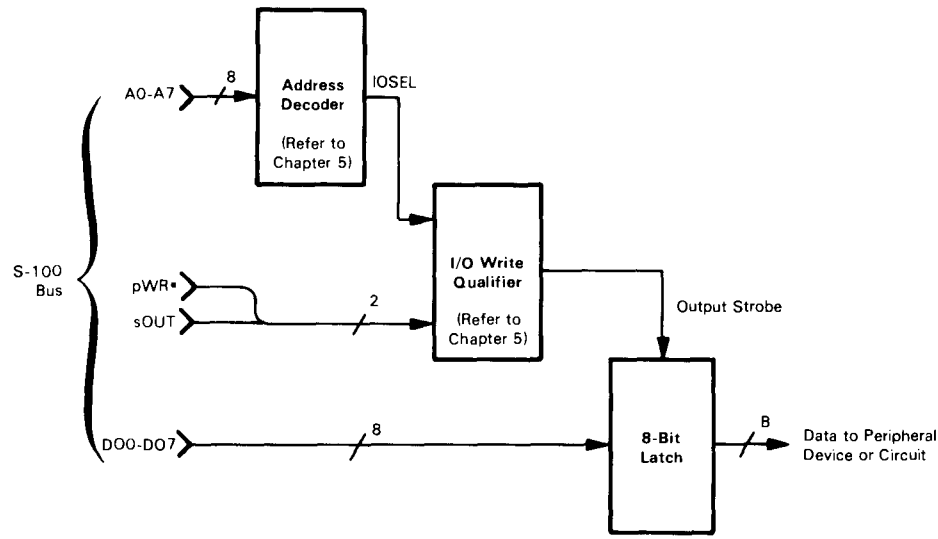


FIGURE 7-2. Block Diagram of a Typical Output Port

An 8-bit master provides an 8-bit address which can select up to 256 output and 256 input ports. The address information is decoded and gated with the S-100 status and control signals to enable the latch or gate circuits. The functional circuit of a typical output port is shown in Figure 7-2.

It is also possible to create a port by using a memory address to select an I/O port. This technique is called "memory-mapped I/O." This is the standard technique used with microprocessors such as the 6502, 6800, 6809, and 68000, which do not have specific input or output instructions. The 8080, 8085, and Z80 microprocessors should generally not require memory-mapped I/O. The block diagram for a memory-mapped I/O port is shown in Figure 7-4.

On the S-100 Bus, most I/O slaves are I/O-mapped; however, memory-mapped slaves are sometimes used. The trend is definitely away from memory-mapped I/O devices. In the case of processors that have no I/O instructions, a block of memory can be decoded by the CPU card logic so that an S-100 I/O cycle is performed on the bus instead of the memory cycle. This of course decreases the usable memory space.

To summarize, the master can address I/O ports and either send data to or receive data from the port. The S-100 Bus allows up to 65,536 unique I/O port locations that do not conflict with the memory address space, and each location can be either input, output, or both. For information on what makes up an S-100 I/O cycle, refer to Chapters 3 and 4.

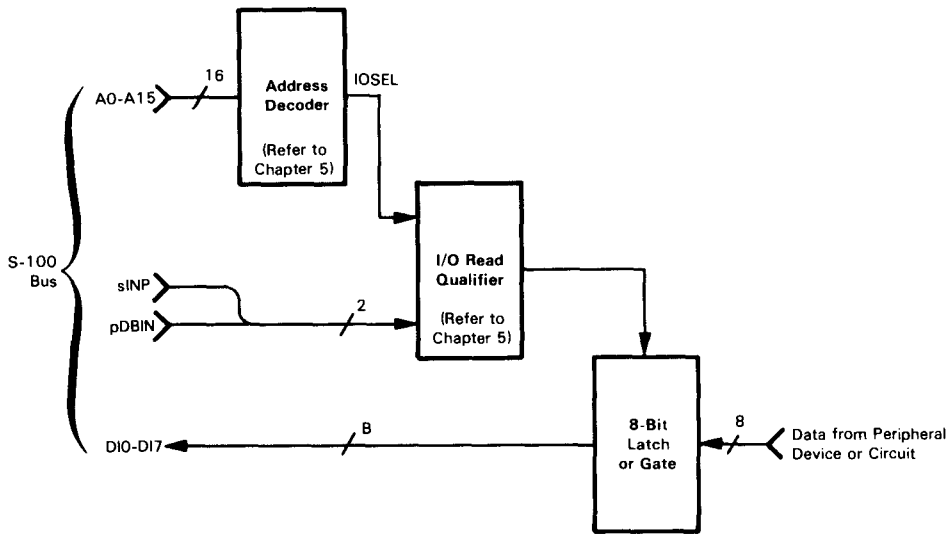


FIGURE 7-3. Block Diagram of a Typical Input Port

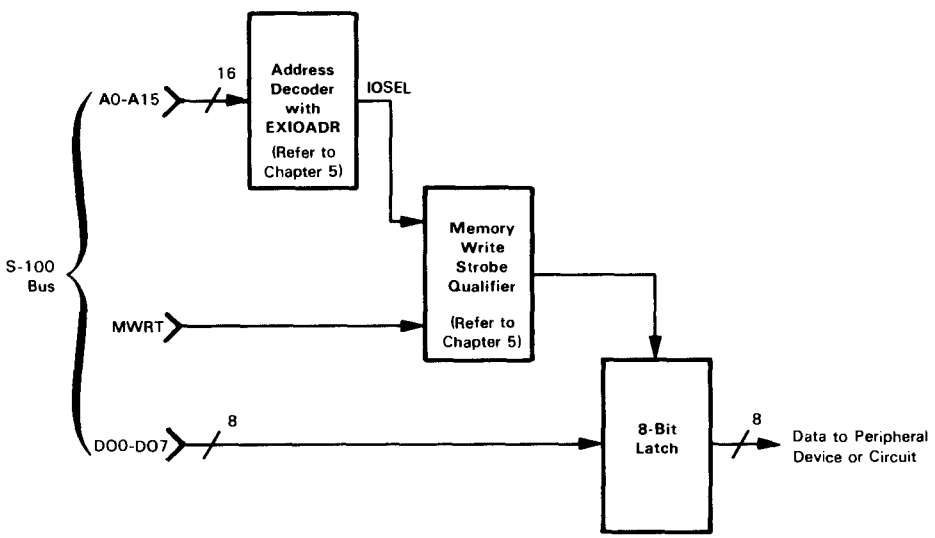


FIGURE 7-4. Block Diagram of a Memory-Mapped Output Port

HANDSHAKING — STROBES AND STATUS

Sometimes when we read data from or write data to an I/O port, we aren't concerned with whether or not the data we sent was received. This may be because the device is just an LED or a device that has no way of telling us that it has received the data. Or it may be that the data it is sending us is valid (because the program may decide what's valid about it).

Since I/O ports are usually used to interface with peripheral devices such as printers, keyboards, terminals, and modems, we would often like to be assured that the device actually received the data sent to it, or that the device has valid data to send. For example, as shown in Figure 7-5, a printer will usually have an output signal line that tells the computer that it is ready to accept data, and an input signal line that tells it when to take the data. A keyboard would have a line that would tell the computer when a key had been pressed and that the data on its data lines was valid. These lines are usually called strobes, and like the S-100 bus strobes, they convey *when* information: when data is valid, when data has been received, etc.

The act of interfacing with these strobes is called "handshaking." The term handshaking comes from the fact that when humans "shake hands" on something, it means that there is an understanding between them. In the case of the computer and the peripheral, one understands that the other is doing or has done something, such as sending data.

In the case of an input port, the computer wants to be informed when the data it is receiving from the peripheral is valid. The peripheral will assert a line that will convey this "data available" information. This line is usually read by the computer on a separate input port, commonly known as the "status port" (because it tells the status of the data that will be available on the "data port"). The computer will read all eight bits of the status port, but this "data available" bit is the only one to be concerned with. This is commonly referred to as the "status bit." The bit will usually be assigned a mnemonic such as DAV. The computer will usually be in a loop waiting for the status bit to go true. When the status bit goes true, the loop will "fall through," and the computer may then read the data from the data port.

An output port has a corresponding line that the computer needs to assert so that the peripheral knows when the data is valid. This is the data strobe, and it may be sent automatically by the port hardware every time the computer writes data to the data port, or we may use a line from another output port that the computer will assert separately. This separate port is usually called the "control port." Most peripherals will not be able to accept data as fast as the computer can send it (which may be many hundred thousand characters per second), so they will usually have a status line that tells the computer when the peripheral is busy and not busy. The computer would normally read this line on a "status" port.

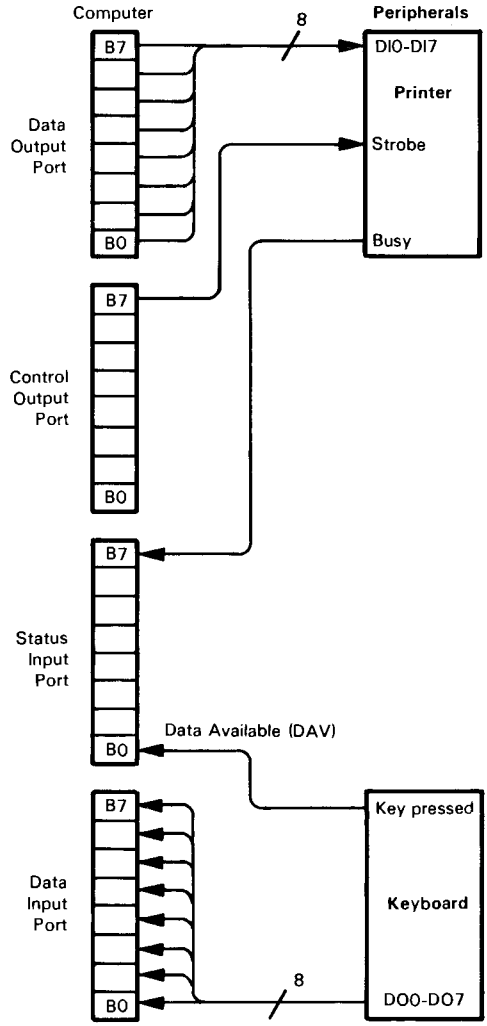


FIGURE 7-5. Handshaking and Data Paths Between a Computer and Peripherals Form a Communications Channel

CHANNELS

To accomplish the transfer of data through I/O ports, we may actually need more than one 8-bit data port to transfer eight bits of data. We call the combination of necessary ports a "channel." A typical channel might consist of two port

addresses and four actual ports: a data input port, a data output port, a status input port and a control output port. In addition, we will need the software routines to conduct the transfer of data. The next chapter will deal with ports, channels, and the associated software.

There are two basic types of channels: parallel and serial. An S-100 computer deals only with parallel data; hardware and/or software is needed to convert the parallel data into serial data and vice versa. We will discuss serial interfacing in a later chapter.

LATCHING THE DATA

Usually when we send data out to a peripheral, we want the data to be present on the output lines for a period of time (usually until we send new data). Recall that the data on the S-100 Bus remains stable for only a short while before and after the write strobe. This could be a very short time indeed. Also, when a peripheral presents data to an input port, the master might not get around to reading it until the data has gone away. We could put the master into a wait state to extend the bus strobes, but that slows down the system (and gets tricky). We need some way to preserve the data, and this is done with a latch.

A latch is a type of flip-flop that has a data input and a clock input (commonly referred to as a D-type latch). The data at the data input is latched into the flip-flop when the clock input is asserted correctly (this could be an edge or a level, depending on the type of latch). A group of latches is often called a register, but a group of latches (usually in the same IC package) may also be called a latch.

Some common 74LS-series latches are shown in Figures 7-6 through 7-10. The 74LS74 has two independent sections, each with clear and preset inputs, inverting and non-inverting outputs, and a positive edge-triggered clock input. It is shown in Figure 7-6. Figure 7-7 shows the 74LS175, essentially a quad version

Function Table

Inputs				Outputs	
$\overline{\text{Preset}}$	$\overline{\text{Clear}}$	Clock	D	Q	$\overline{Q}$
L	H	X	X	H	L
H	L	X	X	L	H
L	L	X	X	H	H
H	H	↑	H	H	L
H	H	↑	L	L	H
H	H	L	X	Q_0	$\overline{Q}_0$

H = High level (steady state)
L = Low level (steady state)
X = Irrelevant
↑ = Transition from low to high level
Q_0 = Level of Q before the indicated steady-state input conditions were established

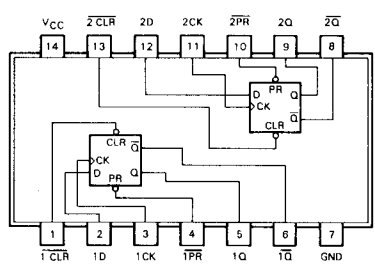
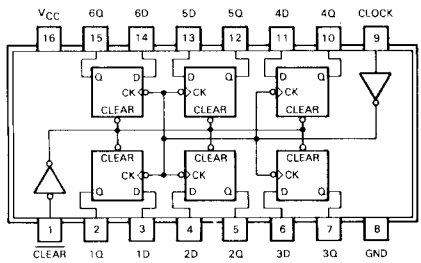
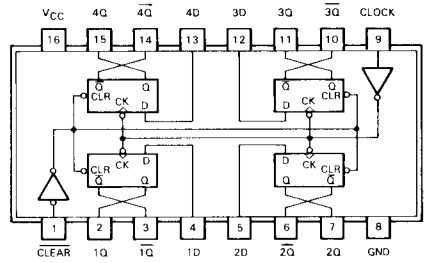


FIGURE 7-6. 74LS74 Dual D-Type Latch



74LS174



74LS175

Function Table
(Each Flip-Flop)

Inputs			Outputs	
$\overline{\text{Clear}}$	Clock	D	Q	$\overline{Q}$
L	X	X	L	H
H	↑	H	H	L
H	↑	L	L	H
H	L	X	Q_0	$\overline{Q_0}$

H = High level (steady state)
L = Low level (steady state)
X = Irrelevant
↑ = Transition from low to high level
Q_0 = The level of Q before the indicated steady-state input conditions were established

FIGURE 7-7. 74LS174 and 74LS175 Hex and Quad D-Type Latches

of the 74LS74, except that the preset inputs are missing and there is a common clock line and clear line. Figure 7-7 also shows the 74LS174, a hex version of the 74LS175 with essentially the same features, except that the non-inverting outputs are missing.

Figures 7-8 through 7-10 show octal latches, which are very convenient and popular because computers usually deal with data in multiples of eight bits. The 74LS273 in Figure 7-8 has eight sections, non-inverting outputs, a common clock input, and a common clear input. It is positive edge-triggered. The 74LS374 in Figure 7-9 is almost the same except that the clear input has been replaced by an output enable input (the outputs of the 74LS374 are tri-state). The 74LS373

Function Table
(Each Flip-Flop)

Inputs			Output
Clear	Clock	D	Q
L	X	X	L
H	↑	H	H
H	↑	L	L
H	L	X	Q_0

H = High level (steady state)
L = Low level (steady state)
X = Irrelevant
↑ = Transition from low to high level
Q_0 = Level of Q before the indicated steady state input conditions were established

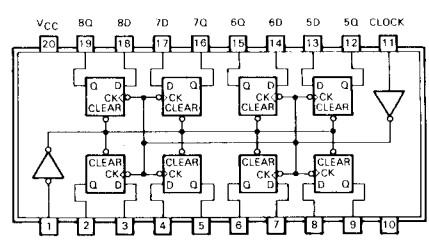
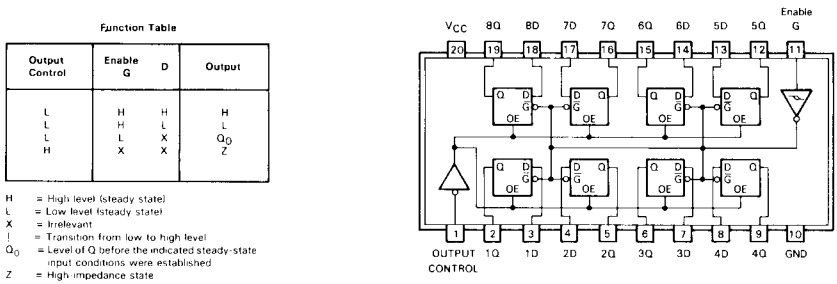


FIGURE 7-8. 74LS273 Octal D-Type Latch



Function Table

Output Control	Enable G	D	Output
L	H	H	H
L	H	L	L
L	L	X	Q_0
H	X	X	Z

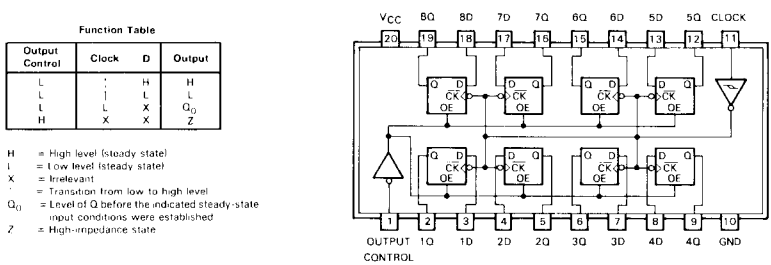
H = High level (steady state)
L = Low level (steady state)
X = Irrelevant
↑ = Transition from low to high level
Q_0 = Level of Q before the indicated steady-state input conditions were established
Z = High-impedance state

FIGURE 7-9. 74LS374 Octal D-Type Latch

in Figure 7-10 is almost identical to the 74LS374 except that its clock input works a little differently. The 74LS373 is known as a transparent latch. That is because when its clock input is high, the data at the D inputs goes straight through to the outputs, as if it were a buffer. But when the clock input goes low, the data is latched. Both the 74LS373 and the 74LS374 meet the current sinking requirements of an S-100 bus driver and may be used as such. (See Chapter 5 for more information.)

PROGRAMMABLE I/O PORT IC'S

An increasingly popular approach to I/O ports employs more sophisticated LSI ICs. These ICs contain control for two or more ports, with provisions for defining port direction, peripheral handshaking, and interrupt processing. However, it should be noted that programmable LSI I/O ports offer no advantages for simple I/O over the non-programmable ICs shown previously. This is due to the fact that



Function Table

Output Control	Clock	D	Output
L	↑	H	H
L	↑	L	L
L	L	X	Q_0
H	X	X	Z

H = High level (steady state)
L = Low level (steady state)
X = Irrelevant
↑ = Transition from low to high level
Q_0 = Level of Q before the indicated steady-state input conditions were established
Z = High-impedance state

FIGURE 7-10. 74LS373 Octal Transparent Latch

the programmable I/O ICs are designed to interface directly to a bidirectional bus, and added ICs are required to use them with the separate S-100 data buses.

Probably the two most popular ICs of this type are the Motorola MC6820 and the Intel 8255.

Motorola MC6820

Motorola calls the MC6820 a PIA (Peripheral Interface Adapter). The functional block diagram for the PIA is shown in Figure 7-11.

The MC6820 has two nearly identical ports, referred to as A and B. Each port has data, direction, and control registers. The data register holds either the input or output data. The register is latched for output and unlatched for input. The direction register determines the data direction for each bit (pin) in the data

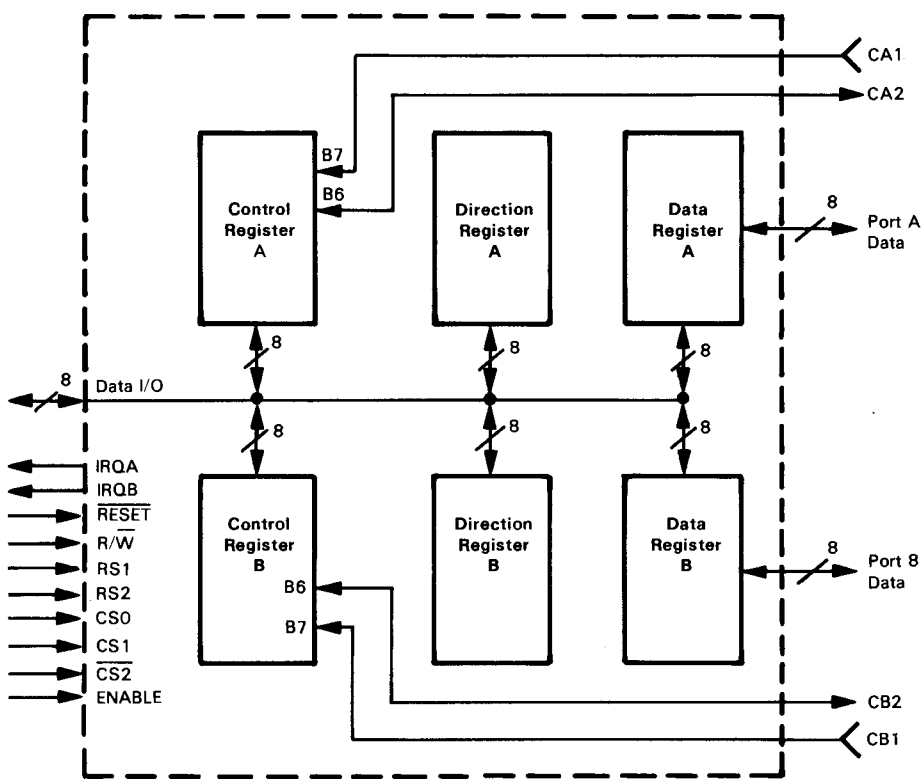


FIGURE 7-11. Functional Block Diagram of the MC6820 PIA

register (0=input and 1=output). This permits some bits to be defined as output and others as input, or even allows the data register to be used for temporary storage.

The control register holds the status and control bits for handshaking and other PIA functions. The function of the two control lines (CA1 and CA2 for port A) is configured by the control register.

The MC6820 uses four consecutive port addresses. Since there are six registers (two data, two direction, and two control) and only four addresses, bit 2 of each control register is used to determine whether the direction register (bit 2=0) or data register (bit 2=1) is being addressed. For a more detailed discussion of the MC6820, consult the *Osborne 4 and 8-Bit Microprocessor Handbook*.

Intel 8255

Called a PPI (Programmable Peripheral Interface) IC, the 8255 provides three 8-bit ports (A, B, and C). C is actually two 4-bit ports, and its bits can be individually set and reset. Ports A and B can be programmed to be either input, output, or bidirectional ports. Port C can be programmed to be either input, output or a pair of control ports, one for port A and the other for port B. An internal control register determines the specific configuration and can be altered by the program. The functional block diagram of the PPI is shown in Figure 7-12.

The 8255 can be programmed to operate in any of three different modes. In all modes, Ports A and B are used as either input, output, or bidirectional data ports. Port C is arranged as two 4-bit ports, so it is used as I/O or control, depending on the mode selected for ports A and B. Further, port C's bits can be set or reset independently to generate device strobes. The modes are as follows:

Mode 0 — Basic I/O (Figure 7-13*a*). A, B, and C are either latched output or non-latched input ports.

Mode 1 — Strobed I/O (Figure 7-13*b*). A and/or B are strobed data ports (input or output) with C providing each port with three control lines (one interrupt line). If desired, one port may be used in mode 0 and the other in mode 1. If both ports are used in mode 1 then there are two remaining bits available in port C for I/O.

The lines operate as follows:

INTR (Interrupt Request): used to interrupt CPU.

$\overline{\text{STB}}$ (Strobe input): loads data into register.

IBF (Input Buffer Full flip-flop): indicates that data has been loaded into register.

$\overline{\text{ACK}}$ (Acknowledge input): informs 8255 that data from Ports A and B has been accepted.

OBF (Output Buffer Full): indicates that CPU has sent data to port.

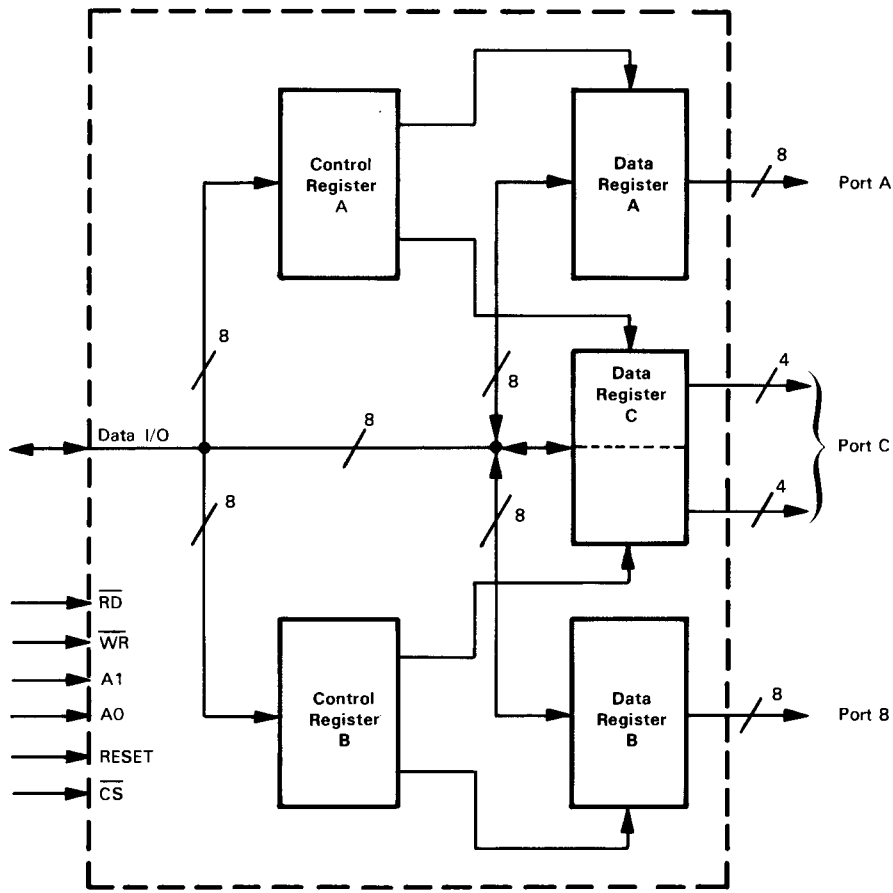


FIGURE 7-12. Functional Block Diagram of the 8255 PPI

Mode 2 — Strobed bidirectional I/O (Figure 7-13*c*). A is a bidirectional data port supported by five strobe control and interrupt lines. B may be operated in either mode 0 or 1, with three associated control lines.

The operation of the control register is shown in Figure 7-14. If bit 7 is high the control byte controls the mode, as shown. For example: 10010101 establishes A as a mode 0 input port, B as a mode 1 output port, C high-order bits as an output port, and C low-order bits as an input port. If Bit 7 is low, port C acts as a set of eight set/reset flip-flops, as indicated in Figure 7-15, controlled by an OUT

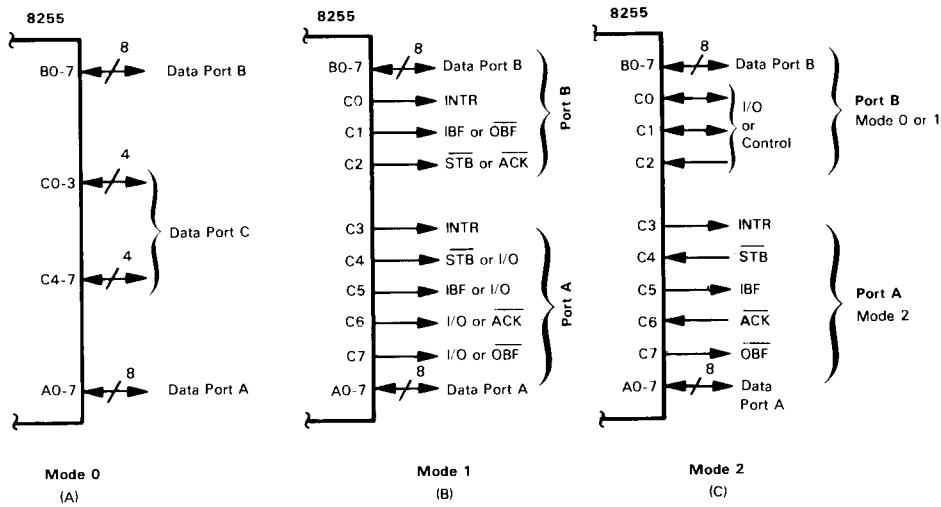


FIGURE 7-13. The Three Modes of Operation of the 8255

instruction. When used as a status/control for A or B, these bits can be set or reset using the bit set/reset, as if they were data output ports. Note that after being reset via the RESET input, the 8255 is set to input mode. A typical S-100 I/O port circuit and software driver will be shown in the next chapter.

OTHER PROGRAMMABLE PORT IC'S

There are a number of other programmable parallel I/O ICs available — in fact, too many to discuss. However, we would like to call your attention to some of them.

Z80-PIO

Made by Zilog, this 40-pin IC is a cross between the 6820 and the 8255. It contains two 8-bit I/O ports plus two associated control lines per port. Each port may be defined independently as input, output, or control port. As a control port, each pin can be individually defined to be input or output. Furthermore, the Z80-PIO contains extensive interrupt logic, including interrupt reset by decoding Z80 instructions from the bus "on the fly."

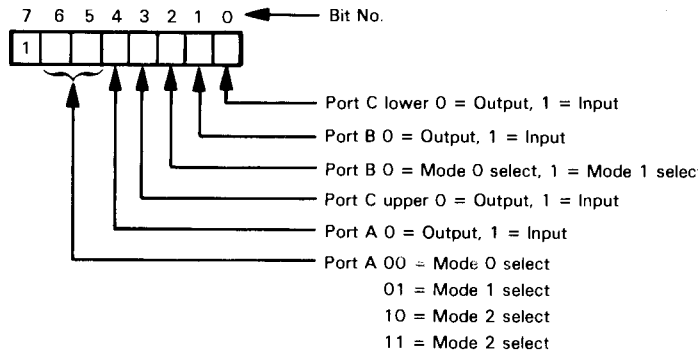


FIGURE 7-14. 8255 Mode Command Format

TMS 5501

Made by Texas Instruments, this is one of the most versatile interface ICs available. It has two 8-bit I/O ports (one is input and one output), an external sense input, and a bidirectional serial link with programmable baud rate generator. Furthermore, the 5501 contains five programmable timers. It also provides several interrupt control functions by receiving external interrupt signals, generating interrupt signals to an 8080 processor, masking out undesired interrupts, establishing priority of interrupts, and generating vectors.

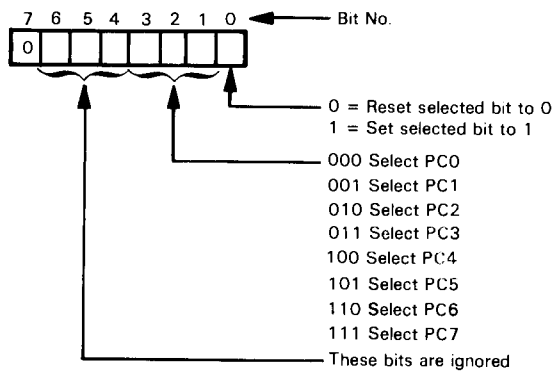


FIGURE 7-15. 8255 Bit Set/Reset Command Format

MCS 6522

Made by MOS Technology and others, the 6522 is an enhanced 6820. It provides more handshaking logic with port A. In addition, a counter/timer and serial I/O have been added to port B.

Each of the three devices mentioned here requires a complicated interface to work properly with the S-100 Bus.

REFERENCES

1. R. Baker, "Put the 'Do Everything' Chip In Your Next Design," *BYTE,* July 1976.
2. P. F. Goldsbrough and P. R. Rony, *Microcomputer Interfacing With The 8255 PPI Chip.* Howard W. Sams, Inc., 1979.
3. A. Osborne and G. Kane, *An Introduction to Microcomputers: Volumes 2 and 3.* Berkeley: Osborne/McGraw-Hill, 1976, 1979.

parallel interfacing

8

In the preceding chapter we discussed the basic principles of creating I/O ports in an S-100 system. In this chapter we will show several specific examples. We will show both the hardware and software used to drive the ports. The software routines used with input and output ports are most often called "I/O drivers." Without these driver routines the hardware does nothing.

A parallel input interface transfers all the bits in the data word at one time to the computer from the peripheral device or circuit. A parallel output interface transfers the data word from the computer to the peripheral device or circuit. In a later chapter we will discuss a serial interface which transfers the data word one bit at a time. The parallel interface is thus faster than the serial interface but requires a separate line for each bit in the word plus a common line. The serial interface requires only one pair of lines in each direction. In this chapter we will deal with common parallel interfaces.

SIMPLE PARALLEL OUTPUT AND INPUT INTERFACES

Figure 8-1 illustrates the use of the 74LS373 or 74LS374 (discussed in the previous chapter) as a simple parallel output interface. The address decoder circuit output is active when the selected port is addressed. The I/O write strobe is developed by a qualifier circuit. The 74LS373 thus latches the data from the data bus when the strobe is high, while the 74LS374, which is edge-triggered, latches the data on the positive edge of the I/O write strobe signal.

A simple parallel input port is shown in Figure 8-2. It uses an octal tri-state buffer/driver IC (refer to Figure 8-3). When this port is read, the data present at the input is placed on the data input bus and transferred to the master.

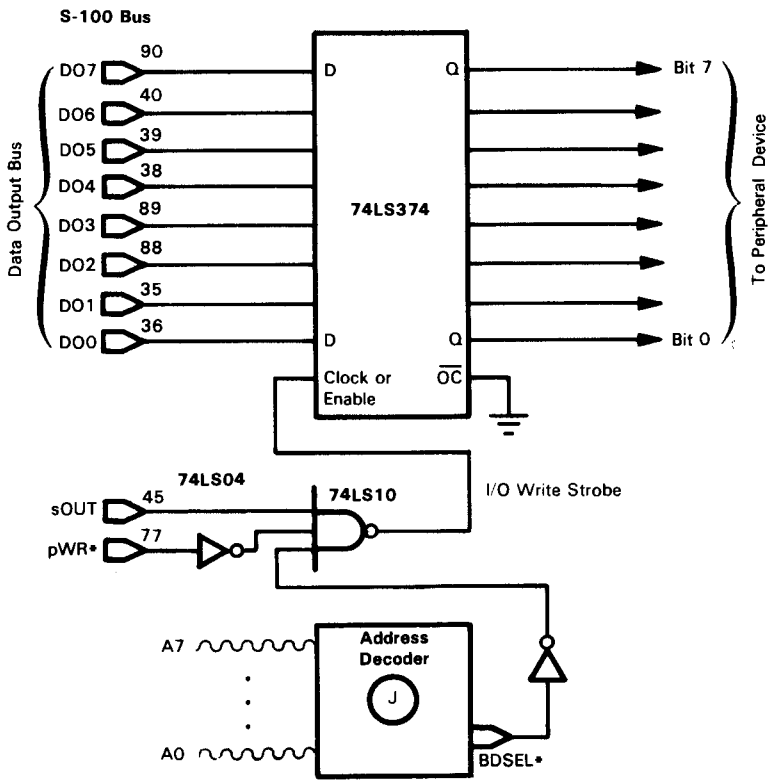


FIGURE 8-1. Simple Latched Output Port

The following simple subroutine reads the data at the input port, inverts it, and sends it to the output port. It assumes that both the input and output ports are at the same address (10_{16}).

```
                        ;SUBROUTINE TO READ DATA FROM A PARALLEL INPUT PORT,
                        ;INVERT IT AND SEND IT TO A PARALLEL OUTPUT PORT.
                        ;WHEN THIS SUBOUTINE IS CALLED, DATA IN ACCUMULATOR
                        ;IS LOST.
                        ;
                        ;TO USE WITH DIFFERENT PORT ADDRESSES, CHANGE THE
                        ;"BASE" EQUATE.
                        ;
0010 =      BASE        EQU   10H          ;BEGINNING PORT ADDRESS
0010 =      IPORT       EQU   BASE         ;INPUT DATA PORT
0010 =      OPORT       EQU   BASE         ;OUTPUT DATA PORT
                        ;
0100 =                  ORG   100H
                        ;
0100 DB10   START       IN    IPORT        ;FETCH DATA FROM INPUT PORT
0102 2F                 CMA                ;COMPLEMENT DATA
0103 D310               OUT   OPORT        ;SEND DATA TO OUTPUT PORT
0105 C9                 RET
```

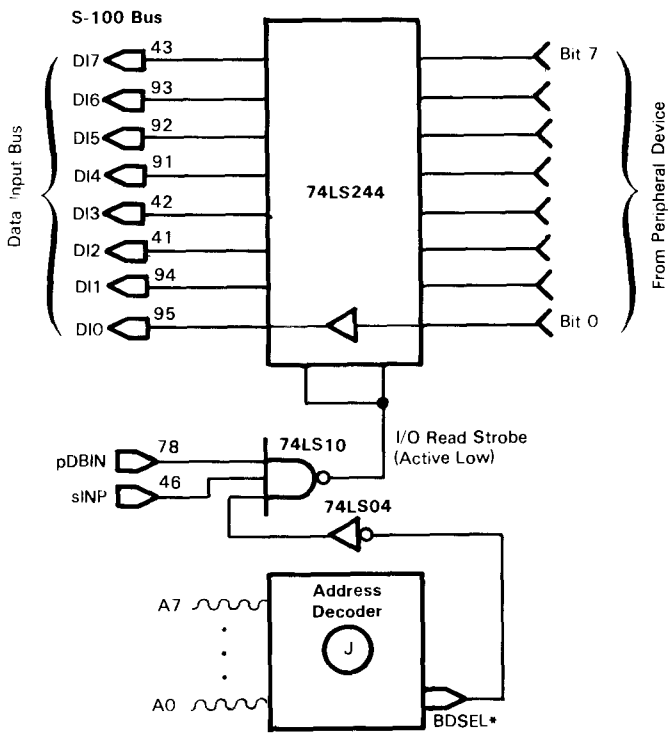


FIGURE 8-2. Simple Input Port (Non-Latched)

HANDSHAKING INTERFACING

In the previous chapter we introduced the concept of handshaking between the master and peripheral devices or circuits. Now let's look at how we accomplish this. Figure 8-4 illustrates a parallel interface with an 8-bit data output port and a 1-bit data latch that holds active high and active low strobe signals for the peripheral device.

The address decoder circuit, taken from Chapter 5, provides address select signals for two ports, A and B. Port A is the control output port and port B is the data output port. The following program is a simple 8080/8085 driver subroutine to output data to the peripheral. The flowchart is shown in Figure 8-5. The data is first sent to the data latch, where it is held. A strobe signal for the peripheral, telling it to load the data from the latch, is generated by first making the 1-bit control port latch high and then returning it to a low logic level.

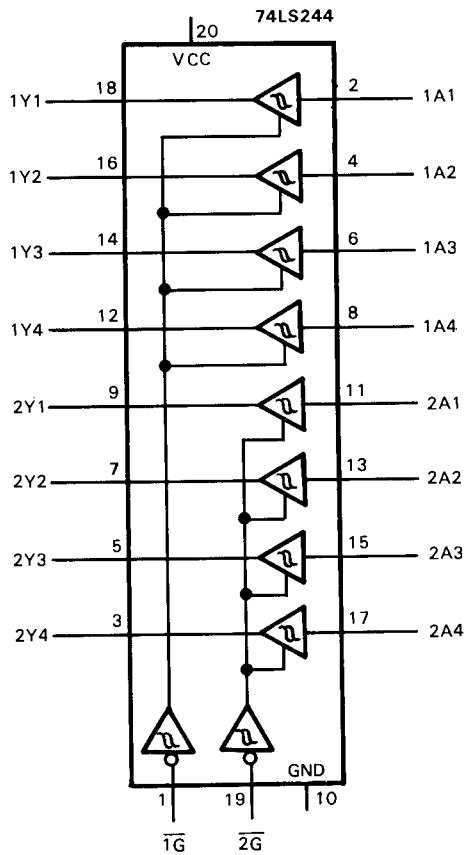


FIGURE 8-3. 74LS244 Logic Symbol with Pin Connections

```
                ;SUBROUTINE TO OUTPUT DATA TO PARALLEL PORT WITH SIMPLE
                ; HANDSHAKING
                ;ASSUMES DATA IN ACCUMULATOR WHEN SUBROUTINE IS CALLED.
                ;ACCUMULATOR CONTENTS ARE LOST.
                ;TO USE WITH A DIFFERENT PORT ADDRESS, CHANGE THE "BASE" EQUATE.
                ;ALL OTHER ADDRESSES RELATIVE TO BASE

0010 =          BASE    EQU   10H       ;BEGINNING PORT ADDRESS
0011 =          DATA    EQU   BASE+1    ;DATA PORT
0010 =          CMND    EQU   BASE      ;COMMAND PORT

0100            ORG     100H

0100 D311   START   OUT     DATA        ;OUTPUT DATA TO DATA PORT
0102 3E01           MVI     A,01        ;SET STB HIGH
0104 D310           OUT     CMND        ;OUTPUT TO COMMAND PORT
0106 3E00           MVI     A,00        ;SET STB LOW
0108 D310           OUT     CMND        ;OUTPUT TO COMMAND PORT
010A C9             RET                 ;DONE
```

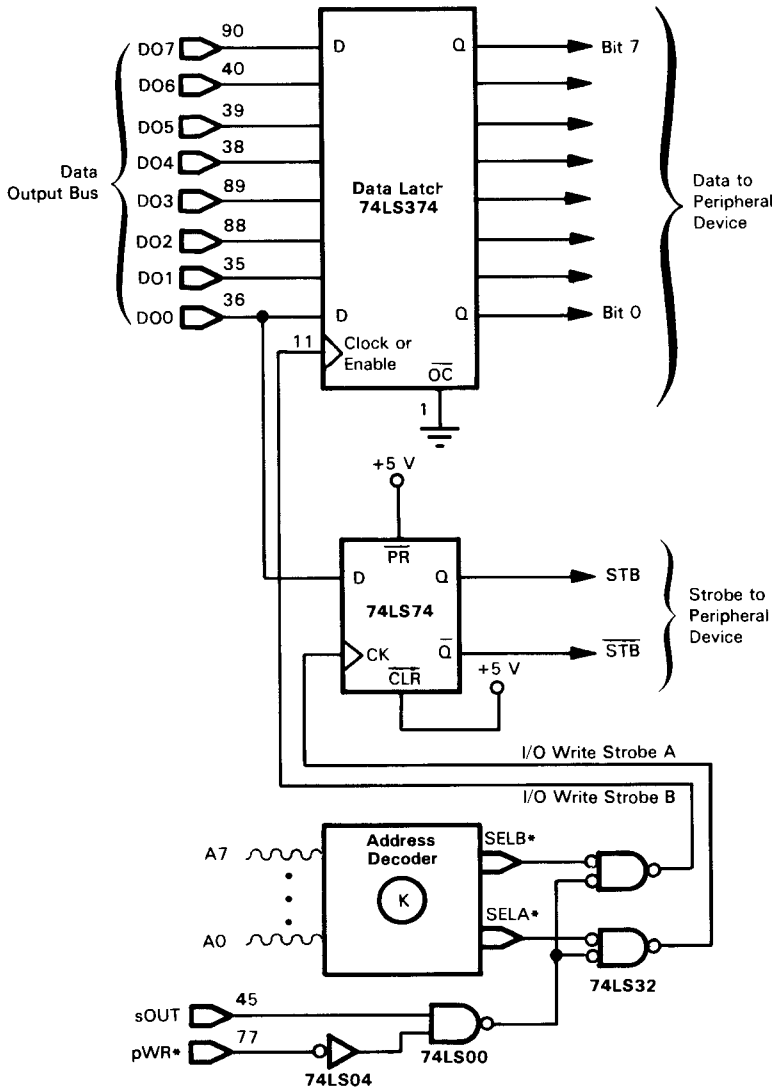


FIGURE 8-4. Latched Output Port with Handshaking Strobe

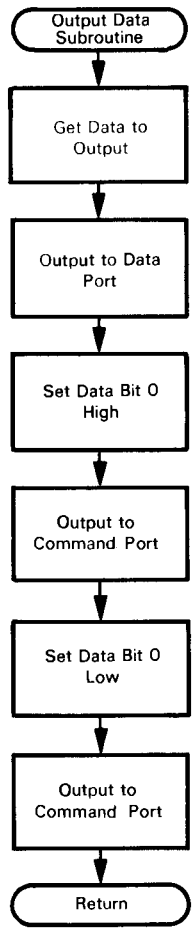


FIGURE 8-5. Flowchart of Program to Output Data and Strobe

A circuit which first checks if the peripheral is ready to receive the output data is shown in Figure 8-6. A 1-bit status input port has been added. The input for this port comes from an output on the peripheral which tells the master that the peripheral is "busy" and is not ready to receive the data from the master. For example, a printer might be busy printing the last character sent to it, and until it is finished printing the character it cannot take in another character. Notice that only two port addresses are decoded. Port A is used as the address for an input port

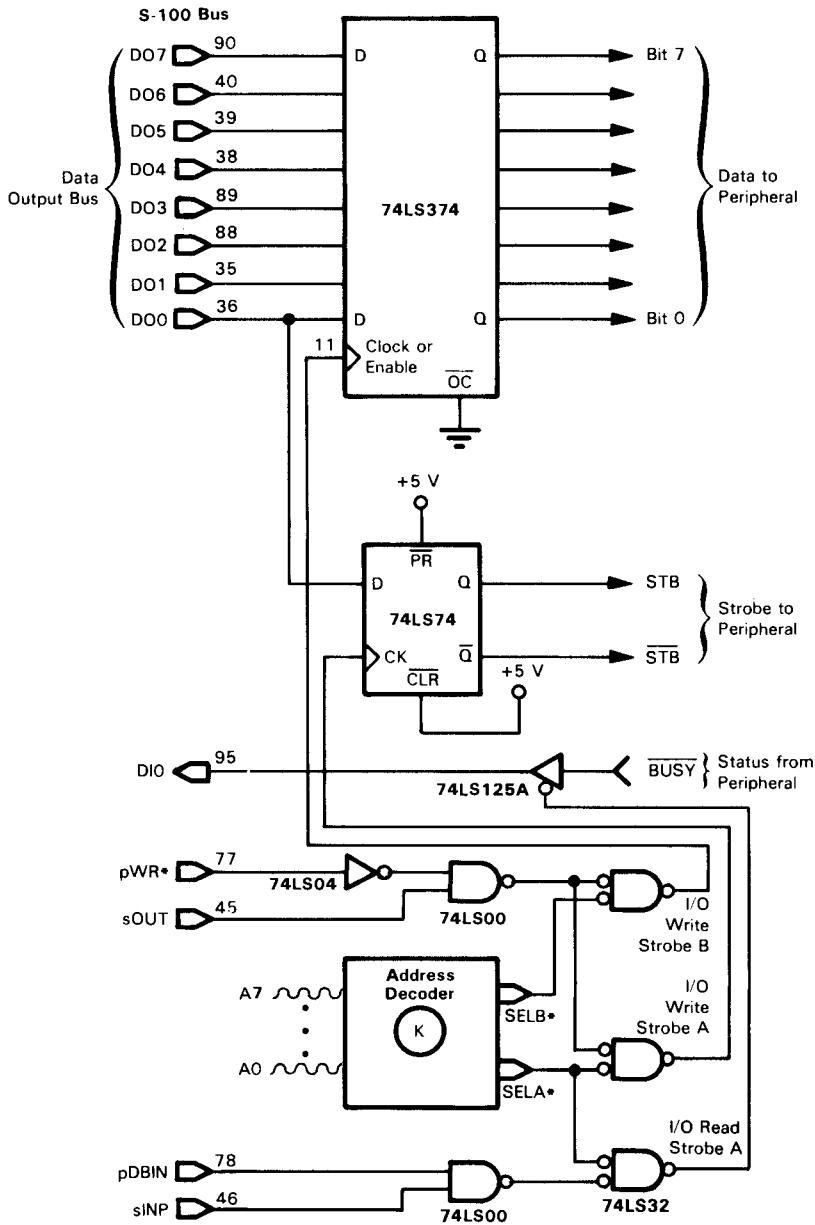


FIGURE 8-6. Latched Output Port with Output Strobe and Busy Handshaking

(busy status) and an output port (strobe control), while port B is used only for data output. The driver subroutine for this interface circuit is shown below. The flowchart is shown in Figure 8-7.

```
                ;SUBROUTINE TO OUTPUT DATA TO PARALLEL PORT WITH BUSY TEST
                ;AND OUTPUT STROBE HANDSHAKING (FIGURE 8-6)
                ;ASSUMES DATA IN ACCUMULATOR WHEN CALLED
                ;CONTENTS OF ACCUMULATOR ARE LOST ON RETURN
                ;ROUTINE WILL NOT RETURN UNTIL DATA HAS BEEN SENT

                ;TO USE WITH DIFFERENT PORT ADDRESSES, CHANGE "BASE" EQUATE
                ;ALL OTHER ADDRESSES RELATIVE TO BASE

0010 =                  BASE    EQU  10H      ;BEGINNING PORT ADDRESS
0011 =                  DATA    EQU  BASE+1   ;DATA PORT
0010 =                  STATUS  EQU  BASE     ;STATUS INPUT PORT
0010 =                  CMND    EQU  BASE     ;COMMAND OUTPUT PORT
0001 =                  MASK    EQU  01H      ;BUSY BIT MASK

0100                    ORG     100H

0100 F5                 PUSH    PSW           ;SAVE DATA IN ACCUMULATOR
0101 DB10       BSYTST  IN      STATUS        ;READ BUSY BIT
0103 E601               ANI     MASK          ;MASK IT FROM OTHER BITS
0105 CA0101             JZ      BSYTST        ;READ AGAIN IF LOW
0108 F1                 POP     PSW           ;GET DATA TO BE SENT
0109 D311               OUT     DADA          ; SEND IT
010B 3E01               MVI     A,01H         ;SET STB BIT HIGH
010D D310               OUT     CMND          ;SEND TO COMMAND PORT
010F 3E00               MVI     A,00H         ;SET STB BIT LOW
0111 D310               OUT     CMND          ;SEND TO COMMAND PORT
0113 C9                 RET                   ;DONE
```

A peripheral which sends data to the master must usually tell the master that it is sending data. For example, a keyboard must tell the master when a key has been pressed, and then, and only then, should the master read the keyboard data input port to determine which key has been pressed. In addition, if the data presented by the peripheral to the master is present only for a short time then a latched data input port must be provided to hold the data until the master can read it. An illustration of this type of input handshaking is shown in Figure 8-8.

In this circuit, a 1-bit latched input port is used to receive the key pressed signal ($\overline{DAV}$) from the keyboard. The same $\overline{DAV}$ strobe signal from the keyboard strobes the data into the data input latch. The setting of the status bit tells the master, when it reads the status input port, that a key has been pressed and that the "data is available" ($\overline{DAV}$) in the data input latch. The master polls the status input port first to check if data is available. If it is, then the data is read from the data input port and, at the same time, the $\overline{DAV}$ latch is reset so that the next keypress will be detected and data latched. The master will not return from the subroutine until a key has been pressed. The subroutine is shown below. The flowchart is shown in Figure 8-9.

```
                ;SUBROUTINE TO READ DATA FROM INPUT PORT WITH HANDSHAKING
                ;RETURNS WITH DATA IN ACCUMULATOR WHEN CALLED.
                ;ROUTINE WILL NOT RETURN UNTIL DATA HAS BEEN READ

                ;TO USE WITH DIFFERENT PORT ADDRESSES, CHANGE "BASE" EQUATE
                ;ALL OTHER ADDRESSES RELATIVE TO BASE.
```

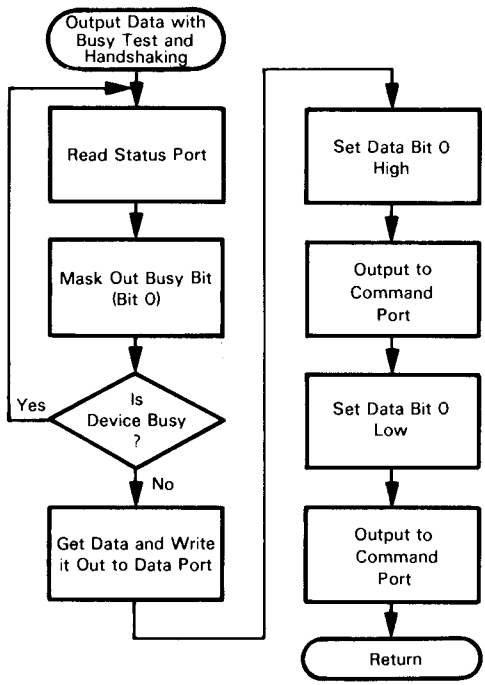


FIGURE 8-7. Flowchart of Program to Read Busy Bit, Send Data and Output Strobe Bit

```
0010 =          BASE    EQU  10H      ;BEGINNING PORT ADDRESS
0011 =          DATA    EQU  BASE+1   ;DATA PORT
0010 =          STATUS  EQU  BASE     ;STATUS INPUT PORT
0001 =          MASK    EQU  01H      ;DAV BIT MASK

0100            ORG  100H

0100 DB10       DAVTST  IN   STATUS   ;READ DAV BIT
0102 E601               ANI  MASK     ;MASK IT FROM REST OF BITS
0104 CA0001             JZ   DAVTST   ;READ AGAIN IF LOW
0107 DB11               IN   DATA     ;READ DATA
0109 C9                 RET           ;DONE
```

MEMORY-MAPPED I/O

Using memory-mapped I/O instead of port I/O can provide faster I/O, which may be needed in some real-time applications such as digital-to-analog conversion. For example, using an 8080 OUT instruction with a 2 MHz clock means

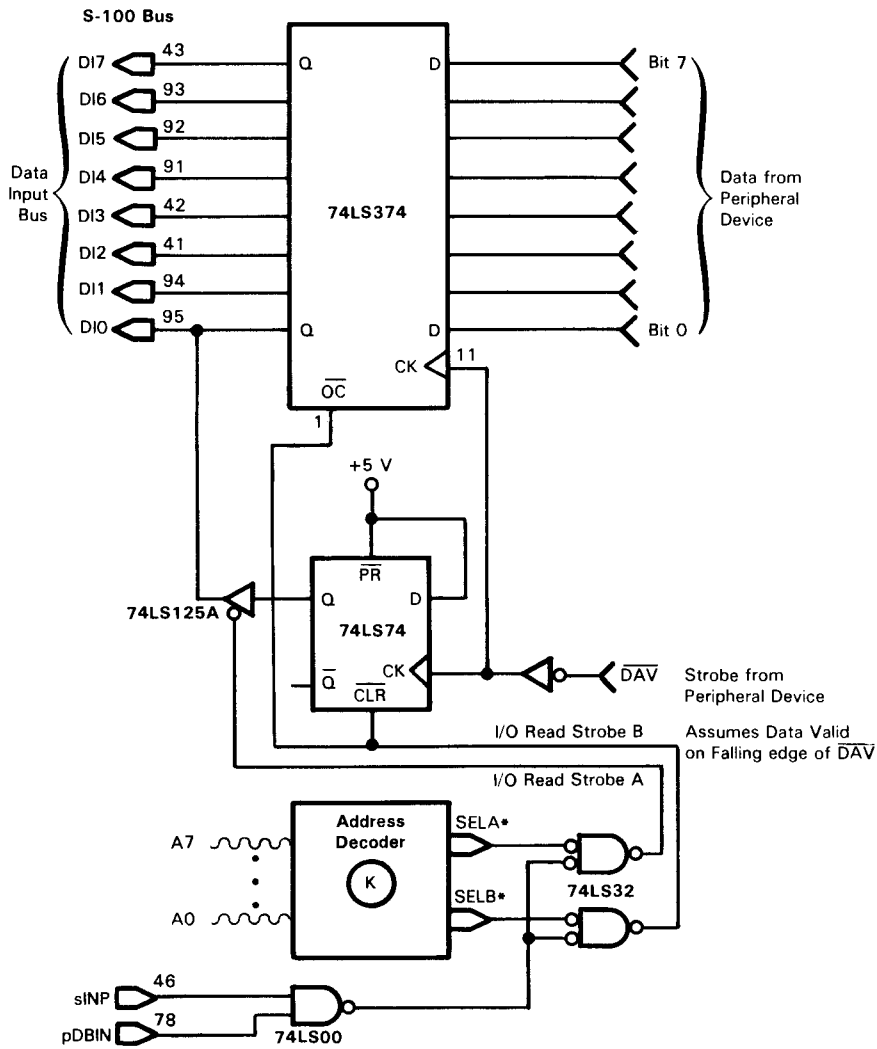


FIGURE 8-8. Latched Input Port with $\overline{\text{DAV}}$ Strobe

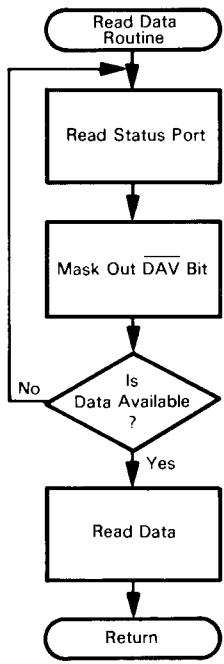


FIGURE 8-9. Flowchart to Read Input Data

5 μs execution time for successive data outputting. With memory I/O this execution time could be reduced to 2 μs.

Compare the following two 8080 program routines to understand why memory-mapped I/O is so much faster than I/O-mapped output. Both output the contents of the B through E registers. The first uses standard I/O.

```
MOV  A,B        ;OUTPUT REGISTER DATA
OUT  PORT
MOV  A,C        ;OUTPUT C-REGISTER DATA
OUT  PORT
MOV  A,D        ;OUTPUT D-REGISTER DATA
OUT  PORT
MOV  A,E        ;OUTPUT E-REGISTER DATA
OUT  PORT
```

In the following example, memory-mapped I/O is used.

```
LXI  H,PORT     ;SET PORT ADDRESS
MOV  M,B        ;OUTPUT B-REGISTER DATA
MOV  M,C        ;OUTPUT C-REGISTER DATA
MOV  M,D        ;OUTPUT D-REGISTER DATA
MOV  M,E        ;OUTPUT E-REGISTER DATA
```

In the first example, fifteen 8080 clock cycles are required to output each register. In the memory-mapped example, only five clock cycles are required, providing a threefold improvement in speed.

interfacing to the real world — input

9

When we connect the CPU to peripheral devices we say that we are interfacing the CPU to the "real world." In other words, now we are connecting the computer to external devices that do things. After all, the CPU does nothing more than "process" data between input and output. In this chapter we will see how we connect such things as keyboards, switches, and sensors to the computer.

Interface driver circuitry and driver software are required to accomplish this interface. This is because these devices typically operate at different voltages and/or currents or require that power be supplied by the interface. Further, a program or software routine, most often called a "driver program," is required to format the data and control information to or from the processor.

All of the circuits in this chapter are designed to be used with the parallel input ports shown in the previous chapter, or with most well designed commercially available parallel interfaces.

INPUTTING FROM SWITCHES

Reading the condition of a switch, whether it is open or closed, is one of the most basic computer input operations. Figure 9-1*a* illustrates a simple switch input. The switch is connected to bit 0 of a parallel input port. When the switch is closed the bit 0 input is low and when the switch is open the bit 0 input is high. Figure 9-1*b* also illustrates the use of a pushbutton (momentary contact) switch. When using this type of switch and pull-up resistor circuit, the "logical" position of the switch is inverted. An open, or off, switch will be read by the computer as a

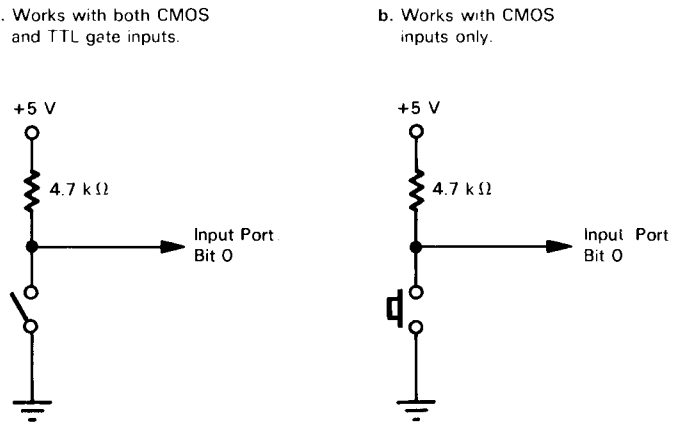


FIGURE 9-1. Connecting Switches to an Input Port

logic one and a closed, or on, switch will be read as a logic zero. This may be confusing because you normally think of "on" as a "one" and "off" as a "zero." To change the logical states you can use inverters after the switches or invert in software. In the case where the switch is not permanently marked with on and off indications, just mount the switch upside down.

Here are some 8080/8085 software drivers for the circuits shown. For example, if we wish to start a process when the pushbutton switch in Figure 9-1*b* is depressed, we could use the following program. The flowchart is shown in Figure 9-2.

```
                ;ROUTINE TO START A PROCESS WHEN
                ;A PUSHBUTTON SWITCH IS DEPRESSED.
                ;IT IS ASSUMED THAT THE PROCESS
                ;SUBROUTINE STARTS AT MEMORY LOCATION 200H
                ;
                ;
0010 =          BASE    EQU   10H       ;PORT STARTING ADDRESS
0010 =          PORT    EQU   BASE      ;SWITCH PORT
0200 =          START   EQU   200H      ;PROCESS SUBROUTINE LOCATION
                ;
                ;
0100            ORG   100H
                ;
0100 DB10       LOOP    IN    PORT      ;CHECK SWITCH
0102 E601               ANI   1
0104 C20001             JNZ   LOOP      ;NOT CLOSED CHECK AGAIN
0107 C30002             JMP   START     ;CLOSED THEN START PROCESS
```

A rotary type switch may also be connected to a port, as shown in Figure 9-3, to determine the switch position. The following short routine will determine which of the six possible positions the switch is in, and cause the processor to branch to a selected routine. The flowchart is shown in Figure 9-4.

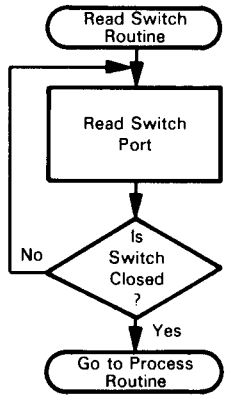


FIGURE 9-2. Read Switch Routine Flowchart

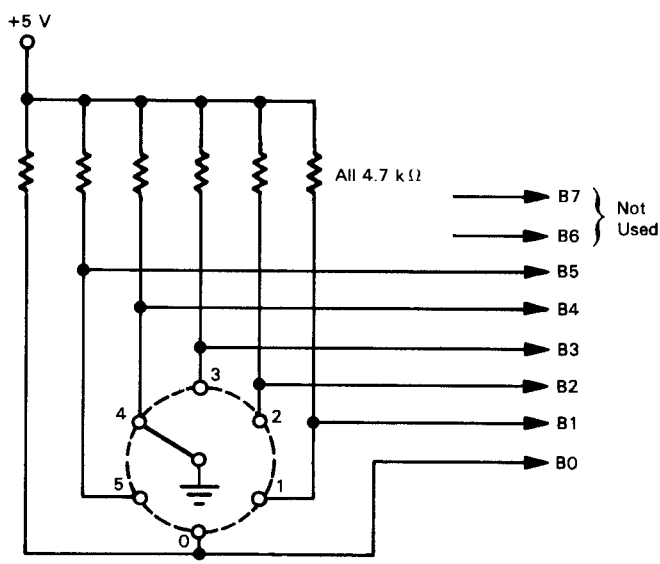


FIGURE 9-3. Connecting Rotary Switch to an Input Port

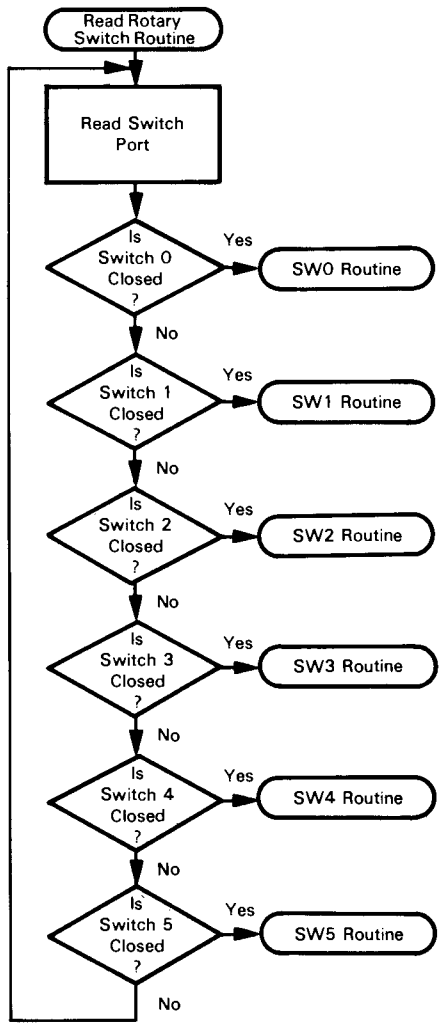


FIGURE 9-4. Rotary Switch Routine Flowchart

```
                ;ROUTINE TO TEST POSITION OF A ROTARY SWITCH
                ;AND BRANCH TO SIX DIFFERENT SERVICE ROUTINES.
                ;IT IS ASSUMED THAT ROUTINE SW0 STARTS AT 200H
                ;
                ;
0010 =          BASE    EQU   10H       ;PORTS STARTING ADDRESS
0010 =          PORT    EQU   BASE      ;SWITCH INPUT PORT
0200 =          SW0     EQU   200H      ;     SERVICE
0300 =          SW1     EQU   SW0+100H  ;     ROUTINE
0400 =          SW2     EQU   SW1+100H  ;    ADDRESSES
```

```
0500 =          SW3         EQU   SW2+100H
0600 =          SW4         EQU   SW3+100H
0700 =          SW5         EQU   SW4+100H
                ;
                ;
0100            ORG   100H
                ;
0100 DB10       RDSW        IN    PORT        ;READ SWITCH
0102 1F                     RAR               ;IF IN POSITION-0
0103 D20002                 JNC   SW0         ;  EXECUTE ROUTINE SW0
0106 1F                     RAR               ;IF IN POSITION-1
0107 D20003                 JNC   SW1         ;  EXECUTE ROUTINE SW1
010A 1F                     RAR               ;IF IN POSITION-2
010B D20004                 JNC   SW2         ;  EXECUTE ROUTINE SW2
010E 1F                     RAR               ;IF IN POSITION-3
010F D20005                 JNC   SW3         ;  EXECUTE ROUTINE SW3
0112 1F                     RAR               ;IF IN POSITION-4
0113 D20006                 JNC   SW4         ;  EXECUTE ROUTINE SW4
0116 1F                     RAR               ;IF IN POSITION-5
0117 D20007                 JNC   SW5         ;  EXECUTE ROUTINE SW5
011A C30001                 JMP   RDSW        ;NOT IN ANY POSITION
                                              ;  READ AGAIN
```

Magnetically operated switches are particularly easy to use to sense position. Figure 9-5*a* illustrates a magnet attached to a rotary moving device or to a linear moving device. When the magnet is adjacent to the reed switch, the switch contacts close (reed switches which open under the influence of a magnet are also available). In the example shown in Figure 9-5*b* the position of the linear moving device can be determined.

Using the setup shown in Figure 9-5*a*, a process can be started when the wheel is positioned correctly, as follows:

```
                ;ROUTINE TO START A PROCESS WHEN WHEEL
                ;IS POSITIONED CORRECTLY.
                ;IT IS ASSUMED THAT THE PROCESS ROUTINE
                ;STARTS AT MEMORY LOCATION 200H
                ;
                ;
0010 =          PORT        EQU   10H         ;SWITCH PORT ADDRESS
0200 =          PROCS       EQU   200H        ;PROCESS ROUTINE ADDRESS
                ;
                ;
0100            ORG   100H
                ;
                ;
0100 DB10       POSIT       IN    PORT        ;CHECK SWITCH
0102 E601                   ANI   01H         ;IF SWITCH IS NOT CLOSED
0104 C20001                 JNZ   POSIT       ;  CHECK AGAIN
0107 C30002                 JMP   PROCS       ;IF CLOSED START PROCESS
```

The position of the linearly moving object shown in Figure 9-5*b* can be determined with the following 8080/8085 routine.

```
                ;ROUTINE TO DETECT THE POSITION OF A MOVING
                ;OBJECT AND EXECUTE ONE OF FOUR DIFFERENT ROUTINES
                ;DEPENDING ON LINEAR POSITION.
                ;FIRST POSITION ROUTINE BEGINS AT ADDRESS 200H
                ;
                ;
0010 =          PORT        EQU   10H         ;SWITCHES PORT ADDRESS
0200 =          ROUT0       EQU   200H        ;ADDRESSES
0300 =          ROUT1       EQU   ROUT0+100H  ;  OF
0400 =          ROUT2       EQU   ROUT1+100H  ;     POSITION
```

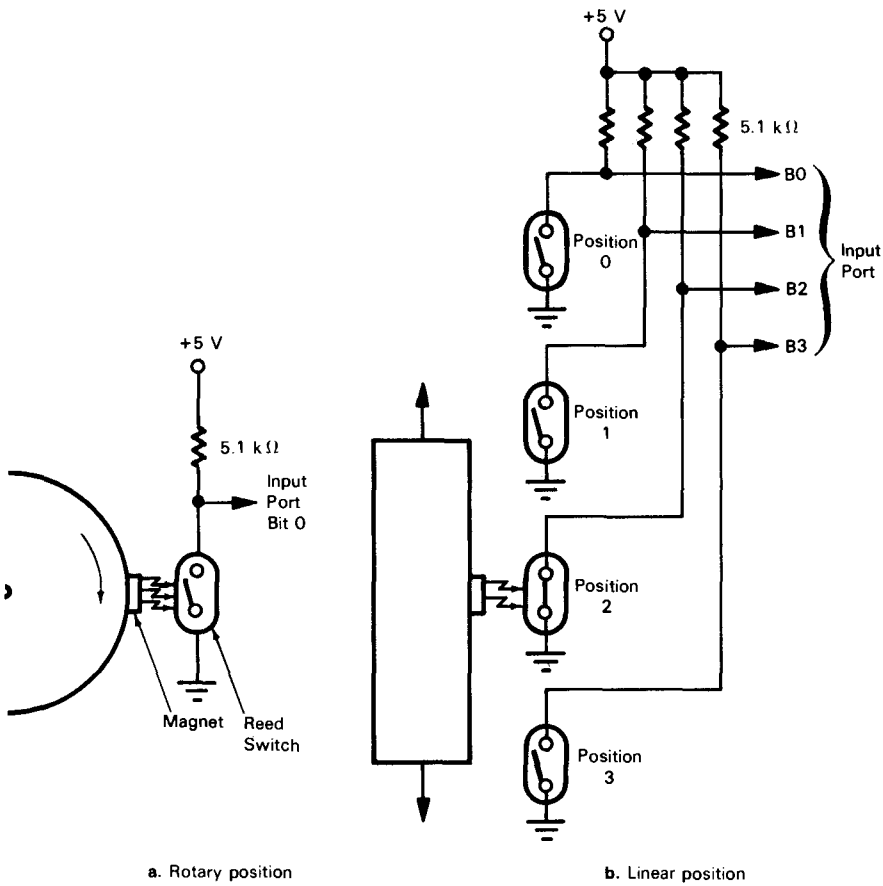


FIGURE 9-5. Determining Position With Magnetically Operated Switches

```
0500 =          ROUT3     EQU   ROUT2+100H ;       ROUTINES
                ;
                ;
0100            ORG   100H
                ;
0100 DB10       POSIT     IN    PORT        ;CHECK LINEAR POSITION
0102 1F                   RAR               ;IF IN POSITION-0
0103 D20002               JNC   ROUT0       ;   EXECUTE ROUTINE-0
0106 1F                   RAR               ;IF IN POSITION-1
0107 D20003               JNC   ROUT1       ;   EXECUTE ROUTINE-1
010A 1F                   RAR               ;IF IN POSITION-2
010B D20004               JNC   ROUT2       ;   EXECUTE ROUTINE-2
010E 1F                   RAR               ;IF IN POSITION-3
010F D20005               JNC   ROUT3       ;   EXECUTE ROUTINE-3
0112 C30001               JMP   POSIT       ;IF NO POSITION DETECTED
                                            ;   TRY AGAIN
```

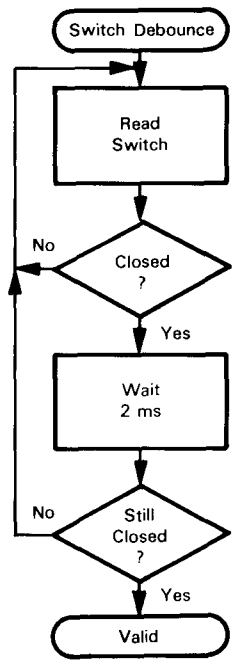


FIGURE 9-6. Flowchart for Switch Debounce Routine

DEBOUNCING SWITCHES

Switches have a kind of noise associated with their operation called "switch bounce." This is produced by the mechanical bouncing of the switch contacts as they close. The switch may go on and off thousands of times a second. In some applications this switch bounce will cause errors. For example, contact bounce in a terminal keyboard will cause many characters to be generated when only one is desired.

Switch bounce can be eliminated by a variety of hardware circuits. However, it is usually more economical to do it in software. This debouncing can be accomplished by sensing a switch closure twice, separated by an appropriate time interval. A time interval of a few milliseconds is usually sufficient. The following is a sample 8080/8085 program for switch debouncing. The flowchart is shown in Figure 9-6.

```
                ;ROUTINE TO DEBOUNCE A SWITCH
                ;
                ;
0010 =          PORT    EQU   10H       ;SWITCH PORT ADDRESS
000A =          TIME    EQU   10D       ;NUMBER OF DELAY LOOPS=10
0085 =          MS      EQU   85H       ;NUMBER OF LOOPS FOR 1 MS
0001 =          MASK    EQU   1         ;MASK OFF ALL BUT BIT-0
                ;
                ;
0100            ORG   100H
                ;
0100 F5         RDSW    PUSH  PSW       ;SAVE REGISTERS
0101 C5                 PUSH  B
0102 DB10               IN    PORT      ;CHECK SWITCH
0104 E601               ANI   MASK
0106 C20001             JNZ   RDSW      ;NOT CLOSED, READ AGAIN
0109 CD1601             CALL  DELAY     ;CLOSED, WAIT FOR 10 MS
010C DB10               IN    PORT      ;CHECK SWITCH AGAIN
010E E601               ANI   MASK
0110 C20001             JNZ   RDSW      ;IF NOT CLOSED START OVER
0113 C1                 POP   B         ;RESTORE REGISTERS
0114 F1                 POP   PSW
0115 C9                 RET             ;CLOSURE WAS VALID
0116 3E0A       DELAY   MVI   A,TIME    ;SET NUMBER OF MS
0118 0685       DELY1   MVI   B,MS      ;SET MS COUNT
011A 05         DELY2   DCR   B         ;COUNT FOR 1 MS
011B C21A01             JNZ   DELY2
011E 3D                 DCR   A         ;COUNT NUMBER OF MS
011F C21801             JNZ   DELY1
0122 C9                 RET
```

The delay time is determined by calculating the time for Register B to count down to zero. This count should be set so that loop DELY2 takes 1 ms. The Accumulator counts the number of milliseconds. For example, if using an 8080/8085 microprocessor with a 2 MHz clock and no wait states, we proceed as follows. First, look up the number of clock cycles for each instruction.

Instruction	Clock Cycles	
DELY1 MVI B,MS	7	1 ms
DELY2 DCR B	5	1 ms
JNZ DELY2	10	15 clock cycles, 1 ms
DCR A	5	15 clock cycles, 1 ms
JNZ DELY1	10	1 ms

The CALL, MVI A, and RET instructions can be ignored since they occur only once. When the system is operating at the standard 2 MHz clock rate, 2000 clock cycles occur every millisecond. The value (MS) in Register B (which is the number of inner loops) is calculated as follows:

$$2000 = 7 + [\,(\,5 + 10\,) \times MS\,]$$
$$2000 - 7 = 15 \times MS$$
$$1993 \div 15 = MS$$
$$MS = 133_{10} = 85_{16}$$

Therefore, if TIME = 10_{10} and MS = 85_{16}, the delay routine will provide approximately a 10 ms delay interval, as follows:

Instruction	Clock Cycles	
CALL DELAY	18	
MVI A,10D	7	
MVI B,85H	7	
DCR B	5*	$15 \times 133 = 1995$
JNZ DELY2	10*	
DCR A	5*	$2(1995 + 7 + 5 + 10) = 2017$
JNZ DELY1	10*	
RET	10	

*On 8085: DCR = 4 cycles
JNZ = 7 cycles except on last time (jump condition false) when it is 10 cycles

The total delay routine is thus:

$$(2 \times 2017) + 18 + 7 + 10 = 4069 \text{ cycles}$$

The total time is:

$$4069 \times 500 \text{ ns} = 2.035 \text{ ms}$$

Notice that the 8085 clock execution times for the DCR and JNZ instructions are slightly different and the calculations must be modified accordingly. Also, on the Z80 the following execution times should be used:

Instruction	Clock Cycles
CALL	17
MVI reg (LD reg)	7
DCR reg (DEC reg)	4
JNZ (JPNZ)	10
RET	10

Longer time delays may require the use of a register pair counter rather than a single register. This may be accomplished as follows:

```
                ;DELAY SUBROUTINE FOR LONGER TIME DELAY
                ;CONTENTS OF REGISTERS A, B, C and D ARE LOST
                ;
FFFF =          TIME        EQU   0FFFFH      ;NUMBER OF SECONDS
FFFF =          LOOPS       EQU   0FFFFH      ;NUMBER OF LOOPS
                ;
                ;
0200            ORG   200H
                ;
0200 11FFFF     DELAY       LXI   D,TIME      ;SET NUMBER OF DELAY LOOPS
0203 01FFFF     DELY1       LXI   B,LOOPS     ;SET LOOP COUNT
0206 0B         DELY2       DCX   B           ;COUNT
0207 78                     MOV   A,B         ;CHECK IF BC = 0
0208 B1                     ORA   C
0209 C20602                 JNZ   DELY2       ;IF NOT ZERO DO AGAIN
020C 1B                     DCX   D           ;COUNT SOME MORE
020D 7B                     MOV   A,E         ;CHECK IF DE = 0
020E B2                     ORA   D
020F C20302                 JNZ   DELY1       ;IF NOT ZERO DO AGAIN
0212 C9                     RET               ;DELAYED LONG ENOUGH
                                              ;  RETURN
```

The MOV A,B and MOV A,C instructions followed by ORA C and ORA D are to test for zero in the respective register pair since the DCX B and DCX D instructions do not affect any flags. The inner loop decrements register pair BC for approximately one second. The outer loop decrements register pair DE to count off the number of seconds.

INTERFACING TO KEYBOARDS AND SWITCH ARRAYS

Typical alphanumeric keyboards, as used in standard video and printer terminals, employ 50 or more switches. These switches could be connected directly to parallel ports. However, this would take six or more ports, each of which would have to be scanned to determine which key has been closed. The interfacing electronic circuitry, and hence the cost, can be substantially reduced by a matrix-scanning technique.

The same situation exists in systems employing a large number of sensing switches. For example, security systems with switch type sensors on all doors and windows, fire and smoke detectors, fault detectors, etc., may have several hundred switch inputs to the computer. In this case the amount of wiring, as well as interfacing electronics, can often be reduced by a matrix-scanning system.

The basic scheme for the matrix-scanning system is illustrated in Figure 9-7. It consists of a matrix of wires with normally open momentary-contact switches at each intersection. In this case there are 64 individual switch contacts. These switches may be keys on an alphanumeric keyboard, or any devices with switch-type closures. Only two I/O ports are needed: one output and one input. This reduces the number of ports from the previous eight to only two, and the number of wires from the previous 65 to only 16.

A program is required to scan the switch matrix and determine whether a

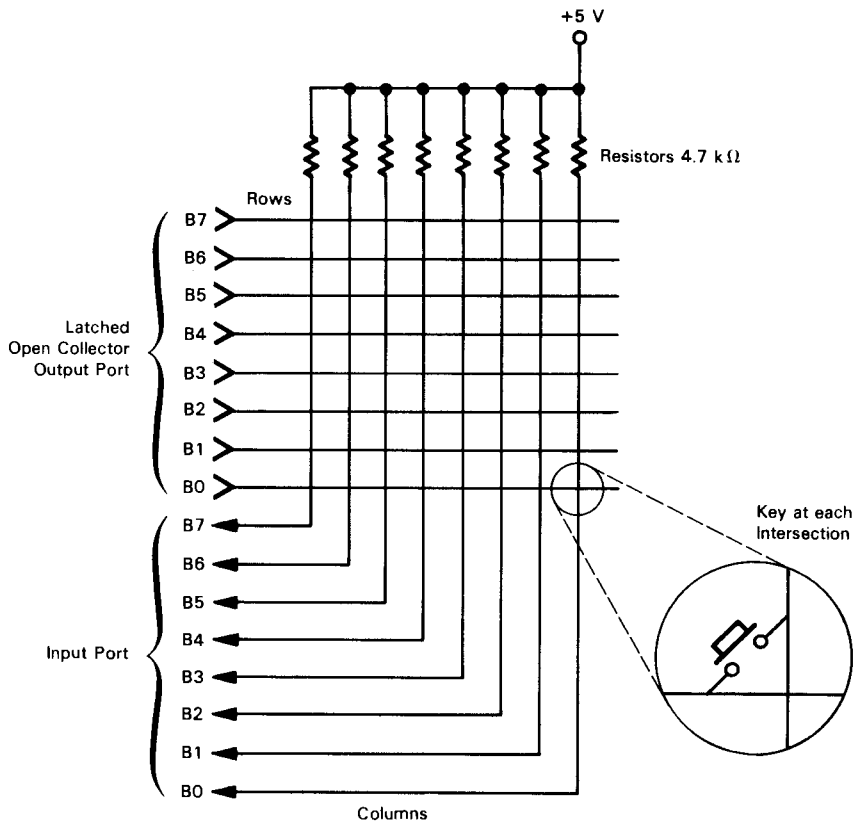


FIGURE 9-7. Keyboard Matrix Scanning System

switch has been closed, and if so, which one. Also, the program must include a switch debounce routine. Further, there is the problem of simultaneous multiple key depressions. This is cured by causing the scan routine to wait until all keys are opened before sensing a closure. This is called "lock-out." In an alarm system a lock-out could postpone timely actuation of an alarm if switches do not open again.

The flowchart for the keyboard scan routine is shown in Figure 9-8. It starts by enabling all output port bits (setting them all low). All the input port bits will be high unless a switch is closed. The program waits for all switches to be open before proceeding. Then, the occurrence of a zero at any bit of the input port indicates a contact closure. The debounce delay is then invoked, after which the

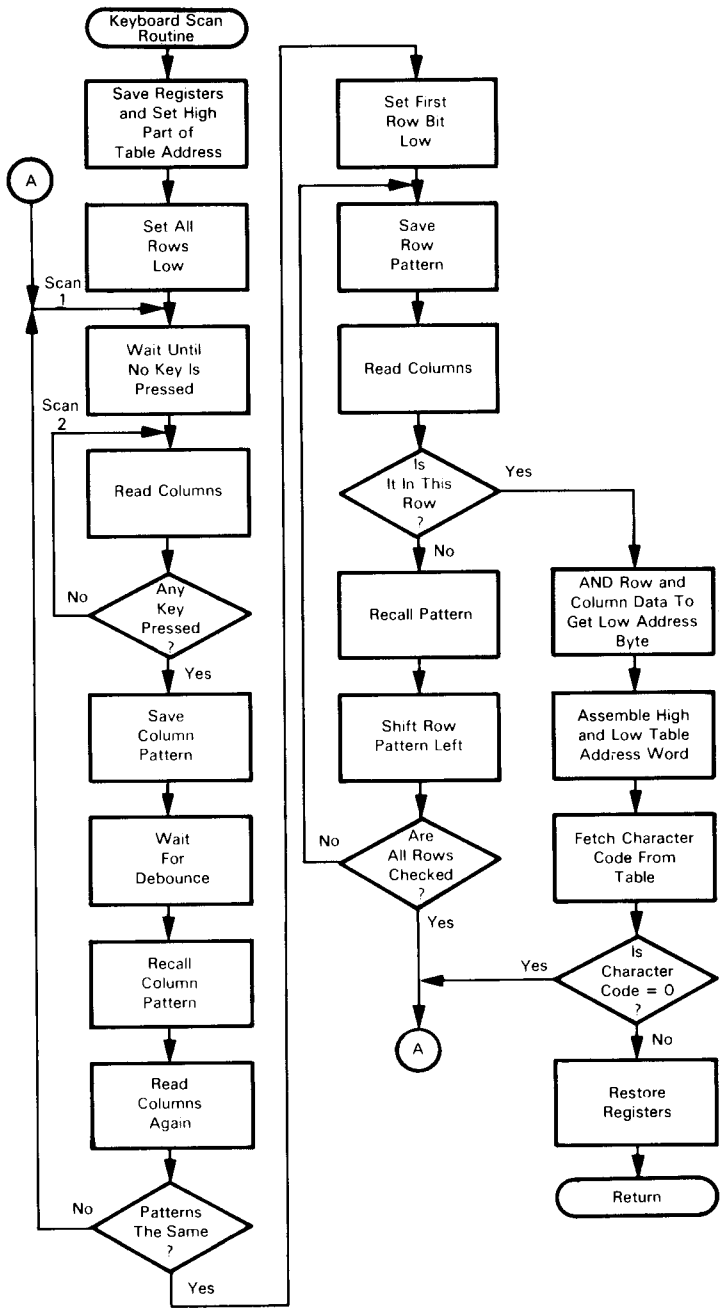


FIGURE 9-8. Flowchart for Keyboard Scanning Routine

matrix is scanned to determine which key is closed. This technique assumes no simultaneous multiple closures.

Each row is enabled, one row at a time, until a switch closure is detected on the column inputs. At this point the Accumulator contains the row count and Register B contains the column count. The LOOKUP subroutine is called which ANDs the Accumulator and Register B to form the low byte of the lookup table address. The low address byte is combined with the high address byte (set to 30_{16} at the beginning of the keyboard routine) to form the 16-bit table lookup address word. The character code is then fetched from the table. The following is the keyboard scanning routine.

```
                ;KEYBOARD SCANNING ROUTINE FOR MATRIX KEYBOARD.
                ;RETURNS WITH CHARACTER CORRESPONDING TO KEY POSITION
                ;IN ACCUMULATOR - DERIVED FROM LOOKUP TABLE

                ;LOOKUP TABLE CONSISTS OF 256 BYTES, OF WHICH ONLY
                ;64 ACTUALLY CONTAIN KEY VALUES. THE REST CONTAIN '00'
                ;WHICH IS AN ERROR CODE.
                ;TABLE ADDRESS IS CONTAINED IN "TABLE" EQUATE

0010 =                  PORT    EQU   10H   ;OUTPUT AND INPUT PORT ADDRESS
0030 =                  TABLE   EQU   30H   ;HI BYTE OF TABLE ADDRESS
000A =                  TIME    EQU   10    ;NUMBER OF DELAY LOOPS
0085 =                  MS      EQU   85H   ;NO. OF LOOPS FOR 1 MS
                ;
0100                    ORG     100H
                ;
0100 C5         SCAN    PUSH    B           ;SAVE BC REGISTER PAIR
0101 E5                 PUSH    H           ;AND HL
0102 2630               MVI     H,TABLE     ;PUT HI BYTE OF TABLE IN H
0104 AF         SCAN1   XRA     A           ;ZERO A REGISTER
0105 D310               OUT     PORT        ;SET ALL ROWS LOW
0107 DB10               IN      PORT        ;READ COLUMNS
0109 FEFF               CPI     0FFH        ;ARE ANY KEYS PRESSED?
010B C20401             JNZ     SCAN1       ;YES, WAIT FOR RELEASE
010E DB10       SCAN2   IN      PORT        ;READ COLUMNS AGAIN
0110 FEFF               CPI     0FFH        ;ARE ANY KEYS PRESSED?
0112 CA0E01             JZ      SCAN2       ;NO, TRY AGAIN
0115 F5                 PUSH    PSW         ;SAVE COLUMNS ON STACK
0116 CD4101             CALL    DELAY       ;WAIT FOR DEBOUNCE
0119 F1                 POP     PSW         ;GET COLUMNS BACK
011A 47                 MOV     B,A         ;PUT THEM INTO B
011B DB10               IN      PORT        ;READ THE COLUMNS AGAIN
011D B8                 CMP     B           ;SEE IF THE SAME AS BEFORE
011E C20401             JNZ     SCAN1       ;NO, TRY THE WHOLE THING AGAIN
0121 3EFE               MVI     A,0FEH      ;ZERO FIRST ROW BIT
0123 D310       SCAN3   OUT     PORT        ;SEND IT OUT
0125 47                 MOV     B,A         ;COPY INTO REGISTER B
0126 DB10               IN      PORT        ;READ THE COLUMNS
0128 FEFF               CPI     0FFH        ;IS IT THIS ROW?
012A C23601             JNZ     LOOKUP      ;YES, LOOK-UP CHARACTER
012D 78                 MOV     A,B         ;NO, GET ROW FOR ROTATE
012E 07                 RLC                 ;MOVE THE ZERO LEFT ONCE
012F D20401             JNC     SCAN1       ;IF ALL ROWS CHECKED
0132 47                 MOV     B,A         ;PUT A BACK IN B
0133 C32301             JMP     SCAN3       ;TRY NEXT ROW
                ;
                ;LOOKUP TABLE ROUTINE
                ;B HAS ROW INFO, A HAS COLUMN INFO
                ;THEY ARE ANDED TO GET LOW-ORDER LOOKUP TABLE BYTE
                ;H HAS HI BYTE OF POINTER TO TABLE, L IS SET BY THE ABOVE
                ;IF VALUE POINTED TO IS ZERO THEN THERE IS AN ERROR
                ;
0136 A0         LOOKUP  ANA     B           ;AND A AND B REGISTERS
```

```
0137 6F              MOV     L,A         ;PUT INTO POINTER
0138 7E              MOV     A,M         ;GET BYTE FROM TABLE
0139 FE00            CPI     00H         ;IS IT AN ERROR?
013B CA0401          JZ      SCAN1       ;YES, TRY THE WHOLE MESS AGAIN
013E E1              POP     H           ;NO, RESTORE THE REGISTERS
013F C1              POP     B           ;AND RETURN WITH THE
0140 C9              RET                 ;CHARACTER IN A
                ;
                ;SUBROUTINE FOR KEYBOUNCE DELAY
                ;
0141 3E0A    DELAY   MVI     A,TIME      ;SET NUMBER OF MS
0143 0685    DELAY1  MVI     B,MS        ;SET NUMBER OF LOOPS IN A MS
0145 05      DELAY2  DCR     B           ;START COUNT
0146 C24501          JNZ     DELAY2      ;LOOP UNTIL B=0
0149 3D              DCR     A           ;THEN DECREMENT OUTER LOOP
014A C24301          JNZ     DELAY1      ;AND REPEAT INNER LOOP
014D C9              RET                 ;AND THEN IT'S DONE
```

INTERFACING TO ENCODED KEYBOARDS

The previous example showed how to interface a matrix of keyswitches (an unencoded keyboard) with a minimum of hardware. The CPU is left to do most of the work, and many tradeoffs were made to keep the software simple. However, there are several "keyboard encoder" ICs that perform all the scanning, decoding, and error detecting functions in hardware. These ICs provide eight data bits and a data-available strobe to the CPU. The circuit in Figure 9-9 shows a typical keyboard encoder interface.

You can obtain the keyboard encoder IC and an unencoded keyboard, and wire up the matrix yourself; however, the encoder ICs are usually part of commercially available ASCII keyboards. In either case, the IC or encoded keyboard may be connected to the computer using the latched parallel input port circuit shown in Figure 8-8. The software for that circuit may also be used to read the keyboard data.

It is also possible to operate the keyboard as an interrupt device to the CPU. In this way the CPU may do other work until interrupted by a pressed key. A discussion of this technique will be left for Chapter 13.

LIGHT SENSORS

There are many different ways of sensing light and using it as a switch type (on-off) input to the CPU. All of the following circuits act in the same manner as a switch, and therefore the associated software would be the same as that for a switch type input.

A simple light sensor can be created using a photocell as shown in Figure 9-10*a*. Light falling on the photocell causes its resistance to drop, in turn causing

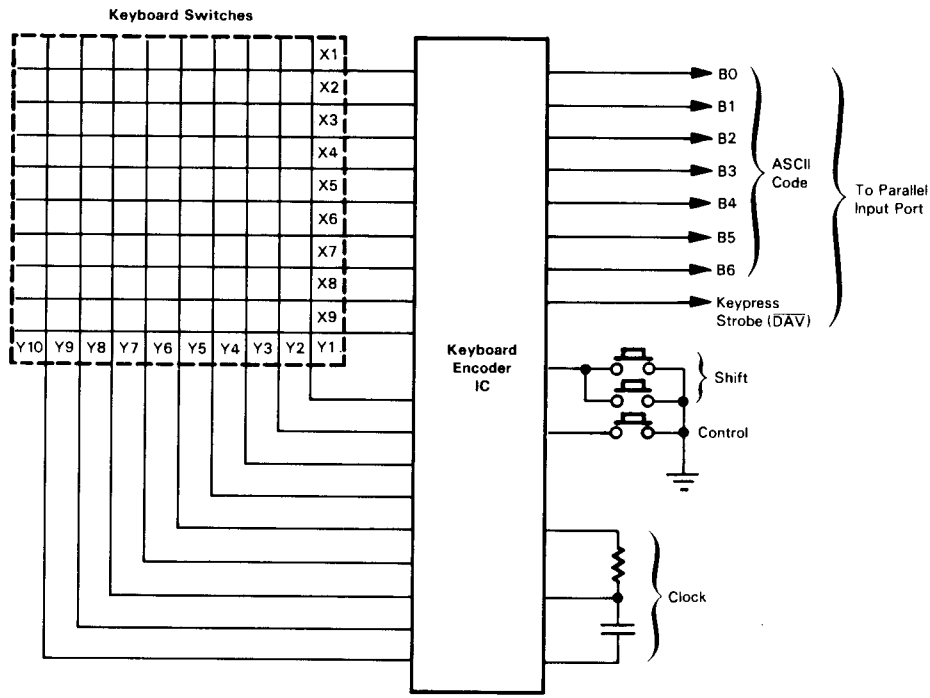


FIGURE 9-9. Typical Encoded Keyboard Circuit

the voltage at the inverter input to decrease to nearly 0 V, switching the inverter output high. A CMOS inverter, with its high input impedance, works best in this application. Even better performance can be achieved using a Schmitt-trigger type CMOS inverter, such as the 74C14. This prevents any oscillation, which might otherwise occur as the input voltage changes.

A phototransistor, as shown in Figure 9-10*b* and 9-10*c*, provides much greater sensitivity to light. These are also available with built-in lenses which provide even higher sensitivity and much better rejection of ambient light sources. An external lens may also be used. Use of a lens typically increases sensitivity by a factor of 5. In any event, a debounce type software routine or Schmitt-trigger circuit will be necessary in most applications to eliminate oscillations occurring during the transition through the linear gate region. Further, bypassing the photocell or phototransistor with a small capacitor (e.g., 5 nF) will also provide some noise filtering.

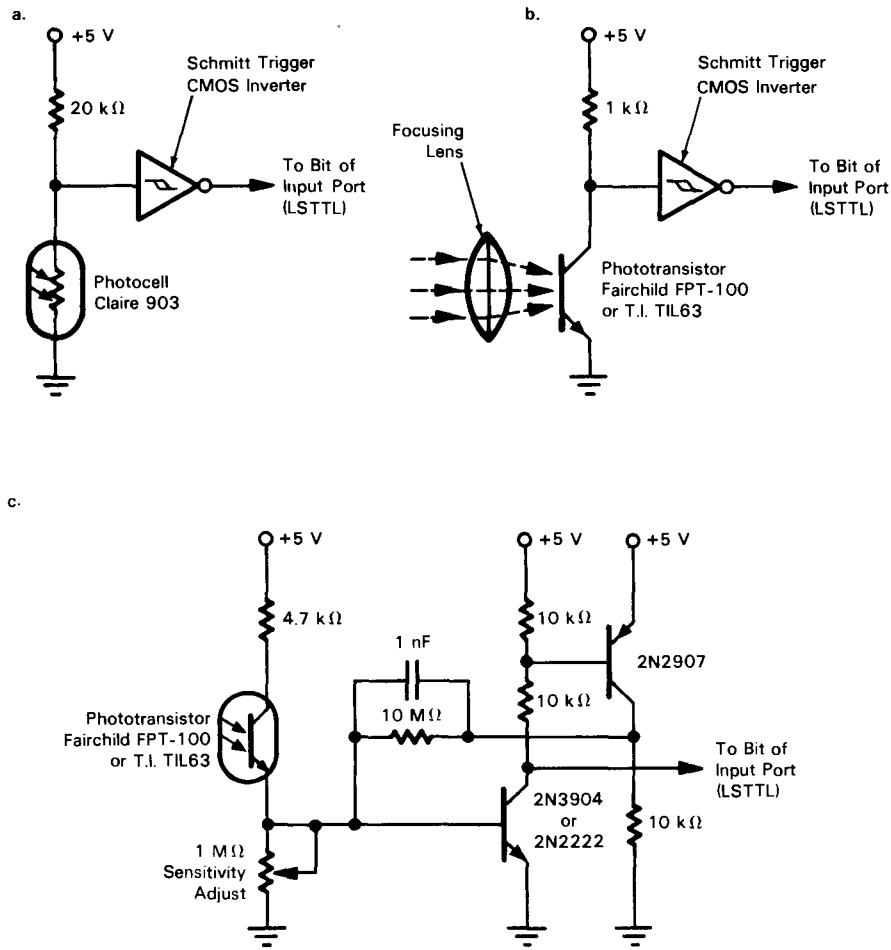


FIGURE 9-10. Using a Photocell and Phototransistors as a Port Input

Combination LED/phototransistor packages are available in a wide variety of mountings. One typical application of such a device is shown in Figure 9-11. Here, the rotational position of a disk is sensed. In fact, an LED/phototransistor combination is usually used to sense the position of floppy disks, printers, etc. The floppy disk contains one or more holes through which the LED shines onto the phototransistor.

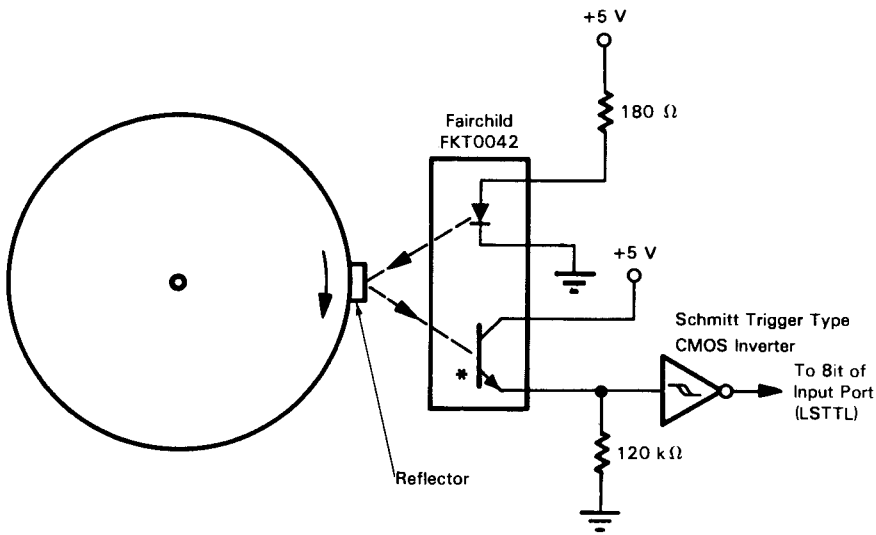
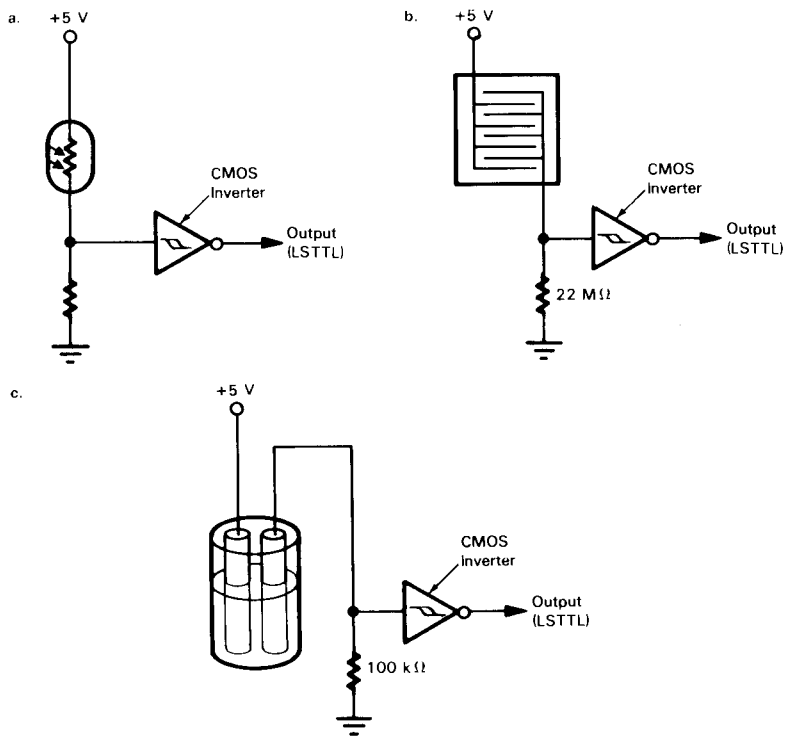


FIGURE 9-11. An LED/Phototransistor Device Used for Detecting Position of a Disk

OTHER TYPES OF SENSORS

A wide variety of sensors are suitable for CPU input. Some examples are shown in Figure 9-12. Figure 9-12*a* illustrates the use of a thermistor as a temperature-operated switch. Figures 9-12*b* and 9-12*c* illustrate how a conductivity change between a pair of conductors will be sufficient to cause a change in a logic level output. Figure 9-12*b* represents a rain sensor, while Figure 9-12*c* represents a conductivity sensor. In each case it is important that a high input impedance CMOS gate be used so as not to load the sensor excessively.

A more accurate and stable temperature sensing circuit can be made using a solid-state temperature sensing IC (in this case a National Semiconductor LM3911) as shown in Figure 9-13. The output of the LM3911 changes at a linear rate of 10 mV/°K. This output is fed to a comparator (LM301) which compares the temperature-related voltage to a trip-point voltage, set by the 50 kΩ potentiometer.



Note: All inverters are CMOS Schmitt Trigger Gates

FIGURE 9-12. Thermal and Conductivity Sensor Inputs

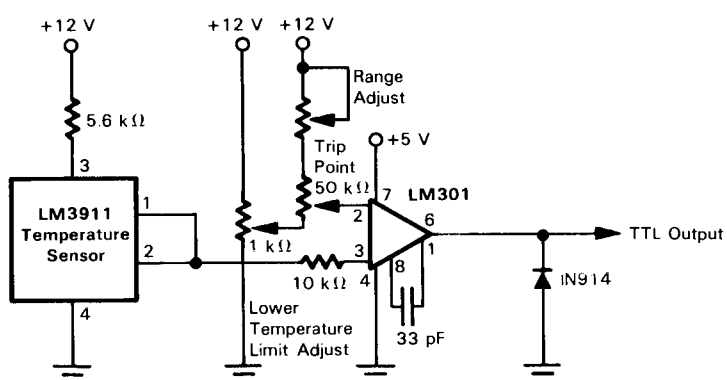


FIGURE 9-13. Precision Temperature Sensing Circuit

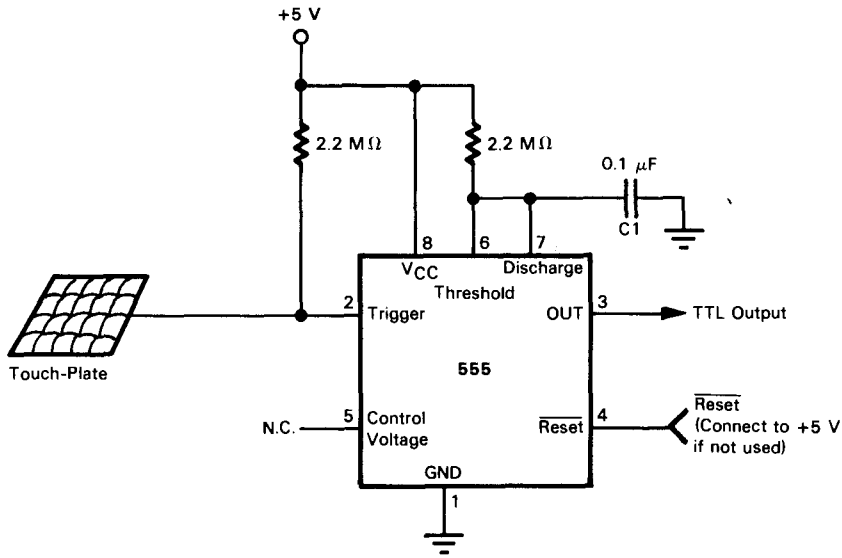


FIGURE 9-14. A Touch-Plate Sensing Circuit

A touch-plate sensor is shown in Figure 9-14. It employs a 555 timer used in a monostable multivibrator mode. It has a time-out of approximately 1 second (set by values of R1 and C1). When a person touches the touch-plate, the trigger input of the 555 picks up an induced 60 Hz signal from alternating current fields present in most buildings. If the touch-plate is touched and held, the 555 will oscillate at a rate near the time-out rate. The circuit is sensitive to noise, so leads must be kept as short as possible. A piece of aluminum screen functions well as a touch-plate.

ISOLATING INPUTS

Frequently we wish to use an input which is not at TTL voltage levels. This necessitates isolating the sensor circuit from the CPU input circuit. This is best accomplished using optical couplers. These devices are essentially a light generator (usually an LED) whose light output is controlled by the input device and a light sensor which is controlled by the light generator. Light is thus the link between input and output, and there is no electrical interconnection. We do not have to worry about the external voltages and ground levels.

The Monsanto MCT-2 (same as the 4N25, Texas Instruments TIL111, Motorola MOC1000, Fairchild FPLA820, and Litronix ISO-LIT-1) is a very widely

used optical coupler for computer interface isolation. It uses an LED and phototransistor. In applications where higher speed is needed (e.g., rates greater than 100 kbps) the Monsanto MCD-2 can be used. The MCD-2 employs a photo diode as the light sensor. In these devices the phototransistor and diode can conduct up to 50 mA and the LED can withstand more than 30 V in the off state.

A typical circuit for sensing the presence of DC voltage or current is shown in Figure 9-15*a*. When the current through the LED is greater than 2 or 3 mA the output will switch states. To calculate the value of the resistor, realize that the LED current must be about 3 mA and the voltage drop across the LED will be approximately 2 V in the on state. Therefore, the remainder of the voltage must be dropped by the current limiting resistor (R). The following formula may be used:

$$R = (Vdc - 2V) \div 0.003\ A$$

Thus, if we wish to sense when 100 V is reached:

$$R = (100 - 2) \div 0.003 = 32{,}667\ ohms$$

Since this R value is not standard and the characteristics of optical couplers vary, it is best to incorporate a potentiometer for fine tuning of the circuit operation. The circuit to detect the 100 Vdc is shown in Figure 9-15*b*. Note that a diode has been added across the input to protect the LED from excessive reverse voltage.

To detect AC voltage a simple rectifier and filter to convert the AC to DC is usually sufficient. Rectifying the AC will produce a DC voltage that is 1.414 times the RMS value of the AC voltage. For example, to detect the presence of the AC power line, at its normal level of 120 V, proceed as follows:

Convert RMS Vac to Vdc:

$$120 \times 1.414 = 170\ V$$

Then calculate the value of R, as before:

$$R = (170 - 2) \div 0.003 = 56{,}000\ ohms$$

The diode should have a PIV (peak inverse voltage) rating of at least 2.8 times the Vac, hence:

$$PIV = 120 \times 2.8 = 336\ V$$

Hence a 1N4004 rectifier, having a PIV of 400 V and rated for 1 A was selected.

The filter capacitor should be 0.1 nF to provide adequate filtering, and its voltage rating should be greater than 1.4 times the Vac. Hence, a 200 V rating was selected. The circuitry for this is shown in Figure 9-16.

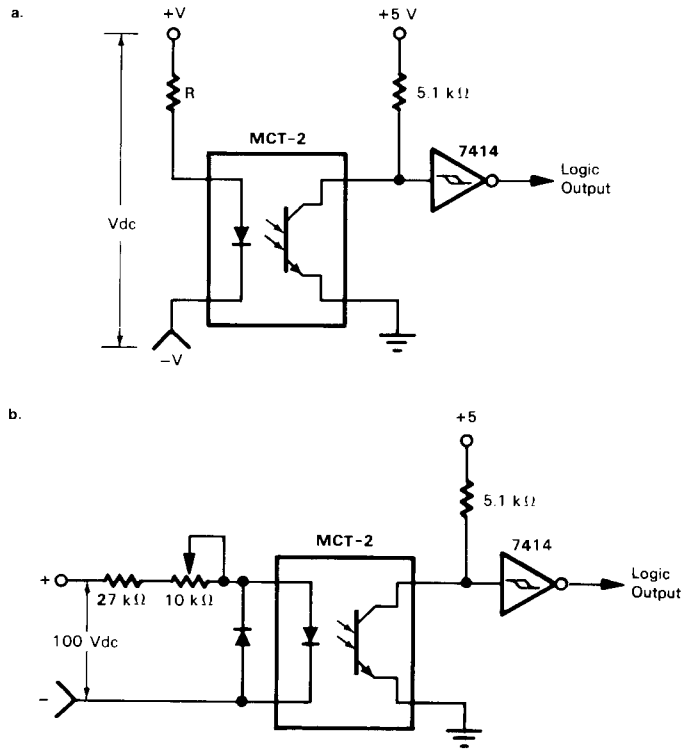


FIGURE 9-15. Using an Optical Coupler to Isolate the CPU Input

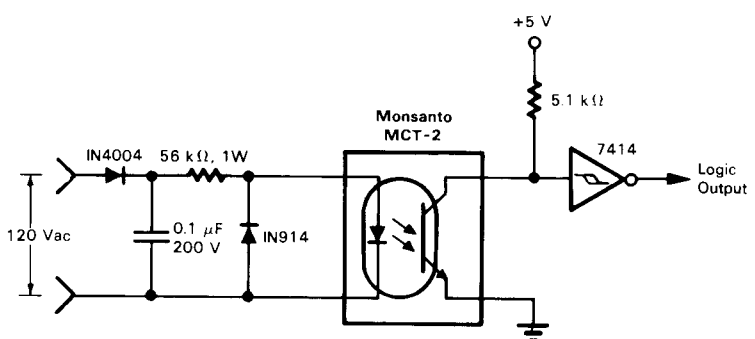


FIGURE 9-16. AC Voltage Sensor Circuit

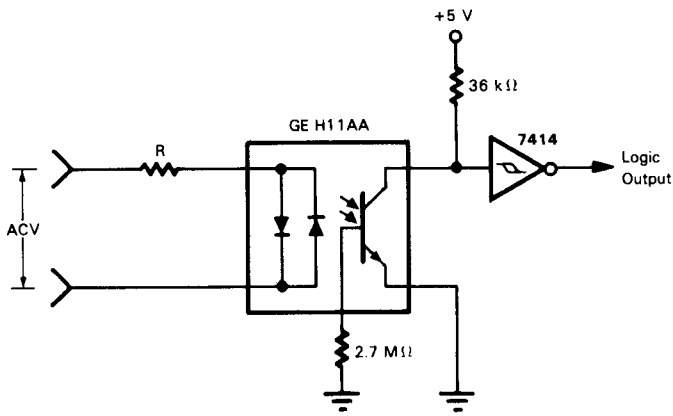


FIGURE 9-17. AC Voltage Sensor Circuit

There are optical couplers made specifically to sense AC voltage. One such device is shown in Figure 9-17. The presence of AC voltage causes a high logic output except at the time the AC voltage is crossing zero.

If this circuit is used as part of a power failure detector circuit then a much smaller size filter capacitor should be used to reduce the discharge time.

ISOLATING LOGIC SYSTEMS

We often need to isolate an input logic system from the CPU. For example, we may wish to connect the output from a piece of equipment which, although it develops TTL signal voltage levels, does not have a power supply which is properly isolated from the AC power line. Such equipment may do harm to the CPU. The isolating circuit shown in Figure 9-18*a* affords such protection.

If we wish to isolate a system employing CMOS logic circuits, the circuitry shown in Figure 9-18*b* can be used. This is necessary since the CMOS gate output cannot provide sufficient current to drive the LED.

It should be pointed out that the slow speed of opto- isolators can be a problem in applications where fast response is necessary.

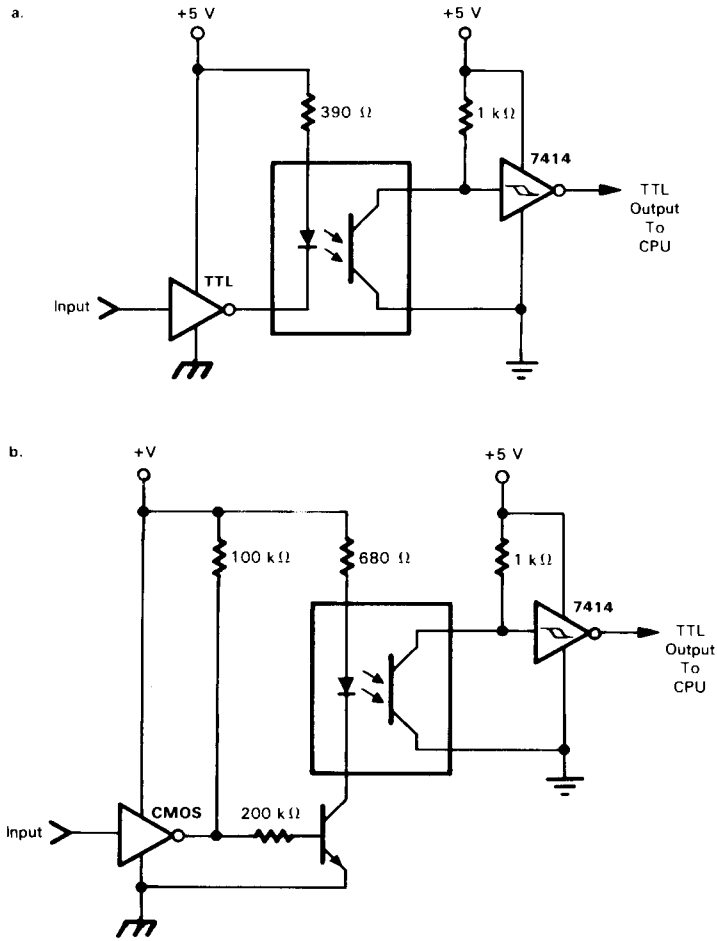


FIGURE 9-18. Isolating Logic Systems From the CPU

REFERENCES

"Applications of Opto-Isolators — Application Note #2." Litronix Inc., 19000 Homestead Rd, Vallco Park, Cupertino, CA 95014.

"Isolation Techniques Using Optical Couplers — Application Note AN-571." Motorola Semiconductor, Box 20912, Phoenix, AZ 85036.

interfacing to the real world — output

10

In the preceding chapter we examined connecting switch type inputs to a microcomputer. In this chapter we will look at how to connect a variety of devices to the output of a microcomputer. These devices will then be controlled directly by the computer.

INTERFACING TO LED'S AND LAMPS

Lamps are connected to the CPU to serve as indicators of states or events. Most often LEDs are used. Actually, an LED may be connected directly to a gate output, as shown in Figure 10-1*a*. In Figure 10-1*a* the LED will be lit when the gate output is low. When the LED is on, typically 16 mA passes through the LED. This is the maximum sink current for a standard TTL gate. However, the brightness of the LED is less than desired. A high-current buffer, such as the 7437, can be used to sink up to 48 mA. By decreasing the current-limiting resistor value to 82 ohms, the LED current is increased to 32 mA and higher brightness is achieved.

An alternative to using a buffer IC is using an LED driver transistor, as shown in Figure 10-1*b*. Essentially the same circuit may be used to drive an incandescent lamp, as shown in Figure 10-1*c*.

Seven-segment LED displays are also frequently connected to latched output ports of a microcomputer. A typical example is shown in Figure 10-2. Each display is connected to a latched output port, as shown in Chapter 8. To display

characters, the character codes are loaded into the registers at each port. For example, to display the characters "6.1", the following routine could be used.

```
;MOVE ".1" TO RIGHT LED DISPLAY
        MVI     A,79H       ;SEVEN SEGMENT CODE FOR "1" AND
                            ;  DECIMAL POINT
        OUT     PORTB
        ;
;MOVE "6" TO LEFT LED DISPLAY
        MVI     A,82H       ;SEVEN SEGMENT CODE FOR "6"
        OUT     PORTA
```

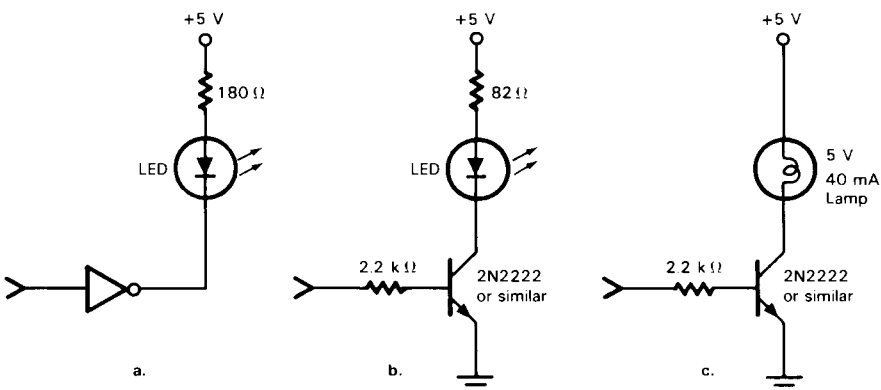
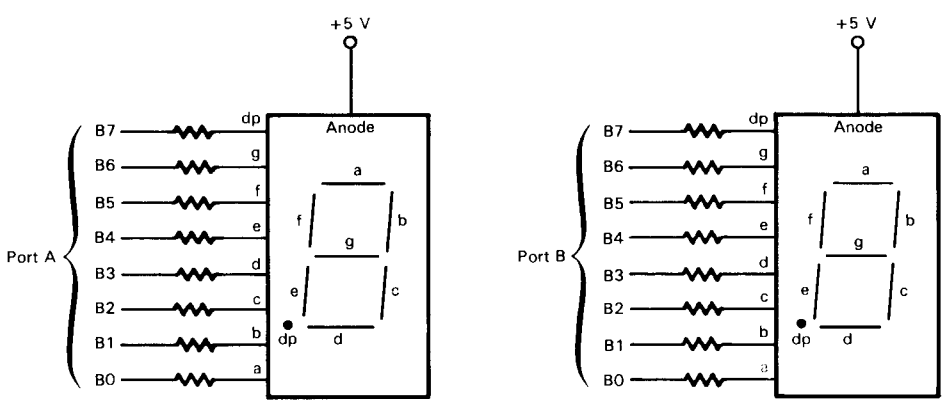


FIGURE 10-1. Interfacing to LEDs and Lamps



All resistors are 330 Ω

(Refer to Figure 4-10 for wiring of typical output port)

FIGURE 10-2. Connecting 7-Segment Display Devices to Output Ports

When many 7-segment display devices are used, the port circuitry becomes quite complex. In such a case it may be worthwhile to consider multiplexing the displays. For example, to drive six 7-segment LED displays will require six separate ports and 48 resistors. Multiplexing will reduce this to two ports and eight resistors. However, a multiplexing software routine will be required, which takes up CPU processing time. A typical multiplexed display circuit is shown in Figure 10-3. It employs a display panel consisting of an array of six 7-segment LED display devices (common cathode type) whose segments are connected in parallel.

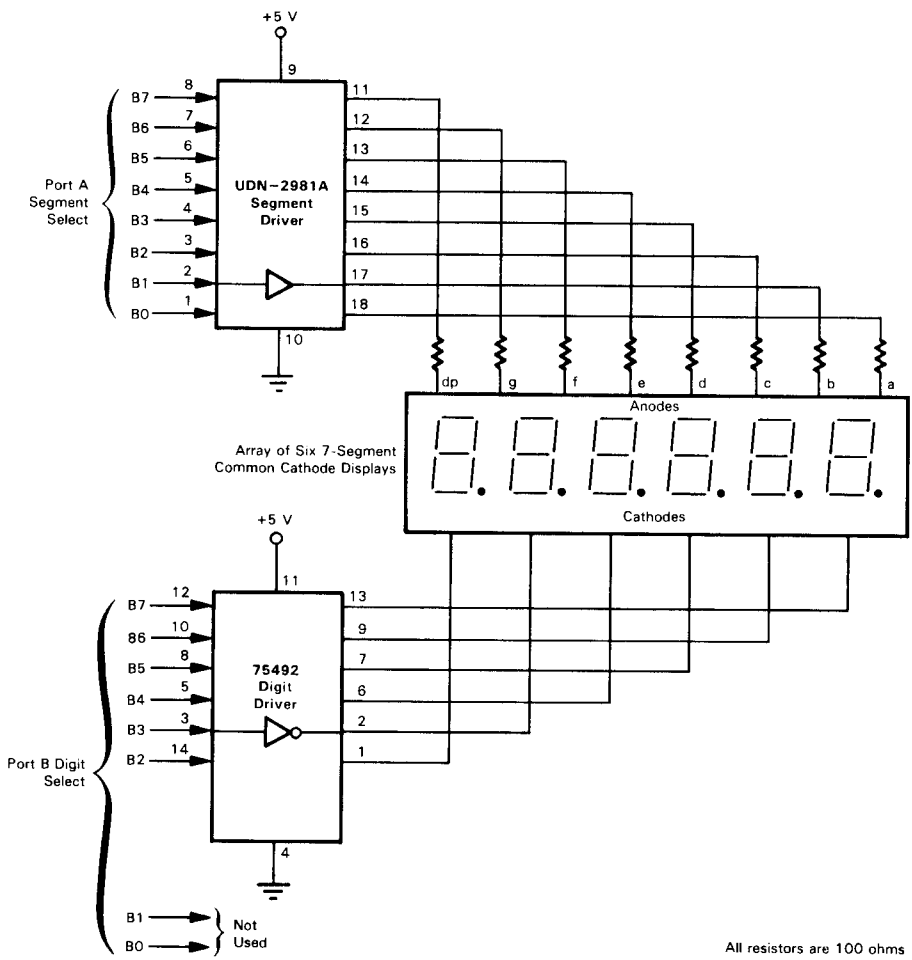


FIGURE 10-3. Wiring for a Multiplexed 6-Digit Display

This reduces the wiring to a minimum. The Sprague UDN-2981A and Texas Instruments 75492 ICs are specifically designed for this application. The 2981 contains eight segment drivers and the 75492 contains six digit drivers. Two latched output ports are employed, one for the segment outputs and the other to select the digits.

The multiplexing procedure is to scan the display one digit at a time. As each digit is enabled, in sequence, the appropriate segment code is provided. The program controls the entire procedure. In the following program, the codes for the characters to be displayed are first loaded into memory addresses DISPLAY through DISPLAY + 5 (six consecutive memory locations). Each segment code is then fetched from memory and sent to the segment port as each digit is selected. After scanning all digits, the process is repeated.

```
                ;ROUTINE FOR CONTROLLING A SIX-DIGIT MULTIPLEXED
                ;DISPLAY
                ;SEVEN-SEGMENT CHARACTER CODES MUST FIRST BE LOADED INTO
                ;MEMORY ADDRESSES STARTING AT "DISPLAY"
                ;
                ;
0010 =          BASE    EQU  10H        ;PORTS STARTING ADDRESS
0010 =          SPORT   EQU  BASE       ;SEGMENT SELECT PORT
0011 =          DPORT   EQU  BASE+1     ;DIGIT SELECT PORT
0004 =          DIGSL   EQU  4          ;DIGIT SELECT CODE
0200 =          DISPLAY EQU  200H       ;STARTING ADDRESS OF CHARACTERS
                ;
                ;
0100            ORG  100H
                ;
0100 210002 SCAN    LXI  H,DISPLAY ;SET SEGMENT CODE POINTER
0103 0604           MVI  B,DIGSL   ;SET DIGIT SELECT CODE
0105 7E     NEXT    MOV  A,M       ;FETCH SEGMENT CODE FROM RAM
0106 D310           OUT  SPORT     ;SEND IT TO SEGMENT PORT
0108 78             MOV  A,B       ;FETCH DIGIT SELECT CODE
0109 D311           OUT  DPORT     ;SEND IT TO DIGIT SELECT PORT
010B 17             RAL            ;SELECT NEXT DIGIT
010C DA0001         JC   SCAN      ;IF ALL DONE, START AGAIN
010F 47             MOV  B,A       ;SAVE DIGIT SELECT CODE
0110 23             INX  H         ;POINT TO NEXT CHARACTER
0111 C30501         JMP  NEXT      ;DISPLAY IT
```

There are currently several LSI Display Controller (driver/decoder/multiplexer) ICs on the market. These devices handle all of the tasks that the previous routine did, as well as simplify the hardware interface. They usually include memory for holding the data to be displayed, the scanning and multiplexing circuitry, the microprocessor interface circuitry, and the display drivers. They can usually be interfaced to the system as memory locations or as an I/O port (or small group of I/O ports). Once the data has been written to the Display Controller IC, it does all the rest of the work, leaving the processor free to do other tasks.

Most display manufacturers also offer these sophisticated display controllers mounted on a miniature PC board along with the displays.

DRIVING RELAYS

Some relays may be driven directly from standard TTL gates or buffers. Many such relays are provided in DIPs to make the mechanical handling easier. These relays are made by most of the larger relay manufacturers. For example, the Sigma 191TE1A2-5S relay is shown in Figure 10-4. It contains a surge suppression diode within the DIP.

Three sample relay driver circuits are shown in Figure 10-5. In Figure 10-5*a* the relay is driven directly by a TTL buffer such as the 7406, which can sink up to 40 mA. An alternative, shown in Figure 10-5*b*, is to use a transistor to control the relay. In Figure 10-5*c* several relays are driven from a 75492 hex driver, which can sink up to 250 mA at each of its outputs. However, no more than 600 mW should be dissipated by the IC continuously.

CONTROL OF DC POWER DEVICES

Control of power devices necessitates isolating the power load from the CPU. Figure 10-6 illustrates the basic circuitry for controlling a DC power device. Isolation is provided by an optical coupler, while the switching of the load is accomplished by a power transistor or power Darlington amplifier, mounted on a heat-sink. If the transistor's gain is 30 or more, the transistor can switch a load of up to 0.24 A. The use of a power Darlington amplifier will allow the switching of several amperes. A high level signal at the input causes the load device to be turned on. Although a relay can be used to switch the load, it is not as reliable and generates radio frequency interference.

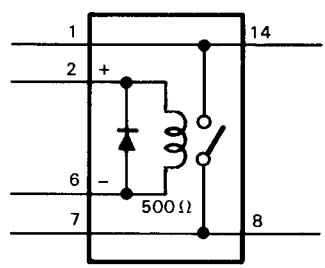


FIGURE 10-4. Pin-Out Diagram for Sigma 191TE1A2-5S Relay

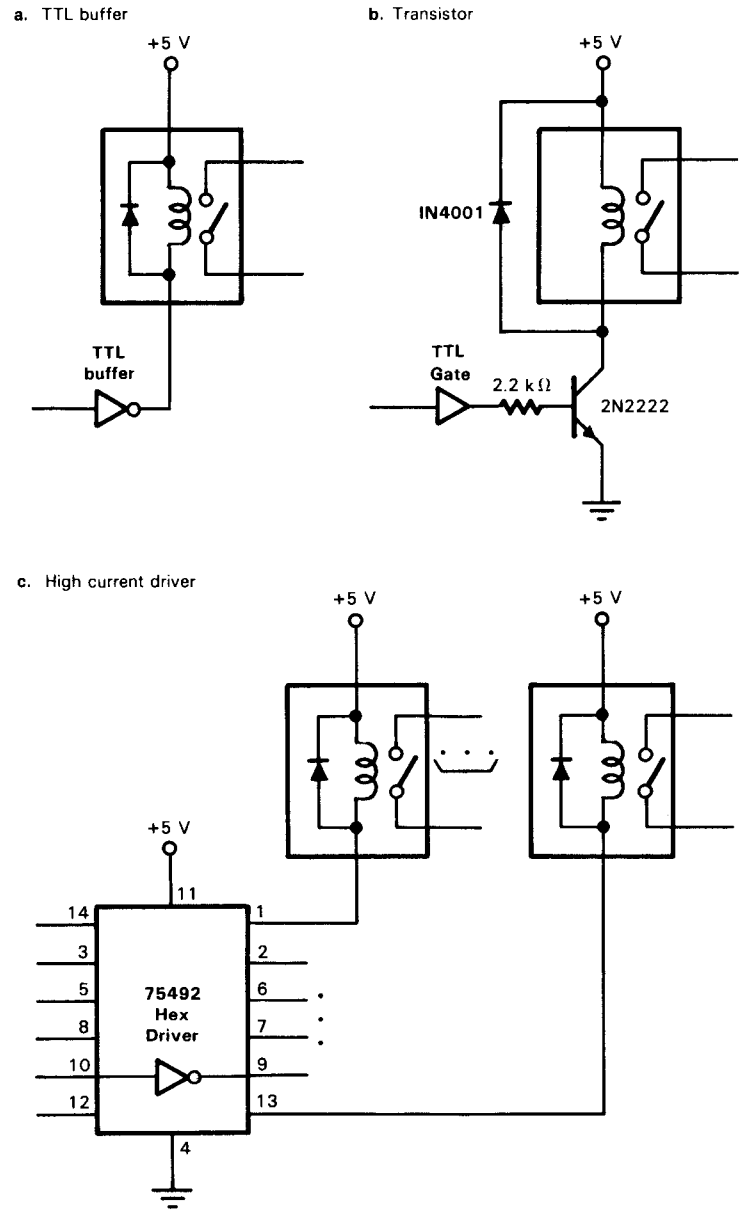


FIGURE 10-5. Typical Relay Driver Circuits

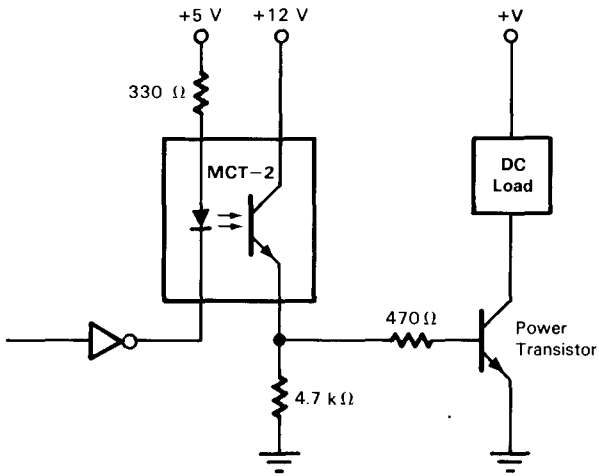


FIGURE 10-6. DC Power Control Circuit

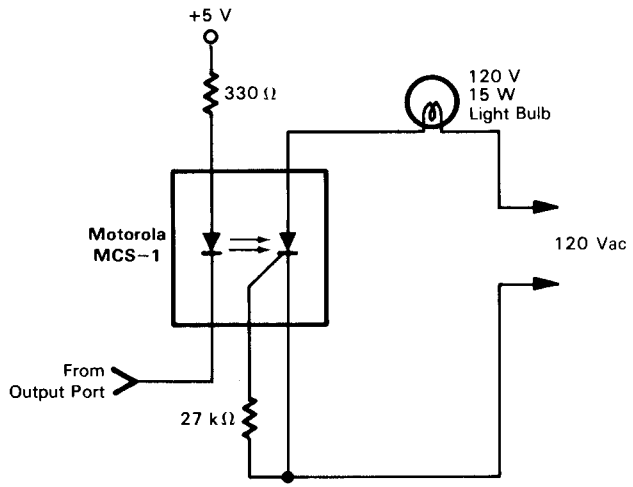
CONTROL OF AC POWER DEVICES

AC power loads are controlled using either an SCR (Silicon Controlled Rectifier) or triac (back-to-back SCRs). The basic SCR control circuit is shown in Figure 10-7*a*. The MCS-1 is a combination opto-coupler/SCR. The lamp is turned on during each positive half-cycle of the AC line voltage. Since only one half of the AC line voltage is used, the circuit is inefficient. The incandescent lamp will be lit at less than half its normal brilliance.

Efficiency can be greatly improved through the use of a full-wave bridge rectifier in conjunction with the opto-coupler/SCR, as shown in Figure 10-7*b*. Here, the SCR provides full-wave control. Actually, optical coupler/dual SCR devices are available which greatly simplify the task. Such a device is shown in Figure 10-8. Note that the Motorola MCS-1 output is rated for 250 mA maximum, while the General Instrument MCS-6200 output will only carry 150 mA.

A triac control circuit is shown in Figure 10-9. This circuit is suitable for controlling most home AC appliances. For higher power AC control a device such as the General Instrument MSR 100/200 solid-state relay can be used. It will control loads drawing up to 10 A at 120 V AC (1.2 kw). The device is shown in Figure 10-10. It features a zero-crossing trigger circuit and filter which greatly improve efficiency and reduce transient and radio frequency noise. Devices are currently available to control up to 40 A (5 kw) from a TTL level input.

a. Basic circuit



b. Improved circuit

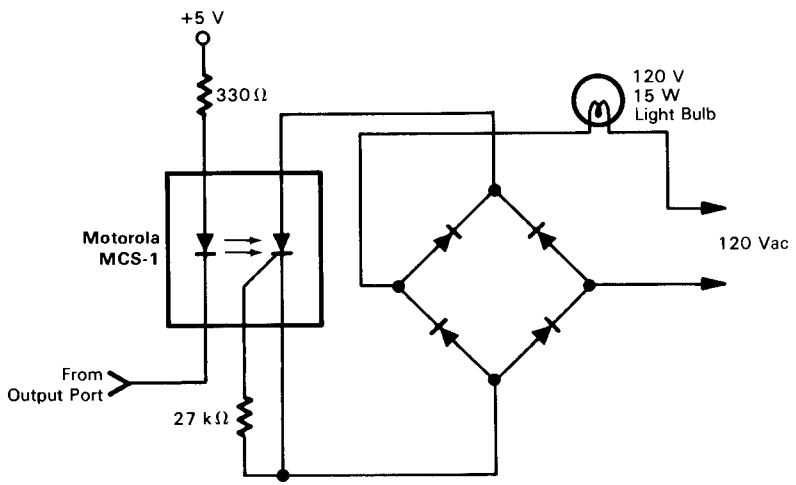


FIGURE 10-7. AC Control Using an Opto-Coupler/SCR

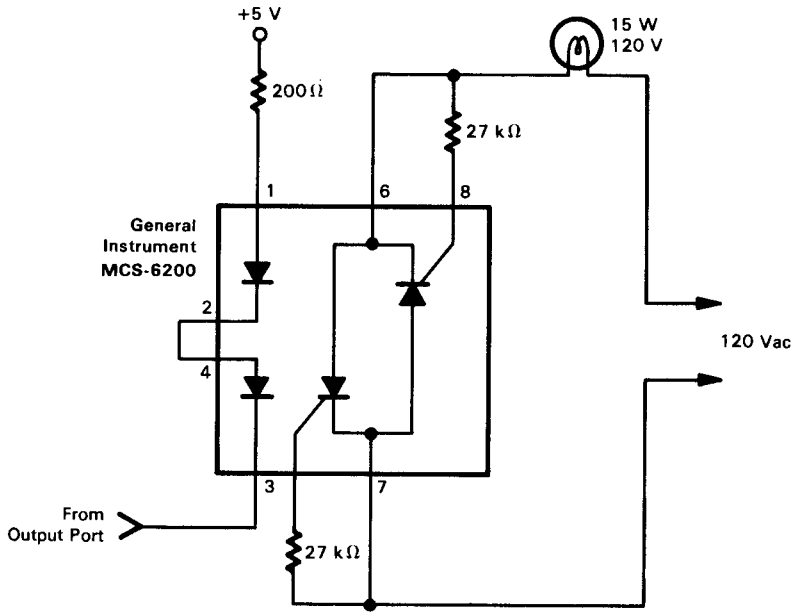


FIGURE 10-8. AC Control Using an Opto-Coupler/Dual SCR

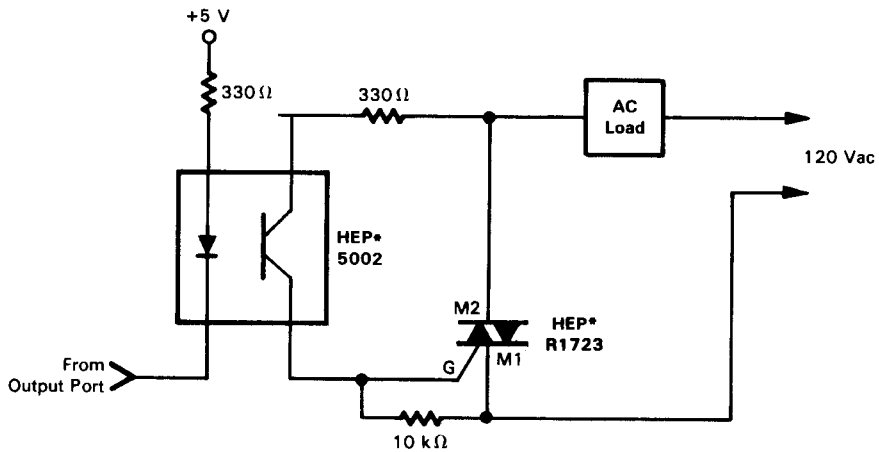


FIGURE 10-9. AC Control Using a Triac

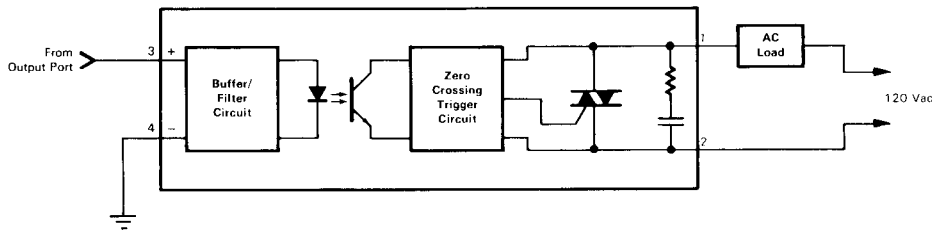


FIGURE 10-10. Monsanto MSR 100/200 Solid State Relay

CONTROL OF MOTORS

DC and AC motors can be controlled using circuits such as those shown in Figures 10-6 through 10-10. The speed of the motor may be controlled by varying the number of on cycles. For example, if the motor is switched on for a very short period of time and off for a long period of time, the average DC current through the motor will be very low, causing a slow speed. If the motor is switched on and off for the same periods, then the average current will be half the maximum. The software driver routine can incorporate a variable time delay to control the average current through the motor. The resultant pulse train to the motor will appear as shown in Figure 10-11.

To cause the DC motor to rotate slowly, the motor is turned on for one time period and off for 14 time periods. The result is that the average current is only 6.7% of maximum, yielding a slow speed. Having equal on and off times yields an average current that is 50% of maximum and results in a medium speed. No off time causes maximum current and motor speed.

The direction of the motor's rotation can be controlled by controlling the direction of current through the motor. The circuitry for controlling a low power DC motor is shown in Figure 10-12. B0 switches the motor on or off, while B1 controls the motor's direction. Note that diodes have been inserted in the base circuit of the motor control transistor. These diodes protect the TTL port logic should the transistor short circuit.

A basic software routine to control the motor's speed and direction is shown in Figure 10-13. A more efficient technique could be accomplished by using a programmable timer and interrupts. This will be discussed in Chapter 13.

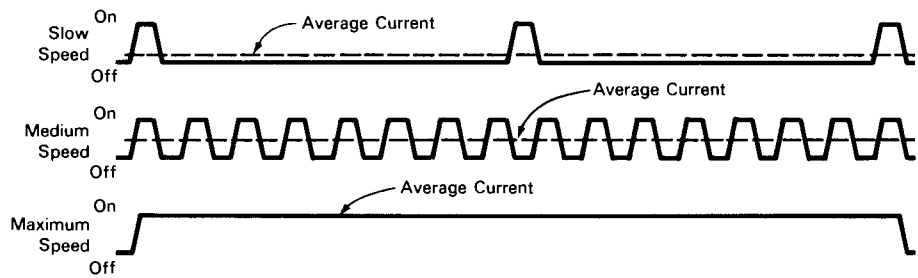


FIGURE 10-11. Varying Current To Motor by Changing On/Off Switching Time

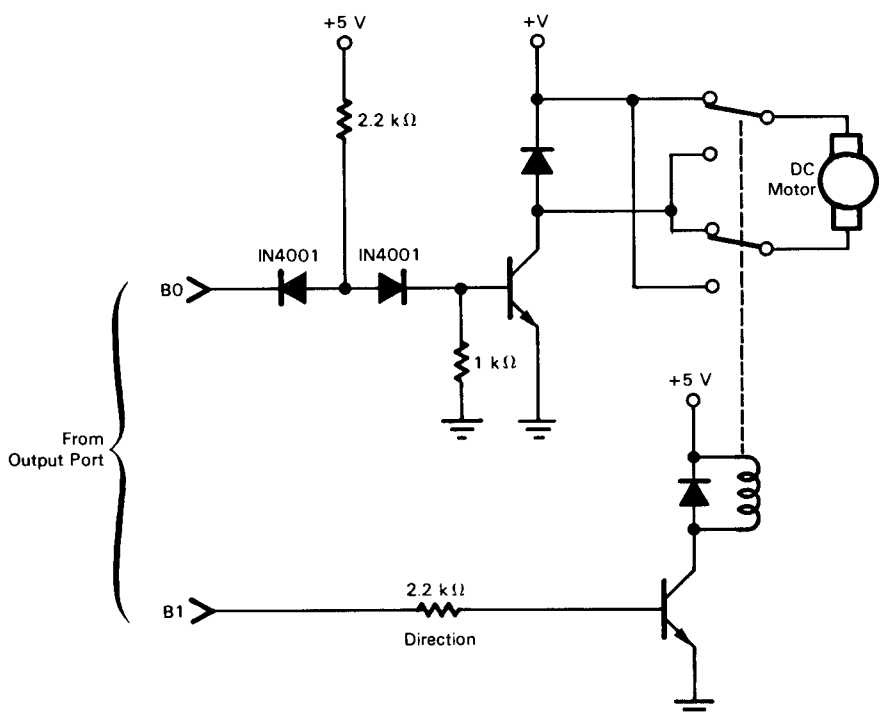


FIGURE 10-12. Simple Low-Power DC Motor Interface Circuit

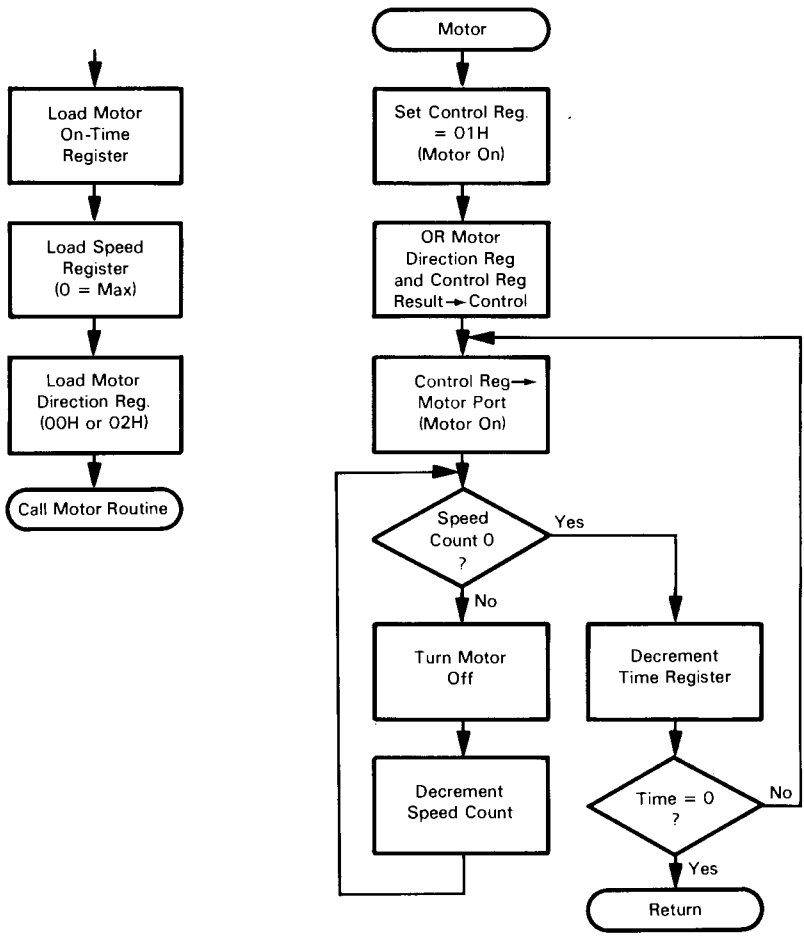


FIGURE 10-13. A Simple Software Routine for Controlling Speed and Direction of DC Motor in Figure 10-12

DRIVING STEPPER MOTORS

Stepper motors are particularly advantageous where precise speed or precise positioning is required. Stepper motors rotate a precise angular displacement each time they are pulsed. Typical step angles are available from 3.75 degrees up to 90 degrees. Further, their operation may be either unidirectional or bidirectional. Operating voltages range from 5 V to 48 V, although 12 V and 24 V are the most common.

The stepper motor contains large numbers of stator pole pairs. These windings must be energized in a certain sequence and hence require a rather complex controller circuit. Fortunately, several stepper motor manufacturers carry IC controllers which permit simple direct interfacing to a microcomputer.

A representative stepper motor and associated IC controller are the North American Phillips model K82944-1 and SAA1027, respectively. The wiring for the devices is shown in Figure 10-14.

A typical stepper motor driver program is shown below. The driver program (STEP) is called as a subroutine. Three values are passed to the STEP subroutine when it is called. They are:

STEP	Number of steps (0 to 255) + 1
DIREC	Direction (04H or 06H)
SPEED	Time between pulses, which establishes speed

A fourth value, PULSE, is established in the subroutine to set the on-time of the pulse.

The controller is initialized by bringing the SET input high while keeping the trigger input high. A low-to-high transition on the trigger input pulses the stepper motor.

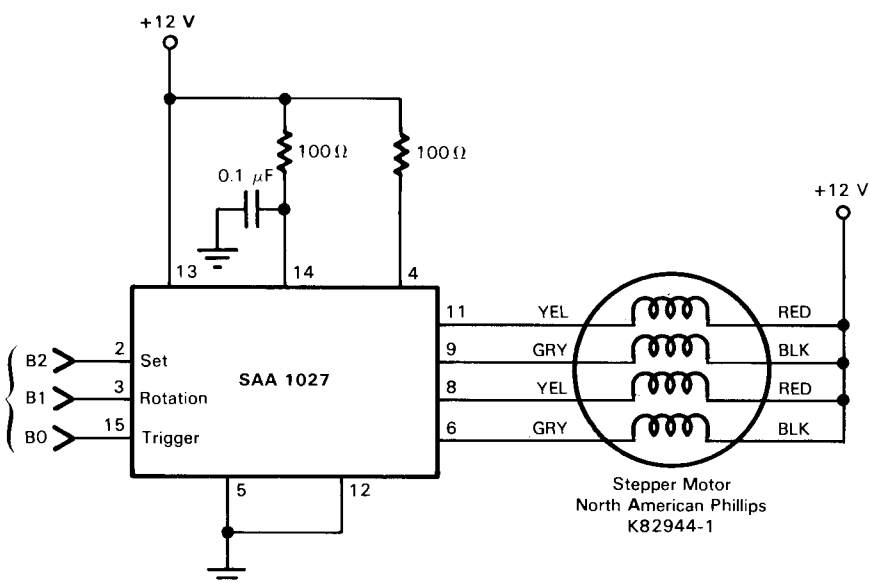


FIGURE 10-14. Stepper Motor Interface Circuit

```
                ;STEPPER MOTOR CONTROLLER PROGRAMMING EXAMPLE
                ;
                ;
0010 =          PORT    EQU  10H        ;OUTPUT PORT ADDRESS
0021 =          STEP    EQU  21H        ;DUMMY STEP NUMBER + 1
0004 =          DIREC1  EQU  04H        ;DIRECTION #1
0006 =          DIREC2  EQU  06H        ;DIRECTION #2
0010 =          SPEED   EQU  10H        ;DUMMY TIME BETWEEN PULSES
1000 =          PULSE   EQU  1000H      ;DUMMY PULSE WIDTH TIME
                ;
                ;
0100            ORG  100H
                ;
0100 0621               MVI  B,STEP     ;SET NUMBER OF STEPS+1
0102 0E04               MVI  C,DIREC1   ;SET DIRECTION
0104 211000             LXI  H,SPEED    ;SET TIME BETWEEN PULSES
0107 CD0002             CALL STEPPER    ;STEP THE MOTOR
                ;
                ;
                ;SUBROUTINE TO DRIVE STEPPER MOTOR
                ;
                ;
0200            ORG  200H
                ;
0200 F5         STEPPER PUSH PSW        ;SAVE REGISTERS
0201 D5                 PUSH D
0202 3E01               MVI  A,01H      ;INITIALIZE CONTROLLER
0204 D310               OUT  PORT
0206 3EFF               MVI  A,0FFH
0208 D310               OUT  PORT
020A 05         LOOP    DCR  B          ;CHECK IF ALL STEPS COMPLETED
020B CA2302             JZ   DONE
020E 79                 MOV  A,C        ;GET DIRECTION ARGUMENT
020F 3C                 INR  A          ;SET TRIGGER BIT HIGH
0210 D310               OUT  PORT
0212 110010             LXI  D,PULSE    ;SET PULSE WIDTH TIME
0215 CD2602             CALL DELAY
0218 3D                 DCR  A          ;SET TRIGGER BIT LOW
0219 D310               OUT  PORT
021B EB                 XCHG            ;GET PULSE OFF TIME
021C CD2602             CALL DELAY
021F EB                 XCHG            ;SAVE PULSE OFF TIME
0220 C30A02             JMP  LOOP       ;STEP AGAIN
0223 D1         DONE    POP  D          ;RESTORE REGISTERS
0224 F1                 POP  PSW
0225 C9                 RET             ;RETURN TO MAIN PROGRAM
                ;
0226 F5         DELAY   PUSH PSW        ;SAVE A REG AND FLAGS
0227 1B         DELY1   DCX  D          ;DECREMENT INPUT ARGUMENT
0228 7A                 MOV  A,D        ;TEST IF DE=0
0229 B3                 ORA  E
022A C22702             JNZ  DELY1      ;IF ARGUMENT NOT 0 KEEP GOING
022D F1                 POP  PSW        ;OTHERWISE RESTORE A & FLAGS
022E C9                 RET             ;AND RETURN
```

GENERATING SOUND

The CPU can be used to control sound generators. Here we will examine a simple on-off sound control circuit. This circuit, shown in Figure 10-15, employs a 556 dual timer (two 555 timers can also be used) with one timer wired as an astable multivibrator and the other as a one-shot multivibrator. If a continuous tone is desired, use the astable circuit only. Whenever the input on pin 4 is high the astable circuit is enabled and the tone is developed.

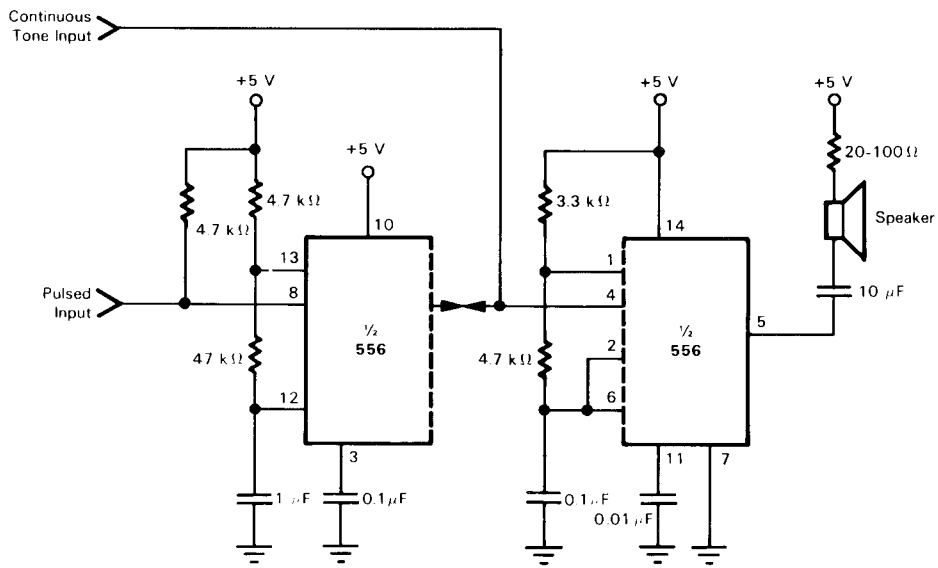


FIGURE 10-15. Tone Generator Interface

The other section of the 556 can be used as a monostable multivibrator to produce a "beep" tone every time the Pulsed Input goes low. Since the one-shot is triggered by a high-to-low transition, it will be necessary to set the Pulsed Input high before attempting to enable the beep circuit again.

In Chapter 12 we will investigate programmed frequency control of a sound generator.

REFERENCES

Bober, Robert E. "Taking The First Step," *BYTE,* February 1978, p. 35.

"Focus on Stepping Motors," *Electronic Design,* October 25, 1977.

Giacomo, Paul. "A Stepping Motor Primer," *BYTE,* February 1979, p. 90.

interfacing to serial ports

11

Serial interfacing is used to transfer data words one bit at a time. It is generally used with microcomputers as a link to peripherals and communications equipment. For example, all data communication via telephone lines is done serially.

Serial data transmission has the advantage of using fewer connecting lines than parallel transmission. Only one pair is required for input and one for output, and in many cases this can be accomplished with one common line, reducing the number of wires to three (in, out, and common). Parallel data communication requires a line for each bit in the data word plus one common line. In other words, two-way parallel I/O typically requires eight inputs, eight outputs, and a common line. So parallel I/O typically uses five times as many lines as serial I/O. On the other hand, parallel I/O, transmitting all bits at one time, versus one bit at a time for serial, is much faster.

Serial I/O requires that the parallel word be converted to or from a serial word. This task can be accomplished with either hardware or software. The hardware approach requires minimal software support and may utilize interrupts. The software approach requires minimal supporting hardware but decreases the operating flexibility of the processor.

A typical serial transmission appears as shown in Figure 11-1. The clock inputs determine the rate at which bits are transmitted and received. The number of bits per second transmitted is usually referred to as the "baud rate." For example, 300 baud means 300 bits per second.

A protocol has been adopted so that the receiving device will know when a serial word starts and ends. The word begins with a start bit (0), continues with the data bits and optional parity bit, and ends with one or two stop bits (1), as

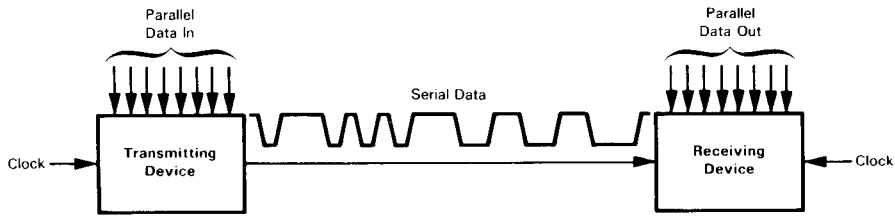


FIGURE 11-1. Serial Data Transmission

shown in Figure 11-2. The 0 logic level is often called a "space" or the spacing condition, and the 1 logic level is called a "mark" or the marking condition. For example, if the data word is 0110101 (7 bits) and even parity is sent, the transmitted word will be 01010110111 (the least significant bit is sent first) and will appear as shown in Figure 11-2. If this data word is an ASCII code (the most widely used serial data code), then the data word 35_{16} represents the number 5. The complete ASCII code is given in Appendix A.

A teletypewriter, commonly abbreviated TTY, is a serial I/O terminal which uses one start bit, seven data bits, one parity bit (generally ignored by the TTY), and two stop bits. Therefore, there are 11 bits in its transmitted and received character code words. It operates at 10 characters per second, so its data rate is 110 baud. Video terminals can generally operate at speeds up to 9,600 baud (960 characters per second) and some operate at speeds as high as 38,400 baud (3,840 characters per second).

Serial I/O in hardware usually involves the use of an LSI controller IC. This device handles many of the operations necessary in serial I/O.

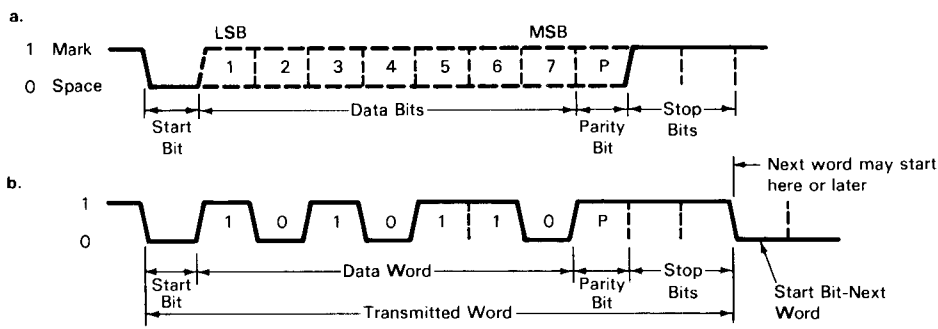


FIGURE 11-2. (a) Serial Transmission Format;
(b) Transmission of ASCII

THE UART

Several ICs have been developed to handle serial-to-parallel (and vice versa) data conversion and provide the proper serial protocol and handshaking signals for the CPU. One such device is called a UART (Universal Asynchronous Receiver/Transmitter). "Asynchronous" transmission means that the serial data is not accompanied to its destination by a separate synchronizing clock signal. Asynchronous transmission uses start and stop bits to delimit characters, and assumes that the bits will be sampled at a fixed rate synchronized to the start bit.

A widely used UART is the General Instrument AY-5-1013A, shown in Figure 11-3. UARTs from a number of manufacturers have standard pinouts. AMI,

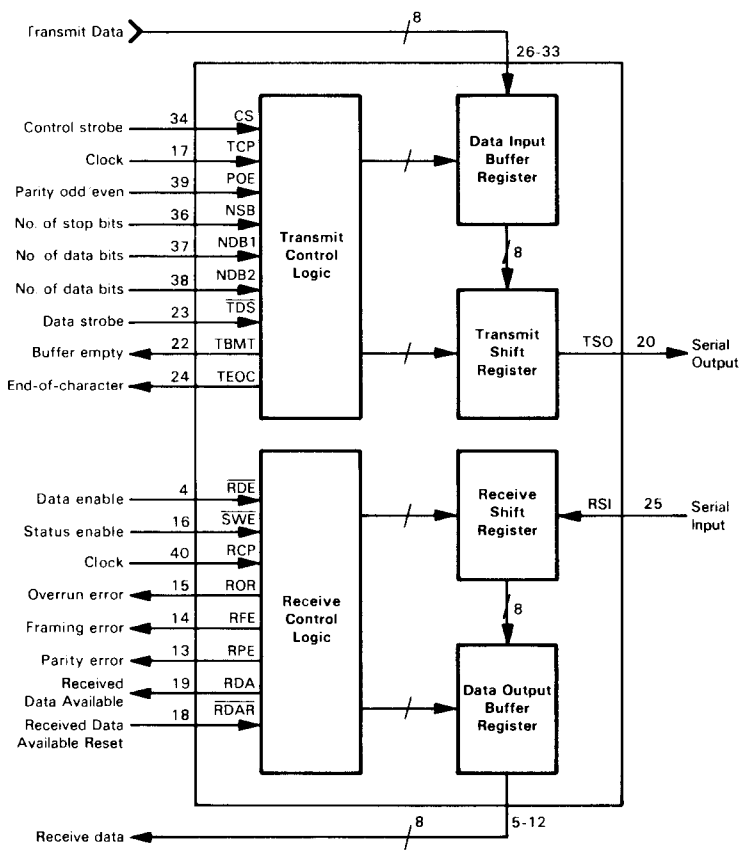


FIGURE 11-3. Block Diagram of the AY-5-1013A UART

Signetics, Standard Microsystems Corporation, Texas Instruments, Western Digital, and others all make UARTs which are pin-compatible with the General Instrument AY-5-1013A. The pin names vary from manufacturer to manufacturer, but the pin functions are generally the same. The AY-5-1013A UART contains a transmitter (parallel-to-serial shift register) and a receiver (serial-to-parallel shift register). Each is clocked separately, and they may be clocked at different rates; usually the same rate is used for both.

The transmitter has a buffer register to latch the parallel input word and control logic to add start, stop, and parity bits. The UART can be configured for the number of data and stop bits and odd/even parity. This is done by latching bits in a control register inside the UART or by tying the appropriate IC control pins high or

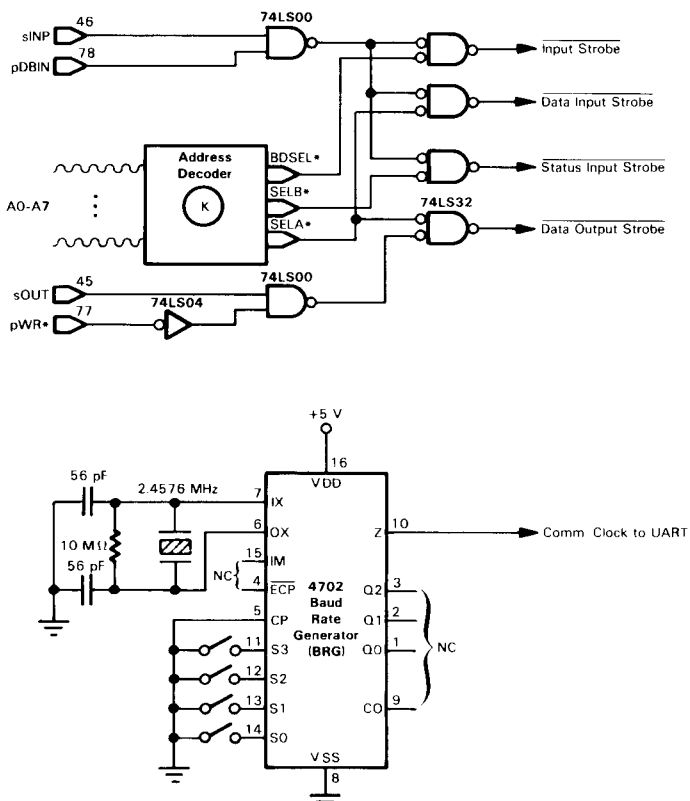


FIGURE 11-4. S-100 Interface for AY-5-1013A UART

(a) I/O Strobes and BRG

low. In addition, there are status signals for the CPU: TBMT indicates that the transmitter buffer is empty (ready to receive a new word) and TEOC (Transmitter End Of Character) indicates that the UART has completed transmitting a word.

The receiver has an output buffer register to hold the received word until the

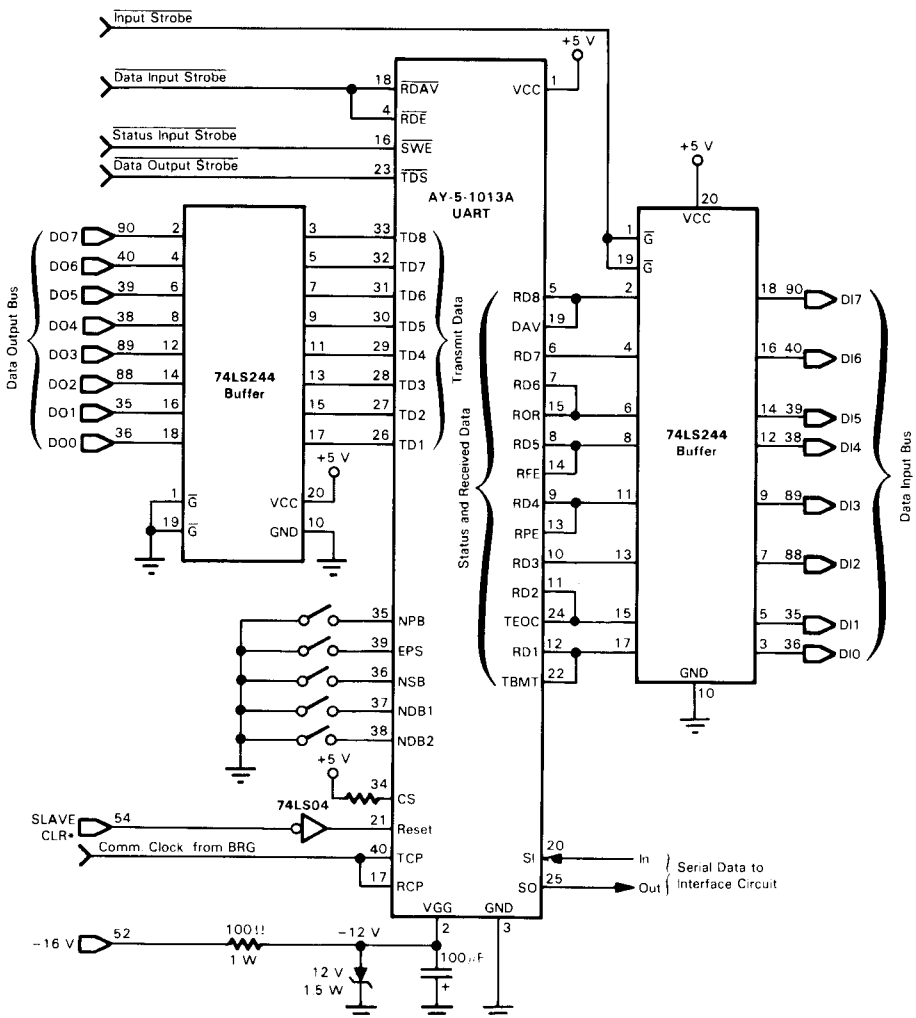


FIGURE 11-4. S-100 Interface for AY-5-1013A UART
(b) UART and Data Bus Buffers

processor is ready to take it. The receiver status signals are DAV (Received Data Available) and RDAV (Received Data Available Reset). In addition, the receiver can detect a framing error (improper stop bits), parity error (single bit error in data), and overrun error (character previously received has not been read by processor). These error outputs are labelled RFE, RPE, and ROR respectively. These error signals are generally ignored in most computer systems where the terminal and processor are in the same local area. In noisy environments (e.g., communication over standard telephone lines) the processor should check these outputs for errors. The error and status signal outputs are tri-state, under control of pin 16 (SWE). The data output signals are also tri-state, controlled by pin 4 (RDE). They may therefore be connected in parallel and controlled by different I/O strobes.

A typical S-100 UART interface is shown in Figure 11-4. The UART looks like parallel ports to the bus while the UART exchanges standard bit-serial I/O signals with the device connected to the serial lines. The transmitting and receiving circuits are clocked at the same rates (pins 17 and 40). A 4702 baud rate generator circuit is used to develop the clock signal, which must be 16 times the baud rate.

The address decoder circuit selects the following port addresses:

Lower port data input
Lower port data output
Upper port status input

The bits in the status word appear on bits 0, 1, 3, 4, 5, and 7, as shown in Figure 11-4.

The following simple software driver subroutines can be used to control the serial interface. The WRCHR routine requires that the data to be serialized be in the Accumulator when the routine is called. The RDCHR routine returns with the data in the Accumulator. If the RET instruction in the RDCHR routine is deleted, the routine will also transmit the character back to the sending device.

```
                ;SUBROUTINES TO SEND AND RECEIVE DATA FROM THE CIRCUIT
                ;SHOWN IN FIGURE 11-4.
                ;RDCHAR RETURNS WITH THE DATA FROM THE UART IN A.
                ;WRCHAR IS CALLED WITH THE DATA TO BE SENT IN A.
                ;BOTH ROUTINES WILL NOT RETURN UNTIL FINISHED.
                ;ALL PORTS ARE RELATIVE TO "BASE".
0010 =      BASE    EQU  10H        ;BASE PORT ADDRESS
0011 =      STATUS  EQU  BASE+1     ;STATUS PORT
0010 =      DATA    EQU  BASE       ;DATA PORT
0080 =      DAV     EQU  80H        ;RECEIVER DATA AVAILABLE BIT
0001 =      TBE     EQU  01H        ;TRANSMITTER BUFFER EMPTY BIT
            ;
0100                ORG  100H
            ;
0100 DB11   RDCHAR  IN   STATUS     ;READ STATUS BYTE
0102 E680           ANI  DAV        ;MASK ALL BUT DAV BIT
0104 CA0001         JZ   RDCHAR     ;IF NOT HIGH, TRY AGAIN
0107 C9             RET             ;AND RETURN
            ;
010A F5     WRCHAR  PUSH PSW        ;SAVE THE DATA TO BE SENT
010B DB11   WR2     IN   STATUS     ;READ STATUS BYTE
```

```
010D E601          ANI   TBE       ;MASK ALL BUT TBE BIT
010F F1            JZ    WR2       ;IF NOT HIGH, TRY AGAIN
0112 F1            POP   PSW       ;GET THE DATA BACK
0113 D310          OUT   DATA      ;AND SEND IT
0115 C9            RET             ;WE'RE DONE
```

The figures that follow show S-100 Bus interfaces for three next generation UARTs that are in common usage today. They are the 8251A, the 8250, and the 2651/2661. The 8251A and the 2651 are also capable of synchronous as well as asynchronous transmission, and they are sometimes called Universal Synchronous/Asynchronous Receiver Transmitters (USARTs). Synchronous transmission differs from asynchronous in that a separate clock is sent along with the data stream; however, this text will not show a synchronous interface. The 8250 and the 2651/2661 contain internal baud rate generators which reduce the amount of external circuitry required as well as putting the baud rate under software control. In all of these devices the "status" byte appears as a register inside the chip, whereas with the AY-5-1013A the status lines had to be buffered and decoded separately. In addition, all of the RS-232 auxiliary control lines are implemented by these three chips.

Figure 11-5 shows the interface for the 8251A. This IC was originally manufactured by Intel, but is now also second-sourced by many other manufacturers. The interface circuitry is very straightforward. A0 is used to select among the internal registers, and sINP and sOUT are combined with the BDSEL* to drive the $\overline{CS}$ input. This combined signal is also used to condition the RD* strobe so that the output buffer turns on only at the proper time. The baud rate clock for this circuit is provided by the 4702, as shown in Figure 11-4. SLAVE CLR* is inverted and used to reset the 8251A, but RESET* can be used if this is more convenient in your system. For more information on how to program the 8251A see the fourth reference at the end of this chapter.

Figure 11-6 shows the 2651 or 2661-3 USART interfaced to the S-100 Bus. The 2651 is manufactured by Signetics and National Semiconductor. The 2661-3, an enhanced version of the 2651, is manufactured by Signetics. The enhancements are all in the synchronous mode, however. Parts of this circuit are very similar to the circuit shown in Figure 11-5, but some oddities in the 2651 bus interface required some special circuitry. The 2651 has an input called $\overline{R}$/W. One might suspect that this, as in so many other LSI peripheral ICs, is the data strobe input. In fact, this input merely tells the chip which direction the data is flowing, and it is not a strobe. It is tied to sOUT, which will be low for reads and high for writes and is stable well in advance of the data strobes. This meets the requirements of the $\overline{R}$/W input.

The $\overline{CE}$ input is the data strobe for the 2651, but it must also be qualified with BDSEL*. In addition, it must pulse low for either a read or a write. The read and write strobes for the S-100 Bus (pDBIN and pWR*) are logically ORed then

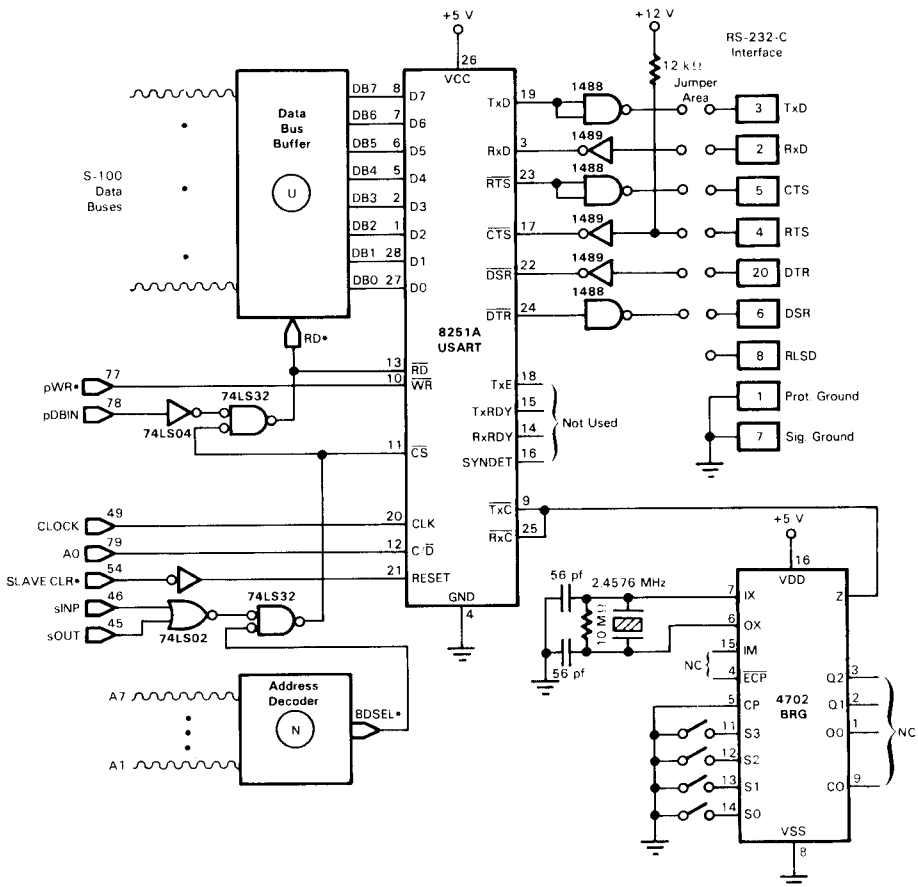


FIGURE 11-5. S-100 Interface for 8251A USART

qualified by BDSEL* to cause a low at $\overline{CE}$ whenever either strobe is active.

The 2651 has its own internal baud rate generator and requires only a 5.068 MHz signal to be applied to the BRCLK input. A classic crystal oscillator is formed with two inverters and a 5.068 MHz crystal. The two 12 kΩ resistors pulling up the CTS and RLSD lines are necessary because if these lines are left floating by the RS-232 device, the 2651 will not function.

For more information on how to program the 2651 or 2661-3, see the last reference at the end of this chapter.

Figure 11-7 shows the 8250 interfaced to the S-100 Bus. The 8250 is produced by National Semiconductor and Western Digital. The 8250 requires a minimum of circuitry for the S-100 Bus interface — little more than an address decoder. The baud rate is generated internally by a 16-bit divide-down counter. The master frequency for this counter is the 2 MHz signal on pin 49 (CLOCK). This requires that a 16-bit divisor value be written into registers inside the 8250. Common baud rates and the associated divisors are shown in Table 11-1.

The 8250 contains internal circuitry which makes its operation with interrupts very sophisticated. The INTRPT output is shown buffered and inverted by a section of a 7406 and may be connected to any of the S-100 vectored interrupt pins or directly to INT* if this circuit is used in a minimal system.

For more information on how to program the 8250 see the third reference at the end of this chapter.

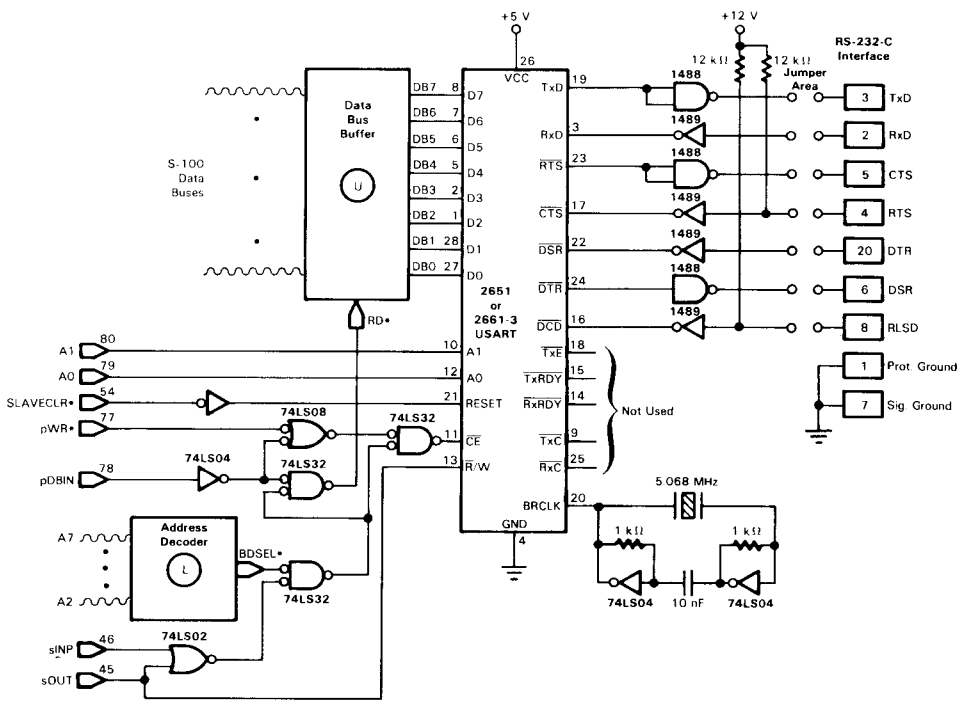


FIGURE 11-6. S-100 Interface for 2651/2661 USART

PROGRAMMABLE BAUD RATE CLOCKS

In some computer systems it may be desirable to change the baud rate under software control, as in a system where different terminals operating at different baud rates are connected to the computer. In such a system it is necessary for the computer to first determine the terminal's baud rate and then adjust the UART to the same baud rate. This means that the baud rate clock must be software programmable, rather than fixed as it is in Figures 11-4 and 11-5.

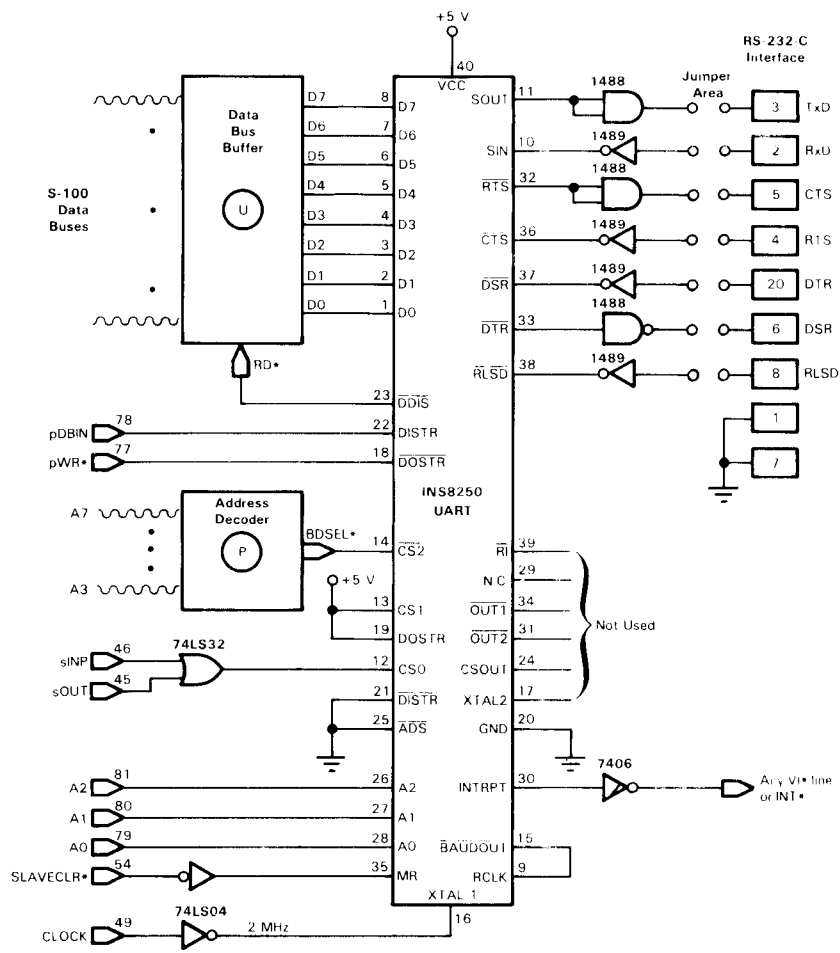


FIGURE 11-7. S-100 Interface for INS8250 UART

TABLE 11-1. Baud Rates and Divisors for INS8250 Using 2 MHz Clock

Baud Rate	Divisor		
	Decimal	High Byte (Hexadecimal)	Low Byte (Hexadecimal)
110	1136	04	70
134.5	929	03	A1
300	416	01	A0
600	208	00	D0
1200	104	00	68
2400	52	00	34
3600	34	00	22
4800	26	00	1A
9600	13	00	0D
19,200	6	00	06

Formula: Divisor = Freq ÷ (Baud Rate × 16)
Example: 1136 = 2,000,000 ÷ (110 × 16)

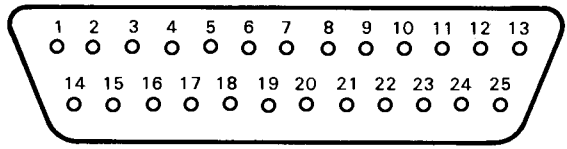
The 4702 baud rate generator circuit shown in Figures 11-4 and 11-5 may be made software programmable by replacing the switches with four bits from any convenient latched parallel output port. Either one that exists in the system or one of those discussed in Chapter 10 may be added to the circuit card directly.

The 2651 and 8250 circuits presented in Figures 11-6 and 11-7 contain built-in software-programmable baud rate generators.

PERIPHERAL SERIAL INTERFACES

The serial I/O pins of UARTs are usually TTL logic voltage levels. This generally limits line lengths to a few meters under ideal conditions. Hence, communication interface links have been developed which overcome these limitations.

The most popular serial interface is the RS-232-C interface used between modems and terminals, printers, computers, and the like. The current loop interface, originally developed for teletypewriters, is also quite popular. Serial interfaces that do not conform to either the RS-232-C standard or the current loop convention are sometimes used. These offer the advantage of lower cost and simplicity. The RS-232-C and current loop approaches will be discussed in the following sections.



Female Connector as Viewed from Wiring Side

Pin Number	EIA Name	RS-232-C Common Mnemonic	Description
1	AA		Protective ground
2	BA	TxD	Data transmitted from terminal
3	BB	RxD	Data received from modem
4	CA	RTS	Request to send
5	CB	CTS	Clear to send
6	CC	DSR	Data set ready
7	AB		Signal ground
8	CF	DCD	Carrier detector
9	*		Reserved for Data Set Testing
10	*		Reserved for Data Set Testing
11	*		Unassigned
12	SCF		Secondary carrier detector
13	SCB		Secondary clear to send
14	SBA		Secondary transmitted data
15	DB		Transmitted bit clock, from DCE
16	SBB		Secondary received data
17	DD		Received bit clock
18	*		Unassigned
19	SCA		Secondary request to send
20	CD	DTR	Data terminal ready
21	CG		Signal quality detector
22	CE		Ring indicator (used by auto answer equipment)
23	CH/CI		Data signal rate selector
24	DA		Transmitted bit clock, from DTE
25	*		Unassigned

*Undefined

FIGURE 11-8. RS-232-C Connector and Pin Definitions

The RS-232-C Standard

The RS-232-C serial interface standard has been defined by the Electronic Industries Association (EIA). "RS" stands for "Recommended Standard" and the "C" indicates that this is the third version of the standard.

The standard utilizes a 25-pin connector, and manufacturers have generally agreed on the DB-25 type connector shown in Figure 11-8. The modem (the data communications equipment, or DCE) should have a female connector, while the terminal, printer, computer, etc. (the data terminal equipment, or DTE) should have a male connector. However, many manufacturers do not adhere to this requirement. The manufacturers generally put female connectors on all equipment and assume the connecting cables have male connectors on both ends.

Twenty signals are defined by EIA. However, generally only nine lines are used between a terminal and a modem, and often as few as three lines are used. RS-232-C does not specify how many of the signals are to be used. Typical cables for terminal-modem and terminal-computer connections are shown in Figure 11-9.

Pins 4, 5, 6, 8, and 20 are handshaking signals used between a modem and a terminal. Pins 15, 17, and 24 are used with high-speed modems (1200 and 2400 baud). Pin 22 indicates that the modem has detected a ring signal on the telephone line. It is used by equipment that automatically answers incoming calls.

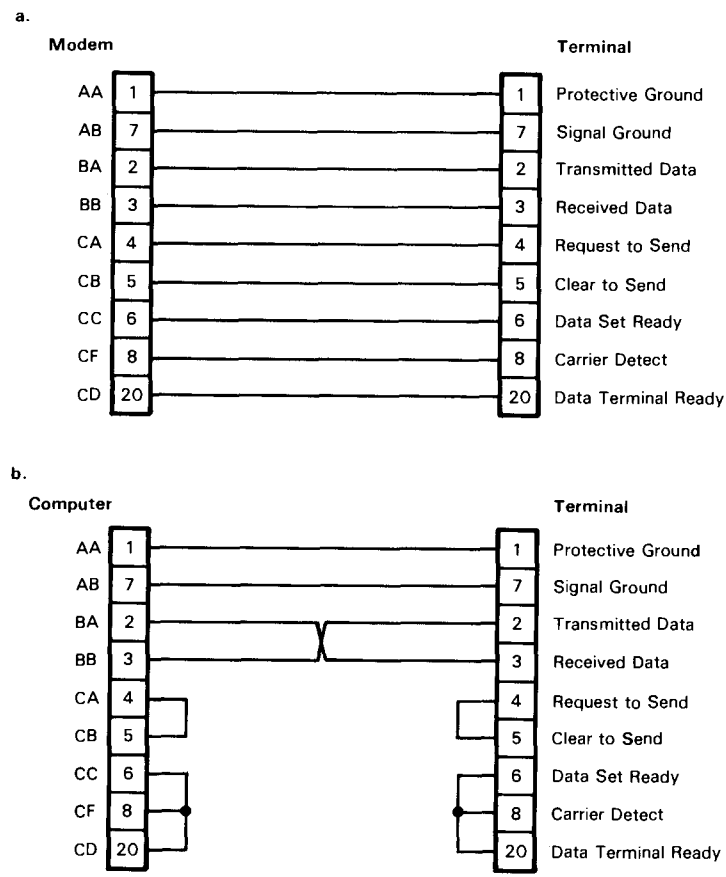


FIGURE 11-9. RS-232 Cable Wiring; (a) Modem-Terminal, (b) Computer-Terminal

The computer's RS-232-C interfaces should contain a jumper area so that the DB-25 connector may be configured as a DCE or DTE device. This allows maximum flexibility in connecting peripherals to the computer. If this is impractical, the most common usage of RS-232 serial ports is connecting terminals and printers, so the computer should be wired as the DCE device.

TABLE 11-2. RS-232-C Signal Level Conventions

Notation	Interchange Voltage	
	Negative	Positive
Signal Condition	Marking	Spacing
Binary State (data lines)	1	0
Function (control lines)	OFF	ON

The standard defines the binary state 1 as a voltage level between −3 V and −15 V. The binary state 0 can range from +3 V to +15 V. Hence the voltage swing can be as little as 6 V or as great as 30 V. The greater the voltage swing the greater the signal-to-noise ratio and hence the greater the noise immunity. The voltage levels do not have to be symmetrical. Note that the signal condition for the data signals is "opposite" that for control signals. For example, a more negative voltage represents a binary 1 state on a data line, but it represents an OFF condition on a control line. These signal levels are summarized in Table 11-2.

An interface circuit is required between the RS-232-C voltage levels and the TTL levels typically used in logic circuits. Fortunately there are ICs specifically made for this purpose. The most widely used are the 1488 (75188) quad line driver and 1489 (75189) quad line receiver. The 1488 requires an external capacitance of 330 pF on each output to meet the RS-232-C slew rate specification of 30 V/μs. If the cable and receiver do not have a capacitance greater than 330 pF, an external capacitor should be connected from each output to ground.

The 1489 quad receiver has external threshold voltage control via pins 2, 5, 9, and 12. If these terminals are not used, any input less than +0.75 V switches the receiver output high, and any input greater than +1.5 V switches the receiver output low. If a 5 kΩ resistor is connected from the threshold input to +5 V, then the input thresholds are shifted about 3 V more negative. The resistor is not required except in high noise situations. Additional noise immunity is afforded by putting a capacitor from the threshold pin to ground. For example, a 500 pF capacitor will cause the receiver to ignore pulses up to 6 V whose durations are less than 800 ns. A typical computer/RS-232-C interface circuit is shown in Figure 11-10. Alternative interface circuits using discrete components are generally undesirable due to their nonstandard performance and decreased reliability.

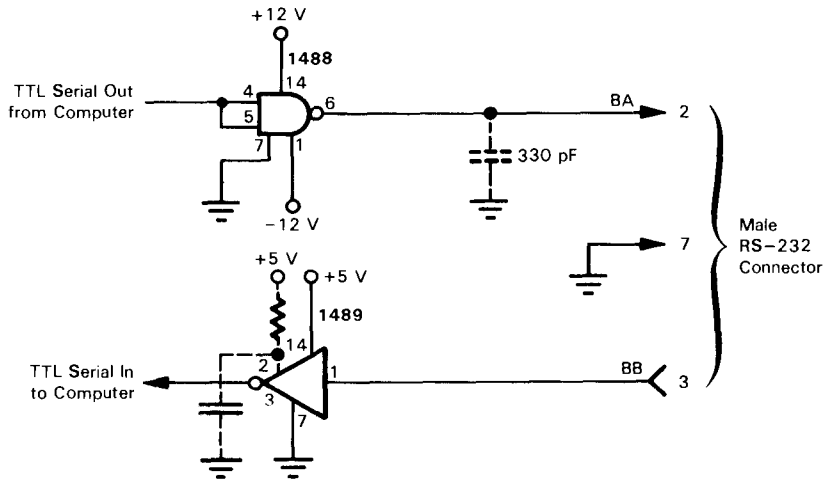


FIGURE 11-10. RS-232-C Serial Interface Circuit

The RS-232-C is currently the most widely used serial interface circuit. Its predecessor, still used with older terminals, is RS-232-B, which is the same as the C version with the exception of voltage levels. It defines a 1 logic level as −5 V to −25 V, while the 0 logic level is the same.

RS-232-C is limited to interconnection lengths of less than 15 meters (50 feet) and data rates less than 20 kilobits per second. In an attempt to update this standard (and provide compatibility during the transition) the Electronic Industries Association has introduced three new standards: RS-449, RS-422-A, and RS-423-A. RS-422-A and RS-423-A cover the electrical characteristics and RS-449 the functional and mechanical characteristics of the interface. The basic interchange functions of RS-232-C have been incorporated in RS-449. In addition to other changes, the newer standards extend the data rate to 2 megabits per second, add ten circuit functions, and delete three. (See the fourth reference at the end of this chapter for a discussion of these standards.)

The Current Loop Interface

The current loop type communication link was developed for teletypewriter (TTY) terminals many years ago and has been used on many other devices as well. It continues to be popular, although it is not as widespread as the RS-232-C link.

The receive line of a TTY is normally connected to the TTY's selector magnet through a resistor. It typically draws 20 mA through a 330 Ω current-limiting

resistor. Older TTYs worked on 60 mA through a 160 Ω resistor.

The TTY provides a series of contact openings on the send line when a key is pressed on the TTY keyboard or the tape reader is operated. When no key is pressed the switch contact (commutator) is closed. A 20 mA loop circuit is also utilized for the send circuit.

The computer interface must usually be able to supply 20 mA on the "send" and "receive" interfaces. The TTY should be checked to determine whether it works on 20 mA or 60 mA. Most current loop interfaces provide only 20 mA capability, which an older 60 mA TTY will not receive. Furthermore, the send circuit of a 60 mA TTY may damage the computer's 20 mA receive circuit.

A typical 20 mA interface circuit is shown in Figure 11-11. The two diodes in the receive circuit afford protection to the phototransistor from current spikes induced by the selector magnet. This interface would have to be modified if the TTY provided the current source for the loops.

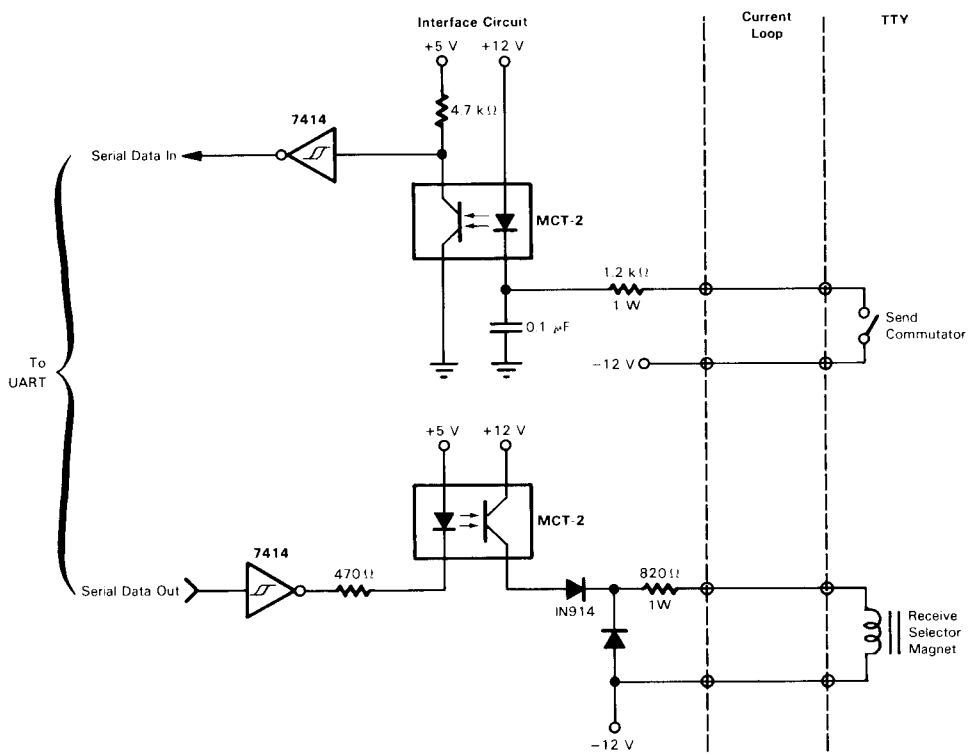


FIGURE 11-11. 20 mA Current Loop Interface Circuit

REFERENCES

Electronic Industries Association. "Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Data Interchange." Electronic Industries Association, 2001 Eye Street N.W., Washington, D.C. 20006.

IEEE. "Peripheral Interface Standards for Microprocessors," *Proceedings of the IEEE,* Vol. 64, No. 6, June 1976.

National Semiconductor. "8250 Data Sheet." National Semiconductor, 2900 Semiconductor Drive, Santa Clara, CA 95051.

Osborne, Adam, and Kane, Jerry. *An Introduction to Microcomputers,* Volumes 2 and 3, Berkeley: Osborne/McGraw-Hill, 1978.

Signetics. "2651 Data Sheet." Signetics Corporation, 811 East Arques Avenue, Sunnyvale, CA 94086.

digital-to-analog and analog-to-digital conversion

12

Digital-to-Analog Converters (DACs) and Analog-to-Digital Converters (ADCs) are often used as outputs from and inputs to microcomputer systems. The basics of DAC and ADC have been covered in some depth in a number of publications (see the references at the end of this chapter). Consult these texts for the theory of operation of these circuits. Our purpose here is to examine how these circuits are interfaced to the S-100 Bus and to look at some popular DAC and ADC applications.

We will examine digital-to-analog conversion first because it is a simpler operation, and many analog-to-digital conversion circuits employ a DAC as part of the ADC circuit.

DIGITAL-TO-ANALOG CONVERSION

In most cases, DACs employ some form of binary weighted current or voltage summing that is controlled by a digital word presented to the summing network. This word, in the case of a computer-DAC interface, is presented via a parallel port, as shown in Figure 12-1. Outputting a word to the DAC port changes the analog output voltage or current.

The binary summing circuit is of either the weighted or the R-2R resistive network type. A typical weighted resistor type DAC circuit is shown in Figure 12-2. The weighted resistors are connected to the 8-bit output of a parallel port via buffers. The effective resistance for bit 7 is 11.75 kΩ, formed by the parallel combination of four 47 kΩ resistors, while the bit 6 resistance is two parallel resistors.

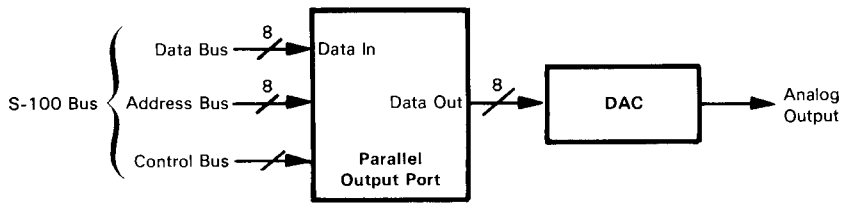


FIGURE 12-1. DAC Interface Block Diagram

The analog output voltage will be:

$$Vout = 5\left(\frac{n}{255}\right)$$

n is the decimal equivalent of a binary number between 0 and 255. Therefore, a value of 10001011 (decimal 139) yields an output of +2.7 V. 255 discrete voltage steps from 0 to +5 V can be developed. The active element serving as a summing point is the LM3900 transconductance operational amplifier. Although a low cost op-amp such as the LM3900 or the 741 will suffice, it is better to use one of the higher cost premium grade IC operational amplifiers if very accurate analog outputs are required.

Although the foregoing circuit will provide a fairly accurate analog output voltage, it lacks the necessary circuit refinements required for a precision analog output. Although these refinements can be made, today it is more economical to utilize one of a number of readily available monolithic DACs. A widely available device is the 1408L-8 shown in Figures 12-3 and 12-4. The 1408 is a multiplying DAC, so the reference voltage may be changed to vary the range of the analog output voltage. The 1408 employs an R-2R summing network and produces an output current which ranges from 0-2 mA. In Figure 12-3 the LM301A operational amplifier functions as a current-to-voltage converter to provide a 0-5 V analog output voltage.

An improved DAC circuit is shown in Figure 12-4. In addition to the LM301A op-amp, it utilizes a special reference regulator IC (MC1403U) which provides a highly regulated positive 2.5 V reference for the DAC. Further, the 741 op-amp is used to develop a negative reference voltage for the current-to-voltage converter circuit, thus improving the accuracy of the circuit.

A very simple DAC circuit can be built using an IC DAC with voltage output (the 1408 produces a current output). One such DAC is the Ferranti Electric ZN425E, shown in Figure 12-5. It also employs an R-2R summing network, and further, it has an internal voltage reference (+2.56 V) and an 8-bit counter. The counter can be used in a counter type ADC circuit (discussed later). The biggest advantage in using this device is that the output is already converted to voltage,

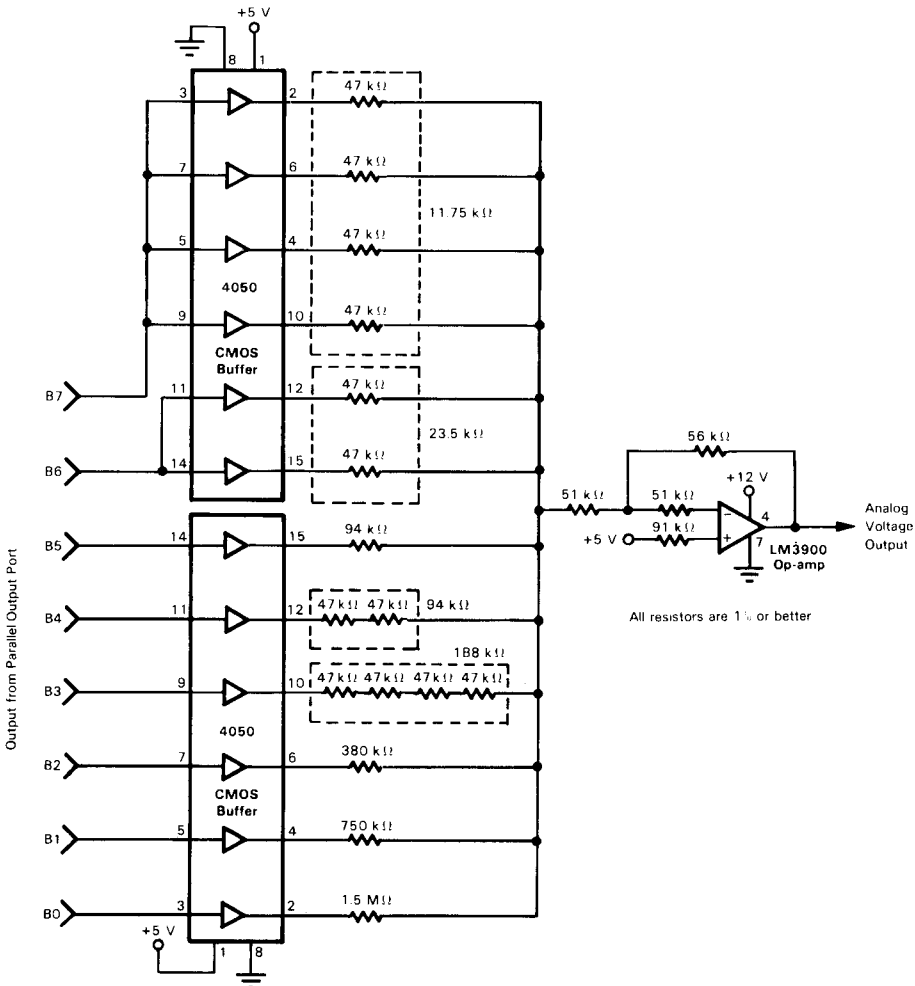


FIGURE 12-2. A Binary-Weighted DAC Circuit

thus eliminating the need for the operational amplifier in applications where a voltage output is necessary. This assumes, of course, that the device can provide the full-scale voltage your application requires (for the ZN425E full-scale output is 2.55 V). Otherwise, a different full-scale output voltage can be achieved by adding an op-amp at the output, or by applying a different reference voltage (0 to +3 V).

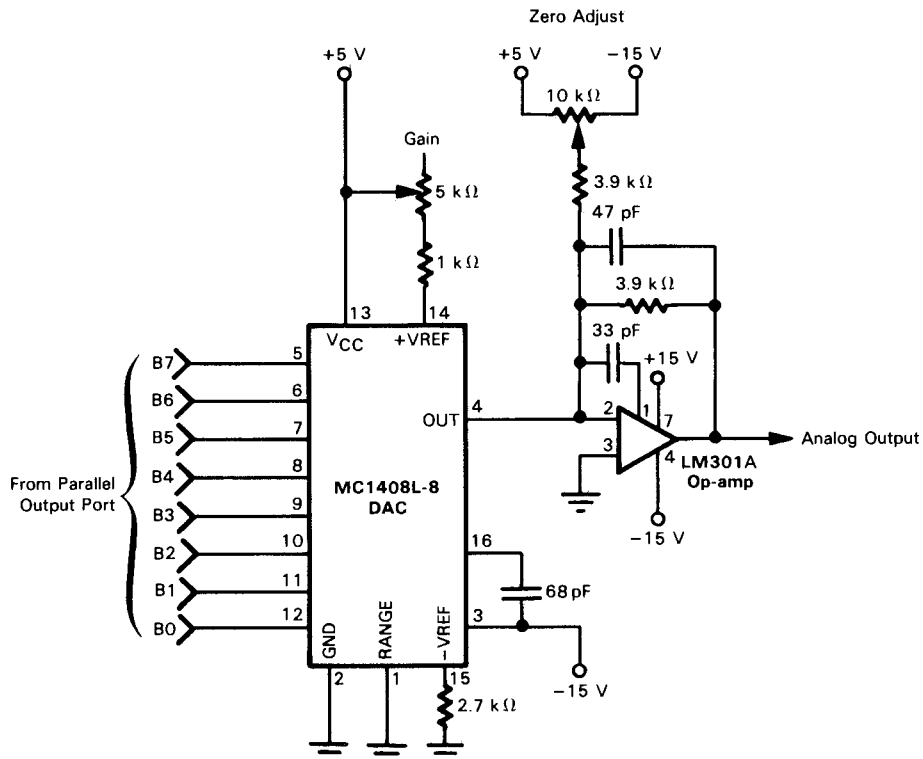


FIGURE 12-3. A Simple 1408 DAC Circuit

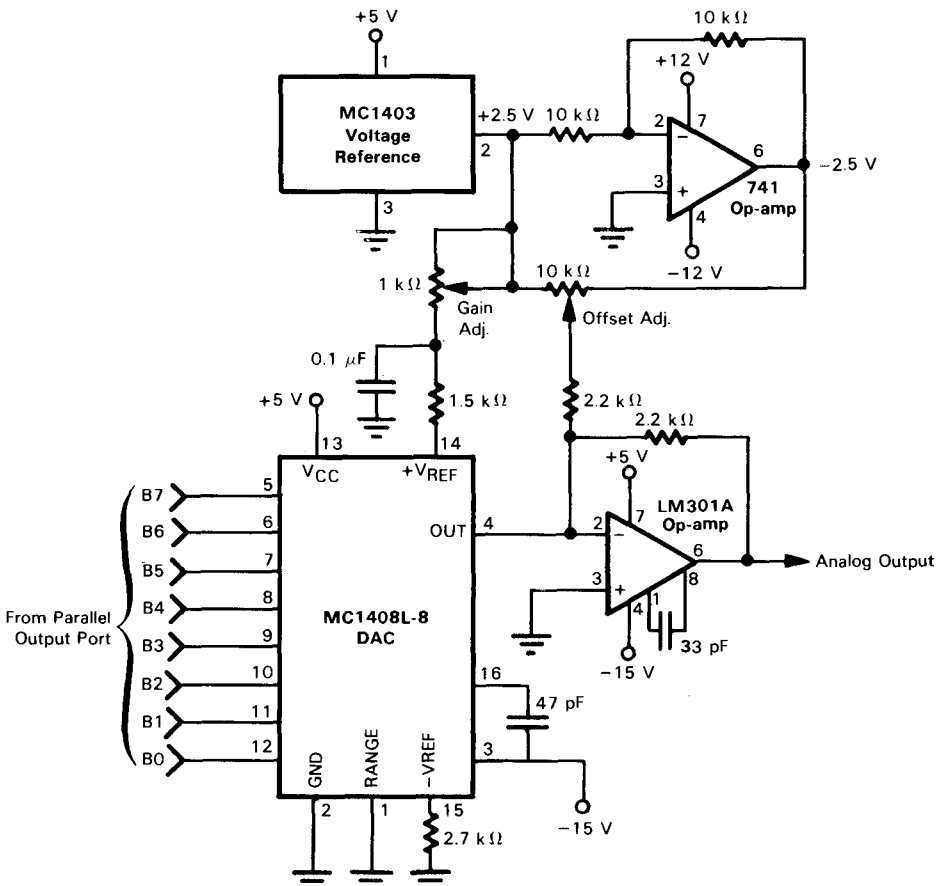


FIGURE 12-4. An Improved 1408 DAC Circuit

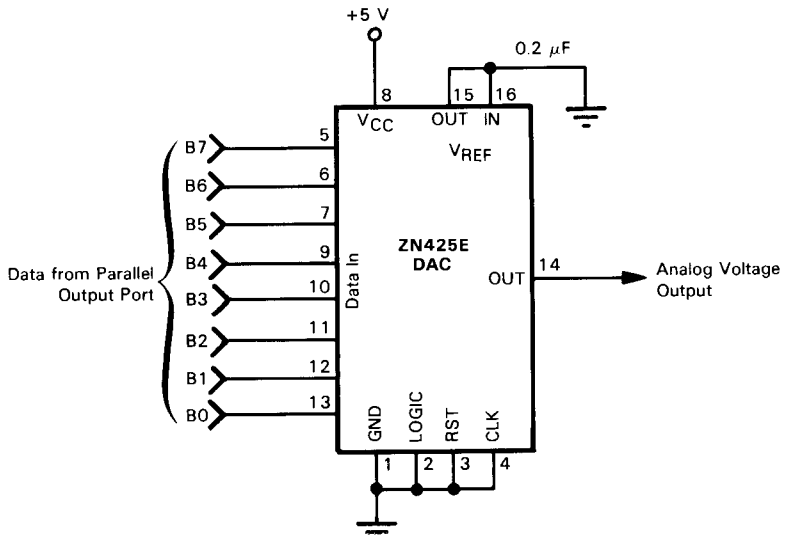


FIGURE 12-5. Voltage Output DAC IC Circuit

Here is a demonstration program which uses the DAC as a programmable sine wave signal generator. The program outputs 64 successive values to the DAC, which converts the binary values to analog voltage values. Each value is stored in a table in memory. The output may be fed to an oscilloscope or audio amplifier. The frequency may be varied by introducing a suitable delay between each value call. A different wave form may be generated by simply changing the values in the table.

```
                    ;DAC SINE WAVE PROGRAM
                    ;
0010 =        DAC       EQU  10H        ;DAC PORT ADDRESS
0200 =        TABLE     EQU  200H       ;START OF TABLE VALUES
0040 =        COUNT     EQU  64         ;NUMBER OF VALUES IN TABLE
                    ;
0100                    ORG  100H
                    ;
0100 210002   WAVE      LXI  H,TABLE    ;SET TABLE POINTER
0103 0640               MVI  B,COUNT    ;SET FOR 64 POINTS
0105 7E       LOOP      MOV  A,M        ;MOVE TABLE VALUE TO DAC
0106 D310               OUT  DAC
0108 23                 INX  H          ;INCREMENT TABLE ADDRESS
0109 05                 DCR  B          ;DECREMENT COUNTER
010A C20501             JNZ  LOOP       ;IF 64 POINTS NOT DONE GET
                                        ;NEXT TABLE VALUE
010D C30001             JMP  WAVE       ;IF SINE WAVE COMPLETE DO
                                        ;AGAIN
                    ;
                    ;LOOK-UP TABLE OF SINE WAVE VALUES
```

```
                        ;
0200                    ORG   TABLE
                        ;
0200 808C98A5           DB    128,140,152,165
0204 B0B7C7D1           DB    176,183,199,209
0208 DAE1EAF0           DB    218,225,234,240
020C F6FAFDFF           DB    246,250,253,255
0210 FFFFFDFA           DB    255,255,253,250
0214 F6F0EAE1           DB    246,240,234,225
0218 DAD1C7BD           DB    218,209,199,189
021C B0A5988C           DB    176,165,152,140
0220 8073675A           DB    128,115,103,90
0224 4F43382E           DB    79,67,56,46
0228 251D150F           DB    37,29,21,15
022C 09050200           DB    9,5,2,0
0230 00000205           DB    0,0,2,5
0234 090F151D           DB    9,15,21,29
0238 252E3843           DB    37,46,56,67
023C 4F5A6773           DB    79,90,103,115
```

Interfacing to DACs Larger than Eight Bits

Eight-bit DACs provide resolution of one part in 256. However, some applications require better resolution, and 10-, 12-, and 16-bit DACs are available for these applications. A 10-bit DAC will provide a resolution of one part in 1024.

A 10-, 12-, or 16-bit DAC is driven by providing the data word in two bytes. For example, a 10-bit DAC can use one byte plus two bits of a second byte for the full 10-bit data word. One possible interface for a 10-bit DAC is shown in Figure 12-6. It uses the Analog Devices AD561 10-bit DAC. In order not to cause glitches in the DAC output, the 10-bit data word is built up in a buffer register composed of IC1 and IC2. After both the upper and lower parts of the word are in the buffer, the ten bits are transferred, as one word, to the register driving the DAC (IC3 and IC4). This technique is referred to as "double buffering."

Notice that the lower bits are loaded with an output to Port 2, the upper bits are loaded with an output to Port 1 and the ten bits are transferred to the DAC with an output to Port 3. The following is a software driver routine. The data to be output is stored in a table with data in successive memory locations.

```
                ;10-BIT DAC PROGRAM (FIGURE 12-6)
                ;
0001 =          HPORT   EQU  1          ;DAC HIGH BYTE PORT
0002 =          LPORT   EQU  2          ;DAC LOW BYTE PORT
0003 =          TPORT   EQU  3          ;TRANSFER PORT
0200 =          TABLE   EQU  200H       ;TABLE STARTING ADDRESS
0040 =          COUNT   EQU  64         ;NUMBER OF VALUES IN TABLE
                ;
0100                    ORG  100H
                ;
0100 210002     WAVE    LXI  H,TABLE    ;SET TABLE POINTER
0103 0640               MVI  B,COUNT    ;SET TABLE COUNT
0105 7E         LOOP    MOV  A,M        ;LOAD LOWER BITS
0106 D302               OUT  LPORT
0108 23                 INX  H          ;LOAD UPPER BITS
0109 D301               OUT  HPORT
010B D303               OUT  TPORT      ;TRANSFER 10 BITS TO DAC
010D 05                 DCR  B          ;DECREMENT COUNT
```

```
010E C26900          JNZ   LOOP       ;64 POINTS NOT DONE GET NEXT
                                      ;POINT
0111 C36400          JMP   WAVE       ;64 POINTS DONE DO NEXT WAVE
                ;
                ;PLACE TABLE HERE
                ;THERE SHOULD BE 64 UPPER BYTES AND 64 LOWER BYTES
                ;
0200                 ORG   TABLE
                ;
0200 00              DB    0          ;8 LOWER BITS
0201 00              DB    0          ;8 UPPER BITS
  .                  .     .             .
  .                  .     .             .
  .                  .     .             .
```

Note that DACs with internal double-buffer registers are available (e.g., the Analog Devices AD7522 10-bit DAC). If such a device is used, the interface is simplified and the number of components required is reduced.

The interface shown in Figure 12-6 can be simplified by eliminating IC2 and feeding the lower eight bits directly to IC3. The software driver routine must first load the two high bits into IC1A (output to Port 1). An output to Port 3 will then load the low byte into IC3 and transfer the upper two bits from IC1A to IC1B, presenting ten bits to the DAC for conversion. This technique eliminates one instruction (output to Port 2). However, it also eliminates the double buffering.

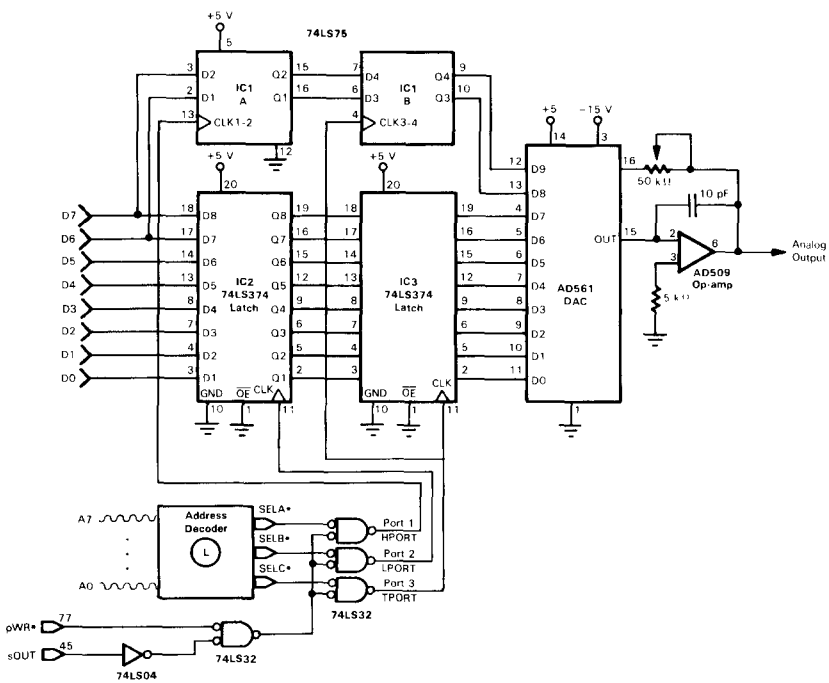


FIGURE 12-6. 10-Bit DAC Interface

A Programmable Signal Generator

The prior example of signal synthesis suffers from two faults: the maximum frequency is limited by the processor speed, and it ties up the processor completely in executing the synthesis routine. A better approach, if such repetitive waveforms are to be generated, is to use an oscillator circuit controlled by a DAC circuit. The DAC analog output is used to control the frequency of the oscillator. Several IC voltage controlled oscillators are available for this purpose. They can be driven from the DACs shown in Figures 12-3, 12-4, and 12-5. Two examples are shown in Figure 12-7.

The Intersil 8038 can provide three simultaneous outputs (sine, square, and triangular waves) over a frequency range of 0.001 Hz to 1 MHz. The frequency is varied by varying the voltage on pin 8 from +5 to +15 V. The Motorola MC4024 is a dual voltage controlled square wave oscillator that operates directly from TTL voltage levels and has a maximum operating frequency of 25 MHz.

ANALOG-TO-DIGITAL CONVERSION

Converting an analog input into a digital word can be accomplished by a variety of ADC techniques. The two most popular are the "ramp" and "successive approximation" methods. Both employ a DAC and a comparator as shown in Figure 12-8. The DAC input is varied while the DAC output is compared to the analog input until a match, or approximate match, is made. The data word input to the DAC then represents the amplitude of the analog input.

The Ramp Type ADC

A ramp type ADC compares the analog input to a ramp function from the DAC. One circuit for doing this is shown in Figure 12-9. It uses the DAC, as shown in Figure 12-3 or 12-4, to develop the ramp function. The DAC is fed to the LM311 comparator circuit, where it is compared to the analog input. The output of the comparator is fed to bit 0 of an input port. The comparator output equals zero until the DAC output is greater than the analog input.

The software routine develops a ramp function from the DAC until the match occurs. It starts by setting the DAC for zero voltage output and increments the voltage until the DC voltage output of the DAC exceeds the analog input voltage (COMP bit 0 = 1). The digital equivalent of the analog voltage is now present at the input to the DAC and in Register B.

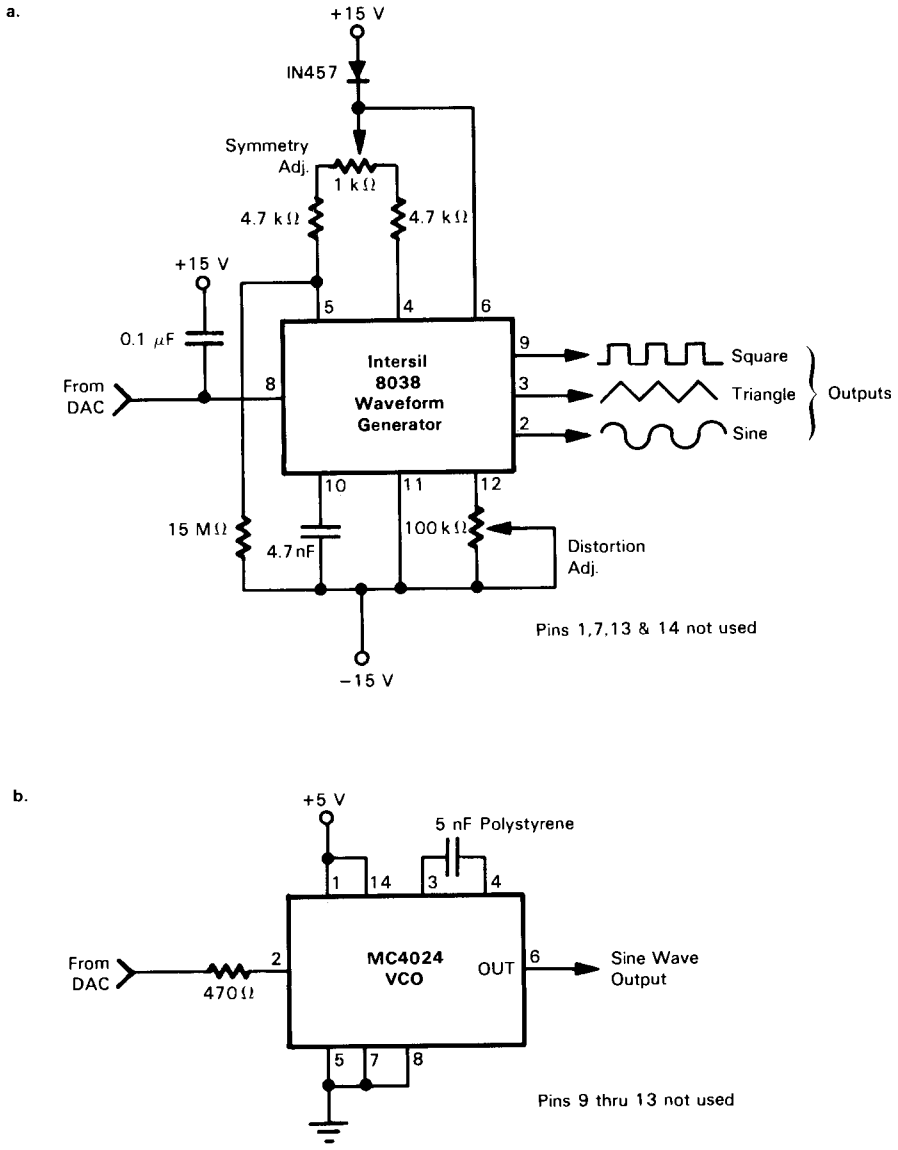


FIGURE 12-7. Two Different Computer Controlled VCO's

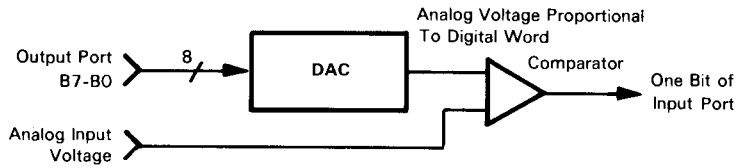


FIGURE 12-8. ADC System Where DAC Output is Compared Against Unknown Analog Input

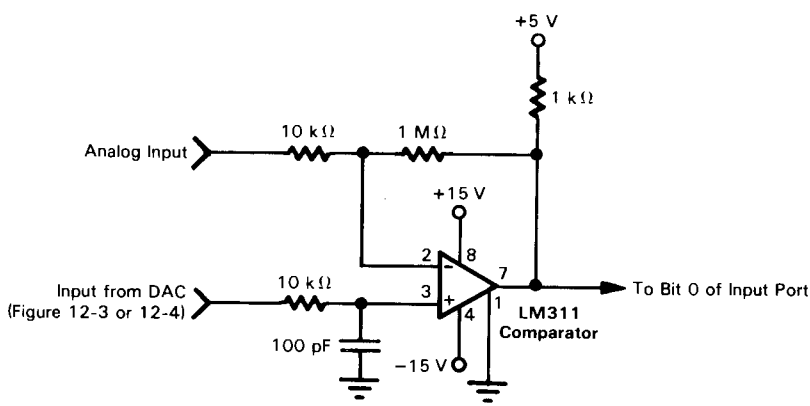


FIGURE 12-9. Added Circuitry For Ramp and Successive Approximation ADC

The following is a typical software driver routine:

```
                ;RAMP ADC CONVERSION ROUTINE
                ;VALUE IN B REGISTER IS LOST
                ;RETURNS WITH ADC VALUE IN REGISTER B
                ;
0001 =          DAC     EQU  1          ;DAC OUTPUT PORT
0001 =          COMP    EQU  1          ;COMPARATOR INPUT PORT
                ;
0100                    ORG  100H
                ;
0100 F5         INIT    PUSH PSW        ;SAVE REGISTERS
0101 AF                 XRA  A          ;RESET DAC
0102 47                 MOV  B,A
0103 D301               OUT  DAC
0105 04         LOOP    INR  B          ;INCREMENT DAC VOLTAGE
0106 78                 MOV  A,B
0107 D301               OUT  DAC
0109 DB01               IN   COMP       ;TEST FOR MATCH
010B 1F                 RAR
010C D26900             JNC  LOOP       ;NO MATCH, TRY NEXT DAC VALUE
010F F1                 POP  PSW        ;RESTORE REGISTERS
```

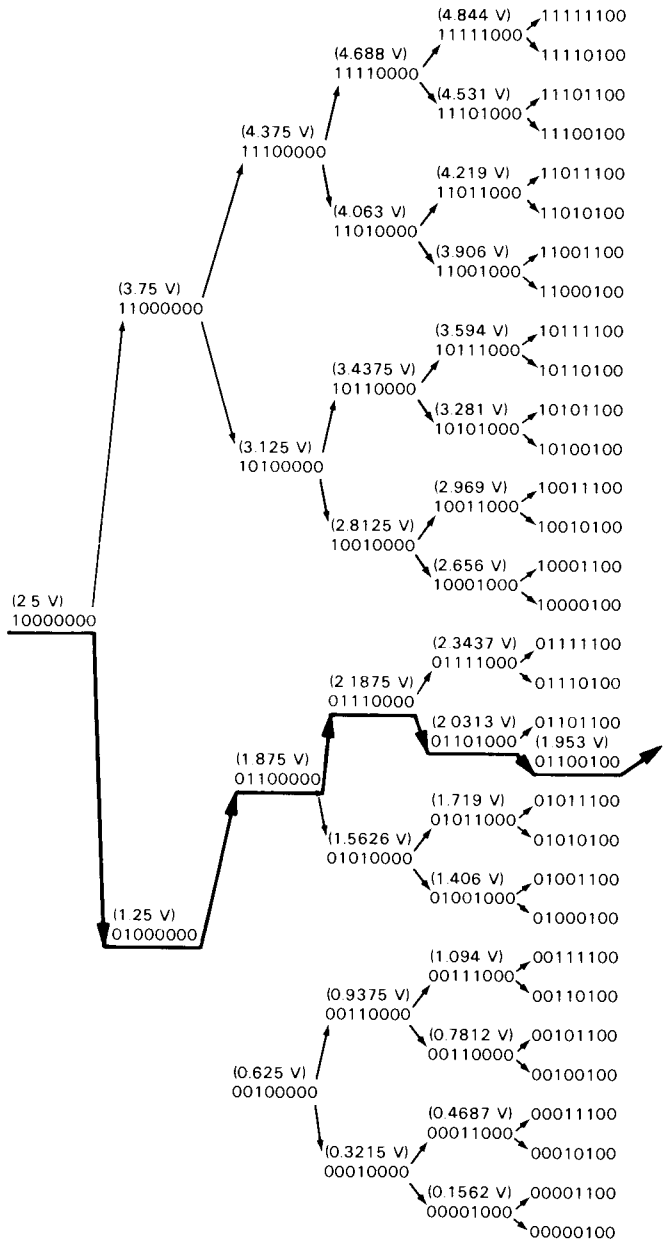
The routine will develop steps that are 1/256th of the full-scale voltage. For example, if the full-scale input is 5 volts, each step would be 19.5 millivolts. At the completion of the ADC routine, Register B will contain the binary equivalent of the analog input voltage. A delay subroutine can be added for slow operational amplifiers.

The Successive Approximation ADC Technique

The ramp method is simple but very slow, and the time required to perform the conversion depends on the value of the analog voltage input. The successive approximation method is typically 15 times faster and always takes the same number of loops in arriving at the result. In the case where speed is not critical, the ramp technique works adequately. The successive approximation technique uses the same circuitry as the ramp method. However, instead of changing the DAC output in incremental steps, it changes the most significant bits, one at a time, to quickly determine the correct digital word.

To comprehend the successive approximation technique, assume that the full-scale voltage is +5 V and the analog input is +2 V. The conversion process starts by setting the DAC input to 10000000, producing a half-scale DAC output (+2.5 V). If the comparator output is high then the CPU knows that the DAC input is too high and sets bit 7 to 0. The CPU then sets the DAC input to 01000000, producing a DAC output of +1.25 V which, when compared to the analog input, causes the comparator output to equal 0. The CPU now knows that the analog input is greater than +1.25 V but less than +2.5 V, and hence leaves bit 6 equal to 1. Next, the processor will output 01100000, producing a DAC output of

+1.87 V. This will still be too low, and bit 5 will be left equal to 1. The process will continue as follows (only the first six approximations are shown):



After eight approximations the result (01100110 = 1.9923 V) will be in Register D. Here is a suggested software subroutine for the successive approximation technique.

```
                ;SUCCESSIVE APPROXIMATION ADC ROUTINE
                ;CONTENTS OF REGISTER D LOST
                ;RETURNS WITH DATA IN REGISTER D
                ;
0001 =          DAC     EQU  1          ;DAC OUTPUT PORT
0001 =          COMP    EQU  1          ;COMPARATOR INPUT PORT
0080 =          MASK    EQU  80H        ;BIT MASK
                ;
0100                    ORG  100H
                ;
0100 F5         INIT    PUSH PSW        ;SAVE REGISTERS
0101 C5                 PUSH B
0102 0680               MVI  B,MASK     ;SET MASK
0104 1600               MVI  D,0        ;CLEAR DAC DATA REGISTER
0106 78         NXBIT   MOV  A,B        ;OUTPUT MASK
0107 D301               OUT  DAC
0109 DB01               IN   COMP       ;READ COMPARATOR OUTPUT
010B 1F                 RAR
010C D21201             JNC  SHIFT      ;IF NOT SET, SHIFT MASK
010F 78                 MOV  A,B        ;IF SET, SAVE DATA
0110 B2                 ORA  D
0111 57                 MOV  D,A
0112 78         SHIFT   MOV  A,B        ;SHIFT MASK
0113 1F                 RAR
0114 DA1B01             JC   DONE       ;IF 8 SHIFTS DONE GO TO END
0117 47                 MOV  B,A        ;IF NOT RECALL MASK
0118 C30601             JMP  NXBIT      ;AND DO NEXT BIT
011B C1         DONE    POP  B          ;RESTORE REGISTERS
011C F1                 POP  PSW
011D C9                 RET
```

A successive approximation ADC can be built in which the successive approximations are done in hardware rather than in software. This has the advantage of not taking up processor time during the conversion process. In other words, the processor can start the conversion process, go on to do other things, and then come back to get the converted value or have the ADC circuit generate an interrupt when it is finished. The hardware approach is also much faster (10-30 times faster) and hence is to be preferred when measuring fast changing signals.

A typical successive approximation ADC is shown in Figure 12-10. It employs the MC1408L8 DAC, used previously, and a special successive approximation register, the National Semiconductor DM2502. The conversion process is started by momentarily enabling the $\overline{S}$ (START) input. When the conversion process is finished the $\overline{EOC}$ line will be active. Hence, the status input port is read until $\overline{EOC}$ is 0, and then the data at the parallel data input port is read.

Fully integrated successive approximation ADCs are also available. One such device, the National Semiconductor ADC0816, also includes an on-chip multiplexer. The multiplexer permits the CPU to select any one of sixteen different analog input signals to be input to the ADC.

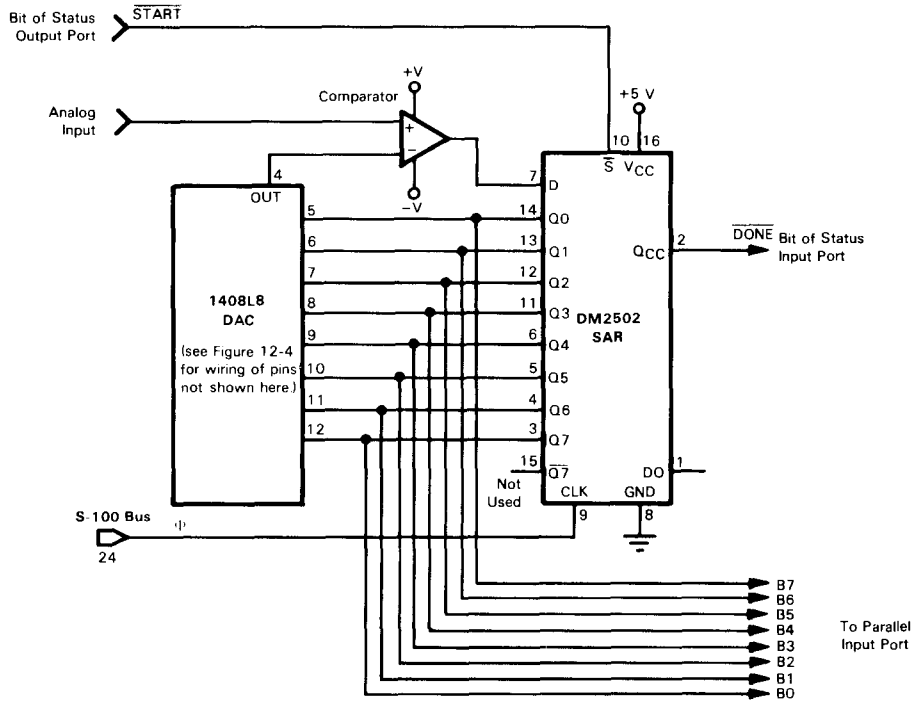


FIGURE 12-10. ADC Using Hardware Successive Approximation Register

Multiplexing ADCs

LSI ICs have recently become available which greatly simplify the multiplexing of ADCs. Multiplexing analog inputs substantially reduces the hardware cost of systems used to monitor or log data from many analog inputs. One such device is the National Semiconductor ADC0816 shown in Figure 12-11. The ADC0816 is a CMOS device incorporating an 8-bit successive approximation ADC, a 16-channel multiplexer, and necessary control logic. The successive approximation is done in hardware so that a conversion takes approximately 100 μs, using a 1 MHz clock rate. This is considerably faster than the software techniques discussed previously.

The device is addressed via the lower four bits of an output port (port 11_{16} in the following example program), allowing selection of any one of 16 single-ended analog signals as an input. Bit 4 of the port is used to start conversion

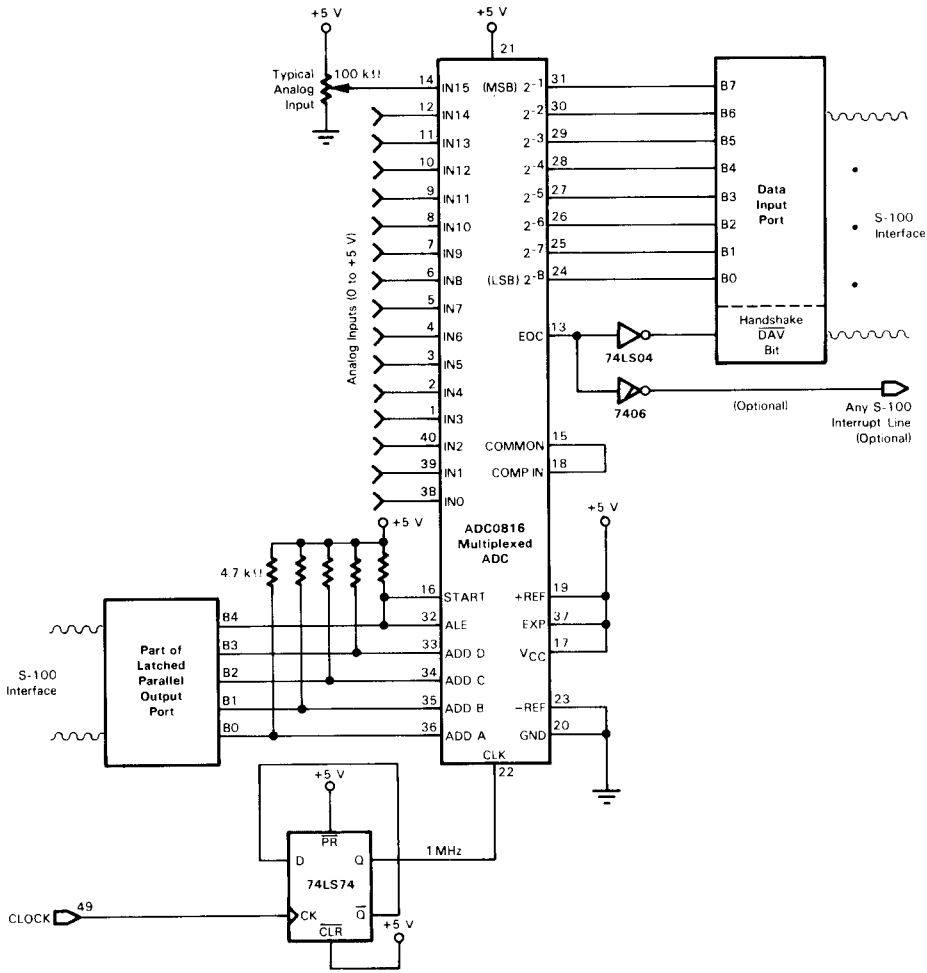


FIGURE 12-11. S-100 Interface of 16 Channel ADC Through Parallel Channels

(START input) and latch the 4-bit ADC multiplexer address code (ALE). The processor waits until the end of the ADC conversion cycle (EOC) is indicated by the ADC0816 and then reads the 8-bit binary output via the data input port. An alternate approach would be to use the EOC signal from the ADC to generate an interrupt signal.

Interrupts will be discussed in the next chapter. If interrupts are used, remember to clear the interrupt when starting another ADC conversion. The following is one possible software driver subroutine for the multiplexed ADC circuit.

```
                ;SUBROUTINE TO READ DATA FROM 16-CHANNEL ADC
                ;RETURNS WHEN ALL 16 CHANNELS HAVE BEEN READ
                ;STORES DATA IN TABLE CALLED "VALUES" IN EQUATES
                ;
                ;TO USE WITH DIFFERENT PORT ADDRESSES, CHANGE
                ;  "BASE" IN EQUATES
                ;ALL OTHER PORTS RELATIVE TO BASE
                ;
0010 =          BASE    EQU  10H        ;BEGINNING PORT ADDRESS
0011 =          DATA    EQU  BASE+1     ;DATA PORT ADDRESS
0010 =          STATUS  EQU  BASE       ;STATUS PORT ADDRESS
0001 =          MASK    EQU  01H        ;DAV BIT MASK
0010 =          START   EQU  10H        ;START CONVERSION BIT MASK
0200 =          VALUES  EQU  200H       ;ADDRESS OF VALUE TABLE
                ;
0100                    ORG  100H
                ;
0100 F5         ANALOG  PUSH PSW        ;SAVE ALL REGISTERS
0101 C5                 PUSH B          ;THAT WE ARE GOING TO USE
0102 E5                 PUSH H
0103 0E10               MVI  C,START    ;PUT START BIT MASK IN C
0105 210002             LXI  H,VALUES   ;SET UP HL TO POINT TO TABLE
0108 0600               MVI  B,0        ;SET B TO 0
010A 78         NEXT    MOV  A,B        ;GET THE CHANNEL NUMBER
010B D311               OUT  DATA       ;OUTPUT THE DATA
010D B1                 ORA  C          ;SAME BUT WITH BIT 4 HIGH
010E D311               OUT  DATA       ;OUTPUT IT
0110 A9                 XRA  C          ;SET BIT 4 LOW AGAIN
0111 D311               OUT  DATA       ;AND OUTPUT IT
0113 DB10       WAIT    IN   STATUS     ;READ THE STATUS PORT
0115 E601               ANI  MASK       ;IS EOC (DAV) TRUE?
0117 CA1301             JZ   WAIT       ;NO, TRY AGAIN
011A DB11               IN   DATA       ;YES, READ THE VALUE
011C 77                 MOV  M,A        ;PUT IT IN TABLE
011D 23                 INX  H          ;INCREMENT TABLE POINTER
011E 04                 INR  B          ;INCREMENT CHANNEL NUMBER
011F 78                 MOV  A,B        ;MOVE B INTO A FOR COMPARE
0120 FE10               CPI  16         ;DONE?
0122 C20A01             JNZ  NEXT       ;NO, NEXT CHANNEL
0125 E1                 POP  H          ;YES, POP ALL THE
0126 C1                 POP  B          ;REGISTERS WE USED
0127 F1                 POP  PSW        ;FOR RETURN
0128 C9                 RET             ;AND WE'RE DONE
```

Note that the positive and negative reference voltages must not be noisy, and the +5 V regulated supply should not be shared with other circuitry if full accuracy is to be achieved.

Analog Data Logging

Digital voltmeter (DVM) ICs offer an easy and very powerful way to measure analog voltages and log them. ICs are available that include virtually all the DVM circuitry and interface easily to a processor. A typical circuit is shown in Figure 12-12. It uses the National Semiconductor ADC3511C DVM IC. The only other IC required, besides the parallel ports, is a +2 V regulated reference IC. Note that good power supply distribution, decoupling, grounding, and regulation techniques should be exercised, as discussed in the previous section.

The DVM IC contains its own internal clock whose frequency is set by an external RC network (fout/fin). The IC provides addressed BCD data output (3-1/2 digits). Each BCD digit (four bits) is available at the outputs (2^0-2^3) on demand via three digit select inputs (D0, D1, and DLE). D0 and D1 are used to select the desired digit data. DLE is the data latch enable and is set to 0 to select a digit.

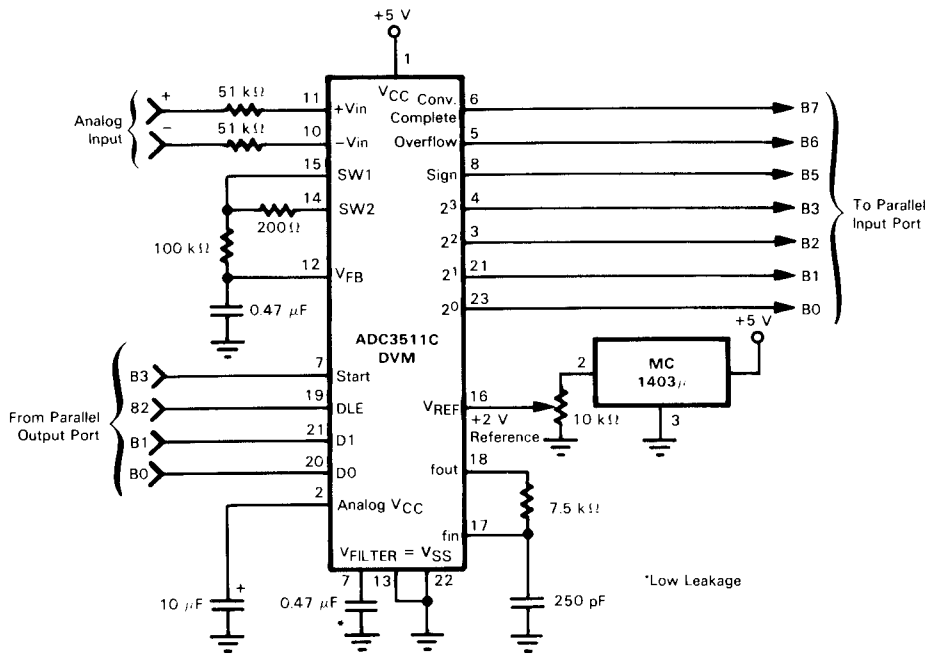
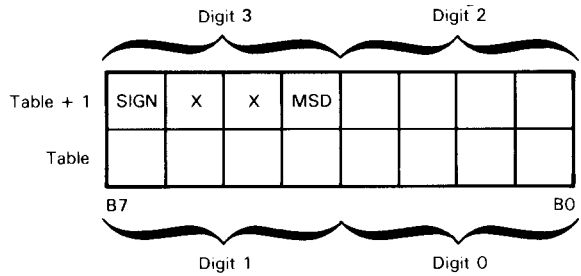


FIGURE 12-12. DVM IC Interface Circuit

Since the output is 3-1/2 digits, only the least significant bit in the most significant digit word is used, and therefore represents decimal digits 1 or 0. The next two bits in the word are not used. The most significant bit (sign) is used to indicate a positive or negative analog voltage input.

A "start of conversion" pulse transfers the BCD data to a set of internal latches and the "conversion complete" output is 1 when the conversion is done. The ADC3511C converts an analog input of 0 to ± 1.999 V to BCD code if the $+V_{REF} = 2$ V. It takes about 200 ms to do a conversion.

The software driver program will read the sign and four BCD digits in succession and store them in two adjacent memory locations as follows:



MSD = Most Significant Digit
X = not used
Sign: 1 = +
0 = −

```
                ;
                ; 3-1/2 DIGIT DVM DRIVER PROGRAM
                ;
                INITL:  LXI   SP,STACK
                        LXI   H,TABLE

                ;DRIVER PROGRAM FOR ADC3511 DVM IC
                ;RETURNS WITH 3-1/2 BCD DIGIT AND SIGN DATA
                ;IN ADMS MEMORY SPACE. IF OVERFLOW, RETURNS
                ;WITH "FF" IN MEMORY SPACE.
                ;
00F0 =          UMASK   EQU   0F0H      ;UPPER NIBBLE MASK
000F =          LMASK   EQU   0FH       ;LOWER NIBBLE MASK
0080 =          SMASK   EQU   80H       ;SIGN MASK
0010 =          DVM     EQU   10H       ;DVM I/O PORTS ADDRESSES
0200 =          ADMS    EQU   200H      ;ADC DATA MEMORY LOCATION
                ;
0100            ORG   100H
                ;
0100 F5         ADIS    PUSH  PSW       ;SAVE REGISTERS
0101 E5                 PUSH  H
0102 C5                 PUSH  B
0103 D5                 PUSH  D
0104 210002             LXI   H,ADMS    ;ADC DATA MEMORY POINTER
0107 B7         START   ORA   A         ;START CONVERSION
0108 D310               OUT   DVM
010A BF                 CMP   A
010B D310               OUT   DVM
010D DB10       WAIT    IN    DVM       ;CONVERSION DONE?
010F 17                 RAL
```

```
0110 D20D01          JNC   WAIT
0113 17              RAL                ;OVERFLOW?
0114 DA4201          JC    OVFLO        ;EXIT PROGRAM
0117 E680            ANI   SMASK        ;MASK OFF SIGN BIT
0119 57              MOV   D,A          ;SAVE SIGN
011A 3E04            MVI   A,04H        ;SET DIGIT SELECT CODE
011C CD2D01          CALL  RDGIT        ;GET DIGITS #1 AND #2
011F 77              MOV   M,A          ;SAVE DIGIT 1-2 DATA
0120 78              MOV   A,B          ;RECALL DIGIT SELECT CODE
0121 3C              INR   A            ;SET CODE FOR NEXT DIGIT
0122 CD2D01          CALL  RDGIT        ;GET DIGITS #3 AND #4
0125 E61F            ANI   1FH          ;MASK OFF UNWANTED BITS
0127 B2              ORA   D            ;COMBINE WITH SIGN BIT
0128 23              INX   H            ;SAVE IT IS MEMORY
0129 77              MOV   M,A
012A C34701          JMP   DONE         ;ALL DONE
012D 47      RDGIT   MOV   B,A          ;SAVE DIGIT SELECT CODE
012E D310            OUT   DVM          ;SELECT DIGIT
0130 DB10            IN    DVM          ;READ DATA
0132 E60F            ANI   LMASK        ;MASK OFF BCD DATA
0134 4F              MOV   C,A          ;SAVE IT
0135 78              MOV   A,B          ;RECALL DIGIT SELECT CODE
0136 3C              INR   A            ;SELECT NEXT DIGIT
0136 3C              INR   A            ;SELECT NEXT DIGIT
0137 47              MOV   B,A          ;SAVE DIGIT SELECT CODE
0138 D310            OUT   DVM
013A DB10            IN    DVM          ;READ BCD DATA
013C 17              RAL                ;SHIFT IT LEFT
013D 17              RAL
013E 17              RAL
013F 17              RAL
0140 B1              ORA   C            ;COMBINE UPPER AND LOWER NIBBLES
0141 C9              RET
0142 36FF    OVFLO   MVI   M,0FFH       ;SET OVERFLOW INDICATION
0144 23              INX   H
0145 36FF            MVI   M,0FFH
0147 D1      DONE    POP   D            ;RESTORE REGISTERS
0148 C1              POP   B
0149 E1              POP   H
014A F1              POP   PSW
014B C9              RET
```

Analog Input Devices

A few of the more common devices that can be connected to ADC circuits are described below.

Joy sticks are essentially two potentiometers that are mechanically linked together so that one or both changes resistance as the "joy stick" is moved in an X-Y plane. Two separate ADC circuits are required, one for each potentiometer. Since joy stick positioning is usually not critical, very economical ADC circuits can be used. The simple ADC arrangement shown in Figure 12-13 is quite popular. The conversion requires that the processor output a 1 to start the capacitor charging and at the same time start a software counter or hardware timer. The capacitor charges exponentially toward +5 V, raising the voltage on the positive input of the comparators. When the voltage equals the analog input voltage, the comparator output goes to 1 level, causing the count to stop. The count is approximately proportional to the analog input voltage (potentiometer setting) if

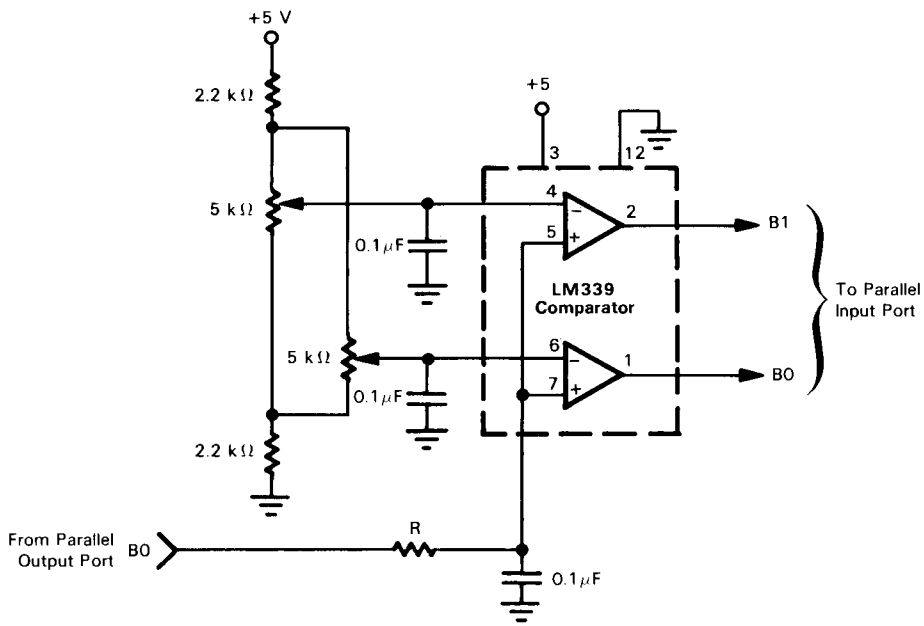


FIGURE 12-13. Simple Joy Stick ADC Circuit

the capacitor is not allowed to charge fully. This value is saved. The processor then discharges the capacitor by outputting a 0 and repeats the process for the other input.

Temperature Sensing can be accomplished simply, as shown in Figure 12-14*a*, using a thermistor. Correction for non-linear characteristics of the thermistor can be accomplished by using a look-up table in memory. For precise temperature measurement a temperature controller IC, such as the National Semiconductor LM3911, should be employed. It includes a temperature sensor, a stable voltage reference, and an operational amplifier. The output voltage is directly proportional to temperature over a range of -25° C to +85° C. The LM3911 is shown in Figure 12-14*b*.

Light Sensing can be done using a photocell divider circuit such as that shown in Figure 12-15*a*. A circuit providing much greater sensitivity is shown in Figure 12-15*b*. Here, a photo Darlington transistor (e.g., GE L14F2, which includes a built-in lens) is used.

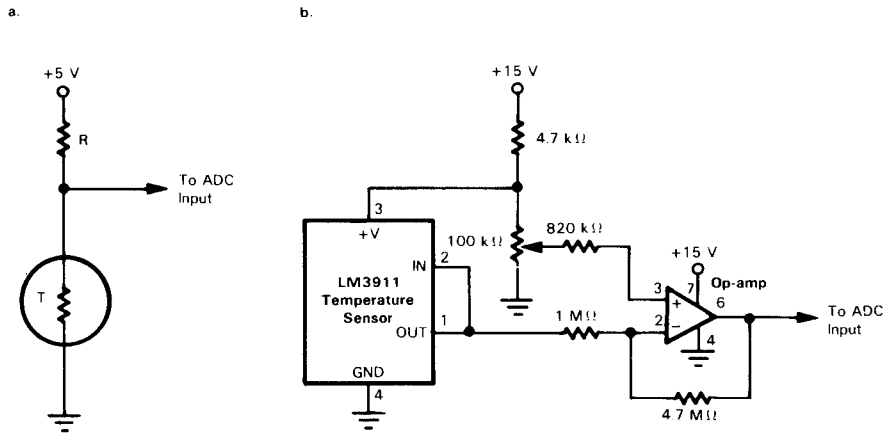


FIGURE 12-14. Two Different Temperature Sensing Circuits

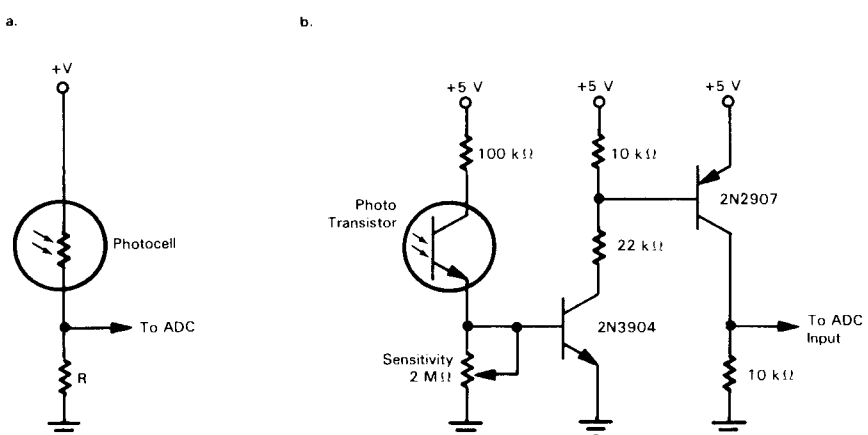


FIGURE 12-15. Light Sensing Circuit

REFERENCES

Kane, J., and Osborne, A. *An Introduction to Microcomputers: Volume 3 — Some Real Support Devices,* Section E. Berkeley: Osborne/McGraw-Hill, 1979.

Libes, S. *Fundamentals and Applications of Digital Logic Circuits,* 2nd edition. Rochelle Park, N. J.: Hayden Book Co., Inc., 1978.

National Semiconductor Corporation, *Pressure Transducer Handbook,* 1977.

Titus, J. A.; Titus, C.A.; Rony, P.R.; and Larsen, D.G. *Bugbook VII — Microcomputer-Analog Convérter Software and Hardware Interfacing.* E & L Instruments, 1978.

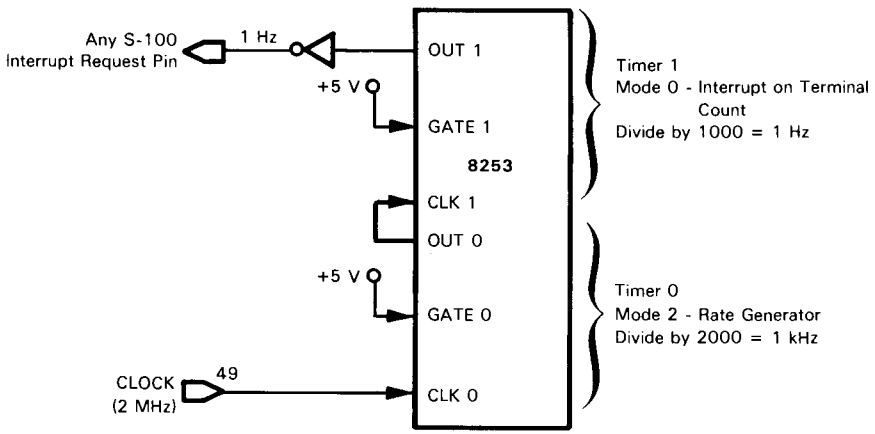


FIGURE 14-7. Timer Configuration for Interrupting Interval Timer Used to Make a "Real Time Clock"

```
                ;ROUTINES TO COMPUTE REAL TIME FROM INTERVAL COUNTER INTERRUPT
                ;ROUTINE MANIPULATES 3 MEMORY LOCATIONS STARTING AT "TABLE"
                ;TABLE HAS SECONDS IN BINARY, TABLE+1=MINUTES, TABLE+2=HOURS
                ;IMPLEMENTS 24 HOUR CLOCK, USES 1 HZ INTERRUPT
                ;TABLE LOCATIONS ASSUMED SET TO PROPER TIME
                ;WHEN "UPDATE" IS CALLED
                ;UPDATE IS THE INTERRUPT SERVICE ROUTINE
                ;INIT IS EXECUTED ONLY ONCE
                ;
                ;TIMER USES FOUR PORTS STARTING WITH BASE
                ;ALL PORTS RELATIVE TO BASE
                ;TO CHANGE PORT ADDRESSES, CHANGE BASE
                ;
0010 =          BASE    EQU  10H      ;BEGINNING PORT ADDRESS
0010 =          CNTR0   EQU  BASE     ;COUNTER 0 REGISTER
0011 =          CNTR1   EQU  BASE+1   ;COUNTER 1 REGISTER
0012 =          CNTR2   EQU  BASE+2   ;COUNTER 2 REGISTER
0013 =          CONTROL EQU  BASE+3   ;CONTROL WORD REGISTER
                ;
07D0 =          COUNT0  EQU  2000     ;DIVISOR FOR COUNTER 0
03E8 =          COUNT1  EQU  1000     ;DIVISOR FOR COUNTER 1
                ;
0080 =          TABLE   EQU  80H      ;ADDRESS OF TIME
                ;
0100                    ORG     100H
                ;
0100 3E34       INIT    MVI     A,00110100B  ;SELECT COUNTER 0, TWO BYTE LOAD
                ;                            ;MODE 2 AND BINARY FORMAT
0102 D313               OUT     CONTROL      ;SEND IT TO CONTROL PORT
0104 21D007             LXI     H,COUNT0     ;COUNT VALUE IN HL
0107 7D                 MOV     A,L          ;LOW BYTE IN A
0108 D310               OUT     CNTR0        ;SEND IT TO COUNTER 0
010A 7C                 MOV     A,H          ;HIGH BYTE IN A
010B D310               OUT     CNTR0        ;SEND IT AND START COUNTER
```

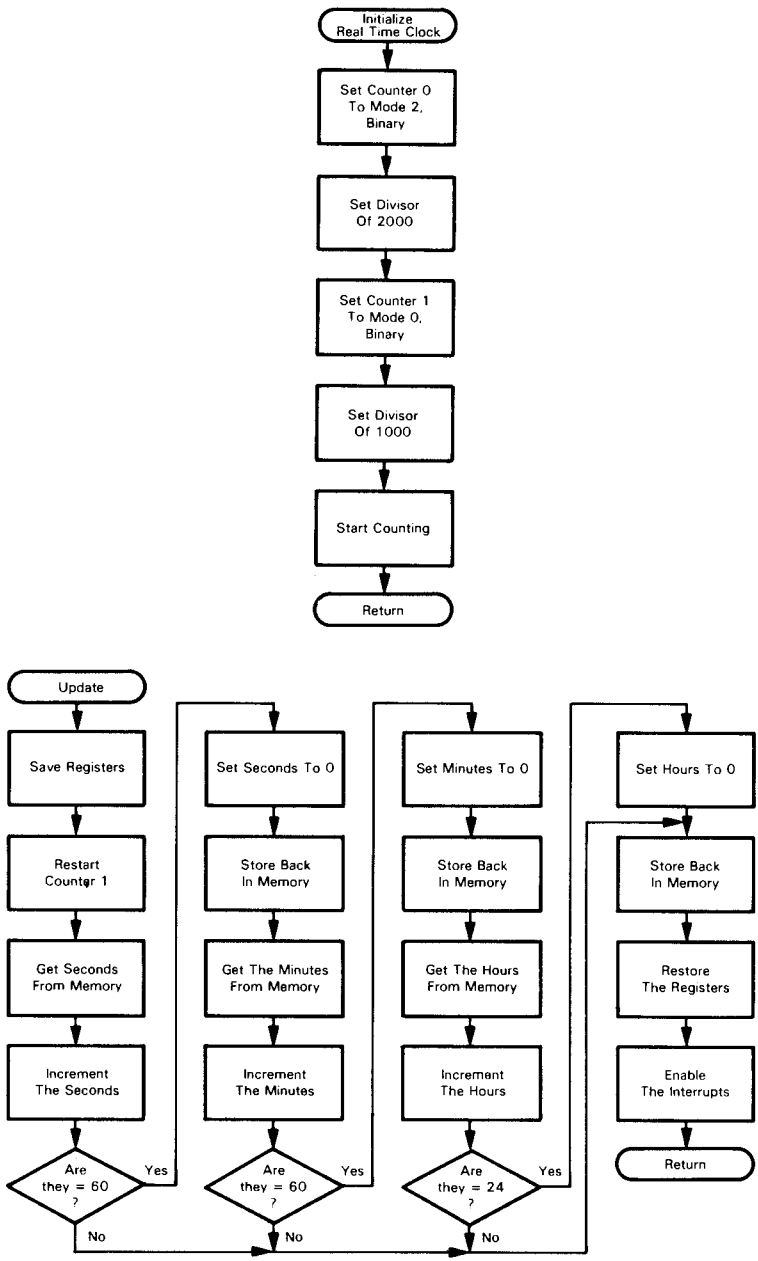


FIGURE 14-8. Flowchart for Real Time Clock Program

```
010D 3E70          MVI     A,01110000B  ;SELECT COUNTER 1, TWO BYTE LOAD
                   ;                    ;MODE 0 AND BINARY FORMAT
010F D313          OUT     CONTROL      ;SEND IT TO THE CONTROL PORT
0111 21E803        LXI     H,COUNT1     ;COUNT VALUE IN HL
0114 7D            MOV     A,L          ;LOW BYTE IN A
0115 D311          OUT     CNTR1        ;SEND IT TO COUNTER 1
0117 7C            MOV     A,H          ;HIGH BYTE IN A
0118 D311          OUT     CNTR1        ;SEND IT AND START COUNTER
011A FB            EI                   ;ENABLE THE INTERRUPTS
011B C9            RET                  ;ON TO SYSTEM TASKS
                   ;
F000               ORG     0F000H
                   ;
F000 F5    UPDATE  PUSH    PSW          ;SAVE THE REGISTERS
F001 E5            PUSH    H
F002 21E803        LXI     H,COUNT1     ;RESEED COUNTER 1
F005 7D            MOV     A,L          ;LOW BYTE IN A
F006 D311          OUT     CNTR1        ;SEND LOW BYTE TO COUNTER 1
F008 7C            MOV     A,H          ;HIGH BYTE IN A
F009 D311          OUT     CNTR1        ;SEND IT AND START COUNTER GOING
F00B 218000        LXI     H,TABLE      ;SET UP H WITH TIME LOCATIONS
F00E 7E            MOV     A,M          ;GET SECONDS IN A
F00F 3C            INR     A            ;INCREMENT SECONDS
F010 FE3C          CPI     60           ;HAS IT REACHED 60?
F012 C22AF0        JNZ     EXIT         ;NO, WE'RE DONE
F015 AF            XRA     A            ;YES, ZERO SECONDS
F016 77            MOV     M,A          ;STORE IT BACK, SECONDS NOW =0
F017 23            INX     H            ;POINT TO MINUTES
F018 7E            MOV     A,M          ;GET MINUTES IN A
F019 3C            INR     A            ;INCREMENT THE MINUTES
F01A FE3C          CPI     60           ;ARE THEY 60 YET?
F01C C22AF0        JNZ     EXIT         ;NO, WE'RE DONE
F01F AF            XRA     A            ;YES, ZERO MINUTES
F020 77            MOV     M,A          ;AND STORE IT BACK
F021 23            INX     H            ;POINT TO HOURS
F022 7E            MOV     A,M          ;GET HOURS IN A
F023 3C            INR     A            ;INCREMENT THE HOURS
F024 FE18          CPI     24           ;ARE THEY 24 YET?
F026 C22AF0        JNZ     EXIT         ;NO, WE'RE DONE
F029 AF            XRA     A            ;YES, ZERO THE HOURS
F02A 77    EXIT    MOV     M,A          ;STORE WHATEVER BACK IN MEMORY
F02B E1            POP     H            ;RESTORE REGISTERS
F02C F1            POP     PSW
F02D FB            EI                   ;ENABLE THE INTERRUPTS
F02E C9            RET                  ;AND RETURN
```

interrupts

13

The normal program execution of the CPU can be suspended in response to a request for *service* from a peripheral device or circuit. This request is called an "interrupt" or "interrupt request." The CPU then executes a software routine which services the device. Upon completion of the interrupt service routine the CPU resumes the interrupted program at the point where it was interrupted. In other words, an interrupt causes the CPU to suspend what it is currently doing, attend to the needs of the interrupting device, and resume what it was doing before the interrupt. For a more detailed discussion of the basics of interrupts see references 2, 3, and 4 at the end of this chapter.

Interrupts usually make the computer system's I/O handling more efficient. In previous chapters we looked at I/O handling for terminals. In the techniques shown, the computer waited for a key to be pressed before responding. Hence, as much as 99% of the CPU's time could be spent in a waiting condition. By having the terminal interrupt the CPU's operation when a key is pressed, we can have the CPU do other tasks instead of waiting, thereby improving the efficiency of the computer system. If we are dealing with a slow output device, such as a teletypewriter, we may use interrupts here, too. After all, an 8080 with a 2 MHz clock can execute as many as 20,000 instructions in the time that it takes a teletypewriter to print one character.

In another application, such as refreshing a display, we may interrupt the CPU at regular intervals. For example, the multiplexed 7-segment display circuit shown in Figure 10-3 might be scanned at a rate of 65 times per second. If eight digits are used, the repetition rate becomes about 500 times per second. Thus we can have a free-running pulse generator circuit which interrupts the CPU once every 2

milliseconds. In this case, every two milliseconds the CPU executes the display scan routine and then returns to the main program.

An interrupt-driven processor system may therefore give the appearance of running more than one program simultaneously. This is often referred to as "foreground/background" operation. For example, a system may have a slow interrupt-driven printer which generates an interrupt when it is ready to have its buffer filled. Thus the CPU may be executing a program and be interrupted by the printer only when the printer is ready to receive a string of characters to be printed. The printer is said to operate in the background, while the main program operates in the foreground, giving the appearance of "multi-processing" or "multi-tasking."

The block diagram of a simple single interrupt system is shown in Figure 13-1. A peripheral device is interfaced to the S-100 bus in essentially the same manner as we have previously shown. In addition, the interface circuit can signal the CPU, via an interrupt request line, that it wishes to interrupt the CPU's operation. The CPU receives the interrupt signal, suspends its current operation, saves its return address, and generates an interrupt acknowledge status signal (sINTA). The sINTA signal is used by the interrupting circuit to tell the CPU, via the data input bus, the address of the Interrupt Service Routine (ISR). The CPU executes the ISR and then returns to the previously interrupted program. The ISR address that the interrupting device provides is usually referred to as a "vector," and the act of jumping to the ISR is called "vectoring."

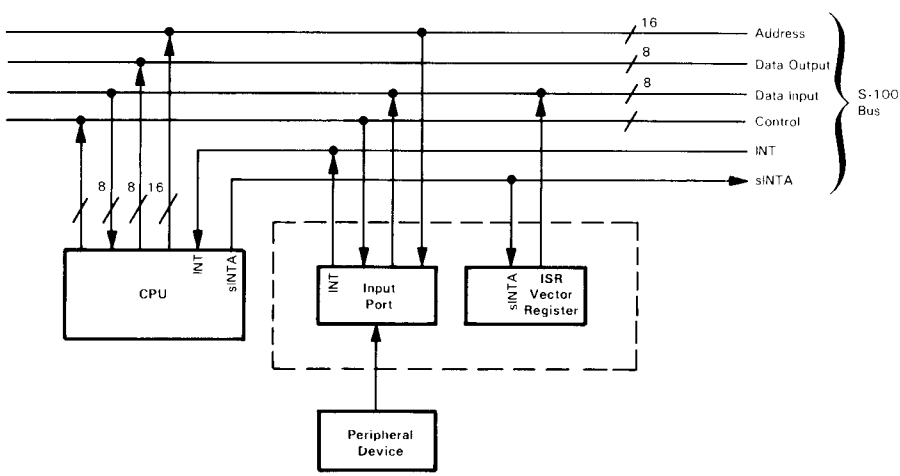


FIGURE 13-1. Block Diagram of an S-100 Input Interrupt System

It is also possible to have more than one device initiate an interrupt and have different ISRs, one for each device. For example, when device #1 causes an interrupt, the processor "vectors" to the device #1 ISR. This system is called a "multiple vectored interrupt" system and is shown in Figure 13-2. Furthermore, it is possible to establish interrupt priorities among interrupting devices, such that should two devices generate simultaneous interrupts, one will have priority over the other. This is called a priority interrupt system.

Interrupts are used primarily for more efficient I/O handling. They are also widely used in the following applications:

1. In systems where there are a large number of inputs to be polled and very fast response is required (e.g., in a safety alarm system).
2. Power failure detection. When a power failure occurs, hardware can be used to detect this failure and by initiating an interrupt save the CPU registers and RAM contents on disk. A small backup power supply sufficient to provide 1 or 2 seconds of operation is usually all that is required. Then when power is restored, the memory and register contents are reloaded and the program resumes running. This interrupt usually has the highest priority in a priority interrupt system.
3. User control panel resetting of system or manual control.
4. Interrupts are sometimes used in setting breakpoints for tracing program operation when debugging programs.

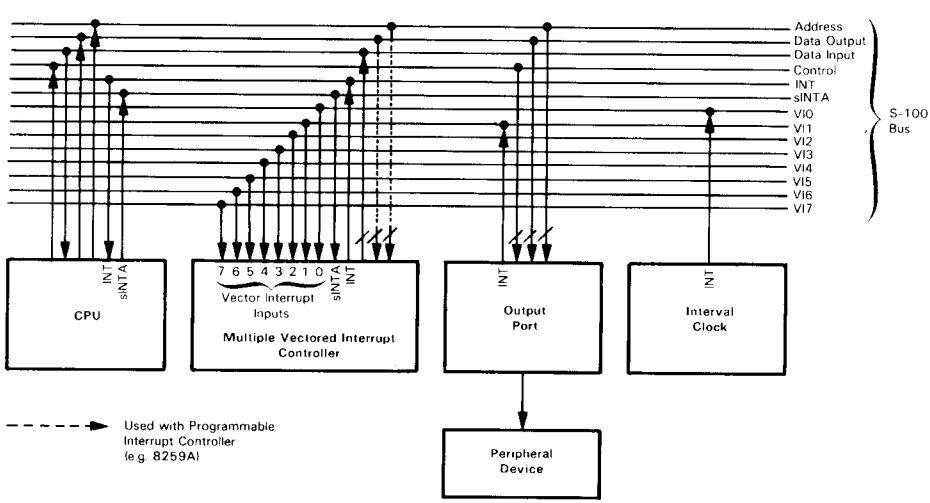


FIGURE 13-2. Block Diagram of an S-100 Multiple Interrupt System

ADVANTAGES AND DISADVANTAGES OF INTERRUPTS

Interrupts have certain disadvantages which in many cases make their use inadvisable. Interrupt-driven programs are much more difficult to write, debug, and test. This is because of the random nature of interrupts. Further, additional hardware is required, which can get quite complex, particularly in multiple interrupt systems.

Interrupts are easily justified on large, expensive, high-speed CPUs. However, in slower low-cost microcomputer systems the large amount of time spent on interrupt housekeeping may leave too little time for useful work. One should consider the possibility of using several smaller processors instead of interrupts. The only case where interrupts are really essential is in power fail interrupt systems.

S-100 INTERRUPT LINES

The S-100 Bus specifies ten interrupt request input lines and one interrupt acknowledge line:

Signal Name	Pin No.	Description
INT*	73	Primary interrupt request input
VI0*	4	Vectored interrupt request line 0
VI1*	5	Vectored interrupt request line 1
VI2*	6	Vectored interrupt request line 2
VI3*	7	Vectored interrupt request line 3
VI4*	8	Vectored interrupt request line 4
VI5*	9	Vectored interrupt request line 5
VI6*	10	Vectored interrupt request line 6
VI7*	11	Vectored interrupt request line 7
NMI*	12	Non-maskable interrupt request line
PWRFAIL*	13	System power failure signal
ERROR*	98	Status signal indicating that an error has occurred
sINTA	96	Interrupt acknowledge line

The INT line and the eight vectored interrupt request lines are enabled by a low logic level which must be kept active until the request is acknowledged. The INT input is a general interrupt request input used by an interrupt controller circuit or directly by the interrupting device when only one such device exists in the system. The VI0*-V17* input lines are used in a multiple interrupt system. If the interrupt

controller circuit utilizes a priority scheme, then when two simultaneous interrupts arrive it processes the highest priority interrupt. In most systems VIO* has the highest priority. All of these lines can usually be "masked" (turned off) by the CPU.

The NMI* input line is a nonmaskable interrupt request input. This input cannot be turned off by the processor. Further, it has priority over all other interrupt requests. NMI* is negative-edge triggered and need not be asserted as a level.

The PWRFAIL* input line is a power-failure-pending signal. It is used to indicate an imminent power failure. This line must be enabled at least 50 ms before the local voltage regulators drift out of specification. The line must also stay low until power is restored and the power-on clear signal is activated. This means that a normally closed relay or a battery powered circuit must drive the power failure line.

The ERROR* line is an error status signal indicating an error condition. It is used to indicate that the current bus operation is producing an error of some sort (for example, a memory parity error, write to protected memory, etc.). When an error occurs the processor should perform a "trap" operation. In other words, all relevant information about the error-causing cycle should be saved (e.g., address, data, register data, status information in memory, etc.).

MICROPROCESSOR INTERRUPT CHARACTERISTICS

The 8080, 8085, and Z80 microprocessors have similar interrupt systems. The 8085 and Z80 accommodate the 8080 interrupt system and have additional interrupt capabilities. We will therefore look at all three microprocessors separately.

8080 INTERRUPT SYSTEM

The 8080 has a single interrupt request input which is connected to the INT line of the S-100 Bus if no vectored interrupt circuit is employed on the CPU board. This interrupt input may be enabled and disabled with software using the EI and DI instructions. Further, the interrupt input is automatically disabled by the processor during a reset operation and during the acceptance of an interrupt.

If the interrupt request occurs when the interrupt input is enabled, the 8080 completes the current instruction and then executes an interrupt acknowledge cycle. It sets the sINTA line active. This is a status signal indicating that the processor acknowledges the interrupt. At the same time the 8080 enters a modified instruction fetch cycle. It is modified in that MEMR is low and pDBIN is high, hence

the instruction is not read from memory. Rather, external hardware must be provided to furnish a RST (restart) or other instruction.

The 8080 has a special instruction (RST) which is in effect a single-byte CALL instruction to one of eight different interrupt vectors (addresses). When RST is executed the return address (PC register) is saved on the stack. The processor then branches to an address indicated by bits 4, 3, and 2 of the RST instruction, as follows:

RST Instructions

Binary	Hex	RST	Branch to (hex)
11 000 111	C7	0	0000
11 001 111	CF	1	0008
11 010 111	D7	2	0010
11 011 111	DF	3	0018
11 100 111	E7	4	0020
11 101 111	EF	5	0028
11 110 111	F7	6	0030
11 111 111	FF	7	0038

Therefore, if the instruction code read by the processor during an interrupt acknowledge cycle is 11010111 ($D7_{16}$) the processor will branch to memory location 0010_{16} and commence execution there.

A very simple single interrupt circuit can be constructed using the parallel input port shown in Figure 13-3. Notice the similarity between this schematic and the one presented in Chapter 8. Here a parallel keyboard can generate an interrupt. No provisions are made for the generation of an RST instruction. Hence, during the interrupt acknowledge cycle the processor reads the data bus, which usually has all highs on it (no data) and interprets it as FF_{16}, or an RST 7 instruction. The CPU therefore vectors to memory location 38_{16} and continues execution. This is not recommended for non-terminated motherboards.

The circuit accepts a strobe from the keyboard, which latches the data from the keyboard and sets the $\overline{DAV}$ flip-flop, interrupting the CPU. The CPU then goes through a service routine, reading the data latched at the input port.

The ISR will usually follow the steps shown in Figure 13-4. The following is a suggested ISR for the circuit shown in Figure 13-4.

```
                ;INTERRUPT SERVICE ROUTINE FOR SIMPLE KEYBOARD
                ;INTERRUPT - HARDWARE AUTOMATICALLY RESETS INTERRUPT
                ;WHEN KEYBOARD DATA HAS BEEN READ
                ;
                ;TO USE WITH OTHER PORT ADDRESSES, CHANGE "BASE" IN EQUATES
0010 =          BASE    EQU     10H         ;BEGINNING PORT ADDRESS
0010 =          PORT    EQU     BASE

0038            ORG     38H                 ;RST 7 VECTORS HERE
```

```
0038 F5    SRVCE  PUSH   PSW              ;PUSH ANY REGISTERS NEEDED
0039 DB10         IN     PORT             ;GET THE DATA
                  ;INSERT ROUTINE TO DO SOMETHING WITH KBD DATA HERE
                  ;OR "CALL" TO ROUTINE
003B F1           POP    PSW              ;POP ALL REGISTERS
003C FB           EI                      ;ENABLE INTERRUPTS AGAIN
003D C9           RET                     ;BACK TO OTHER TASK
```

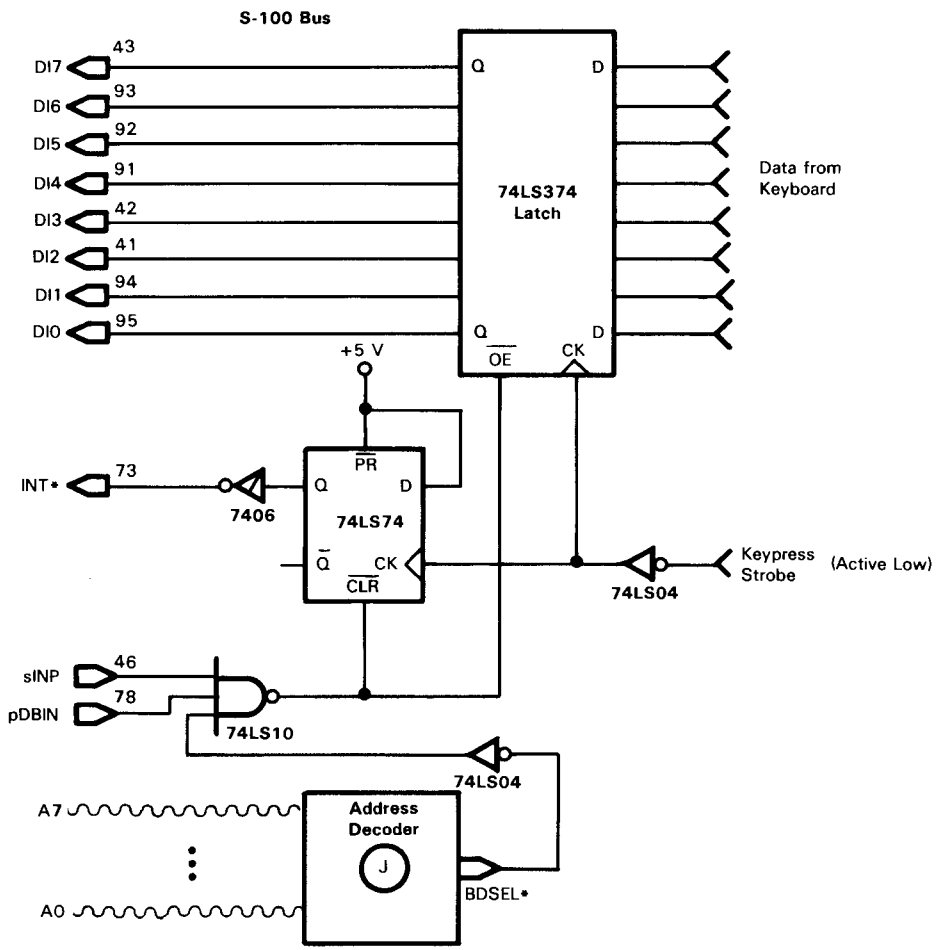


FIGURE 13-3. Simple Single Interrupt Circuit

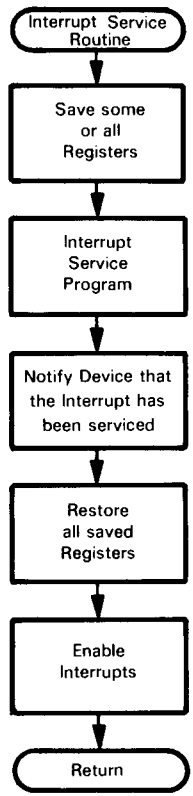


FIGURE 13-4. Interrupt Service Routine Flow Diagram

If there is not enough room to put the entire service routine at location 38H, then put a "jump" to the routine there instead. If it is desired to vector to a different restart location, an 8-bit buffer can be used as an interrupt instruction port. This circuit is shown in Figure 13-5. During the interrupt acknowledge cycle, the buffer gates the RST instruction onto the data bus. The sINTA signal is used to gate the RST instruction on the data input bus.

Multiple Interrupt System

The single interrupt circuit can be expanded to a multiple interrupt system by the addition of an 8-line to 3-line encoder IC (74LS148) as shown in Figure 13-6. Each interrupt device strobe is connected to an input of the encoder IC and will

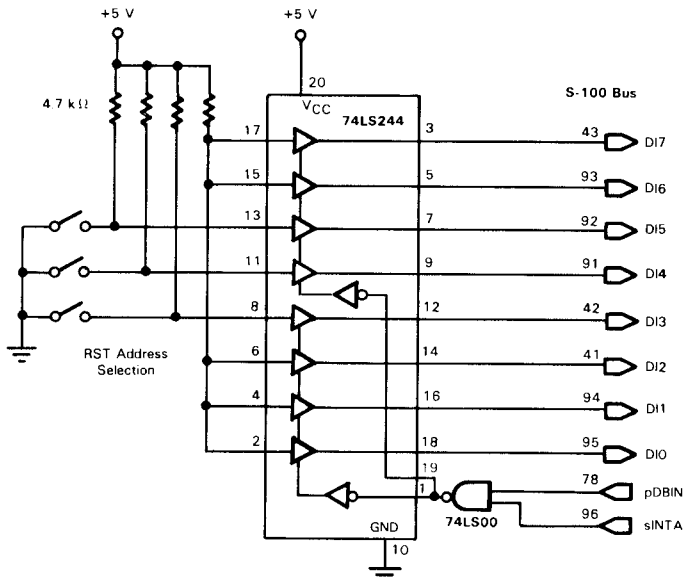


FIGURE 13-5. Interrupt Instruction Port

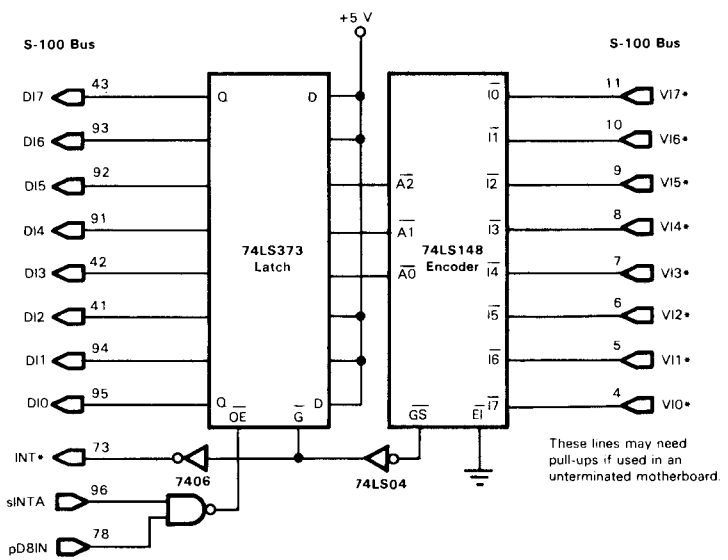


FIGURE 13-6. Simple Priority Interrupt Controller

cause a different RST instruction to be placed on the DI bus. For example, a low on VI7* will generate an RST 7 instruction.

The RST vector locations are too close together for entire ISRs. There are only eight bytes between RST addresses, and most ISRs require more than eight bytes. Hence, a jump instruction must be used to vector to the actual ISRs for RST 0 through RST 6. The following is a typical routine to initialize the CPU and set up the ISR vectors.

```
                ;SYSTEM INITIALIZATION ROUTINES
                ;
                ;RESET & RST-0 ENTRY POINT
                ;USE FOR COLD START OR TO RESET SYSTEM
                ;
FE00 =          STACK     EQU  0FE00H     ; THESE
0100 =          MAIN      EQU  100H       ;  ARE
1100 =          ISR1      EQU  1100H      ;   DUMMY
1200 =          ISR2      EQU  ISR1+100H  ;    VALUES
1300 =          ISR3      EQU  ISR2+100H  ;
1400 =          ISR4      EQU  ISR3+100H  ; CHANGE
1500 =          ISR5      EQU  ISR4+100H  ;  AS
1600 =          ISR6      EQU  ISR5+100H  ;   REQUIRED
1700 =          ISR7      EQU  ISR6+100H  ;
                ;
                ;
0000                      ORG  0          ;RESET & RST 0 ENTRY POINT
                ;
0000 FB                   EI              ;ENABLE INTERRUPTS
0001 C30001               JMP  MAIN       ;GO TO MAIN PROGRAM
                ;
                ;
0008                      ORG  8          ;RST 1 ENTRY POINT
                ;
0008 C30011               JMP  ISR1       ;INTERRUPT SERVICE ROUTINE-1 VECTOR
                ;
                ;
0010                      ORG  10H        ;RST 2 ENTRY POINT
                ;
0010 C30012               JMP  ISR2       ;INTERRUPT SERVICE ROUTINE-2 VECTOR
                ;
                ;
0018                      ORG  18H        ;RST 3 ENTRY POINT
                ;
0018 C30013               JMP  ISR3       ;INTERRUPT SERVICE ROUTINE-3 VECTOR
                ;
                ;
0020                      ORG  20H        ;RST 4 ENTRY POINT
                ;
0020 C30014               JMP  ISR4       ;INTERRUPT SERVICE ROUTINE-4 VECTOR
                ;
                ;
0028                      ORG  28H        ;RST5 ENTRY POINT
                ;
0028 C30015               JMP  ISR5       ;INTERRUPT SERVICE ROUTINE-5 VECTOR
                ;
                ;
0030                      ORG  30H        ;RST6 ENTRY POINT
                ;
0030 C30016               JMP  ISR6       ;INTERRUPT SERVICE ROUTINE-6 VECTOR
                ;
                ;
0038                      ORG  38H        ;RST 7 ENTRY POINT
```

The circuit in Figure 13-7 is very similar to the one in Figure 13-6 but it has a set of eight gates between the VI* lines and the inputs to the 74LS148. One input

to each gate is connected to a VI* line and the other input is connected to a bit from a latched parallel output port.

The purpose of this circuit is to allow each interrupt to be "masked." This means that the interrupt will be ignored by the circuitry. When the bit from the parallel port is high, the interrupt input will be masked, and when the bit is low the interrupt will be gated through to the 74LS148.

One of the simple latched parallel output ports from Chapter 8 can be used for the mask bits.

The 8255 PPI Interrupts

The Intel 8255 Programmable Parallel Interface (PPI) is a versatile LSI parallel I/O device with three operating modes. While mode 0 provides basic I/O, modes 1 and 2 provide strobed I/O. Figure 13-8 shows the 8255 being used to handle two interrupt driven devices. Each parallel port (A and B) generates its own interrupt strobe which would go to the interrupt inputs of a circuit such as those shown in Figures 13-6, 13-7, and 13-9.

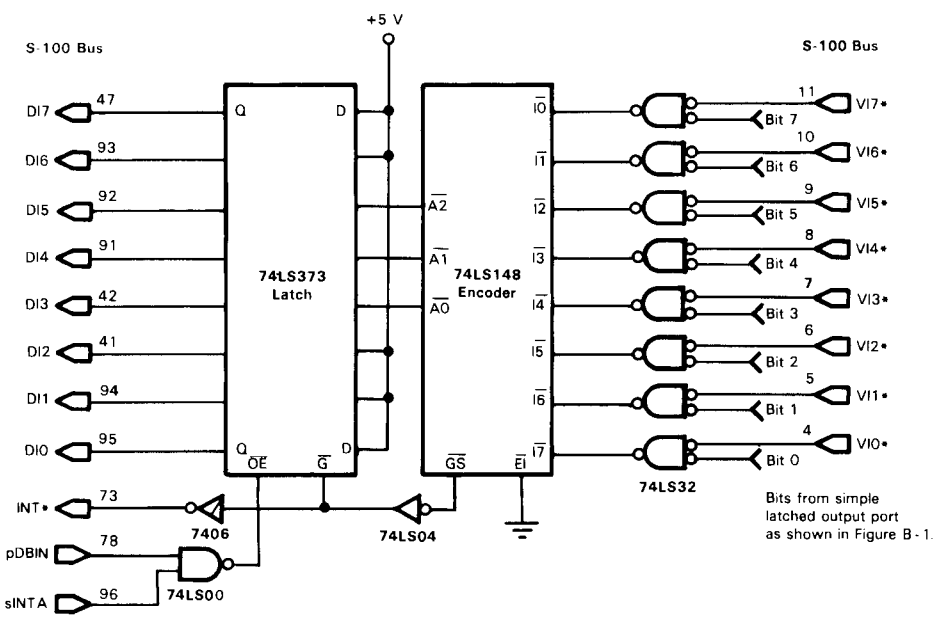


FIGURE 13-7. Simple Priority Interrupt Controller with Full Masking

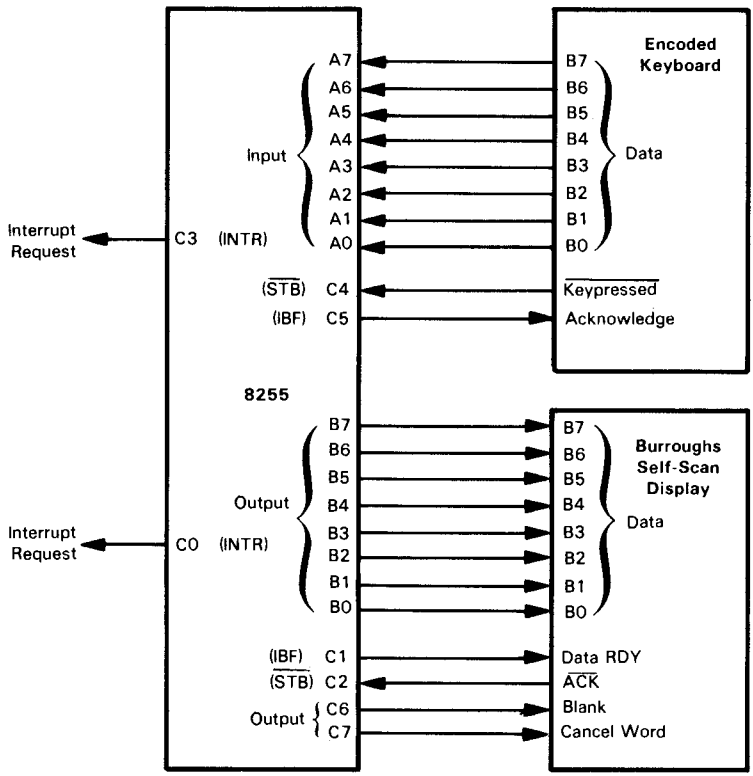


FIGURE 13-8. Using the 8255 PPI to Interface Two Interrupt Driven Devices (Mode 1)

The 8259A Programmable Interrupt Controller and CPUs

The 8259A Programmable Interrupt Controller (PIC) is a very complex and versatile IC and has many features and operating modes. It would take up a whole chapter just describing what this chip does. Instead of reprinting all the information here, we suggest that you obtain it from the source — Intel. Intel has published an excellent application note on the 8259A (see the references at the end of this chapter).

A block diagram of the 8259A appears in Figure 13-9.

The 8259A PIC handles up to eight vectored priority interrupts and can be cascaded for up to 64 vectored priority interrupts. It has several modes of operation.

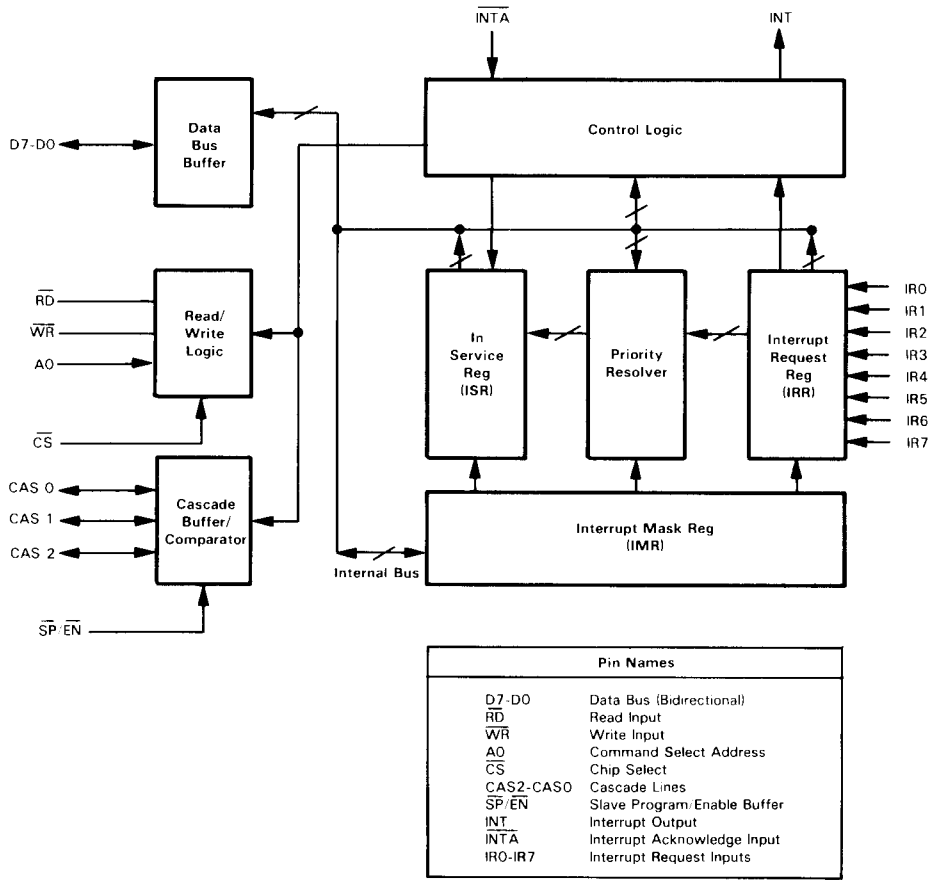


FIGURE 13-9. Block Diagram of 8259A Programmable Interrupt Controller (Courtesy of Intel Corp.)

The 8259A responds to an interrupt acknowledge cycle by placing the op-code for a call instruction on the data lines. When the CPU decodes the call op-code, it must fetch two more bytes to get the call address. The 8259A was originally designed to work with an 8085 processor. The 8085 will issue two more interrupt acknowledge cycles so that it may read the next two bytes. However, the 8080 and the Z80 will not respond in this manner. Both read two more bytes, but they will no longer provide any status information to signify that an interrupt acknowledge cycle is still occurring. Instead, they will issue a memory read status.

In terms relating to the S-100 Bus, if an 8085 is the CPU on the master and an interrupt occurs, the master will assert sINTA and read the data from the DI bus by asserting pDBIN. Because the data is a CALL op-code, the master will leave sINTA asserted and read the next byte by asserting pDBIN. The same procedure occurs to read the last byte.

If an 8080 or Z80 is the CPU on the master and an interrupt occurs, the master will assert sINTA and read the data by asserting pDBIN. So far the actions of the 8085 and 8080/Z80 masters have been the same. Because the data was a CALL op-code, the master will try to fetch the next two bytes of data. However, the 8080/Z80 master will no longer assert sINTA, but will instead assert sMEMR and read the data by asserting pDBIN, just as if it were fetching the next two bytes from memory. The interrupt controller slave card will no longer drive the data bus during pDBIN because the interrupt acknowledge status is no longer present (causing the slave to deselect). Some memory cards (depending on the random address on the address bus) will select and drive the data bus instead. The result is chaos, most likely causing the system to crash.

Something has to be done to trick the interrupt controller slave to remain selected for the entire three-byte fetch, and at the same time insure that all the system memory is deselected.

Interfacing to the 8259A

An 8259A priority interrupt circuit for the S-100 Bus is shown in Figure 13-10. To understand how the circuit works, realize that the next cycle that will be executed after the three bytes of the CALL instruction have been fetched is a memory write cycle. This occurs because the CPU is pushing the return address onto the stack, which is a memory write operation. A memory write operation is signified on the bus by sWO* being asserted. (sWO* is also asserted for an output, but we can assume in this case that an output cycle will not occur immediately following an interrupt acknowledge cycle.)

Here's how the circuit works: pSTVAL* is inverted and applied to one input of AND gate C. The other input is pSYNC. When pSYNC is high and pSTVAL* is low, the output of the AND gate will go high. This output is a pulse that indicates that the address and status buses are now valid. This signal is applied to one input of NAND GATES A and B.

sINTA is applied to the other input of NAND gate A so that the output of gate A will go low for the duration of the pulse from C, when sINTA is high. This will occur during the first interrupt acknowledge cycle which will set the flip-flop. The Q output will go high and is inverted by the 7406 which drives the PHANTOM* line low, disabling system memory. The $\overline{Q}$ output is applied to one input of gate D. When pDBIN is asserted the other input of gate D will go low, causing the output to go low. This becomes the INTA* signal to the 8259A. Since the output of the

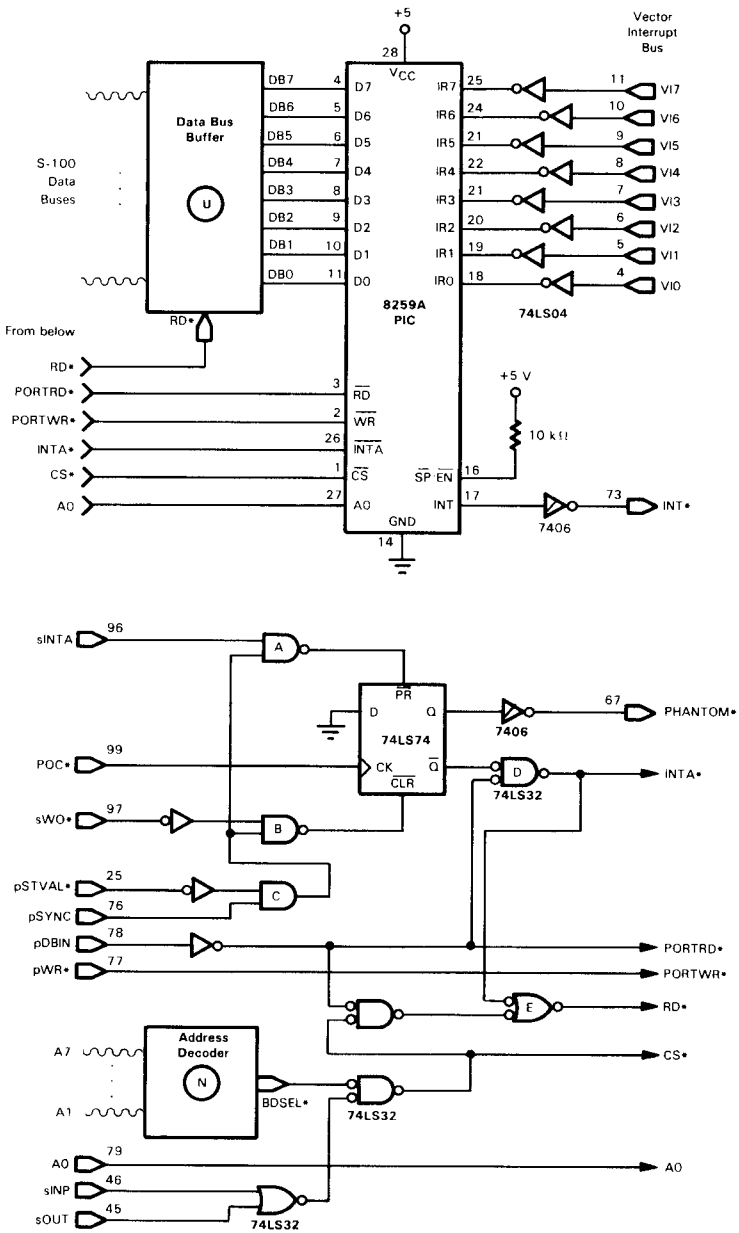


FIGURE 13-10. Priority Interrupt System Using the 8259A

flip-flop is latched, the next two pDBIN pulses will also cause INTA* to go low, causing the next two bytes to be issued from the 8259A. PHANTOM* will also remain asserted so that the system memory will not respond. The INTA* signal is also applied to gate E so that the board's data bus drivers will be turned on in the proper direction.

The flip-flop will be cleared when sWO* is asserted and there is a "status valid" pulse at the input to gate B, causing B's output to go low which resets the flip-flop. This will inhibit further pDBIN signals from asserting the INTA* line, and release the PHANTOM* line as well, allowing system memory to be active again.

A counter could have been used to count three pDBINs after sINTA* is first asserted, but the 8259A has a mode that allows it to work with the 8086/8088 series of processors. These processors issue only two INTA* pulses, and this circuit allows the 8259A to work with any number of pulses. The 8088/86 processor is currently the most popular 16-bit processor on the S-100 Bus, but this circuit should also be compatible with the Z8000 interrupt structure.

```
            ;ROUTINES FOR THE 8259A
            ;FIRST ROUTINE IS TO INITIALIZE THE 8259A
            ;SETS INTERRUPT VECTOR TABLE TO 200H
            ;ROUTINES AT 200H VECTOR TO A SERVICE ROUTINE (SERV)
            ;IN THIS EXAMPLE, ALL THE SERVICE ROUTINES ARE THE SAME
            ;IN THE REAL WORLD, YOU'D WANT THEM ALL DIFFERENT

            ;ALL I/O PORTS ARE RELATIVE TO "BASE"

0010 =      BASE    EQU    10H
0010 =      PORT0   EQU    BASE
0011 =      PORT1   EQU    BASE+1
0300 =      SERV    EQU    300H          ;DUMMY SERVICE ROUTINE
            ;
0100                ORG    100H
            ;
0100 3E1F   INIT    MVI    A,00011111B   ;SET FOR A5-7 OF SERVICE ROUTINE=0
                                          ;LEVEL TRIGGERED, ROUTINE INTERVAL=4
                                          ;SINGLE 8259, AND ICW4 NEEDED
0102 D310           OUT    PORT0         ;SEND IT (THIS IS ICW1)
0104 3E02           MVI    A,02H         ;UPPER BYTE OF SERVICE ROUTINE TABLE
0106 D311           OUT    PORT1         ;SEND IT (ICW2)
0108 3E00           MVI    A,00H         ;8259 HAS NO SLAVES
010A D311           OUT    PORT1         ;SEND IT (ICW3)
010C 3E02           MVI    A,00000010B   ;SET NOT FULLY NESTED, NON-BUFFERED MODE
                                          ;USE AUTO END OF INTERRUPT AND NOT 8086
010E D311           OUT    PORT1         ;SEND IT (ICW4)
            ;
0110 3E00           MVI    A,00H         ;ENABLE ALL INTERRUPTS (ALL MASK BITS=0)
0112 D311           OUT    PORT1         ;SEND IT (OCW1)
0114 3EA0           MVI    A,10100000B   ;SET AUTO ROTATING PRIORITY MODE
0116 D310           OUT    PORT0         ;SEND IT (OCW2) NO OCW3 NEEDED
0118 FB             EI                   ;ENABLE INTERRUPTS
                    ;JMP TO MAIN SYSTEM TASK, NO MORE INITIALIZATION NEEDED
            ;
0200                ORG    200H          ;SERVICE ROUTINE JUMP TABLE
                                          ;PUT JUMP EVERY FOUR BYTES
0200 C30003 ISR0    JMP    SERV
0203 00             NOP
0204 C30003 ISR1    JMP    SERV
0207 00             NOP
0208 C30003 ISR2    JMP    SERV
020B 00             NOP
020C C30003 ISR3    JMP    SERV
```

```
020F 00           NOP
0210 C30003 ISR4  JMP    SERV
0213 00           NOP
0214 C30003 ISR5  JMP    SERV
0217 00           NOP
0218 C30003 ISR6  JMP    SERV
021B 00           NOP
021C C30003 ISR7  JMP    SERV
021F 00           NOP
          ;
0300              ORG    SERV          ;LOCATION OF DUMMY SERVICE ROUTINE
                  ;THESE ROUTINES SHOULD FIRST PUSH ALL THE REGISTERS
                  ; THAT THE ROUTINE WILL USE, THEN SERVICE THE PERIPHERAL,
                  ; POP ALL THE REGISTERS, THEN...
          ;
0300 FB           EI                   ;REENABLE THE INTERRUPTS
0301 C9           RET                  ;AND RETURN TO THE MAIN SYSTEM TASK
```

8085 INTERRUPT SYSTEM

The 8085 has the same interrupt operation as the 8080. Further, it has four interrupt request inputs in addition to the INTR input. These additional interrupt inputs are labelled RST 5.5, RST 6.5, RST 7.5, and TRAP. The TRAP input has the highest priority (for simultaneously occurring interrupts), RST 7.5 second highest priority, and so on. INTR has the lowest priority. The INTR input functions in the same manner as the INT input of the 8080. The RST inputs may be connected to the VI lines. The TRAP input of the 8085 is typically connected to the NMI* line of the S-100 Bus.

Each of the RST inputs, 5.5, 6.5, and 7.5, has a programmable bit mask which allows each input to be selectively enabled or disabled using the SIM (Set Interrupt Mask) instruction. It is also possible to read the state of the mask using the RIM (Read Interrupt Mask) instruction. The TRAP input is nonmaskable, and can be used for power failure or bus error detection.

The 8085 interrupt inputs cause a restart at the following addresses:

Priority	Input	Name	Address called
Highest	TRAP	TRAP	24
	RST 7.5	RST 7.5	3C
	RST 6.5	RST 6.5	34
	RST 5.5	RST 5.5	2C
Lowest	INTR	RST 0 - RST 7, same as 8080	

The RST 5.5 and RST 6.5 inputs respond to high-level inputs. The RST 7.5 input responds to a positive-going edge which sets an internal interrupt request flip-flop. The flip-flop is reset when the interrupt is serviced or by a SIM instruction or a RESET to the 8085. The TRAP input responds to a high-level input or a positive-going edge. Since the S-100 interrupt lines are active low, these must be inverted.

Z80 INTERRUPT SYSTEM

The Z80 has two interrupt request inputs, INT* and NMI*. NMI* is a non-maskable, negative-edge triggered interrupt input which has priority over both the INT* interrupt and bus requests. Enabling NMI* causes a CALL to memory location 0066_{16}. No RST vector is needed on the data bus. NMI* is typically connected to the S-100 NMI* bus line.

INT* functions in the same manner as the INT* input on the 8080. This input is connected to the INT* line of the S-100 Bus if there is no vectored interrupt circuit on the CPU board. Further, the interrupt response to INT* operates in one of three possible modes: mode 0, 1, or 2. The mode is established using the IM instruction. When the processor is RESET the interrupts are set to operate in mode 0. Mode 0 is the same as the 8080 interrupt operation.

When the Z80 is set for mode 1 operation, on receiving an interrupt it immediately vectors to memory location 0066_{16}. No external interrupt vector need be provided. When in mode 2 operation, you must first create a table of 16-bit interrupt address vectors, which can reside in any page of memory and then initialize the I register. On receiving an interrupt via INT*, the CPU reads the data bus (with memory off), as does the 8080, for the interrupt acknowledge vector. The Z80 then combines the 8-bit data word with the contents of the 8-bit I register to form the 16-bit address of the interrupt response vector. The CPU then reads the data at the vector address and vector address + 1 to obtain the effective memory location to vector to.

The interrupt address table in memory can therefore consist of up to 128 two-byte address vectors. The interrupt response vector read from the data bus must be even, with data bit 0 set to 0, since two data bytes are accessed at one time. For example, if the I register contains FF_{16} and in response to an interrupt the CPU reads a 00_{16} from the data bus, the CPU reads the 2-byte data at locations $FF00_{16}$ and $FF01_{16}$ as the actual interrupt vector address for the ISR.

Again a word of caution is in order. When considering a system with a large number of interrupts it will probably prove wise to break the system up into smaller ones, each having its own processor, or to use a polling technique. These alternatives are far easier to program and debug. Further, the software housekeeping of a multiple-interrupt system may negate the usual time savings of an interrupt system over a polled system.

POLLED INTERRUPTS

In some systems it is desirable to use interrupts in a "polled mode." A polled interrupt system usually causes only one "master" interrupt. The software then "polls" the interrupt controller or individual interrupt sources to identify the source of the interrupt.

The 8259A can be used in the polled mode by setting a bit in one of its mode registers. This causes the 8259A to ignore INTA* pulses, and it also generates no CALL instructions. Instead, it waits for the next read from the device and creates a special byte that contains a code that corresponds to the highest priority interrupting device. The service routine then uses this code to determine what service routine to branch to.

Since no CALL instructions are issued by the 8259A, the S-100 interface circuitry is a lot simpler. Figure 13-11 shows a circuit that can replace the S-100 interface portion of the schematic in Figure 13-10 that allows the 8259A to be used in the polled mode. Like the simple interrupt system presented in Figure 13-3, this circuit assumes the Data Input bus will float high during the interrupt acknowledge cycle and the CPU will interpret this as an RST 7. (This is not recommended for unterminated motherboards.) The circuit in Figure 13-5 could be added if something other than an RST 7 is desired.

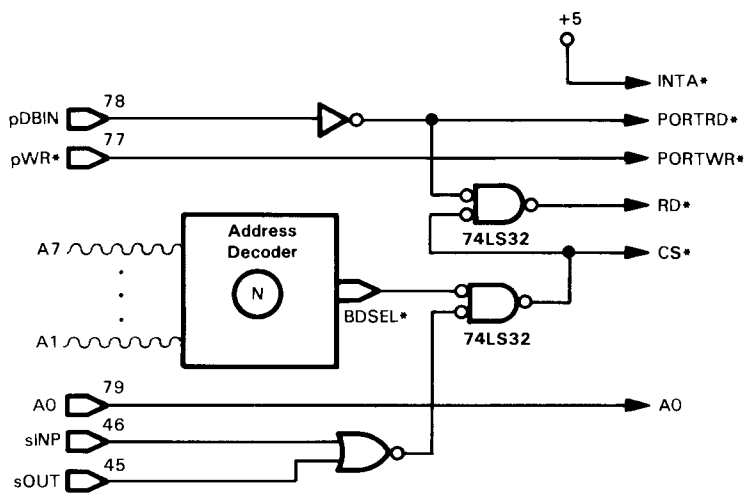


FIGURE 13-11. Alternative S-100 Interface for 8259A Using Polled Mode (See Figure 13-10)

The following program can be used with the circuit for an eight-level polled interrupt system. The program assumes that an RST 7 will be generated. The program consists of two parts: one is the routine to initialize the 8259A, which is executed only once, and the other is the routine that reads the byte from the 8259A and then calculates a jump vector from it.

```
                ;ROUTINES FOR THE 8259A WHEN USED IN THE POLLED MODE
                ;FIRST ROUTINE IS TO INITIALIZE THE 8259A
                ;SETS INTERRUPT VECTOR TABLE TO 200H, BUT THAT IS A "DUMMY" VALUE
                ;ASSUMES AN RST 7 INTERRUPT ROUTINE AT 38H
                ;THE "VECTOR GENERATION" ROUTINE READS THE PRIORITY STATUS BYTE
                ;FROM THE 8259A AND USES THAT TO GENERATE A VECTOR TO A TABLE
                ;OF JUMPS TO THE ACTUAL SERVICE ROUTINES
                ;TO CHANGE THE ADDRESS OF THE TABLE, CHANGE "TABLE" IN EQUATES
                ;JUMPS IN THE TABLE ARE SPACED 4 BYTES APART
                ;
                ;ALL I/O PORTS ARE RELATIVE TO "BASE"
                ;
0010 =          BASE    EQU     10H
0010 =          PORT0   EQU     BASE
0011 =          PORT1   EQU     BASE+1
0300 =          SERV    EQU     300H            ;DUMMY SERVICE ROUTINE
0200 =          TABLE   EQU     200H

0000                    ORG     0H

0000 C30001     START   JMP     100H            ;JUMP TO INIT ROUTINE

0038                    ORG     38H             ;RST 7 ADDRESS
                ;
0038 F5         VCTRGN  PUSH    PSW
0039 D5         PUSH    D
003A E5         PUSH    H
003B 3E0C       MVI     A,00001100B             ;POLL COMMAND TO 8259A
003D D310       OUT     PORT0                   ;SEND IT (OCW3)
003F DB10       IN      PORT0                   ;READ THE PRIORITY STATUS
0041 07         RLC                             ;SHIFT BIT 7 INTO CARRY
0042 D25200     JNC     EXIT                    ;NO INTERRUPT, LEAVE
0045 07         RLC                             ;THERE WAS AN INTERRUPT
                                                ;SO WE SHIFT A AGAIN TO OFFSET THE BITS
0046 210002             LXI     H,TABLE         ;SO WE CAN USE IT FOR THE LOW BYTE
0049 5F                 MOV     E,A             ;OF THE TABLE BY PUTTING IT IN E
004A 1600               MVI     D,0             ;ZERO HIGH BYTE FOR ADD
004C 19                 DAD     D               ;ADD DE TO HL, RESULT IN HL
004D 115200             LXI     D,EXIT          ;SNEAKY WAY TO DO AN
0050 D5                 PUSH    D               ;INDIRECT CALL
0051 E9                 PCHL                    ;TO ADDRESS IN HL
0052 E1         EXIT    POP     H               ;WITH A RETURN TO HERE
0053 D1                 POP     D               ;POP THE REGISTERS
0054 F1                 POP     PSW             ;FOR THE RETURN
0055 FB                 EI                      ;ENABLE THE INTERRUPTS
0056 C9                 RET                     ;AND RETURN
                ;
                ;
0100                    ORG     100H            ;INITIALIZE THE 8259A

0100 3E1F       INIT    MVI     A,00011111B     ;SET FOR A5-7 OF SERVICE ROUTINE=0
                                                ;LEVEL TRIGGERED, ROUTINE INTERVAL=4
                                                ;SINGLE 8259, AND ICW4 NEEDED
0102 D310               OUT     PORT0           ;SEND IT (THIS IS ICW1)
0104 3E02               MVI     A,02H           ;UPPER BYTE OF SERVICE ROUTINE TABLE
0106 D311               OUT     PORT1           ;SEND IT (ICW2)
0108 3E00               MVI     A,00H           ;8259 HAS NO SLAVES
010A D311               OUT     PORT1           ;SEND IT (ICW3)
010C 3E02               MVI     A,00000010B     ;SET NOT FULLY NESTED,
                                                ;  NON-BUFFERED MODE
                                                ;USE AUTO END OF INTERRUPT
```

```
                                        ;  AND NOT 8086
010E D311           OUT    PORT1        ;SEND IT (ICW4)

0110 3E00           MVI    A,00H        ;ENABLE ALL INTERRUPTS
                                        ; (ALL MASK BITS=0)
0112 D311           OUT    PORT1        ;SEND IT (OCW1)
0114 3EA0           MVI    A,10100000B  ;SET AUTO ROTATING PRIORITY MODE
0116 D310           OUT    PORT0        ;SEND IT (OCW2) NO OCW3 NEEDED
0118 FB             EI                  ;ENABLE INTERRUPTS
             ;JMP TO MAIN SYSTEM TASK, NO MORE INITIALIZATION NEEDED
             ;
0200                ORG    TABLE        ;SERVICE ROUTINE JUMP TABLE
             ;                          ;PUT JUMP EVERY FOUR BYTES
0200 C30003 ISR0    JMP    SERV
0203 00             NOP
0204 C30003 ISR1    JMP    SERV
0207 00             NOP
0208 C30003 ISR2    JMP    SERV
020B 00             NOP
020C C30003 ISR3    JMP    SERV
020F 00             NOP
0210 C30003 ISR4    JMP    SERV
0213 00             NOP
0214 C30003 ISR5    JMP    SERV
0217 00             NOP
0218 C30003 ISR6    JMP    SERV
021B 00             NOP
021C C30003 ISR7    JMP    SERV
021F 00             NOP
             ;
0300                ORG    SERV         ;LOCATION OF DUMMY SERVICE ROUTINE

             ;THESE ROUTINES SHOULD FIRST PUSH ALL THE REGISTERS
             ;  THAT THE ROUTINE WILL USE, THEN SERVICE THE PERIPHERAL
             ;  POP ALL THE REGISTERS, THEN...
             ;
0300 C9             RET    ;... RETURN TO THE VECTOR GENERATOR ROUTINE
```

MAKING USE OF INTERRUPTS

Interrupts are a useful addition to any system where high throughput is essential, or where you wish to have some background task occurring concurrent with the main task you are performing.

Higher throughput is achieved with interrupts because the CPU can be doing something other than sitting in a loop waiting for a key to be pressed or a character to be typed. If the peripheral or slave device is intelligent, meaning that it can do some processing of its own, it can perform a task while the CPU is doing something else. This is known as "parallel processing." The interrupt in this case would occur when the peripheral had completed its task.

In a multi-user system (where high throughput is essential), interrupts are almost always a necessity. The CPU will usually get a recurring interrupt every millisecond or so to tell it to quit working for this user and work for the next user until the next interrupt occurs, and so on. Thus every user gets an equal share of the processor's time. This type of interrupt is called an "interval clock interrupt."

Another common use of the interval clock is to keep track of the "real time" (hours, minutes, date, etc.). If one knows the interval between interrupts, then

seconds, minutes, hours, etc. can be computed. The interval clock is used so often to keep track of real time that many people call it a "real time clock." In the strictest sense, this is a misnomer because it is the computer that computes real time, not the clock itself. An actual "real time clock," therefore, is a circuit that acts like a wristwatch or wall clock because it keeps track of the real time (and usually the date) all by itself. The computer can then read the date on command without the need to do any computing.

For more information on interval clocks and a real time clock routine, see Chapter 14, which discusses programmable timer/counters.

POWER FAILURE INTERRUPT

A power failure can have disastrous results in some systems. In such a case it is wise to consider a power failure interrupt. The power failure interrupt is usually activated at least 50 ms before the local voltage regulators drift out of specification. This is usually enough time to execute a hundred or more instructions.

Typically, on sensing power failure the interrupt service routine saves all registers, the stack, and possibly data buffers on disk. On restoration of power, the system is initialized, all data is loaded from the disk back into registers, stack, and buffers, and the CPU returns to the execution of the program.

The PWRFAIL* bus line (pin 13) is held low until the power-on clear signal is activated. This means that either a normally closed relay or a battery powered circuit drives the PWRFAIL* line. The circuit driving this line must meet the electrical specifications for an open collector circuit given in the IEEE specifications.

REFERENCES

Jigour, Robin. *Using the 8259A Programmable Interrupt Controller,* AP-59, Intel Corporation.

Leventhal, Lance. *8080A/8085 Assembly Language Programming,* Berkeley: Osborne/McGraw-Hill, 1978.

Leventhal, Lance. *Z80 Assembly Language Programming,* Berkeley: Osborne/McGraw-Hill, 1979.

Osborne, Adam. *An Introduction to Microcomputers, Volumes 1, 2 and 3,* Berkeley: Osborne/McGraw-Hill, 1976, 1979.

programmable timer/counters

14

In previous chapters we saw numerous examples in which precise time intervals or frequencies were developed by the processor as part of processing operations. These invariably required the use of delay loops and counting registers. These routines often caused the CPU to spend most of its time in a counting loop. This reduced the CPU's useful processing time to a small percentage of its total time. This is a distinct disadvantage. Further, these delay time routines add to the software size and complexity and are dependent on the speed of the CPU.

It is therefore advantageous, particularly when it is necessary to increase system data throughput, to relegate this timing/counting function to hardware. Fortunately, several manufacturers produce timer/counter ICs specifically designed for use with microprocessors.

The timer/counter circuit, when programmed to operate as a timer, is triggered by either an internal or external clock signal. Also, most of the timer/counter ICs allow the timer/counter circuits to be triggered by an external signal, and hence can be used as event counters. These circuits may also be used as interval timers, precisely measuring the time between events. These devices are directly addressable as ports or memory locations and usually contain more than one timer/counter, allowing simultaneous or overlapping delays to be generated.

We will look at some of these ICs and some timer/counter applications in this chapter.

PROGRAMMABLE COUNTER/INTERVAL TIMERS

The Intel 8253 IC is an example of a powerful and flexible programmable counter/interval timer device. Other ICs in this class are the Motorola MC6840 and Zilog CTC.

The 8253 contains three independent 16-bit counter/interval timers, as shown in Figure 14-1. Each counter operates as a presettable down-counter capable of either straight binary or binary-coded-decimal (BCD) counting. It can be operated in six different modes, as follows:

Mode	Function
0	Output = 1 on terminal count
1	Programmable one-shot
2	Rate generator; divide-by-N
3	Square wave generator
4	Software triggered strobe
5	Hardware triggered strobe

The counters are independent, so any combination of modes may be used. Additionally, the current value of each counter may be read by the CPU.

Each counter has clock, gate, and output lines. Although their functions vary with the operating mode, they generally operate as follows:

CLK Supplies the events to be counted or serves as a reference timing signal. The counter decrements on the falling edge of the CLK input (3 MHz maximum).

GATE Either inhibits or enables counting.

OUT Indicates the terminal count or supplies the divided CLK output.

Each counter is controlled by writing a control word to the control register and then loading the counter with the desired count value. The format of the control word is shown in Figure 14-2. Bits 6 and 7 specify which counter is being configured. Bits 4 and 5 specify which counter byte is being addressed. Bits 1, 2, and 3 select the operating mode. Bit 0 selects either binary or BCD counting.

A typical S-100 to 8253 interface circuit is shown in Figure 14-3. The 8253 is selected as four consecutive I/O ports, as follows:

OUT 10 = load counter 0
OUT 11 = load counter 1
OUT 12 = load counter 2
OUT 13 = load control word
IN 10 = read counter 0
IN 11 = read counter 1
IN 12 = read counter 2

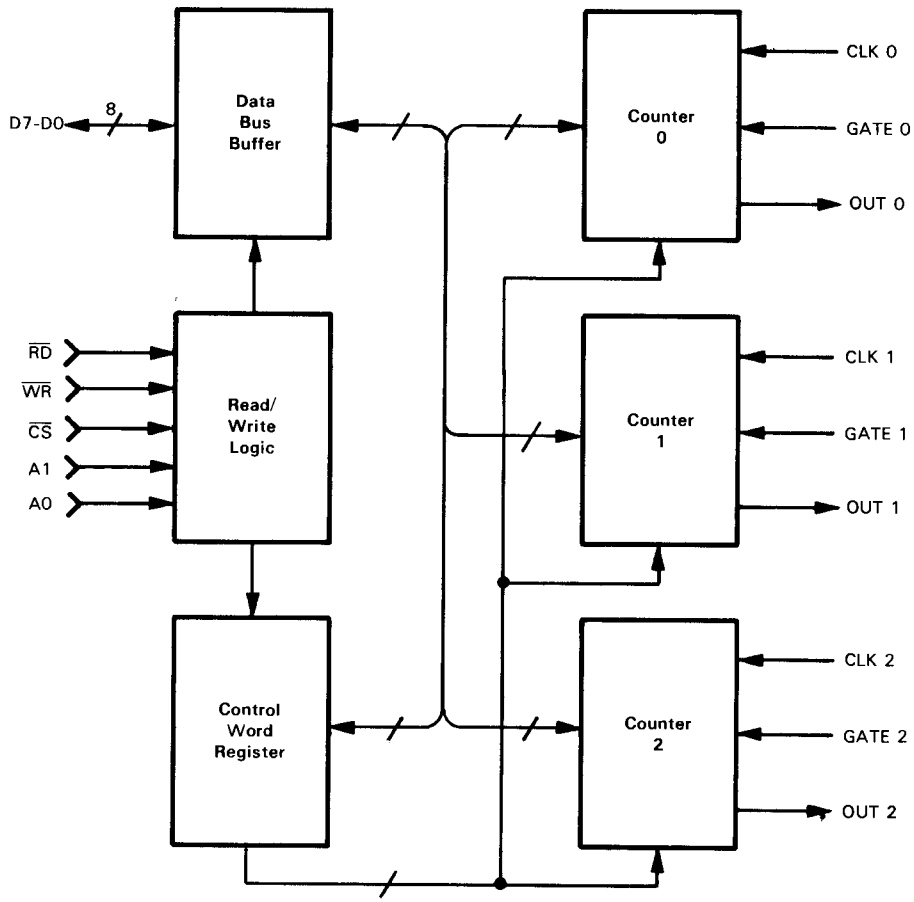


FIGURE 14-1. Block Diagram of 8253 Programmable Interval Timer

Mode 0 operation. This mode can be used to provide programmed time intervals. This is accomplished by connecting a timer output to an interrupt request input. The CLK may be connected to the S-100 CLOCK signal. Control bits 4 and 5 are then used to load the counter register as follows:

B5	B4	Function
0	1	Load low byte
1	0	Load high byte
1	1	Load low byte first, then load high byte

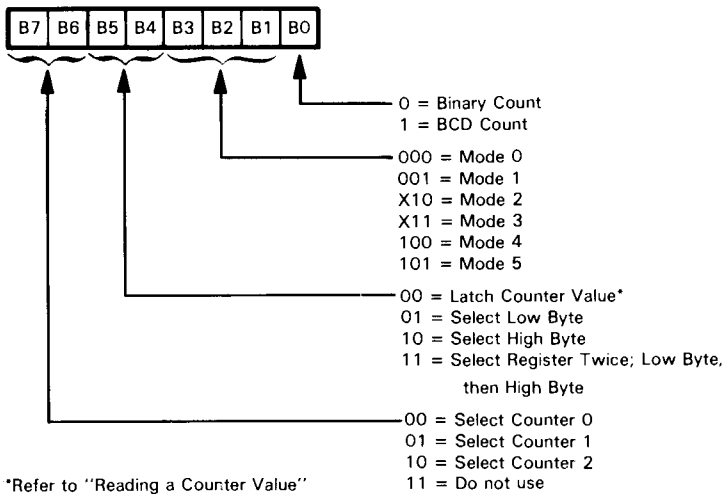


FIGURE 14-2. 8253 Control Word Format

The following routine causes counter 0 to count 128 binary states before the OUT terminal goes high.

```
                ;ROUTINE TO INITIALIZE COUNTER AND HAVE IT COUNT TO 128
                ;IN BINARY FORMAT, TIMER MODE = 0
                ;COUNT = COUNTDOWN VALUE; LXI H,COUNT PUTS 16 BIT # IN HL
                ;LETS ASSEMBLER DO DECIMAL TO BINARY CONVERSION
                ;TIMER USES FOUR PORTS, BASE=FIRST PORT
                ;TO USE WITH OTHER PORT ADDRESSES, CHANGE BASE
                ;
0010 =          BASE    EQU  10H          ;BEGINNING PORT ADDRESS
0010 =          CNTR0   EQU  BASE         ;COUNTER 0 REGISTER
0011 =          CNTR1   EQU  BASE+1       ;COUNTER 1 REGISTER
0012 =          CNTR2   EQU  BASE+2       ;COUNTER 2 REGISTER
0013 =          CONTROL EQU  BASE+3       ;CONTROL WORD REGISTER
0080 =          COUNT   EQU  128          ;GETS COUNTDOWN AMOUNT

0000 3E30       START   MVI  A,00110000B ;SELECT COUNTER 0, TWO BYTE LOAD
                                         ;MODE 0 AND BINARY FORMAT
0002 D313               OUT  CONTROL     ;OUTPUT TO THE CONTROL PORT
0004 218000             LXI  H,COUNT     ;PUT COUNT VALUE IN HL
0007 7D                 MOV  A,L         ;PUT LOW BYTE IN A
0008 D310               OUT  CNTR0       ;SEND IT TO COUNTER
000A 7C                 MOV  A,H         ;PUT HIGH BYTE IN A
000B D310               OUT  CNTR0       ;SEND IT AND START COUNTING
                ;DONE
```

The following routine causes counter 1 to count 308 BCD states before the OUT terminal goes high.

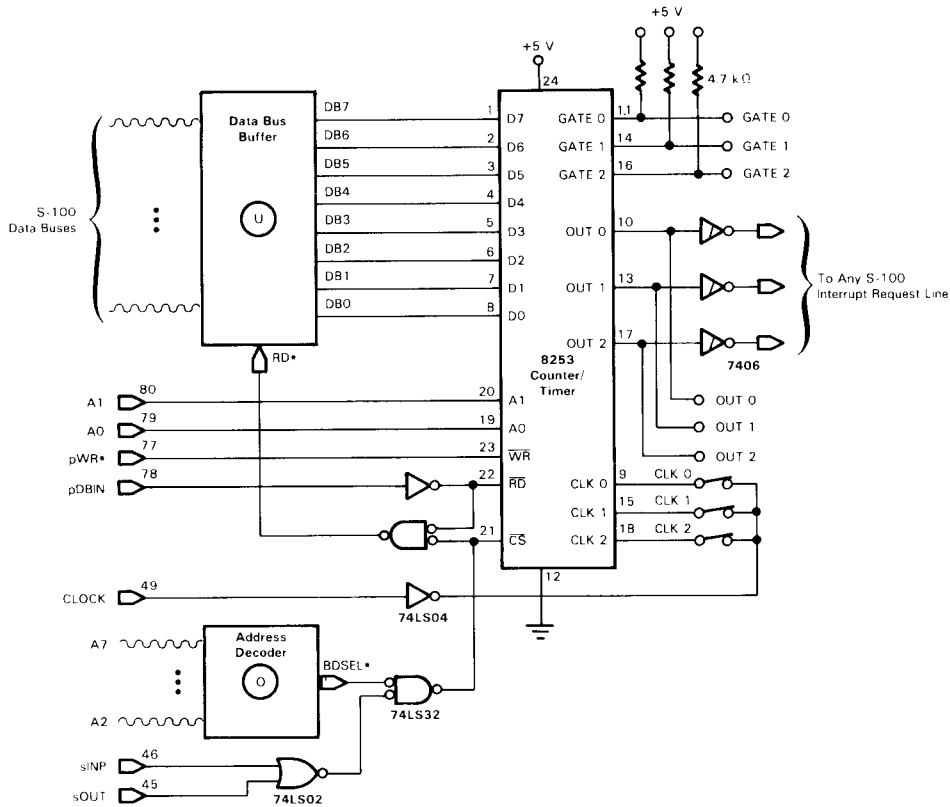


FIGURE 14-3. 8253 Timer/Counter to S-100 Interface With Interrupt Capability

When the terminal count (0000_{16}) is reached, the OUT line goes high and remains high until the counter is reloaded. Reloading a counter during a count causes the counter to start decrementing from the new value loaded into the counter. The GATE input can be used to enable (H) and disable (L) the counting.

```
                ;ROUTINE TO INITIALIZE COUNTER AND HAVE IT COUNT TO 308
                ;IN BCD FORMAT, TIMER MODE=0
                ;COUNT=COUNTDOWN VALUE, DIVIDED INTO TWO BCD NUMBERS
                ;WHICH ARE HICNT AND LOCNT AND
                ;LETS ASSEMBLER DO ALL THE WORK
                ;TIMER USES FOUR PORTS, BASE=FIRST PORT
                ;TO USE WITH OTHER PORT ADDRESSES, CHANGE BASE

0010 =          BASE    EQU  10H    ;BEGINNING PORT ADDRESS
```

```
0010 =     CNTR0    EQU  BASE    ;COUNTER 0 REGISTER
0011 =     CNTR1    EQU  BASE+1  ;COUNTER 1 REGISTER
0012 =     CNTR2    EQU  BASE+2  ;COUNTER 2 REGISTER
0013 =     CONTROL  EQU  BASE+3  ;CONTROL WORD REGISTER
0134 =     COUNT    EQU  308     ;GETS COUNTDOWN AMOUNT
0003 =     HICNT    EQU  COUNT/100
0008 =     LOCNT    EQU  COUNT-HICNT*100
           ;
0000 3E71  START    MVI  A,01110001B ;SELECT COUNTER 1, TWO BYTE LOAD
                                    ;MODE 0 AND BCD FORMAT
0002 D313           OUT  CONTROL     ;OUTPUT TO THE CONTROL PORT
0004 3E08           MVI  A,LOCNT     ;PUT LOW BYTE IN A
0006 D311           OUT  CNTR1       ;SEND IT TO COUNTER 1
0008 3E03           MVI  A,HICNT     ;PUT HIGH BYTE IN A
000A D311           OUT  CNTR1       ;SEND IT AND START COUNTING
           ;DONE
```

Mode 1 operation. This provides a programmable one-shot, as shown in Figure 14-4. The OUT line goes low on the count following the rising edge of the GATE input. The OUT line returns high on terminal count. If a new count value is loaded into the counter before the terminal count is reached, the duration will not be affected until the next gate trigger (Figure 14-4b). The one-shot can be retriggered to extend the output pulse width (Figure 14-4c).

Mode 2 operation. This provides a programmable rate generator. The OUT line goes low for one clock period, as shown in Figure 14-5. It returns high until the counter reaches 0000_{16}. The counter automatically reloads and repeats the operation. If the counter is reloaded between output pulses the present period is not affected, but subsequent periods reflect the new value. When GATE is low the counter is disabled and OUT will be high. The counter is reset to the initial value so that when GATE goes high the counter restarts. Thus the GATE input can be used to synchronize the counter.

Mode 3 operation. This provides a square wave rate generator. Operation is the same as Mode 2 except that the OUT line will be high for one of half the count and low for the other half. If the count value is not an even number then OUT will be high for an extra count.

Mode 4 operation. This provides a software triggered strobe. OUT will normally be high. When the terminal count is reached, OUT will go low for one clock period and then return high. Reloading the count between OUT pulses causes the subsequent period to reflect the new value. When GATE is low counting is inhibited.

Mode 5 operation. This provides a hardware triggered strobe. OUT will normally be high. The counter starts counting on the rising edge of the GATE input and OUT will go low for one clock period when the terminal count is reached. The counter is retriggerable.

Reading a counter value. The 16-bit count value of any counter may be read at any time. If reading while the counter is counting, a write command must be executed first to latch the count into a storage register from which the value can then be read. This allows reading the count value "on the fly."

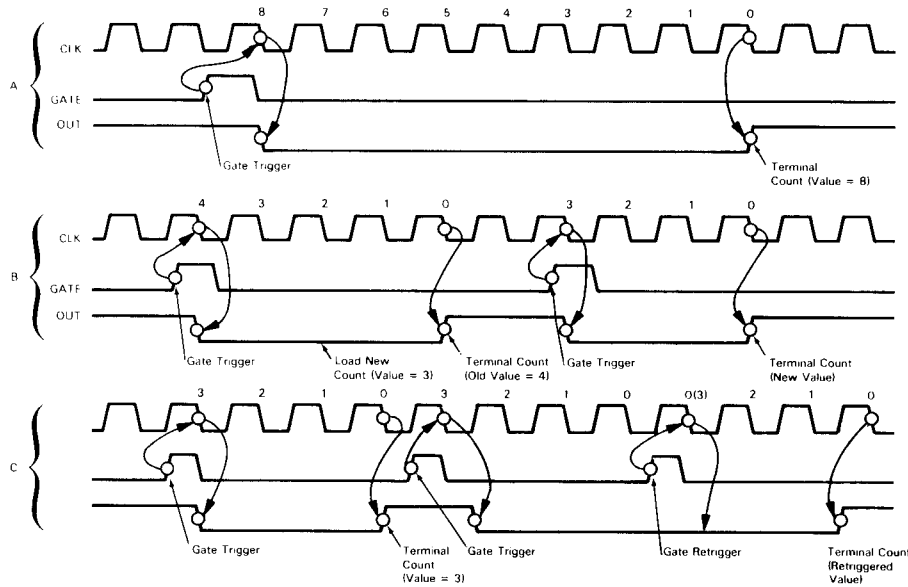


FIGURE 14-4. 8253 Mode 1 Operation: (A) Normal One-Shot Operation, (B) New Count Value Loaded During Count, (C) Retrigger Operation

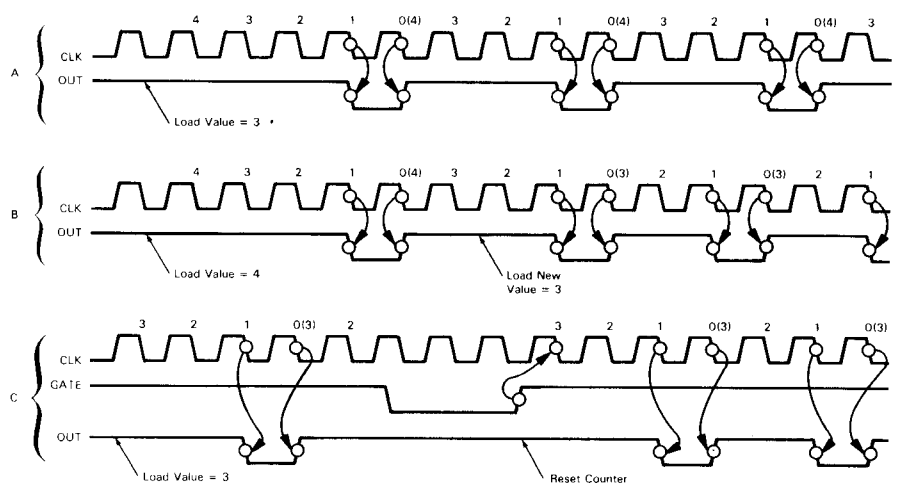


FIGURE 14-5. 8253 Mode 2 Operation: (A) Normal Rate Generation, (B) New Count Value Loaded During Count, (C) Reset Operation

To read a count value that is not changing it is necessary to read the counter only twice. For example, the following routine reads the value of counter 0 and saves it in the BC register pair.

```
                ;ROUTINE FOR READING 8253 COUNTER-0 AND SAVING
                ;IT IN BC REGISTERS
                ;
                ;
0010 =          BASE    EQU  10H     ;BEGINNING PORT ADDRESS
0010 =          CNTR0   EQU  BASE    ;COUNTER 0 REGISTER
                ;
                ;
0100                    ORG  100H
                ;
0100 DB10               IN   CNTR0   ;READ COUNTER 0 LOW BYTE
0102 4F                 MOV  C,A     ;SAVE IT
0103 DB10               IN   CNTR0   ;READ COUNTER 0 HIGH BYTE
0105 47                 MOV  B,A     ;SAVE IT
```

To read a counter "on the fly," write a control word with bits 4 and 5 = 0 (refer to Figure 14-3) while bits 6 and 7 specify the counter to be latched. Bits 0-3 are not used. In the following example counter 1 is read "on the fly" and its value saved in the BC register pair.

```
                ;ROUTINE FOR READING 8253 COUNTER 1 ON-THE-FLY
                ;AND SAVING IT IN BC REGISTERS
                ;
                ;
0010 =          BASE      EQU  10H       ;STARTING PORT ADDRESS
0011 =          CNTR1     EQU  BASE+1    ;COUNTER 1 REGISTER PORT
0013 =          CNTRL     EQU  BASE+3    ;CONTROL REGISTER PORT
                ;
                ;
0100                      ORG  100H
                ;
0100 3E40                 MVI  A,40H     ;SET CODE TO LATCH COUNTER 1
0102 D313                 OUT  CNTRL
0104 DB11                 IN   CNTR1     ;READ COUNTER 1 LOW BYTE
0106 4F                   MOV  C,A       ;SAVE IT
0107 DB11                 IN   CNTR1     ;READ COUNTER 1 HIGH BYTE
0109 47                   MOV  B,A       ;SAVE IT
```

APPLICATIONS OF PROGRAMMABLE TIMER/COUNTERS

Programmable timer/counter ICs lend themselves to a wide variety of applications: event counters, baud rate generators, frequency generators, rate multipliers, real time clocks, complex motor controllers, etc. The following are three typical examples, utilizing the 8253 IC.

Measuring Speed

A timer, in conjunction with a CPU, can be used to measure time intervals with great accuracy. In this example we will measure a camera's shutter "speed" from 1/1000 of second to 10 seconds. The circuitry is quite simple and is shown in Figure 14-6. A phototransistor is used to sense light passing through the open camera shutter. A 7414 Schmitt Trigger Inverter is used to minimize noise interference. The S-100 crystal-controlled clock signal is used as the reference.

Counter 0 divides the 2 MHz clock signal by 2000 to produce 1 kHz clock inputs for counters 1 and 2. Counters 1 and 2 decrement from their programmed count values. Counter 1 will stop counting when the shutter closes, and hence measures the time period of the open shutter (shutter "speed"). Since the counter counts down in binary it holds a value that is the difference between the initial programmed value (2710_{16} = $10{,}000_{10}$). Counter 2 will continue to count to its terminal value and then produce an interrupt. This occurs when 10 seconds have elapsed.

The interrupt routine reads the binary count value of counter 1 and subtracts it from 2710_{16} (10,000 decimal) to produce the true time value. Since counter 1 is clocked by the 1 kHz clock it provides shutter "speed" measurement in thousandths of seconds, up to a maximum of 10 seconds.

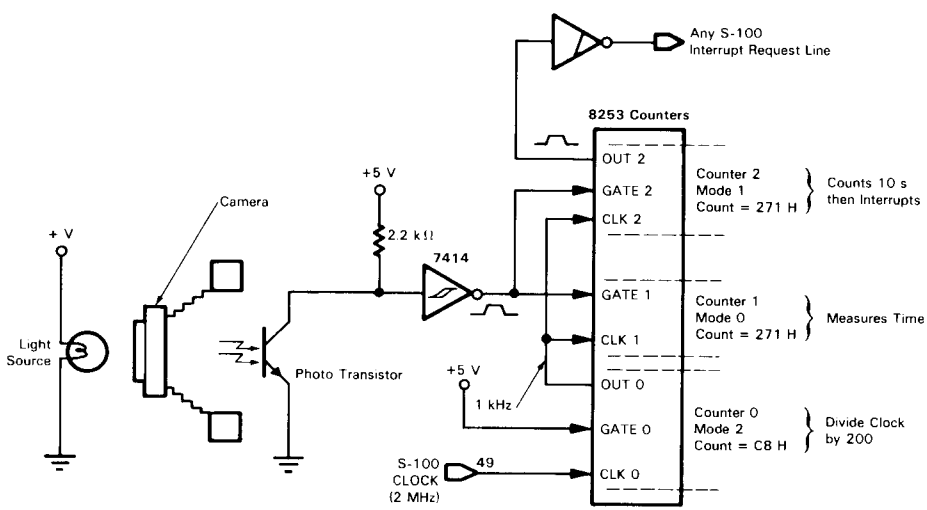


FIGURE 14-6. Using Programmable Counter/Timer to Measure a Camera's Shutter Speed

The measurement process begins by calling the initialization routine, which programs the timer/counters and returns to the main program. The opening of the shutter starts the measuring process, which is completed when timer 2 times out, generating the interrupt. The interrupt service routine reads the time value from counter 1 and does the necessary subtraction, storing the data in a pair of memory locations before returning to the main program.

```
                ;CAMERA SHUTTER SPEED MEASUREMENT PROGRAM
                ;USING 8253 COUNTER/TIMER
                ;
                ;
0010 =      BASE      EQU  10H        ;BEGINNING PORT ADDRESS
0010 =      CNTR0     EQU  BASE       ;COUNTER 0 REGISTER
0011 =      CNTR1     EQU  BASE+1     ;COUNTER 1 REGISTER
0012 =      CNTR2     EQU  BASE+2     ;COUNTER 2 REGISTER
0013 =      CNTRL     EQU  BASE+3     ;CONTROL PORT
1000 =      TABLE     EQU  1000H      ;TABLE POINTER
                ;
                ;
                ;
0100                  ORG  100H
                ;
0100 3E35   INITL     MVI  A,35H      ;CONTROL WORD FOR COUNTER 0
0102 D313             OUT  CNTRL
0104 3EC8             MVI  A,0C8H     ;SET COUNTER 0
0106 D310             OUT  CNTR0      ;  VALUE=2000 COUNTS
0108 3E07             MVI  A,07H
010A D310             OUT  CNTR0
010C 3E70             MVI  A,70H      ;CONTROL WORD FOR COUNTER 1
010E D313             OUT  CNTRL
0110 3E10             MVI  A,10H      ;SET COUNTER 1
0112 D311             OUT  CNTR1      ;  VALUE=10,000 COUNTS
0114 3E27             MVI  A,27H
0116 D311             OUT  CNTR1
0118 3EB2             MVI  A,0B2H     ;CONTROL WORD FOR COUNTER 2
011A D313             OUT  CNTRL
011C 3E10             MVI  A,10H      ;SET COUNTER 2
011E D312             OUT  CNTR2      ;  VALUE=10,000 COUNTS
0120 3E27             MVI  A,27H
0122 D312             OUT  CNTR2
0124 C9               RET
                ;
                ;
                ;INTERRUPT SERVICE ROUTINE
                ;
0125 E5     ISR       PUSH H          ;SAVE REGISTERS
0126 D5               PUSH D
0127 C5               PUSH B
0128 F5               PUSH PSW
0129 210010           LXI  H,TABLE    ;SET TABLE POINTER
012C 111027           LXI  D,2710H    ;SET MINUEND=10,000 DECIMAL
012F A7               ANA  A          ;CLEAR CARRY BIT
0130 DB11             IN   CNTR1      ;READ & STORE TIME VALUE
0132 77               MOV  M,A        ;LSB
0133 23               INX  H
0134 DB11             IN   CNTR1      ;MSB
0136 77               MOV  M,A
0137 79               MOV  A,C        ;FETCH MINUEND LOW-BYTE
0138 2B               DCX  H          ;SUBTRACT TIME VALUE LOW-BYTE
0139 9E               SBB  M
013A 5F               MOV  E,A        ;SAVE DIFFERENCE LOW-BYTE
013B 78               MOV  A,B        ;FETCH MINUEND HIGH-BYTE
                                                    ;PAGE 2
013C 23               INX  H          ;SUBTRACT TIME VALUE HIGH-BYTE
013D 9E               SBB  M
013E 77               MOV  M,A        ;SAVE DIFFERENCE HIGH-BYTE
```

```
013F 2B             DCX  H          ;SAVE VALUES BACK IN TABLE
0140 73             MOV  M,E
0141 CD0001         CALL INITL      ;RESET COUNTERS
0144 F1             POP  PSW        ;RESTORE REGISTERS
0145 C1             POP  B
0146 D1             POP  D
0147 E1             POP  H
0148 FB             EI              ;ENABLE INTERRUPTS
0149 C9             RET
```

Baud Rate Generation

A popular use of programmable counter/timers is for programmable baud rate generation. As such, the timer provides the clock for a UART or USART. Obtaining this clock is a simple matter in an S-100 system. A single timer driven from the S-100 CLOCK signal can be programmed to provide the desired baud rate clock. The timer's gate input must be high. The CLK input is connected to the S-100 CLOCK line and the timer's OUT line is connected to the clock inputs of the UART transmitter and receiver. The timer is set for mode 3 operation and a binary count and then the counter divisor is loaded as follows:

```
            ;SOFTWARE ROUTINE FOR PROGRAMMABLE BAUD
            ;RATE GENERATION USING 8253 TIMER/COUNTER
            ;
            ;
0010 =      BASE      EQU  10H        ;PORTS STARTING ADDRESS
0010 =      CNTR0     EQU  BASE       ;COUNTER 0 REGISTER PORT
0013 =      CNTRL     EQU  BASE+3     ;CONTROL REGISTER PORT
0000 =      LSB       EQU  0          ;DUMMY VALUES FOR
0000 =      MSB       EQU  0          ;  DIVISOR
            ;
            ;
0100                  ORG  100H
            ;
0100 3E36             MVI  A,36H      ;SET CONTROL CODE
0102 D313             OUT  CNTRL      ;  FOR COUNTER 0
0104 3E00             MVI  A,LSB      ;SET LOW-BYTE DIVISOR VALUE
0106 D310             OUT  CNTR0
0108 3E00             MVI  A,MSB      ;SET HIGH-BYTE DIVISOR VALUE
010A D310             OUT  CNTR0
```

The divisors (LSB and MSB) for the baud rates and different UART and USART modes of operation are shown in Table 14-1.

Real Time Clocks

At the end of the previous chapter we discussed the use of a Real Time Clock (RTC). The RTC interrupts the processor at a regular time interval so that the processor performs operations at accurately timed intervals or keeps track of the passage of time.

TABLE 14-1. Baud Rate Divisors (2 MHz clock)

UART Mode	Baud Rate (Hz)	UART Clock (Hz)	Divisors		
			(Decimal)	(Hexadecimal)	
				MSB	LSB
×16	50	800	2500	09	C4
	110	1760	1136	04	70
	150	2400	833	03	41
	300	4800	417	01	A1
	600	9600	208	00	D0
	1200	19.2 k	104	00	68
	2400	38.4 k	52	00	34
	4800	76.8 k	26	00	1A
	9600	153.6 k	13	00	0D
×64	50	3200	625	02	71
	110	7040	284	01	1C
	150	9600	208	00	D0
	300	19.2 k	104	00	68
	600	38.4 k	52	00	34
	1200	76.8 k	26	00	1A
	2400	153.6 k	13	00	0D
×1	300	300	6667	1A	0B
	600	600	3333	0D	05
	1200	1200	1667	06	83
	2400	2400	833	03	41
	4800	4800	417	01	A1
	9600	9600	208	00	D0
	19.2 k	19.2 k	104	00	68
	38.4 k	38.4 k	52	00	34
	56 k	56 k	36	00	24

A typical RTC circuit, using an 8253 timer, is shown in Figure 14-7. The 8253 derives its clock input from the S-100 CLOCK signal and divides it down to develop the interrupt signal.

The circuit shown in Figure 14-7 is arranged so that timer 0 and timer 1 are cascaded to develop a time-of-day clock interrupt signal. Timer 0 divides the 2 MHz CLOCK signal by 2000 and timer 1 divides timer 0's signal by 1000 to yield a total division of 2 million. The result is a 1 pulse-per-second interrupt of the processor.

The following program is used to store and update the values for hours, minutes, and seconds. A flowchart for this program is shown in Figure 14-8. The program can be expanded to include dates, day of the week, etc. The time-of-day interrupt should have a very high priority since delays in servicing this interrupt will cause time errors.

temporary master access and temporary bus masters

15

Using interrupts improves the efficiency of a computer system, allowing better utilization of processor time while handling I/O operations. However, processing interrupts requires an amount of time which in some applications may be intolerable. In applications where data is transferred directly between I/O and memory, the TMA (Temporary Master Access) technique affords transfers that are limited only by the access time of the I/O and memory. TMA is particularly desirable in applications where data is transferred in blocks between memory and I/O devices (e.g., disk and cassette storage). TMA in effect replaces software with hardware.

A TMA controller (TMAC) circuit is required to manage the transfer operations. The TMAC takes over the entire I/O-memory transfer operation from the CPU, usually transferring entire data blocks. When a data block is transferred this is called a "burst mode," since several data bytes are moved in one burst. The burst mode is the fastest I/O method.

A basic S-100 TMA system is shown in Figure 15-1. The associated timing signals are shown in Figure 15-2. Upon receiving a TMA request from a peripheral device the TMAC circuit signals the permanent master, via the Hold (HOLD*) line, that it wishes to take control of the address, data, and control buses. The TMAC then waits for the permanent master to finish the execution of its current instruction, which is signalled by the hold acknowledge signal (pHLDA). The TMAC then takes control of the address, data, status, and control buses. This is done by the TMAC turning off the permanent master's tri-state bus buffers and turning on the

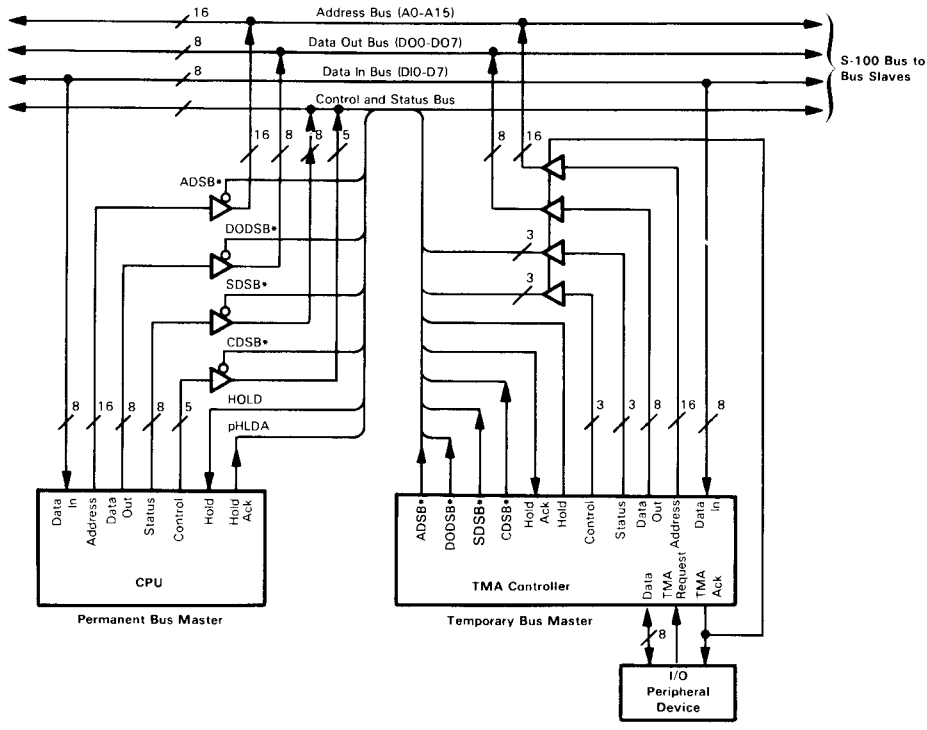
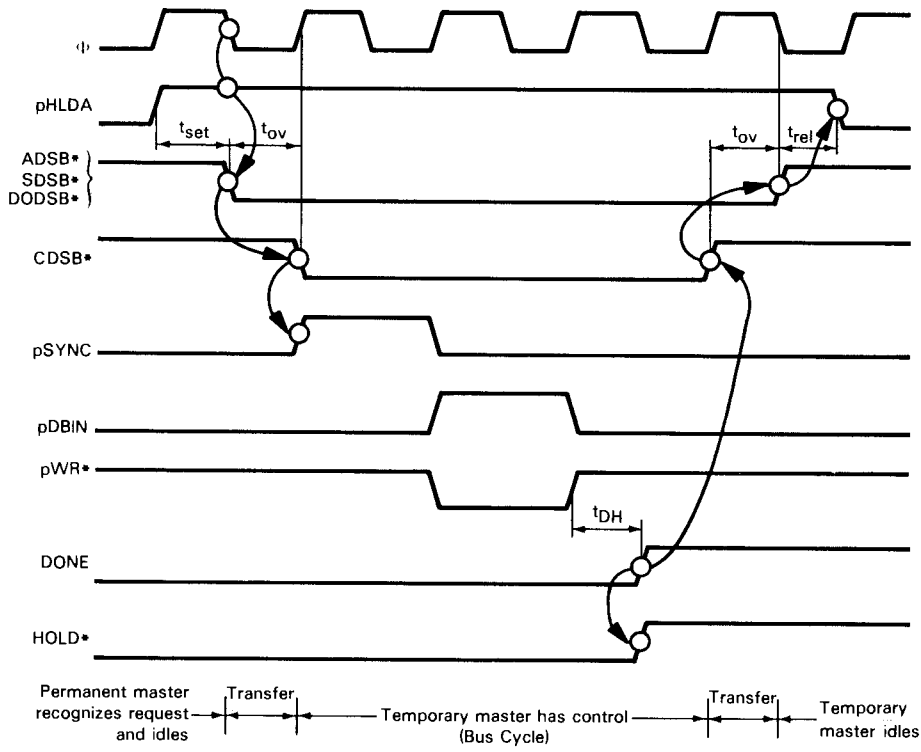


FIGURE 15-1. Block Diagram of Basic TMA System for S-100 Bus

TMAC's tri-state bus buffers to the buses. The TMAC now provides the necessary address, I/O, and memory read/write control signals to transfer a data byte or block of data bytes between memory and I/O. Upon completing the data transfer the TMAC returns control to the permanent master.

The TMAC becomes the master of the system during the TMA operation and is therefore called a "temporary bus master." The memory and I/O devices follow the commands given by the permanent bus master (CPU) or temporary bus master (TMAC), and hence memory and I/O devices are called "bus slaves."

Although TMA provides the advantage of higher speed data transfer, it has the disadvantage of considerably more complex circuitry. Further, the permanent master is disabled so that no interrupts can be acknowledged while TMA is occurring. Thus the permanent master cannot handle any time-dependent operations such as an interrupt-driven time-of-day routine. Suspending interrupts would cause errors to occur in such a system. Further, a power-failure interrupt request during a TMA operation would not be serviced.



t_{set} = 30 ns minimum
t_{ov} = 83 ns minimum/1 μs maximum
t_{rel} = 20 ns minimum
t_{DH} = 33 ns minimum/400 ns maximum

FIGURE 15-2. Timing of Bus Transfer Between Permanent and Temporary Masters

Another caution should be given. Dynamic memory systems may not operate properly with TMA. In such a system, the TMA devices cannot continuously use the address and data buses for more than a millisecond or so, since memory control logic must be able to use the buses to refresh the dynamic memory cells.

One last caution. The TMA transfer process is inherently complex and hence there is a complex interaction between all the parts of the system. Timing problems are likely to occur, and extra circuitry, careful design, and extensive testing are required to develop a reliable TMA system.

TMA TECHNIQUES

Two basic techniques are used for TMA: the "cycle-stealing" and the "direct TMA" methods. One kind of cycle-stealing method does the TMA transfer operation during clock cycle times when the permanent master is not accessing memory or I/O. For example, the 8080 CPU accesses memory and I/O during states T2 and T3, leaving two or three T-states during which the TMA transfer may be made. Circuitry is required to identify the proper T-states. While this type of cycle-stealing does not slow down the operation of the processor or affect time dependent operations, it does require complex timing circuits and does slow down the TMA transfer rate. If this type of cycle-stealing TMA were to be implemented on the S-100 Bus, it would more properly be called "bus state stealing."

Another kind of TMA that is quite popular on the S-100 Bus is another type of cycle-stealing TMA. In this case we mean "bus cycle-stealing." It does slow the system down, but if a peripheral's data rate is not very high then this type of TMA is preferred over the burst mode. Even the data rate of a floppy disk controller is slow compared to the maximum data transfer rate on the bus. This form of cycle-stealing TMA allows the processor to execute quite a few instructions between TMA transfers.

In a system with one permanent master and one temporary master the permanent master normally has control of the bus. The temporary master may request control by generating an "IWANT" signal indicating that it wants the bus. A timing transfer circuit, such as the one shown in Figure 15-3, then generates a HOLD* signal to the permanent master. The permanent master completes its current instruction cycle, goes into an idle mode and sets the Hold Acknowledge signal (pHLDA) true, which causes MINE to be asserted. The timing transfer circuit then performs an orderly transfer of bus control from the permanent master to the temporary master. The permanent master is disconnected from the bus by enabling the ADSB*, SDSB*, and DODSB* lines. These disable the permanent master's address, status, and data out bus drivers and enable the control drivers of the temporary master.

Both the permanent and temporary masters are now driving the control output bus lines. This is necessary to prevent spurious signals from being generated on the control lines. This is shown as the "transfer" state in Figure 15-2. During this transfer time the control lines must have the following levels: pSYNC and pDBIN must be low, while pSTVAL*, pWR*, and pHLDA must be high. The transfer state ends when CDSB* is enabled, disabling the permanent master's control drivers and enabling the address, status, and data out drivers of the temporary master. The temporary master is now signalled that it has complete control of the bus and can begin its first cycle.

The temporary master signals that it is finished using the bus by generating a DONE signal. The timing transfer circuit then does a mirror-image transfer

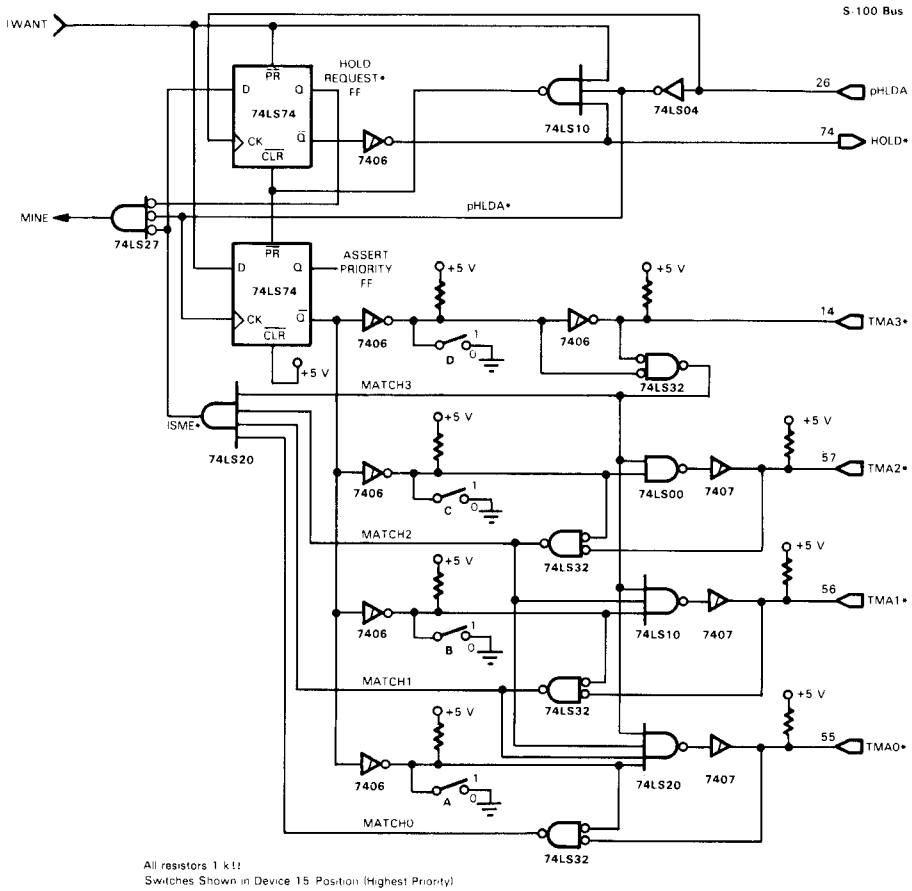


FIGURE 15-3. Arbitration Logic for S-100 TMA Controller

sequence from the temporary master to the permanent master. At the same time it releases the HOLD* line to the permanent master.

MULTIMASTERS

Up to 16 temporary masters may be accommodated on the S-100 Bus. To accomplish this, an arbitration circuit is necessary to determine which temporary master will be allowed to control the bus at any given time. The S-100 Bus uses a

four-line arbitration bus for arbitrating among the 16 temporary masters. The lines are TMA0* (55), TMA1* (56), TMA2* (57), and TMA3* (14). These lines are driven by open collector drivers and are pulled high by pull-up resistors somewhere in the system. Each temporary master has a unique priority which it asserts (active low) on the arbitration bus at an appropriate time. A typical arbitration controller circuit is shown in Figure 15-3.

Each temporary master has its own arbitration controller. The controller circuit contains either a switch register or a programmable register which establishes the priority of the temporary master. All switches open ($0F_{16}$) is the highest priority. No two temporary masters may have the same priority.

The controllers compare the priority appearing on the bus with the priority they are asserting, starting with the most significant bit. If disagreement occurs at any bit position, another temporary master or controller is asserting that priority bit and thus must have a higher priority. In that case, all less significant bits are removed by the temporary master with the lower priority. All more significant bits agree and thus need not be removed, and the bit which disagreed must have been a logical zero and thus was not asserted. Having the agreeing bits asserted reduces system noise caused by the redistribution of driving currents in the bus and speeds settling of the correct priority on the arbitration bus. This process is a continuous parallel process in which the incorrect comparisons occur and are removed.

A temporary master or controller requests the bus by asserting its internal IWANT line. If pHLDA is not asserted (permanent master has bus) and HOLD* is not already asserted, the temporary master or controller may assert its priority and enable the HOLD* line. This process guarantees ample time to settle the arbitration bus before the granting of the bus on the rising edge of pHLDA.

This scheme usually results in the first requestor winning the bus. Only if simultaneous bus requests occur will the arbitration have any effect. This, however, is not improbable, since multiple requestors will become synchronized by waiting for the falling edge of pHLDA.

The master or controller that is successful in gaining control of the bus will have asserted its MINE line. Thereupon the bus transfer, as described earlier, must begin. All requestors will continue to assert their priorities on the arbitration bus until the falling edge of pHLDA. Thus the priority number of the current bus master is available on the TMA arbitration bus while pHLDA is asserted. A master or controller that loses the bus continues to assert its priority bits, but removes its assertion of the HOLD* line so that the winner may indicate that it is done by releasing HOLD*.

Figures 15-4 and 15-5 show two possible cases of bus arbitration. In Figure 15-4 the requestor has no competition. It requests the bus and the bus is granted. In Figure 15-5 the requestor waits for the bus to be free, arbitrates for the bus and loses, then arbitrates again and wins.

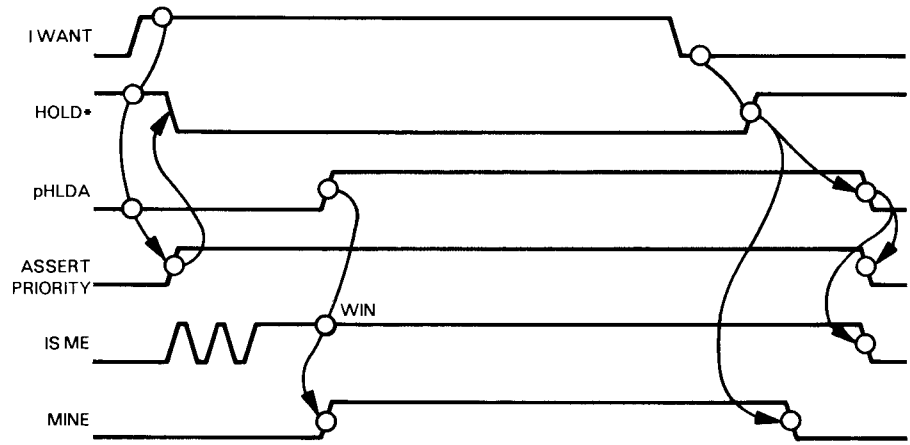


FIGURE 15-4. Arbitration Timing — No Competition

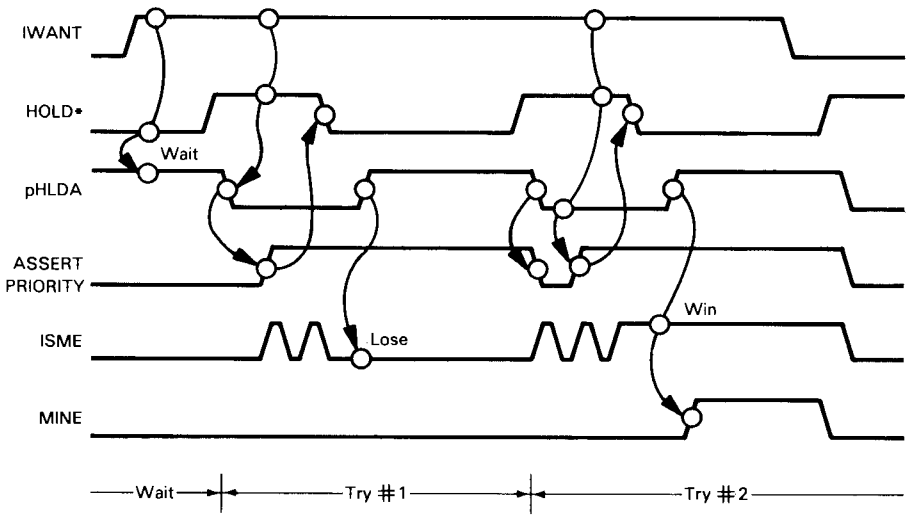


FIGURE 15-5. Arbitration Timing — Wait, Lose, then Win

Looking at this in more detail, in Figure 15-4 a temporary master asserts its IWANT line and finds pHLDA unasserted (permanent master has bus) and HOLD* unasserted (no other temporary masters requesting bus). The temporary master or controller asserts HOLD* and its priority on the TMA arbitration bus. The ISME signal is asserted if none of the bit comparisons on the arbitration bus fails. ISME is clocked on the rising edge of pHLDA, creating the MINE signal. When the temporary master is finished with the bus the IWANT signal is released, releasing HOLD* and resetting MINE. The permanent master releases pHLDA, gaining control of the bus.

In Figure 15-5 the requestor raises its IWANT line but finds the bus already busy and must wait to assert its request and priority until the falling edge of pHLDA. The requestor arbitrates for the bus during try #1, but another requestor has a higher priority and ISME is low at the rising edge of pHLDA, indicating a loss in the arbitration process. The losing requestor removes its assertion of the HOLD* signal, but continues to assert its losing priority until the falling edge of pHLDA. The process repeats, but this time results in a win for the requestor.

S-100 TMA CONTROLLER CIRCUIT

After the arbitration interval, assuming that this particular master is the winner, the MINE signal will be asserted high. This signal originates in the arbitration circuit shown in Figure 15-3. This is an indication to the rest of the TMAC circuitry that it may begin to take control of the bus.

This takeover has to be synchronized to the master system clock (Φ). The circuitry to perform this task is shown in Figure 15-6. Timing diagrams for this circuit appear in Figures 15-7 and 15-8. In the following discussion, it may be helpful to refer to these timing diagrams.

On the first falling edge of Φ after MINE is asserted, the XFERI signal will be set high. This signal is applied to the CLR* input of flip-flop 2 (FF2), so when XFERI goes high it will allow flip-flop 2 to be clocked. The following rising edge of Φ will then clock the MINE signal through flip-flop 2, setting the XFERII signal high. At the same time, XFERI is clocked through flip-flop 3 to produce SYNC, which later becomes pSYNC, signifying the beginning of a bus cycle.

XFERII is applied to the PR* input of flip-flop 6, which was holding point C low. This line (point C) is applied to one input of an OR gate, the other input of which is the SYNC signal. Since one input is low and SYNC is now high, the output of the OR gate (point D) will be high. At the next rising edge of Φ this high will be clocked through flip-flop 4, which sets STB high. STB will later become either pDBIN or pWR*, depending on the direction of the data transfer. The other output of flip-flop 4, STB*, will reset SYNC to a low because it is tied to the CLR* input of flip-flop 3, which will end SYNC. This connection also ensures that SYNC will be

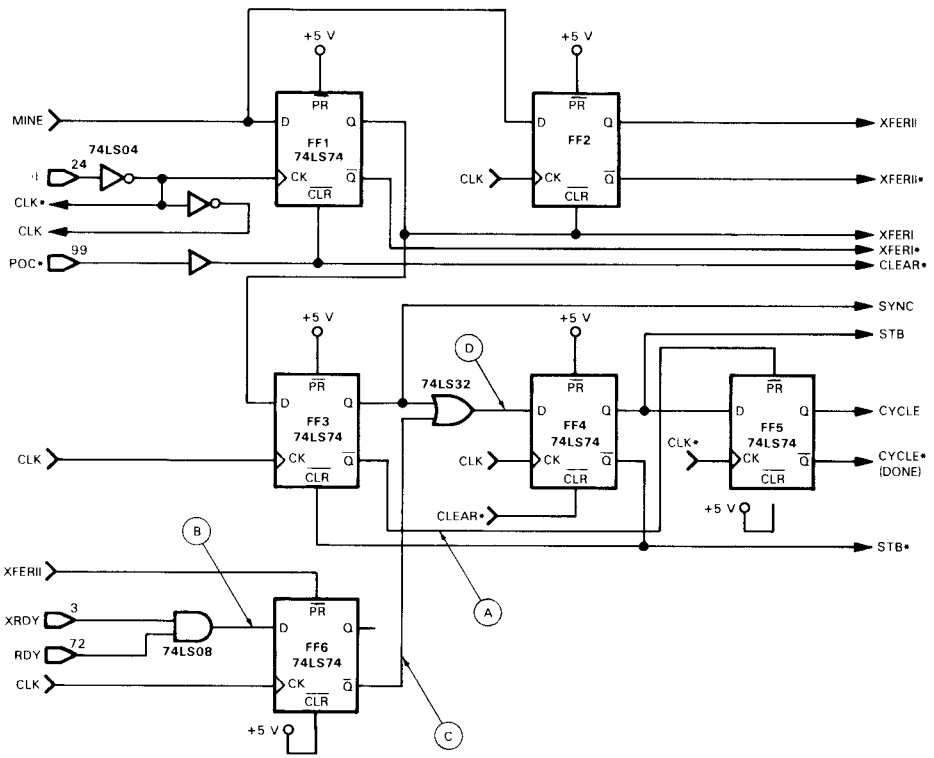


FIGURE 15-6. Timing Logic for S-100 TMA Controller

inhibited during the data strobes should it become extended by a wait state.

If point B was still high at that last rising edge of Φ, then point C will still be low. Since SYNC is now low, point D will also go low. On the next rising edge of Φ, this low will be clocked through flip-flop 4, ending STB.

Meanwhile, the inversion of the SYNC signal (point A) sets the output of flip-flop 5 high, causing the CYCLE signal to be asserted. This signal indicates that a bus cycle has started, since it goes high right after SYNC goes high (point A goes low). On the falling edge of the clock during SYNC, nothing will happen to the output of flip-flop 5 because its clock input will be overridden by point A being low. When point A returns high (when SYNC goes away), the STB signal that is applied to the D input will also go high. Since the PR* input is now high, the flip-flop may be clocked. But since the STB signal at the D input is high, the next falling edge of Φ (the one during STB) will clock the STB signal through, and the output (CYCLE) will remain high.

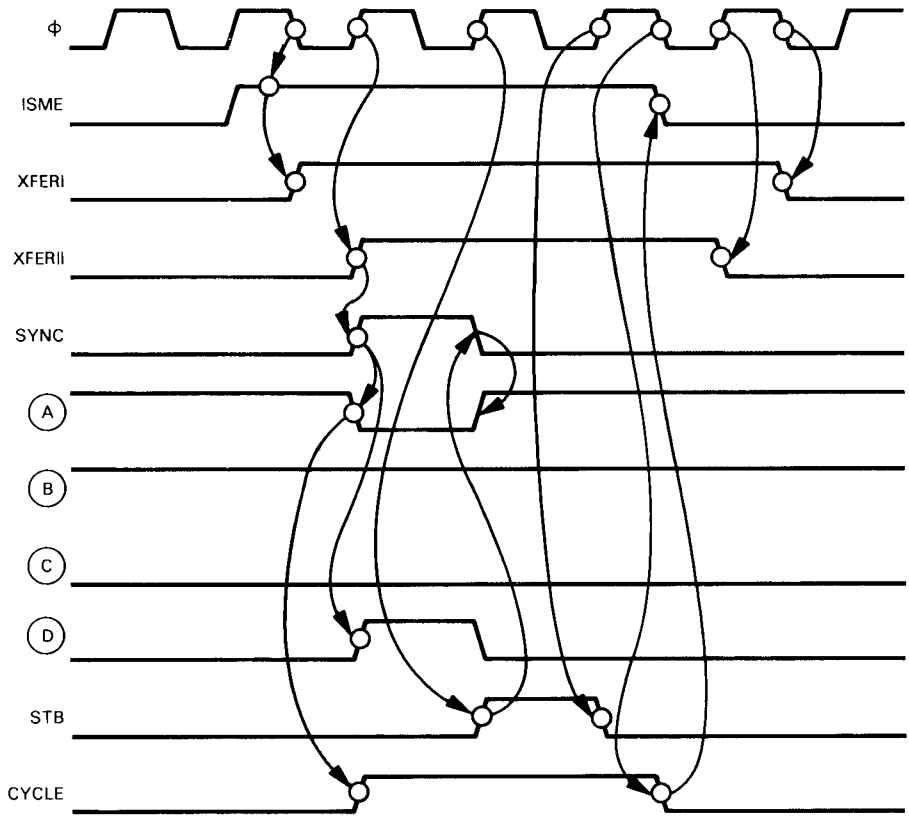


FIGURE 15-7. Timing Diagram for Logic of Figure 15-6 — No Wait States

At the next rising edge of Φ, STB will be clocked low, so that at the next falling edge of Φ the CYCLE output will go low, indicating that the bus cycle is complete. The half-cycle delay after STB falls is to cause the information being asserted on the S-100 Bus to remain stable, as indicated in the IEEE specification. When the signal CYCLE* goes high, this becomes the "DONE" indicator to the rest of the controller circuitry. If there are no more bytes to be transferred, the controller will drop its IWANT signal, causing MINE to go low. At the next rising edge of Φ this low will be clocked through flip-flop 2, ending XFERII.

Since MINE is now low, the next falling edge of Φ will clock it through flip-flop 1 and end XFERI. At the same time the permanent master will regain control of the bus, thus ending the TMA transfer.

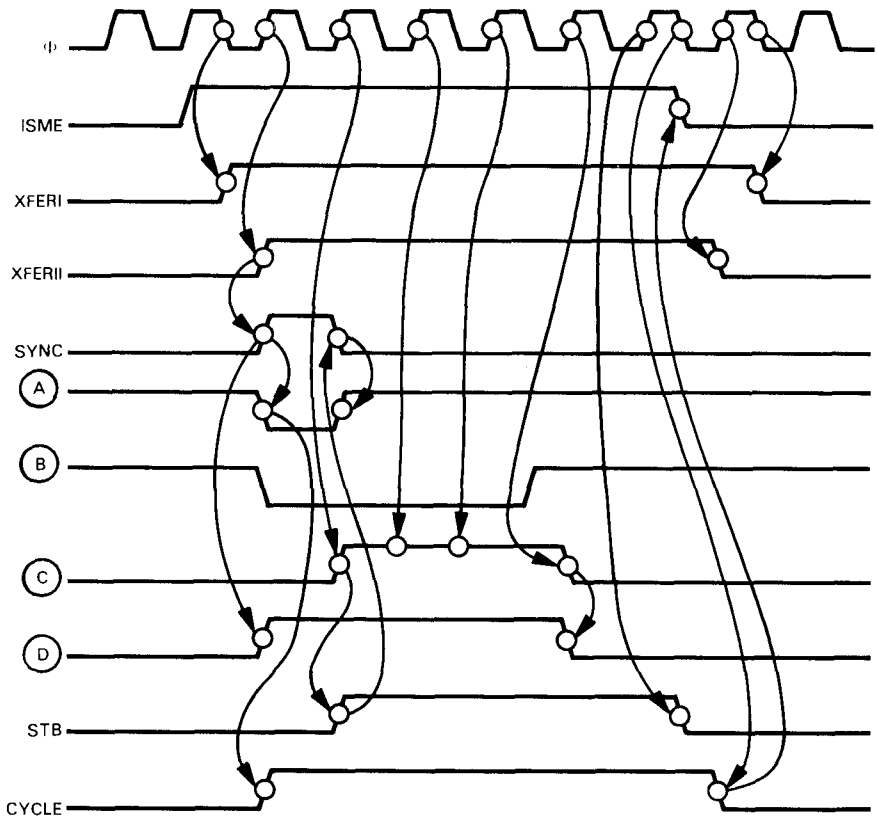


FIGURE 15-8. Timing Diagram for Logic of Figure 15-6 — 3 Wait States

The above discussion assumes that no wait states were requested by the bus slave (neither RDY nor XRDY was low). The timing diagram in Figure 15-8 shows the same TMA cycle except with three wait states added. If a wait state is requested the following occurs.

Assume that SYNC has just been asserted high. XFERII will have just risen as well, which will allow flip-flop 6 to be clocked. Point C was set low by XFERII, which allows point D to go high. Sometime after SYNC is asserted, one of the RDY lines will go low, causing point B to go low. The next rising edge of Φ is going to do two things: it will clock point D through flip-flop 4, starting STB, and it will clock point B through flip-flop 6, setting point C high.

The next rising edge of Φ would normally clock STB low, but since point C is high, STB will remain high. If point B is still low at this rising edge, point C will still be high. As long as point B is low at succeeding rising edges of Φ, STB will continue to remain high. This is the object of the wait state: to extend the strobes. When point B finally goes high, the next rising edge of Φ will set point C low. Since SYNC is low and point C is now low, point D will go low. The last rising edge of Φ which clocked point C low, would still clock a 1 into flip-flop 4 because point D was still high at the time of the rising edge. STB will therefore remain high until the next rising edge of Φ, which will clock point D through flip-flop 4, when STB will end.

The next falling edge of Φ will clock CYCLE low, indicating the end of the bus cycle. If this is the last byte of data or the circuit is cycle-stealing, the transfer will end, as above.

This circuit is capable of either cycle-stealing (bus cycle) or burst mode. If MINE is not dropped after the rising edge of CYCLE* (indicating the end of the bus cycle), then the rising edge of Φ that would have clocked XFERII low will instead start another SYNC and leave the XFER lines asserted. As long as MINE remains asserted, multiple bus cycles will be executed. It is therefore up to the rest of the circuit to either drop its IWANT signal in response to DONE or leave it asserted, causing another cycle to be executed.

Now that we have seen how the timing transfer circuitry works, we can discuss the rest of the TMA controller circuitry.

Figure 15-9 shows the address decoder circuitry, address counters, and bus buffers for the S-100 address lines. The port decoder is used along with the negated-input NAND gates (really 74LS32 OR gates) to provide four I/O write strobes. Three are used to load data into the 74LS191 counters to preset them to the desired starting address of the TMA transfer. This is done by writing the data as three separate bytes to three successive port locations. This must be done prior to starting the TMA transfer.

When XFERI goes high the output of the 7406 will go low, asserting ADSB*, which will cause the address lines from the permanent master to float. At the same time XFERI* will go low, which will cause the address in the counters to be asserted on the address lines.

The value written into the counters will be incremented when the CYCLE* signal goes high, which it will do at the end of the bus cycle. Thus the counters now hold the next address and are ready for the next transfer to occur.

CONTROL* is the fourth write strobe, and is used to program the TMAC with direction of transfer information (R/W*), as well as issue a start transfer command (START*) to the TMAC. This circuit is shown in Figure 15-10. At the end of the CONTROL* strobe, the information on DB1 (internal data bus bit 1) will be latched into flip-flop 1. The output becomes the R/W* signal, which will be low for TMA writes (transfer of data from the TMAC to memory). Thus, if a write occurs to the control port with bit 1 high, the TMAC will be set to the read mode,

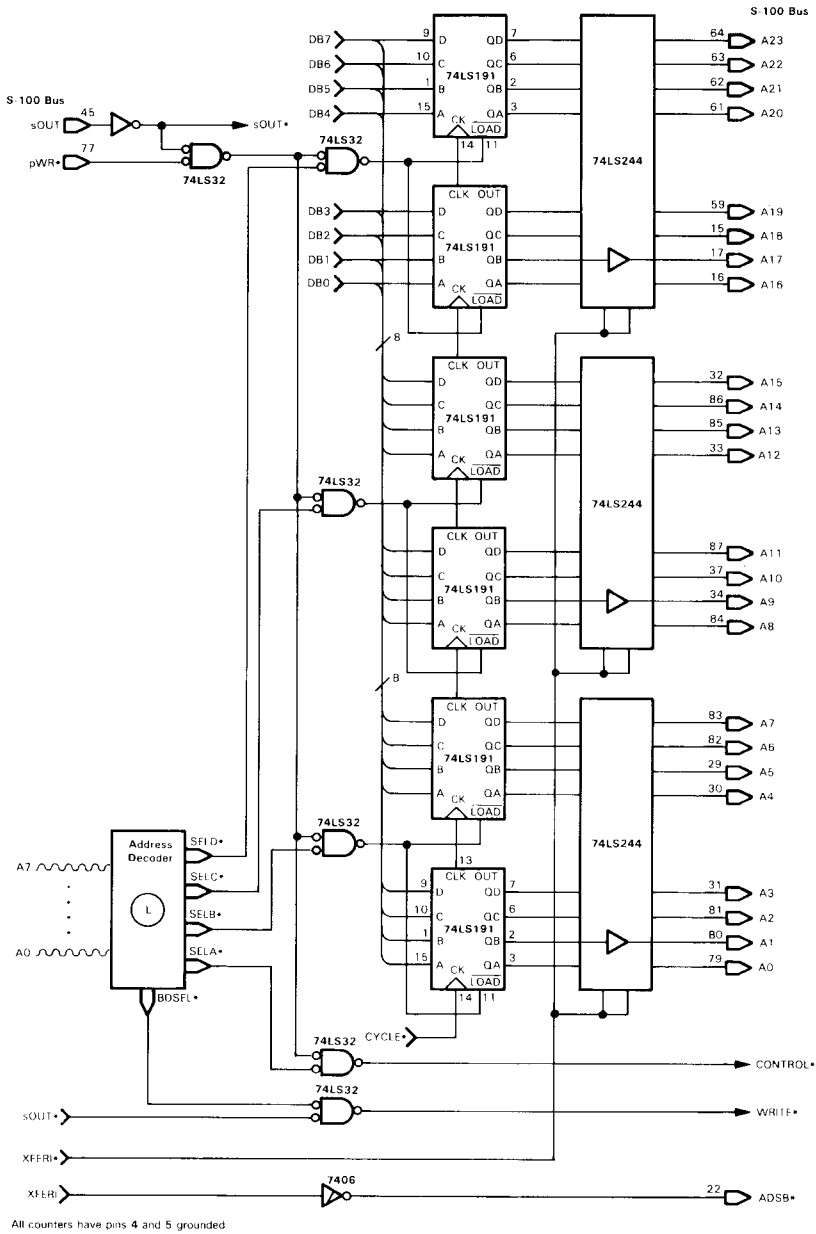


FIGURE 15-9. Address Counters and Port Decoding Logic for S-100 TMA Controller

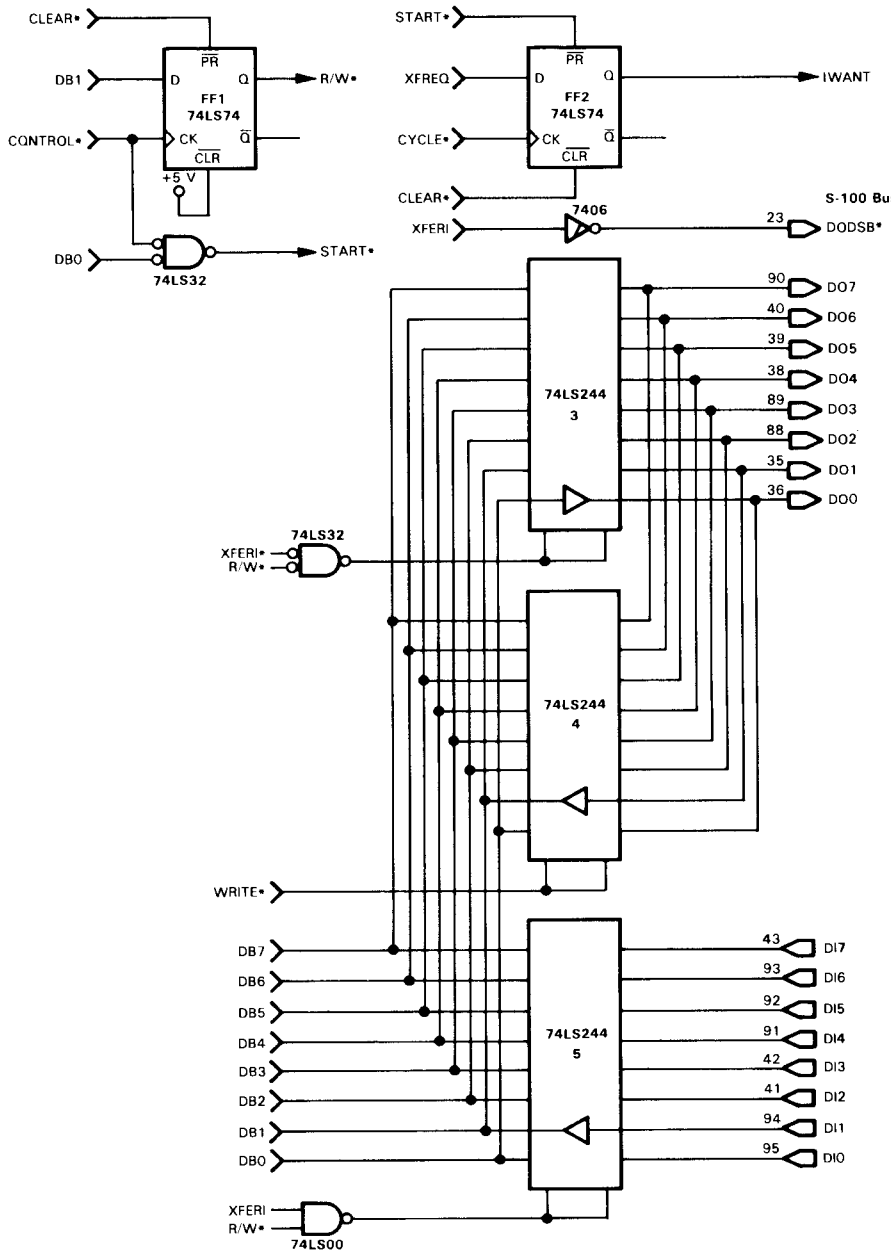


FIGURE 15-10. Data Bus Control Logic; Start Pulse and Direction Bit Logic

and if a write occurs with bit 1 low the write mode will be selected. The TMAC is in the read mode at power-on.

If a write occurs to the control port (CONTROL* low) with DB0 low, a low will appear at the output of the negated-input NAND gate (really an OR gate). This signal, called START*, will remain low for the duration of the CONTROL* strobe. This signal is used to start the TMA transfer cycle or cycles. How this occurs will be discussed later.

To set the TMAC to the read mode and begin the transfers, a CPU will write to the control port with bit 1 high (sets read mode) and bit 0 low (starts transfer). To set the TMAC to the write mode and begin transfers, the CPU will write bit 1 low (write mode) and bit 0 low (starts transfer). To change the read/write mode without starting the transfer, the write should occur with bit 0 set high.

Another portion of the circuit that appears in Figure 15-8 is the circuitry to handle the data bus direction and provide buffering. The data bus of a TMAC is quite different from that of a normal slave device. A normal slave always receives data on the data output bus and transmits data on the data input bus. A master must do just the opposite: it must transmit its data on the data output bus and receive data on the data input bus. However, a TMAC must look like a slave to the bus at times (when the address counters are being loaded, for example) and look like a master when it is doing TMA transfers.

This particular TMAC has no readable registers. The WRITE* signal will go low when any of the four ports is selected and an output cycle is occurring (OUT* and BDSEL* low in Figure 15-9). WRITE* is applied to the tri-state controls of buffer 4 in Figure 15-10. This enables the data on the data output bus to appear on the internal bidirectional data bus so that it may be written to the address counters and the control port. When WRITE* is high, buffer 4 is disabled.

When XFERI is high, DODSB* will be asserted low by the 7406, and the data output bus will float. When XFERI* is low and R/W* is low, buffer 3 will be enabled, and the data on the internal bus will be asserted on the data output bus. It is up to the peripheral device to place data onto the internal data bus. It should use the CYCLE signal to do this. If R/W* had been high (signifying the read mode) then buffer 3 would have been disabled.

When XFERI is high and R/W* is high, the output of the NAND gate will go low, enabling buffer 5. This will allow the data on the data input bus to appear on the internal data bus (TMAC is reading data from memory) so that the peripheral device may read the data. It may use any of the available control signals (e.g., STB ANDed with R/W*) to determine that data from memory is valid. This will depend on the peripheral involved. When XFERI goes low, buffer 5 will be floated.

To start the transfer of data, the TMAC will assert the IWANT signal high. The TMAC does this via the START* pulse, which was discussed previously. START* is applied to the preset input of flip-flop 2, setting IWANT high. The peripheral device should have the transfer request line (XFREQ) asserted high. If this is to be

a single byte, then XFREQ should go low about the time the STB signal goes away. When CYCLE* goes high, this low will be clocked into flip-flop 2 and IWANT will go low, causing the transfer to end.

If this is to be a burst or multiple-byte transfer, the peripheral should leave XFREQ asserted high. When CYCLE* goes high, IWANT will also remain high, causing another transfer to occur, and so on until XFREQ goes low.

Figure 15-11 shows the last two sections of the TMAC: the circuits for driving the control output and status buses. When XFERI* goes low, both the 74LS367A and the 74LS244 will be enabled, and XFERI will cause SDSB* to go low, disabling the permanent master's status bus. Both the status and control buses will be driven by the TMAC, and the control bus will also be driven by the permanent master.

At this time SYNC and STB will both be low. This will cause pSYNC to be low, pSTVAL* high, pDBIN low, and pWR* high. pHLDA will always be high because its buffer's input is always high. This is called for in the IEEE standard. When XFERII goes high CDSB* will go low, floating the permanent master's control bus. At the same time SYNC will go high, causing pSYNC to go high, starting the bus cycle. When pSYNC is high and CLK* is high pSTVAL* will go low. SYNC will then go low, causing pSYNC to go low and pSTVAL* to return high. At the same time STB will go high. If R/W* is high, pDBIN will go high. If R/W* is low, pWR* will go low. After a while STB will go low, causing either pDBIN or pWR* to stop being asserted.

On the status bus, the buffer sections that drive sOUT, sINP, sHLTA, sINTA, and sM1 have their inputs tied low, so when the 74LS244 is enabled all those lines will be driven low on the bus. sXTRQ* will be driven high. If R/W* is high, then sMEMR will be high and sWO* will be high, signifying a memory read cycle. If R/W* is low, then sMEMR and sWO* will be low. sWO* low and sOUT low signifies a memory write status.

When XFERII goes low CDSB* will return high, allowing both masters to drive the control bus. When XFERI goes low (and XFERI* goes high) the two buffers will be floated and SDSB* will go high, allowing the permanent master to again drive the status bus.

This completes the description of the circuitry. What we have not shown is the peripheral device that receives data from and transmits data to the TMAC. This could be anything from a floppy disk controller to a high-speed hard disk. It is the responsibility of the peripheral to drive the internal data bus with data (at the right time) when it wants to transfer data to memory, and to read the data from the internal bus (at the right time) when it wants data from memory. The peripheral must assert XFREQ high to transfer data in either direction. If a single byte is to be transferred, XFREQ must be low before CYCLE* goes high. If a burst of data is to be transferred, XFREQ must be asserted until all the data has been transferred. The peripheral must know when all transfers have been completed and stop

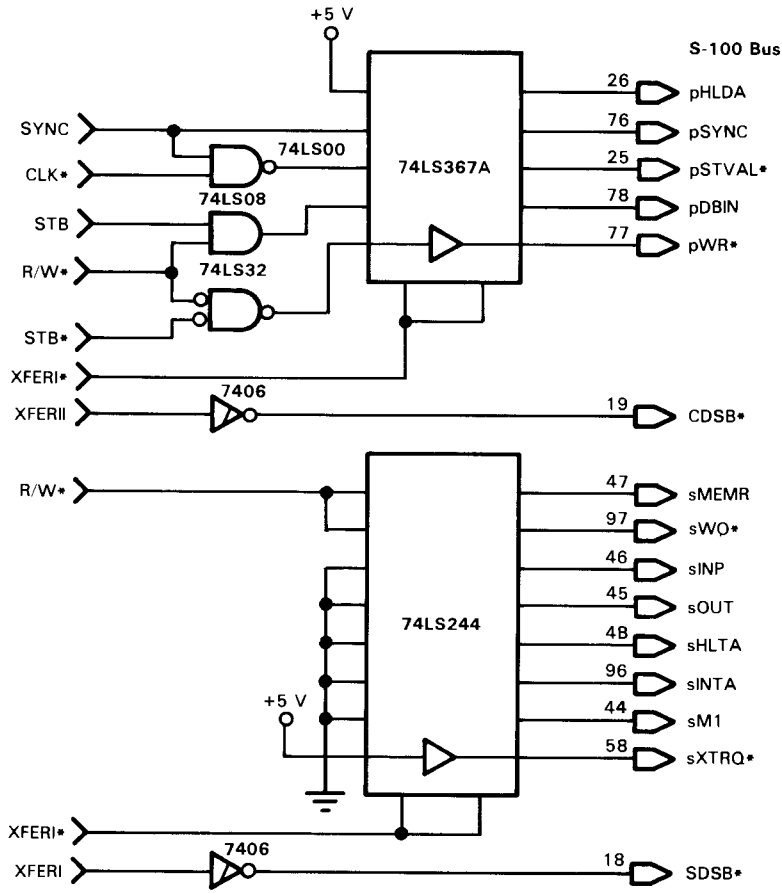


FIGURE 15-11. Control and Status Bus Logic for S-100 TMA Controller

asserting XFREQ. A master must load the address counters with the starting address at which the transfer will occur and select the direction of the transfer. The master must also start the transfer.

When used in burst mode, the permanent master will know that the transfer is completed because it will get the bus back when it is done. When cycle-stealing, however, the peripheral must signal the controlling master that the transfer is complete. This may be done with an interrupt (usually from some kind of terminal count output from the peripheral), or a status register may be added to allow the master to read the peripheral's status.

You may well ask "Why didn't you use one of the LSI DMACs that are available?" The answer is because:

1. None of them has more than a 16-bit address counter.
2. Most of them have trouble running at speeds beyond 4 MHz.
3. They need address and data buffers anyway.
4. None of them can easily be made to meet the S-100 standard.

DUMMY MASTERING

In cases where a number of processors coexist in a single system as temporary masters, it may prove inefficient, from a systems point of view, to implement a permanent master.

Another important consideration is that any temporary master must synchronize its operations with the master system clock. Therefore, the speed of the temporary master is limited by the speed of the permanent master.

In such cases, it is permissible that the permanent master be implemented as a dummy — that is, a device that conducts no bus cycles, but only supplies the master system clock and an arbitration interval so that the TMA control bus may settle. The dummy master takes control of the bus between temporary masters, asserting the control output bus in the null state, and passes the bus to the next requestor after an arbitration interval of one clock cycle.

Output signals required for dummy masters are the control output signals and the system clock (Φ). Input signals required are HOLD* and CDDSB*.

MULTIPROCESSING

As the cost of CPUs drops and the complex demands on the computer system increase, it becomes advantageous to use more than one CPU in the computer system. For example, one CPU might be assigned the task of communicating with I/O devices, another searching memory, another editing, another checking, and so on. The use of separate, interconnected CPUs, each having a dedicated, sole task, reduces the complexity of the operating system software and improves reliability and backup.

Multiprocessing is a form of TMA. In standard TMA the master CPU relinquishes its control of the bus to a controller whose sole function is to transfer data between memory and an I/O device. The TMAC is thus a temporary master,

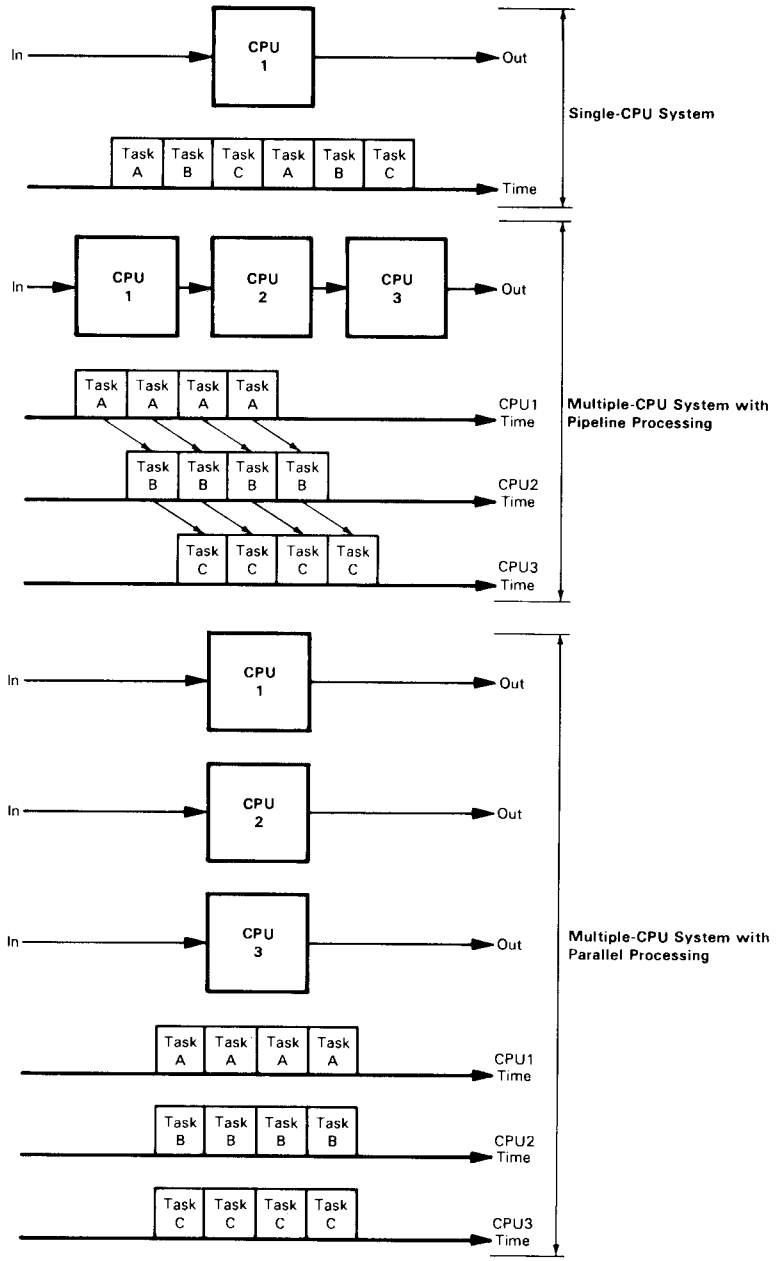


FIGURE 15-12. Multiprocessing Techniques Compared

while the memory and I/O devices act as slaves, following the commands of whichever master is in control of the bus. Multiprocessing is essentially the same as TMA except that the temporary master is also a CPU or CPU-like device.

MULTIPROCESSING SYSTEMS

In an S-100 based multiprocessing system there is actually more than one CPU on the S-100 Bus, and the bus serves as the communicating link between the CPUs. Each CPU contains its own memory and possibly I/O circuits. Each CPU is assigned a function to which it is best suited. For example, an 8-bit CPU with its own ROM, RAM, and I/O on the CPU card may be assigned to execute the DOS (Disk Operating System) and I/O handling, while another 16-bit CPU card with its own ROM and RAM may be assigned to execute a high-level language such as BASIC. Tasks are therefore partitioned in the system.

Since each CPU has local resources it may access the bus only when it needs to communicate with other masters or slaves. In many applications less than 10% of the time is taken by system bus accesses. The IEEE S-100 Bus standard therefore provides facilities for up to 16 prioritized masters on the bus. These masters may be either other CPUs or dedicated controllers.

In a multiprocessing system CPUs may share resources such as a high-speed mathematics board or a disk controller. In addition, throughput can often be enhanced by using "pipeline" or "parallel" processing techniques. These are shown in Figure 15-12. In the pipeline system, system functions (tasks) are divided among CPUs, so that data flows through the system serially. Each CPU performs its portion of the system functions. It then calls upon the other processors to perform other sets of functions. For example, one CPU may do data acquisition and buffering while another CPU processes the data.

In the parallel processor system each CPU performs a separate task without any dependence on the other CPUs.

REFERENCES

Barthmaier, Joseph P. "Multiprocessing System Mixes 8- and 16-Bit Microcomputers," *Computer Design,* February 1980.

Kane, Jerry, and Osborne, Adam. *An Introduction to Microcomputers: Volume 3 — Some Real Support Devices,* Berkeley: Osborne/McGraw-Hill, 1976, 1979.

some useful circuits

16

This chapter examines some useful circuits that didn't quite fit into any of the other chapters. Three of these circuits are used for debugging, another is an ERROR* trap circuit, and the last is a Jump-on-Reset circuit.

ADDING LED'S TO MONITOR S-100 SIGNALS

When debugging an S-100 system it is usually useful to "see" what is happening on the various signal lines. This can be accomplished with a simple logic probe, an oscilloscope, or even a logic analyzer. Moving the logic probe or scope lead around to examine all the lines can quickly become tedious; therefore it is very convenient to attach LEDs to the signal lines to monitor their state. The LEDs can be attached permanently, like a front panel, or on a board that is plugged into the bus for trouble-shooting.

The basic circuit shown in Figure 16-1 is the recommended way of monitoring an S-100 signal line with an LED. The inverter can be a tri-state type buffer with its enable input tied so that the buffer is permanently enabled, or a high current TTL inverter, such as a 7406 (open collector inverters will work just fine). The incoming signal line is inverted, causing current to flow through the current limiting resistor and LED to ground. Thus when the signal line is high, the LED will light. TTL type devices can sink current much better than they can source it, so it is not recommended that the LED current be sourced by the TTL device.

Any number of these circuits may be used to view as many signal lines as you need to monitor. The most common choices would be the address, status, and data buses.

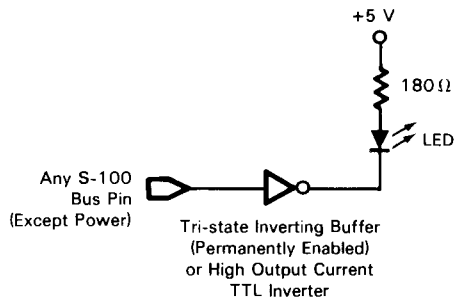


FIGURE 16-1. Adding LEDs to Monitor S-100 Signal Lines

A SINGLE STEPPER

Being able to see what is happening on the S-100 signal lines with LEDs is not of much use if the lines are changing states millions of times a second. What we need is a circuit to stop the processor between bus cycles so that we can examine the state of the LEDs. This circuit is called a "single stepper," and is shown in Figure 16-2. Here's how it works. If the RUN/STOP switch is closed, one input to the 74LS38 will be low. This will cause the other input to be ignored and the XRDY line will never be driven low. This will cause the CPU to run just as if this circuit did not exist. When the first pSYNC pulse on the bus occurs, the flip-flop will be preset and the Q output will be high. Each succeeding pSYNC will set the output high. This output forms the other input to the 74LS38. If we want the CPU to stop, the RUN/STOP switch is opened. Both inputs to the 74LS38 will now be high and the output driving XRDY will go low. This will cause the CPU to stop in an indefinite wait state. The address, data, and status buses will no longer be changing and we can now examine the LEDs connected to them. To step the processor to the next bus cycle, push the SINGLE STEP button. The cross-coupled gates debounce the switch (otherwise the CPU would advance hundreds or thousands of bus cycles instead of one) and clock a low out of the flip-flop. This causes XRDY to go high, allowing the CPU to run again. When pSYNC goes high at the beginning of the next cycle the flip-flop's output will be set high, stopping the CPU again. We may then examine the next bus cycle's information.

This whole process may be repeated as often as desired, "single stepping" through the program. When we wish the CPU to run again, the RUN/STOP switch should be closed.

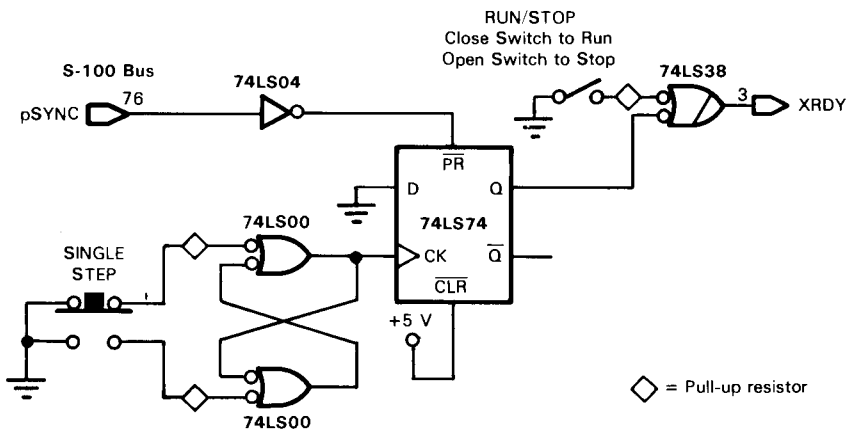


FIGURE 16-2. Single Stepper for the S-100 Bus

A HARDWARE BREAKPOINT TRAP

The previous circuit did not allow us to choose where in the program we wished to stop (unless the RUN/STOP switch is open at power-on, in which case the single stepper will stop the CPU in its first cycle). To stop the processor at a specific point in the program requires that we set a "breakpoint." The term breakpoint is used because it is the point at which the program execution breaks.

A circuit to stop the processor at a predetermined memory address is shown in Figure 16-3. The address at which we want the program to stop is set in the switches, in binary. A closed switch will match a high on a particular address line. When the TRAP on/off switch is open the breakpoint trap is enabled. When the address set in the switches appears on the bus, the XRDY line will go low, causing the processor to stop. It may then be single stepped with the circuit in Figure 16-2.

To restart the processor, press the RESTART button. This will cause the XRDY line to go high again, causing the CPU to run. If the TRAP on/off switch is closed, the breakpoint circuitry will have no effect on the XRDY line.

Here's how the circuit works: the address lines are compared to the settings of the address switches by the 74LS136 exclusive-OR gates, exactly the same as they are in our address decoder circuits. (See Chapter 5 for a discussion of how this works.) The XOR gate outputs are tied together in three separate groups. This was done to minimize the load on the outputs. The three outputs are combined and inverted by a 74LS10. This output will go low only when there is a match on

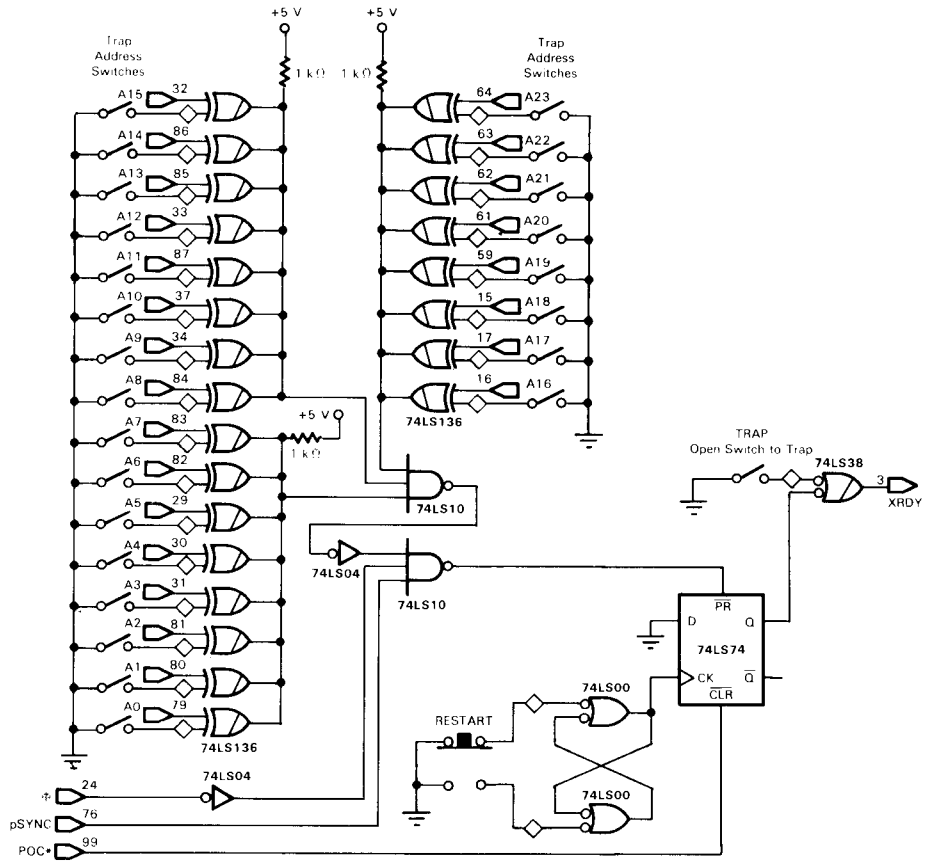


FIGURE 16-3. S-100 Breakpoint Trap Circuit

all of the address lines. This signal is inverted by a 74LS04 and applied to one input of another section of the 74LS10. One of the other inputs to this gate is the pSYNC signal, and the third input is the inverted clock (Φ). When all three inputs are high (which will occur at the beginning of a bus cycle if there is a match) the output will go low, presetting the output of the flip-flop high. This will cause the XRDY line to be driven low (if the TRAP on/off switch is open) and the CPU will be stopped. To restart the processor, press the RESTART button. This will be debounced by the cross-coupled gates and will clock a low out of the flip-flop, causing the XRDY line to return high again, and the CPU to continue.

The three circuits discussed above (LEDs, single stepper, and breakpoint trap) can be combined on a single circuit board to make a powerful debugging tool.

AN ERROR* TRAP CIRCUIT

In some of the earlier chapters we discussed the possibility of an error occurring somewhere in the system. In response to that condition we stated that the ERROR* line should be asserted and that some form of hardware should latch the information from the bus so that the error may be attended to. But we never showed you any hardware to implement this "trap." The circuit in Figure 16-4 is one such solution.

The idea behind this circuit is to latch the address bus on every M1 or instruction fetch cycle. When an error occurs (ERROR* goes low) the address in the latches is frozen. This means that no further latching will occur. The latches will then contain the address of the most recently executed instruction. The error recovery routine may then read the contents of the latches to determine what caused the error. Without the address of the most recent instruction, the system may never be able to determine the cause of the error. The error recovery routine must reenable the trap circuit so that new M1 cycles may be latched again. This will usually be the last thing the recovery routine does before returning to the system.

Here's how the circuit works: the address bus is connected to the inputs of three 74LS374 octal latches. When the system is powered up the 74LS74 will be preset by POC*. The flip-flop Q output is applied to one input of an AND gate, allowing the other input to pass through to the output. The other input is a signal consisting of pSYNC, inverted pSTVAL*, and sM1. All these signals will go high near the beginning of the bus cycle in which an instruction fetch is occurring. The output of the AND gate will go high as pSTVAL* falls (indicating a valid status) and will latch the addresses into the 74LS374s.

When ERROR* falls, a low will be clocked out of the 74LS74 and will set one input to the AND gate low. This will inhibit further M1 cycles from latching the address, and therefore the error-related address will be preserved. ERROR* low will also cause a nonmaskable interrupt which should cause the error recovery routine to be invoked.

The error recovery routine can then read the data in the latches by addressing the three input ports that are decoded by the address decoder and the 74LS32 gates. The fourth I/O port location is used to reset the error latch so that new M1 cycles may be latched and another error can be detected.

A JUMP-ON-RESET CIRCUIT

A very useful circuit to have in a system is one that will automatically cause the processor to jump to a specific memory location at power-up and every time a reset occurs. All processors have some mechanism for doing this, but the starting

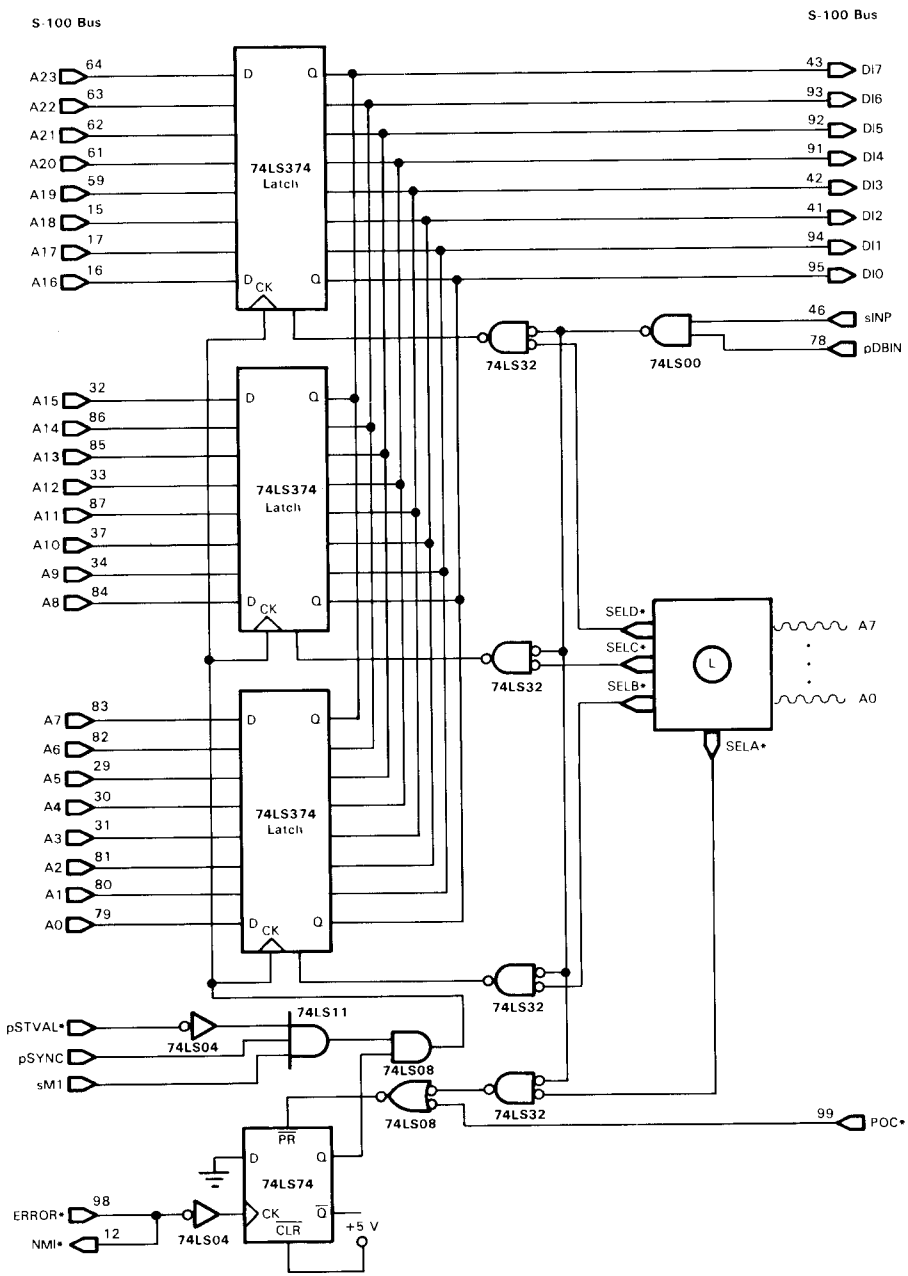


FIGURE 16-4. ERROR* Trap Circuit

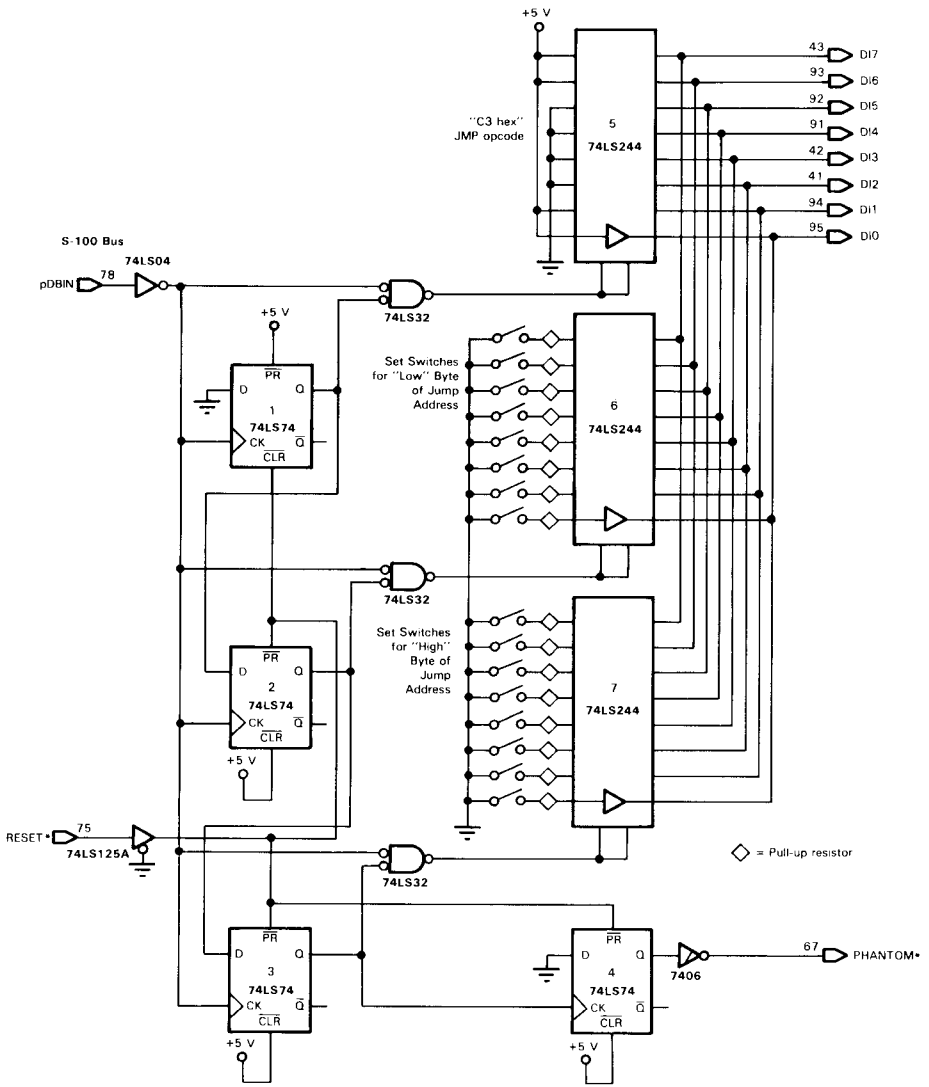


FIGURE 16-5. Jump-On-Reset Circuit

address is usually "hard-wired" inside the CPU chip itself. This address may not be convenient to use. The circuit in Figure 16-5 allows the CPU to jump to any address. This will occur at power-up and every time RESET* is asserted (usually by pushing the RESET button).

The "jump" op-code for 8080/8085/Z80 processors is hard-wired to the inputs of buffer 5 (in this case a $C3_{16}$). If other processors are used, a different op-code may be selected. The low byte of the jump address is set with the switches connected to the inputs of buffer 6 and the high byte of the address is set with the switches connected to the inputs of buffer 7. An open, or off, switch represents a 1 in the jump address, and a closed, or on, switch represents a 0.

Here's how the circuit works: when a RESET* occurs (remember that a RESET* will be generated along with POC* at power-up), flip-flop 1's output will be set low, and the outputs of flip-flop 2, 3, and 4 will be set high. The output of flip-flop 4 is inverted by the 7406, which will drive PHANTOM* low, disabling all the memory on the bus. The low at the output of flip-flop 1 is applied to one input of a low-true AND gate (74LS32). The other input gets the inverted pDBIN. When pDBIN goes high, the output of the OR gate will go low, enabling buffer 5. This places the JMP op-code onto the data input bus, which the processor will read.

When pDBIN falls, the low at the output of flip-flop 1 will be clocked into flip-flop 2, setting its output low. Flip-flop 1's output will now be high, which will inhibit buffer 5. Buffer 6 will now turn on when the next pDBIN is high, driving the low byte of the jump address onto the data input bus. When this pDBIN falls, the low at the output of flip-flop 2 will be clocked into flip-flop 3 and will in turn enable buffer 7 which will drive the high byte of the jump address onto the data input bus. When this read cycle is complete, flip-flop 3's output will go high, which will clock a low into flip-flop 4 (which has been high all this time, driving PHANTOM* low). This will cause PHANTOM* to return high, enabling the system memory and causing the processor to continue execution from the desired address.

The flip-flops will all remain in this state until another RESET* occurs, which will restart the process.

circuits not covered in this book

17

In this book we have presented many circuits and interfaces, but there are certain kinds of devices that we didn't describe or show how to construct. Some of these devices are: video and graphics circuits, dynamic memory interfaces, floppy and hard disk interfaces, cassette interfaces, CPU boards, and S-100 interfaces for non-S-100 based computers. The purpose of this chapter is to briefly explain why these items were not addressed in this text.

The basic reason is that the circuits we have presented are fairly complex in nature; the above circuits are much more complex than the ones we have shown you. Another reason is that most of the peripherals involved in the above interfaces (i.e., a hard disk drive or a dynamic memory IC) are tricky enough by themselves. Up to 90% of the S-100 card circuitry may be dedicated to accommodating the peripheral device and only 10% to the actual S-100 interface. We did not feel it was worthwhile to explain the 90% so that the 10% relevant to this text would be understandable. Most of that 10% ends up looking like a few parallel ports anyway, so you will still understand the underlying concepts should you ever encounter such interfaces.

We did not tell you how to build a cassette interface for two reasons: 1) There is no real "standard" for cassette interchange in the S-100 world, and 2) you should not waste your time with cassettes. We recommend that you buy a floppy disk system instead. You will quickly recover the cost of the floppy disk system in the time you won't spend waiting for the cassette to load.

We did not show you how to build a CPU board because each CPU IC has different problems associated with getting it on the S-100 Bus and meeting the IEEE standard. Some of these problems are minor and some are major. In any case, not only is building a CPU board a complex task (one error here can mess up

your whole system's timing), but if we showed you how to use CPU "A" then you would still need information on CPU "B", etc. You are much better off buying a CPU board from a manufacturer who guarantees it to meet the IEEE standard.

We did not show you how to interface a non-S-100 based computer (such as a TRS-80) to the S-100 Bus for several good reasons. One is that although the actual hardware interface may not be too difficult, the software considerations are usually overwhelming. Another is that the hardware interface might not be easy. In fact, it may take radical modifications to some of these computers to make an S-100 interface meet the IEEE standard (and even then some would not meet it). The last reason is that the expansion signals of most other computers are not standardized, and may change from model to model. Trying to keep up with that would be a nightmare.

We did not show you how to interface dynamic memory to the S-100 Bus because the problems involved would require almost a whole book by themselves. In addition to timing, board layout plays a major part in the proper operation of a dynamic memory card. A wire-wrapped dynamic memory board would be unreliable at best.

We did not show you how to interface floppy and hard disks to the S-100 Bus because the interface to the drives themselves is quite complicated. As with dynamic memories, the problems and considerations here would take a whole book. You are much better off taking advantage of a manufacturer's expertise in this area and buying a floppy or hard disk system that is already designed and debugged.

Finally, we did not show you how to build video and graphics interfaces. Devices of this type are best left to those who know the intricacies of video inside and out. If you are interested in graphics or video displays, the basic concepts we have presented in this text should help you in interfacing those circuits to the S-100 Bus. We recommend that you refer to *The CRT Controller Handbook* by Gerry Kane (Osborne/McGraw-Hill, 1981). Also, a number of excellent video and graphics boards are available commercially.

the ASCII character codes

b7 →					0	0	0	0	1	1	1	1
b6 →					0	0	1	1	0	0	1	1
b5 →					0	1	0	1	0	1	0	1
b4	b3	b2	b1	Row \ Column	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	¦
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

NUL	Null	DC1	Device control 1
SOH	Start of heading	DC2	Device control 2
STX	Start of text	DC3	Device control 3
ETX	End of text	DC4	Device control 4
EOT	End of transmission	NAK	Negative acknowledge
ENQ	Enquiry	SYN	Synchronous idle
ACK	Acknowledge	ETB	End of transmission block
BEL	Bell, or alarm	CAN	Cancel
BS	Backspace	EM	End of medium
HT	Horizontal tabulation	SUB	Substitute
LF	Line feed	ESC	Escape
VT	Vertical tabulation	FS	File separator
FF	Form feed	GS	Group separator
CR	Carriage return	RS	Record separator
SO	Shift out	US	Unit separator
SI	Shift in	SP	Space
DLE	Data link escape	DEL	Delete

hex, decimal, octal, binary conversion table

b

X \ Y	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111	Binary
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Hex
0000	000	001	002	003	004	005	006	007	008	009	010	011	012	013	014	015	Decimal
0	000	001	002	003	004	005	006	007	010	011	012	013	014	015	016	017	Octal
0001	016	017	018	019	020	021	022	023	024	025	026	027	028	029	030	031	Decimal
1	020	021	022	023	024	025	026	027	030	031	032	033	034	035	036	037	Octal
0010	032	033	034	035	036	037	038	039	040	041	042	043	044	045	046	047	Decimal
2	040	041	042	043	044	045	046	047	050	051	052	053	054	055	056	057	Octal
0011	048	049	050	051	052	053	054	055	056	057	058	059	060	061	062	063	Decimal
3	060	061	062	063	064	065	066	067	070	071	072	073	074	075	076	077	Octal
0100	064	065	066	067	068	069	070	071	072	073	074	075	076	077	078	079	Decimal
4	100	101	102	103	104	105	106	107	110	111	112	113	114	115	116	117	Octal
0101	080	081	082	083	084	085	086	087	088	089	090	091	092	093	094	095	Decimal
5	120	121	122	123	124	125	126	127	130	131	132	133	134	135	136	137	Octal
0110	096	097	098	099	100	101	102	103	104	105	106	107	108	109	110	111	Decimal
6	140	141	142	143	144	145	146	147	150	151	152	153	154	155	156	157	Octal
0111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	Decimal
7	160	161	162	163	164	165	166	167	170	171	172	173	174	175	176	177	Octal
1000	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	Decimal
8	200	201	202	203	204	205	206	207	210	211	212	213	214	215	216	217	Octal
1001	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	Decimal
9	220	221	222	223	224	225	226	227	230	231	232	233	234	235	236	237	Octal
1010	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	Decimal
A	240	241	242	243	244	245	246	247	250	251	252	253	254	255	256	257	Octal
1011	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	Decimal
B	260	261	262	263	264	265	266	267	270	271	272	273	274	275	276	277	Octal
1100	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	Decimal
C	300	301	302	303	304	305	306	307	310	311	312	313	314	315	316	317	Octal
1101	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	Decimal
D	320	321	322	323	324	325	326	327	330	331	332	333	334	335	336	337	Octal
1110	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	Decimal
E	340	341	342	343	344	345	346	347	350	351	352	353	354	355	356	357	Octal
1111	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	Decimal
F	360	361	362	363	364	365	366	367	370	371	372	373	374	375	376	377	Octal

memory addressing, hexadecimal/decimal

Extended Address Byte				High Byte				Low Byte			
Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

Examples

1. Hex-to-Decimal

```
Hex = C31B          49,152   C
                       768   3
                        16   1
                   +    11   B
          Decimal =  49,947
```

2. Decimal-to-Hex

```
Decimal =  43,390
           -40,960   A
           - 2,430
           - 2,304   9
               126
           -   112   7
Hex = A97E      14   E
```

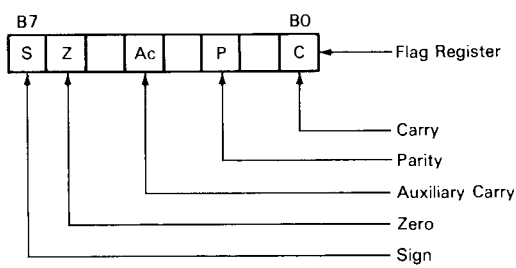
3. Extended Address Hex-to-Decimal

```
Hex = 2B5F6A  2,097,152   2
                720,896   B
                 20,480   5
                  3,840   F
                     96   6
              +      10   A
Decimal = 2,842,474
```

8080/8085 instructions

d

	A	B	C	D	E	H	L	M	Immed†	
AND*	87	80	81	82	83	84	85	86	C6	Add register to A
ADC*	8F	88	89	8A	8B	8C	8D	8E	CE	Add register to A with carry
ANA*	A7	A0	A1	A2	A3	A4	A5	A6	E6	AND register with A
CMP*	BF	B8	B9	BA	BB	BC	BD	BE	FE	Compare register with A
DCR**	3D	05	0D	15	1D	25	2D	35	-	Decrement register
INR**	3C	04	0C	14	1C	24	2C	34	-	Increment register
MOV A	7F	78	79	7A	7B	7C	7D	7E	3E	A ← register
MOV B	47	40	41	42	43	44	45	46	06	B ← register
MOV C	4F	48	49	4A	4B	4C	4D	4E	0E	C ← register
MOV D	57	50	51	52	53	54	55	56	16	D ← register
MOV E	5F	58	59	5A	5B	5C	5D	5E	1E	E ← register
MOV H	67	60	61	62	63	64	65	66	26	H ← register
MOV L	6F	68	69	6A	6B	6C	6D	6E	2E	L ← register
MOV M	77	70	71	72	73	74	75	-	36	(HL) ← register
ORA*	B7	B0	B1	B2	B3	B4	85	B6	F6	OR register with A
SBB*	9F	98	99	9A	9B	9C	9D	9E	DE	Subtract register from A with borrow
SUB*	97	90	91	92	93	94	95	96	D6	Subtract register from A
XRA*	AF	A8	A9	AA	AB	AC	AD	AE	EE	Exclusive-OR register with A



† Change last letter of mnemonic to I, in most cases (e.g., ADI instead of ADD)

	PSW, A	BC	DE	HL	SP	
DAD***	-	09	19	29	39	Add pair to HL
DCX**	-	0B	1B	2B	3B	Decrement register pair
INX**	-	03	13	23	33	Increment register pair
LDAX	-	0A	1A	7E	-	Load A indirect (register pair holds address)
STAX	-	02	12	77	-	Store A indirect (register pair holds address)
LXI	-	01	11	21	31	Load register pair immediate
POP	F1	C1	D1	E1	-	Pop register pair from stack
PUSH	FS	C5	D5	E5	-	Push register pair onto stack
LHLD	-	-	-	2A	-	Load HL direct (addr)
SHLD	-	-	-	22	-	Store HL direct (addr)

	0	1	2	3	4	5	6	7	
RST i	C7	CF	D7	DF	E7	EF	F7	FF	Restart call to location (i × 8)

	Uncond.	Z Zero	NZ Not Zero	C Carry	NC Not Carry	P Plus	M Minus	PE Even Parity	PO Odd Parity	
C(ALL)	CD	CC	C4	DC	D4	F4	FC	EC	E4	Call subroutine if condition is true
J(MP)	C3	CA	C2	DA	D2	F2	FA	EA	E2	Jump if condition is true
R(ET)	C9	C8	C0	D8	D0	F0	F8	E8	E0	Return if condition is true

CMC***	3F	Complement carry flag
CMA	2F	Complement A
DAA*	27	Decimal adjust A
DI	F3	Disable interrupts
EI	FB	Enable interrupts
XCHG	EB	Exchange DE and HL
XTHL	E3	Exchange HL and top of stack
HLT	76	Halt processor
IN Port	DB	Input to A
OUT Port	D3	Output from A
NOP	00	No operation

PCHL	E9	Jump to (HL)
RAL	17	Rotate A left through carry
RAR	1F	Rotate A right through carry
RLC	07	Rotate A left circular
RRC	0F	Rotate A right circular
STC***	37	Set carry flag
STA	32	Store A
LDA	3A	Load A
SPHL	F9	Load SP with HL
RIM	20	Read interrupt mask } 8085
SIM	30	Set interrupt mask } only

* All flags affected
** All flags except carry flag affected
*** Only carry flag affected

Z80 instructions with 8080 cross references

8080 Mnemonic	Z80 Mnemonic	A	B	C	D	E	H	L	M	Immed	(IX + d)	(IY + d)	Description
ADD	ADD	87	B0	81	82	83	84	85	86	C6	DD86	FD86	Add to A
ADC	ADC	8F	88	89	8A	8B	8C	8D	8E	CE	DD8E	FD8E	Add with carry to A
ANA	AND	A7	A0	A1	A2	A3	A4	A5	A6	E6	DDA6	FDA6	AND with A
	BIT0	CB47	CB40	CB41	CB42	CB43	CB44	CB45	CB46	-	DDCB46	FDCB46	Test bit 0
	BIT1	CB4F	CB48	CB49	CB4A	CB4B	CB4C	CB4D	CB4E	-	DDCB4E	FDCB4E	Test bit 1
	BIT2	CB57	CB50	CB51	CB52	CB53	CB54	CB55	CB56	-	DDCB56	FDCB56	Test bit 2
	BIT3	CB5F	CB58	CB59	CB5A	CB5B	CB5C	CB5D	CB5E	-	DDCB5E	FDCB5E	Test bit 3
	BIT4	CB67	CB60	CB61	CB62	CB63	CB64	CB65	CB66	-	DDCB66	FDCB66	Test bit 4
	BIT5	CB6F	CB68	CB69	CB6A	CB6B	CB6C	CB6D	CB6E	-	DDCB6E	FDCB6E	Test bit 5
	BIT6	CB77	CB70	CB71	CB72	CB73	CB74	CB75	CB76	-	DDCB76	FDCB76	Test bit 6
	BIT7	CB7F	CB78	CB79	CB7A	CB7B	CB7C	CB7D	CB7E	-	DDCB7E	FDCB7E	Test bit 7
CMP	CP	8F	B8	B9	BA	BB	BC	BD	BE	FE	DDBE	FDBE	Compare with A
DCR	DEC	3D	05	0D	15	1D	25	2D	35	-	DD35	FD35	Decrement
INR	INC	3C	04	0C	14	1C	24	2C	34	-	DD34	FD34	Increment
	IN(C)	ED78	ED40	ED48	ED50	ED58	ED60	ED68	-	-	-	-	Input to register from port addressed by C
MOVA	LDA	7F	78	79	7A	7B	7C	7D	7E	3E	DD7E	FD7E	Move to A
MOVB	LDB	47	40	41	42	43	44	45	46	06	DD46	FD46	Move to B
MOVC	LDC	4F	48	49	4A	4B	4C	4D	4E	0E	DD4E	FD4E	Move to C
MOVD	LDD	57	50	51	52	53	54	55	56	16	DD56	FD56	Move to D
MOVE	LDE	5F	58	59	5A	5B	5C	5D	5E	1E	DD5E	FD5E	Move to E
MOVH	LDH	67	60	61	62	63	64	65	66	26	DD66	FD66	Move to H
MOVL	LDL	6F	68	69	6A	6B	6C	6D	6E	2E	DD6E	FD6E	Move to L
MOVM	LD(HL)	77	70	71	72	73	74	75	-	-	-	-	Move to memory
	LD(IX+d)	DD77	DD70	DD71	DD72	DD73	DD74	DD75	-	DD36	-	-	Move to memory (IX + d)
	LD(IY+d)	FD77	FD70	FD71	FD72	FD73	FD74	FD75	-	FD36	-	-	Move to memory (IY + d)
ORA	OR	B7	B0	B1	B2	B3	B4	B5	B6	F6	DDB6	FDB6	OR with A
	OUT(C)	ED79	ED41	ED49	ED51	ED59	ED61	ED69	-	-	-	-	Output register to port addressed by C
	RES0	CB87	CB80	CB81	CBB2	CBB3	CBB4	CB85	CB86	-	DDCBB6	FDCB86	Reset bit 0
	RES1	CB8F	CB88	CB89	CB8A	CB8B	CB8C	CB8D	CB8E	-	DDCB8E	FDCB8E	Reset bit 1
	RES2	CB97	CB90	CB91	CB92	CB93	CB94	CB95	CB96	-	DDCB96	FDCB96	Reset bit 2
	RES3	CB9F	CB98	CB99	CB9A	CB9B	CB9C	CB9D	CB9E	-	DDCB9E	FDCB9E	Reset bit 3
	RES4	CBA7	CBA0	CCA1	CBA2	CBA3	CBA4	CBA5	CBA6	-	DDCBA6	FDCBA6	Reset bit 4
	RES5	CBAF	CBAB	CBA9	CBAA	CBAB	CBAC	CBAD	CBAE	-	DDCBAE	FDCBAE	Reset bit 5
	RES6	CBB7	CB80	CBB1	CBB2	CBB3	CBB4	CBB5	CBB6	-	DDCBB6	FDCBB6	Reset bit 6
	RES7	CBBF	CBBB	CBB9	CBBA	CBBB	CBBC	CBBD	CBBE	-	DDCBBE	FDCBBE	Reset bit 7
	RL	CB17	CB10	CB11	CB12	CB13	CB14	CB15	CB16	-	DDCB16	FDCB16	Rotate left through carry
	RLC	CB07	CB00	CB01	CB02	CB03	CB04	CB05	CB06	-	DDCB06	FDCB06	Rotate left with branch carry
	RR	CB1F	CB18	CB19	CB1A	CB1B	CB1C	CB1D	CB1E	-	DDCB1E	FDCB1E	Rotate right through carry
	RRC	CB0F	CB08	CB09	CB0A	CB0B	CB0C	CB0D	CB0E	-	DDCB0E	FDCB0E	Rotate right with branch carry
SBB	SBC	9F	98	99	9A	9B	9C	9D	9E	DE	DD9E	FD9E	Subtract from A with borrow
	SET0	CBC7	CBC0	CBC1	CBC2	CBC3	CBC4	CBC5	CBC6	-	DDCBC6	FDCBC6	Set bit 0
	SET1	CBCF	CBC8	CBC9	CBCA	CBCB	CBCC	CBCD	CBCE	-	DDCBCE	FDCBCE	Set bit 1
	SET2	CBD7	CBD0	CBD1	CBD2	CBD3	CBD4	CBD5	CBD6	-	DDCBD6	FDCBD6	Set bit 2
	SET3	CBDF	CBD8	CBD9	CBDA	CBDB	CBDC	CBDD	CBDE	-	DDCBDE	FDCBDE	Set bit 3
	SET4	CBE7	CBE0	CBE1	CBE2	CBE3	CBE4	CBE5	CBE6	-	DDCBE6	FDCBE6	Set bit 4
	SET5	CBEF	CBE8	CBE9	CBEA	CBEB	CBEC	CBED	CBEE	-	DDCBEE	FDCBEE	Set bit 5
	SET6	CBF7	CBF0	CBF1	CBF2	CBF3	CBF4	CBF5	CBF6	-	DDCBF6	FDCBF6	Set bit 6
	SET7	CBFF	CBF8	CBF9	CBFA	CBFB	CBFC	CBFD	CBFE	-	DDCBFE	FDCBFE	Set bit 7
	SLA	CB27	CB20	CB21	CB22	CB23	CB24	CB25	CB26	-	DDCB26	FDCB26	Shift left into carry and clear LSB
	SRA	CB2F	CB2B	CB29	CB2A	CB2B	CB2C	CB2D	CB2E	-	DDCB2E	FDCB2E	Shift right into carry and preserve MSB
	SRL	CB3F	CB3B	CB39	CB3A	CB3B	CB3C	CB3D	CB3E	-	DDCB3E	FDCB3E	Shift right into carry and clear MSB
SUB	SUB	47	90	91	92	93	94	95	96	D6	DD96	FD96	Subtract contents from A
XRA	XOR	AF	A8	A9	AA	AB	AC	AD	AE	AF	DDAE	FDAE	Exclusive-OR contents with A

8080 cross references

8080 Mnemonic	Z80 Mnemonic	(PSW'A)	(BC)	(DE)	(HL)	SP	IX	IY	
DAD	ADD HL	-	09	19	29	39	-	-	Add to HL
	ADD IX	-	DD09	DD19	-	DD39	DD29	-	Add to IX
	ADD IY	-	FD09	FD19	-	FD39	-	FD29	Add to IY
	ADC HL	-	ED4A	ED5A	ED6A	ED7A	-	-	Add to HL with carry
	SBC HL	-	ED42	ED52	ED62	ED72	-	-	Subtract from HL with borrow
DCX	DEC	-	0B	1B	2B	3B	DD2B	FD2B	Decrement register pair
INX	INC	-	03	13	23	33	DD23	FD23	Increment register pair
LDAX	LDA(r)	-	0A	1A	7E	-	-	-	Load A from memory (register pair)
STAX	LD (r) A	-	02	12	77	-	-	-	Store A in memory (register pair)
LXI	LD	-	01	11	21	31	DD21	FD21	Load register pair immediate
POP	POP	F1	C1	D1	E1	-	DDE1	FDE1	Pop register pair from stack
PUSH	PUSH	F5	C5	D5	E5	-	DDE5	FDE5	Push register pair onto stack
LHLD	LD r,(addr)	-	ED4B	ED5B	2A	ED7B	DD2A	FD2A	Load register from memory
SHLD	LD (addr),r	-	ED43	ED53	22	ED73	DD22	FD22	Store register in memory

Z80 Mnemonic	I (Increment)	IR (Increment and repeat)	D (Decrement)	DR (Decrement and repeat)	
CP	EDA1	EDB1	EDA9	EDB9	Compare A to memory (HL), increment or decrement HL, decrement BC; stop when BC = 0 or match is found
LD	EDA0	EDB0	EDA8	EDB8	Move from memory (HL) to memory (DE), increment or decrement address, decrement BC; stop when BC = 0
OUT	EDA3	EDB3	EDAB	EDBB	Output memory (HL) to port, increment or decrement BC; stop when BC = 0
IN	EDA2	EDB2	EDAA	EDBA	Input to memory (HL) from port, increment or decrement HL, decrement BC; stop when BC = 0

8080 Mnemonic	Z80 Mnemonic	0	1	2	3	4	5	6	7	
RST	RST	C7	CF	D7	DF	E7	EF	F7	FF	Restart call at location (i × 8)

8080 Mnemonic	Z80 Mnemonic	Uncond.	Zero	Not Zero	Carry	Not Carry	Plus	Minus	Even Parity	Odd Parity	
CALL	CALL	CD	CC	C4	DC	D4	F4	FC	EC	E4	Call subroutine if condition is true
JMP	JP	C3	CA	C2	DA	D2	F2	FA	EA	E2	Jump if condition is true
	JR	18	28	20	38	30	-	-	-	-	Jump relative if condition is true

S-100 bus electrical specifications

(0°C to 70°C)

POWER BUS

	Instantaneous		Average	
Line	**Max.**	**Min.**	**Max.**	**Min.**
+8 V	+25 V	+7 V	+11 V	--
+16 V	+35 V	+14.5 V	+21.5 V	--
−16 V	−14.5 V	−35 V	−21.5 V	--
all logic	5 V	0 V	+5 V	0 V

DRIVERS

	Output Voltage
Low state:	$V_{OL} \leq +0.5$ V @ $I_{OL} = 24$ mA
High state:	$V_{OH}, I_{OH} \geq +2.4$ V @ $I_{OH} = -2$ mA (except open collector types)
Leakage current (high state):	$\leq \pm 25\ \mu$A
Rise time:	5-50 ns (at rated capacitive load)

RECEIVERS

Source current:	<0.5 mA @ 0.5 V
Sink current:	$<50\ \mu$A @ 2.4 V
Voltage low state:	≥ 0.8 V
Voltage high state:	≤ 2.0 V

INTERNAL CAPACITIVE LOADS (@ 25° C)

Drivers:	≤ 15 pF
Receivers:	≤ 10 pF
Transceivers:	≤ 20 pF
Card level inputs:	≤ 25 pF

standard specifications for S-100 bus interface devices

This proposed standard eliminates many of the problems in the S-100 bus and upgrades it for 16-bit microprocessors. It is offered here for public comment before submission to the IEEE Standards Board.

IEEE Task 696.1 / D2

Kells A. Elmquist, InterSystems Inc.
Howard Fullmer, Parasitic Engineering Inc.
David B. Gustavson, Stanford Linear Accelerator Center
George Morrow, Thinker Toys

Introductory comments by
Robert G. Stewart, Chairman,
IEEE-CS Computer Standards Committee

The following draft of a proposed standard for the S-100 bus is the culmination of over a year and a half of effort to eliminate many of the bus's problems and to upgrade it to be suitable for 16-bit microprocessors. The address bus has been extended to 24 bits, the data in and data out buses ganged to form a 16-bit wide data bus for 16-bit transactions, and two additional handshaking lines added to permit intermixing of 8- and 16-bit memory cards.

A binary encoded multiple master arbitration bus permits up to 16 masters on the bus. The necessary logic can be implemented in one chip. Additional ground lines, a power fail line, and an error line have been added. Three lines termed NDEF—for not to be defined—have been allotted to allow leeway to implementers for specialized use. Such use must be specified in all literature. Five lines are RFU—reserved for future use. Some lines formerly used for front panel purposes have been deleted, with the intention that such lines can best be handled by a jumper cable from the CPU card to the front panel. A DMA protocol is specified which provides overlap of the control lines at the beginning and end of the transition between permanent and temporary masters. This allows the address, data, and control buses to settle before information is transferred.

As a bit of personal testimony, I implemented the new DMA protocol on my own system, which includes a Digital Systems dual floppy disk interfaced to a MITS Altair 8800, using DMA for disk transfers. The soft error rate, presumably due to glitching on the positive true logic lines, dropped from a situation where a file would be seriously munged in a few hours to the present situation where I can work for days on end without an observable error.

We have observed a new typographic convention in publishing the proposed standard. The use of an overbar to denote electrically low active or negative true logic lines has been replaced by a postfix asterisk to avoid confusion with Boolean negation and permit typing on word processing systems. This is verbalized by the word "star," replacing the prior word "bar." The Boolean negation overbar can be optionally replaced by a prefix minus sign, with parentheses if needed.

The named authors of the standard were evenly divided as to whether the asterisk should be included in logic equations and state diagrams as well as in electrical signal names and timing diagrams. Two authors believe that the asterisk, when thought of as a designator rather than as the negation operator, adds clarity and consistency, and lessens the need to remember or look up the electrically active level when converting from logic to electrical representations. Such use makes logic state diagrams more directly useful for interpreting oscilloscope or logic analyzer waveforms.

The other two authors feel that the inclusion of the asterisk in the name of a logic state or variable is likely to carry with it the implication of logical negation, thus causing the logic statements to be interpreted incorrectly. Furthermore, they assert that many designers think mostly in terms of electrical levels, with high being true, which again causes logic statements to be interpreted incorrectly. They propose to resolve this hazard by removing the electrical information, i.e., the asterisk, from the variable name when it is us-

ed in a logic context as opposed to an electrical or timing context.

A compromise has been reached where the asterisk is not used in the context of logic equations, but is included elsewhere in the document. We solicit feedback from the readers on these two points of view.

The S-100 bus subcommittee has been ably chaired by George Morrow and Howard Fullmer. Both of them provided invaluable technical insights which have been incorporated throughout the draft standard. John Walker of Marinchip Systems suggested the method of using 16-bit memory and interface cards interchangeably with 8-bit cards. David Gustavson and Leo Paffrath of SLAC suggested the bus arbitration scheme which has also been implemented on the Department of Energy's Fastbus. Howard Fullmer suggested the DMA overlap protocol which lowers glitching noise. Kells Elmquist of InterSystems offered a critique of the draft published in May 1978 in *Computer* and provided many useful suggestions for improvement. He carefully investigated numerous timing and electrical alternatives and resolved many open questions relating to the standard. Kells wrote the final version of the draft for submission to and revision by the subcommittee.

The IEEE Computer Society is publishing this standard in draft form to allow you to comment upon it prior to submission to the IEEE Standards Board for adoption as an IEEE standard. For example, should the data bus be extended to 32 bits, and if so, how? Your comments should be sent to George Morrow by August 15, 1979, with copies to Gordon Force. Mr. Morrow's address is:

George Morrow
Thinker Toys
5221 Central Avenue
Richmond, California 94804

If you would like to participate in other standardization efforts of the Microprocessor Standards Committee, please contact its chairman:

Gordon Force
Logical Solutions
1128 Amur Creek Court
San Jose, California 95051

Finally, preparation of this proposed standard has benefited from the contributions of many individuals and companies. We indeed thank them all.■

The proposed standard

1.0 General

1.1 Scope

This standard applies to interface systems for computer system components interconnected via a 100-line parallel backplane commonly known as the S-100 bus.

It applies to microprocessor computer systems, or portions of them, where

1) Data exchanged among the interconnected devices is digital (as distinct from analog).
2) The total number of interconnected devices is small (22 or fewer).
3) The total transmission path length among interconnected devices is electrically short (25" or less). That is, transmission line propagation delays are not important.
4) The maximum data rate of any signal on the bus is low (less than or equal to 6 MHz).

1.2 Object

This standard is intended:

1) To define a rational, general-purpose interface system for designers of new computer system components that will ensure their compatibility with present and future S-100 computer systems.
2) To provide the microprocessor computer system user with compatible device families which will communicate in an unambiguous way without modification, from which a modularly expandable computer system may be constructed.
3) To enable the interconnection of independently manufactured devices into a single system.
4) To specify terminology and definitions related to the system.
5) To define a system with the minimum number of restrictions on the performance characteristics of devices connected to the system.
6) To define a system that, of itself, is of relatively low cost, and allows the interconnection of low cost devices.
7) To define a system that is easy to use.

1.3 Definitions

The following definitions apply for the purpose of this standard. This section contains only general definitions. Detailed definitions are given in other sections as appropriate.

1.3.1 General system terms

Compatibility. The degree to which devices may be interconnected and used without modification, when designed as defined in Sections 2, 3, and 4 of this standard.

Interface. A shared boundary between parts of a computer system, through which information is conveyed.

Interface system. The device independent functional, electrical, and mechanical elements of an interface necessary to effect unambiguous communication among a set of devices. Driver and receiver circuits, signal line descriptions, timing and control conventions, message transfer protocols, and functional logic circuits are typical interface system elements.

System. A set of interconnected elements constituted to achieve a given objective by performing specified functions.

1.3.2 Signals and paths

Assert. To drive a signal line to the true state. The true state is either a high or low state, as specified for each signal.

Bidirectional bus. A bus used by any individual device, or set of devices, for the two-way transmission of messages, that is, both input and output.

Bit-parallel. A set of concurrent data bits present on a like number of signal lines used to carry information. Bit-parallel data bits may be acted upon concurrently as a group or independently as individual data bits.

Bus. A set of signal lines used by an interface system, to which a number of devices are connected, and over which messages are carried.

Byte. A set of bit-parallel signals corresponding to binary digits operated on as a unit. Connotes a group of eight bits where the most significant bit carries the subscript 7 and the least significant bit carries the subscript 0.

Byte-serial. A sequence of bit-parallel data bytes used to carry information over a common bus.

High state. The electrically more positive signal level used to assert a specific message content associated with one of two binary logic states.

Low state. The electrically less positive signal level used to assert a specific message content associated with one of two binary logic states.

Signal. The physical representation which conveys data from one point to another. For the purpose of this standard, this applies to digital electrical signals only.

Signal level. The magnitude of a signal when considered in relation to an arbitrary reference magnitude (voltage in the case of this standard).

Signal line. One of a set of signal conductors in an interface system used to transfer messages among interconnected devices.

Signal parameter. That parameter of an electrical quantity whose values or sequence of values convey information.

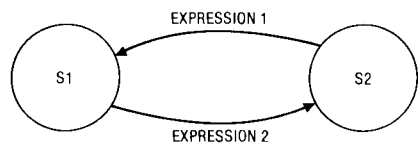
Unidirectional bus. A bus used by a device for one-way transmission of messages, that is, either input only or output only.

Word. A set of bit-parallel signals corresponding to binary digits and operated on as a unit. Usually connotes a group of 16 bits where the most significant bit carries the subscript 15 and the least significant bit carries the subscript 0.

1.4 State diagram notation

Each state that an interface function can assume is represented graphically by a circle. A mnemonic is used within the circle to identify the state.

All permissible transitions between states of an interface function are represented graphically by arrows between them. Each transition between states may be qualified by an expression whose value must be either true or false. If a state transition is not qualified by an expression it is assumed that transition from one state to another will occur after a minimum time period, as indicated in the timing specifications. An interface function must enter the state pointed to if and only if the driving expression becomes true, or in the case of a time dependent transition, as soon as the minimum specified time has passed.



An expression consists of two parts, a driving expression and a driven expression, separated by a slash (/). The driving expression is mandatory and specifies the conditions necessary for the state transition. The driven expression is optional and is used to indicate signal transitions as a result of the state transition. A signal transition is indicated by the signal name followed by an equal sign (=), followed by an indication of the state attained by the signal as a result of the transition. A driving expression consists of one or more messages used in conjunction with the operators AND (a•b), OR (a+b), and NOT (−a). Precedence is defined by parentheses. An example expression is: (driving/driven)

$$A \cdot (B+C) \ / \ D=F(ALSE), \ E=T(RUE)$$

If A AND (B OR C) is true, then D is forced false and E is forced true, and the state transition takes place.

1.5 Logical and electrical state relationships

This standard makes a distinction between the logical function of a signal and its electrical implementation. All equations in this standard are logic equations, not electrical equations (unless otherwise stated), and are written in terms of logic states. The use of the term "active" for the purpose of this standard is synonymous with the logic state true.

There are two types of electrical implementation of the logic states:

Active high signals. Active high signals are represented without a suffix after the signal name mnemonic (i.e. ABCD).

LOGIC STATE	BINARY STATE	ELECTRICAL SIGNAL LEVEL	ELECTRICAL STATE
FALSE (F)	0	CORRESPONDS TO ≤.8 V, CALLED THE LOW STATE.	L
TRUE (T)	1	CORRESPONDS TO ≥2.0 V, CALLED THE HIGH STATE.	H

Active low signals. Active low signals are represented with an asterisk suffix after the mnemonic (i.e. ABCD*).

LOGIC STATE	BINARY STATE	ELECTRICAL SIGNAL LEVEL	ELECTRICAL STATE
FALSE (F)	0	CORRESPONDS TO ≥2.0 V, CALLED THE HIGH STATE.	H
TRUE (T)	1	CORRESPONDS TO ≤.8 V, CALLED THE LOW STATE.	L

In translating a logic equation into an electrical implementation, care must be taken to account for the active-high or active-low character of the electrical signal. For example, the logic equation

$$MWRT = pWR \cdot -sOUT, \text{ (logic equation)}$$

when implemented electrically, becomes

$$MWRT = (-pWR^*) \cdot -sOUT, \text{ (electrical equation)}$$

since pWR* is the electrical signal carrying the pWR information on the bus.

Note that this is equivalent to

$$MWRT = -(pWR^* + sOUT), \text{ (electrical equation)}$$

by deMorgan's theorem; consequently, a single two-input NOR gate is sufficient to implement MWRT, if it meets the loading and drive requirements.

The edge or change of electrical value of an electrical signal on a timing diagram which causes a transition change of the variable as a logic variable from false to true is:

Signal	*Edge*
active high	rising
active low	falling

Logic equations in state diagrams are written in terms of logic state, not electrical state.

The suffix asterisk "*" is not a negation operator. It is a designator (like a comment or footnote) attached to a name, telling the reader what the relationship is between the truth state and the electrical state. That is, this variable is true when the line on the bus is low.

A prefix minus sign "–" represents the logical negation operator and is equivalent to the use of an overbar. Parentheses are used to enclose the negated variable when required for clarity.

1.6 Interface system overview

1.6.1 Interface system objective

The overall purpose of the interface system is to provide an effective communication link over which messages are carried in an unambiguous way among a group of interconnected devices.

Messages in an interface system belong to either of two broad categories:

1) Messages used to manage the interface system itself, called interface messages.
2) Messages used by the devices interconnected by the interface system, and carried by that system, but not part of the interface system itself (i.e. data). These are called device dependent messages.

The interface system herein described comprises the necessary functional and electrical specifications for interface messages to effect the objective of this standard, but it is beyond the scope of this standard to specify the nature or meaning (other than electrical signal level) of device dependent messages.

1.6.2 Fundamental communication capabilities

An effective communication link requires two basic functional elements to organize and manage the flow of information among devices:

1) A device acting as a bus master.
2) A device acting as a bus slave.

All data transfer communications between a bus master and a bus slave are carried out in terms of a generalized bus cycle generated by the bus master and responded to by the addressed bus slave.

In the context of the interface system described by this standard:

1) A device acting as a bus master has the capability to address all bus slaves, or some portion of them, by generating all interface messages necessary to effect a bus cycle, and has the capability to transfer device dependent messages to or from the addressed slave as a part of that bus cycle.
2) A device acting as a bus slave monitors all bus cycles, and has the capability, thus, to be addressed by the bus master and to transfer device dependent messages to or from the bus master.

Bus master and bus slave capabilities occur both individually and collectively in devices interconnected via the S-100 interface system.

1.6.3 Message paths and bus structure

The S-100 interface system consists of a set of signal lines used to carry all information, interface messages and device dependent messages among interconnected devices.

The bus structure is organized into eight sets of signal lines:

1) Data bus— 16 signal lines.
2) Address bus— 16 or 24 signal lines.
3) Status bus— 8 signal lines.
4) Control output bus— 5 signal lines.
5) Control input bus— 6 signal lines.
6) DMA control bus— 8 signal lines.
7) Vectored interrupt bus— 8 signal lines.
8) Utility bus— 20 signal lines.

2.0 Functional specification

2.1 Functional partition

Functional devices interconnected via the interface system are divided into two broad classifications, bus masters and bus slaves, according to their relationship to the generation and reception of interface messages.

Devices acting as bus masters are responsible for the initiation of all bus cycles, and for the generation of all signals necessary for the conduction of an unambiguous bus cycle. These signals are termed type M signals, and consist of the address, status, and control buses. Device dependent messages are transmitted and received on the data bus.

Bus masters are subdivided into two classifications, permanent masters and temporary masters. A permanent bus master (generally a CPU) is the highest priority master in the interface system. A temporary master may request the bus from the permanent master for an arbitrary number of bus cycles, and then returns control of the bus to the permanent master. The transfer of bus control from a permanent master to a temporary master and back to the permanent master is termed a DMA cycle.

The difference between a permanent bus master and a temporary bus master is that:

1) Only one permanent master may exist within the interface system, whereas up to 16 temporary masters may co-exist in a single system.
2) A temporary master is not subject to a DMA cycle, that is, there are no nested DMA operations.

Devices acting as bus slaves are bus cycle receptors. A bus slave monitors all bus cycles and, if addressed during a particular bus cycle, accepts or sends the requested device dependent message on the data lines. While bus masters must generate a specific set of signals in order to assure an unambiguous bus cycle, a bus slave need only examine and generate that subset of bus signals necessary to communicate with bus masters.

2.2 Signal lines

2.2.1 General

The bus is a collection of message paths defined relative to the current bus master. They are:

1) Address bus.
2) Status bus.
3) Data input/output bus.
4) Control output bus.
5) Control input bus.
6) DMA control bus.
7) Vectored interrupt bus.
8) Utility bus.

The nature and use of each bus is specified in the following sections.

2.2.2 Address bus

The address bus consists of 16 or 24 bit-parallel signal lines used to select a specific location in memory or a specific input/output device for communication during the current bus cycle.

All bus masters must assert at least 16 address bits, but may assert 24 address bits if extended address capability is desired. Validity of the address bus is defined in 2.7.3.

Table 1 summarizes address usage for various bus cycles.

Table 1.
Address usage for different bus cycles.

CYCLE TYPE	STANDARD ADDRESSING	EXTENDED ADDRESSING
MEMORY READ MEMORY WRITE M1 (OP-CODE FETCH)	A0-A15	A0-A23
INPUT OUTPUT	A0-A7†	A0-A15
INTERRUPT ACKNOWLEDGE	NONE	NONE
HALT ACKNOWLEDGE	NONE	NONE

† see 2.2.2.3

2.2.2.1 Standard memory addressing

The standard memory address bus consists of 16 lines specifying 1 of 64K memory locations. These 16 lines are named A0 through A15, where A15 is the most significant bit.

2.2.2.2 Extended memory addressing

The extended memory address bus consists of 24 lines specifying 1 of 16 million memory locations. These 24 lines are named A0 through A23, where A23 is the most significant bit.

2.2.2.3 Standard input/output device addressing

The standard I/O device address bus consists of 8 lines, A0 through A7, specifying 1 of 256 I/O devices. A7 is the most significant bit.

NOTE: The I/O device address has traditionally been duplicated onto the high order address byte, A15-A8. While this is considered acceptable procedure, it is not recommended for new designs as it complicates expansion to extended I/O device addressing.

2.2.2.4 Extended input/output device addressing

The extended I/O device address bus consists of 16 lines, A0 through A15, specifying 1 of 64K devices. A15 is the most significant bit.

2.2.3 Status bus

The status bus consists of eight lines which identify the nature of the bus cycle in progress, and qualify the nature of the address on the address bus.

The mnemonics for status lines always begin with a lower-case s.

The 8 status lines are:

1) Memory read— sMEMR.
2) Op-code fetch— sM1.
3) Input— sINP.
4) Output— sOUT.
5) Write cycle— sWO*.
6) Interrupt acknowledge— sINTA.
7) Halt acknowledge— sHLTA.
8) Sixteen-bit data transfer request— sXTRQ*.

The 8 lines on the status bus must be generated by the current bus master.

Validity of the status bus is given in 2.7.3.

2.2.3.1 Status memory write

One relevant status signal is not directly available on the bus, but may be created by the combination of two others. Status Memory Write is defined as:

sMemory Write =
(−sOUT) • sWO, (logic equation)

that is, status memory write is true when sOUT is false and sWO (write) is true.

2.2.3.2 Status usage chart

Table 2 gives the status word definition for all possible bus cycles. (W) refers to word (16-bit data path) operations; (B) refers to byte (8-bit data path) operations. H=high state. L=low state. X=don't care.

Table 2.
Status usage chart.

STATUS BITS		sMEMR	sM1	sWO*	sOUT	sINP	sINTA	sHLTA	sXTRQ*
CYCLE TYPE									
MEMORY READ	(B)	H	L	H	L	L	L	L	H
	(W)	H	L	H	L	L	L	L	L
OP-CODE FETCH	(B)	H	H	H	L	L	L	L	H
	(W)	H	H	H	L	L	L	L	L
MEMORY WRITE	(B)	L	L	L	L	L	L	L	H
	(W)	L	L	L	L	L	L	L	L
OUTPUT	(B)	L	L	L	H	L	L	L	H
	(W)	L	L	L	H	L	L	L	L
INPUT	(B)	L	L	H	L	H	L	L	H
	(W)	L	L	H	L	H	L	L	L
INTERRUPT ACKNOWLEDGE	(B)	L	X	H	L	L	H	L	H
	(W)	L	X	H	L	L	H	L	L
HALT ACKNOWLEDGE		X	X	H	L	L	L	H	X

WHERE:
H = HIGH STATE
L = LOW STATE
X = DON'T CARE
W = 16-BIT OPERATION
B = 8-BIT OPERATION

2.2.4 Data bus

Data input and data output are always specified relative to the current bus master. Data transmitted by the current bus master to a bus slave is called data output. Data received by the current bus master from a bus slave is called data input.

The data bus consists of 16 lines grouped as two unidirectional 8-bit buses for byte operations and as a single bidirectional bus for 16-bit word operations.

2.2.4.1 Byte operations

Two unidirectional 8-bit buses are used for byte data transfers. Data output appears on the data output bus (DO0-DO7), where DO7 is the most significant bit.

Data input appears on the data input bus (DI0-DI7), where DI7 is the most significant bit.

2.2.4.2 Word operations

For 16-bit data transfers the DI and the DO buses are ganged together, creating a single 16-bit bidirectional bus. Two signal lines control the ganging of the data buses, sixteen request (sXTRQ*) and sixteen acknowledge (SIXTN*). When both of these lines are true (in the low state), the data buses are ganged with DO0 corresponding to DATA 0 and DI7 corresponding to DATA 15, the most significant bit.

Complete specification of the 8/16-bit protocol is given in 2.6.

2.2.5 Control output bus

The 5 lines of the control output bus determine the timing and movement of data during any bus cycle. The mnemonics for the control output lines always begin with a lower-case p.

The five lines are:

1) pSYNC, which indicates the start of a new bus cycle.
2) pSTVAL*, which in conjunction with pSYNC indicates that stable address and status may be sampled from the bus in the current cycle.
3) pDBIN, a generalized read strobe that gates data from an addressed slave onto the data bus.
4) pWR*, a generalized write strobe that writes data from the data bus into an addressed slave.
5) pHLDA, the hold acknowledge signal that indicates to the highest priority temporary master that the permanent master is relinquishing control of the bus.

The control output signals are subject to the functional and timing disciplines given in 2.7, 3.8, and 3.9.

2.2.6 Control input bus

The six lines of the control input bus allow bus slaves to synchronize the operations of bus masters with conditions internal to the bus slave (e.g., data not ready), and to request operations of the permanent master (e.g., interrupt or hold).

The six control input lines are:

1) RDY
2) XRDY

3) INT*
4) NMI*
5) HOLD*
6) SIXTN*

2.2.6.1 Ready lines

The ready lines are used by bus slaves to synchronize bus masters to the response speed of the slave. Thus cycles are suspended and wait states inserted until both ready lines are asserted.

The RDY line is the general ready line for bus slaves. It is specified as an open collector line.

The XRDY line is a special ready line commonly used by front panel devices to stop and single step bus masters. As it is not specified as an open collector line, it should not be used by other bus slaves, since a bus conflict may exist.

2.2.6.2 Interrupt lines

The two interrupt lines, INT* and NMI*, are used to request service from the permanent bus master.

The INT* line may be masked off by the bus master, usually via an internal software operation. If the master accepts the interrupt request on the INT* line, it may respond with an interrupt acknowledge bus cycle, accepting vectoring information from the data bus. The INT* line is often implemented as a "group interrupt" line in conjunction with the vectored interrupt bus. In this case, INT* indicates the presence of one or more vectored interrupt requests.

The NMI* line is a non-maskable interrupt request line, that is, it may not be masked off by the bus master. Accepting an interrupt on the NMI* line need not generate an interrupt acknowledge bus cycle.

An interrupt request on the INT* line is asserted as a level, that is, the line is asserted until interrupt service is received. An interrupt request on the NMI* line, on the other hand, is asserted as a negative going edge, since no interrupt acknowledge cycle need be generated.

Both these lines are specified as open collector lines.

2.2.6.3 Hold request

The hold request line, HOLD*, is used by temporary bus masters to request control of the bus from the permanent bus master. The HOLD* line may be masked by the permanent bus master to prevent temporary masters from gaining bus control.

The HOLD* line is specified as an open collector line, and may only be asserted at certain times. See 2.8.3.

2.2.6.4 Sixteen acknowledge

The sixteen acknowledge line, SIXTN*, is a response to the status signal sixteen request (sXTRQ*), and indicates that the requested 16-bit data transfer is possible.

The SIXTN* line is specified as an open collector line. Detailed specification of the use of this line is given in 2.6.

2.2.7 DMA control bus

The eight lines of the DMA control bus are used in conjunction with control bus signals HOLD* and pHLDA. They arbitrate among simultaneous requests for control of the bus by temporary masters and disable the signal drivers of the permanent bus master, thus effecting an orderly transfer of bus control.

All eight lines of the DMA control bus are specified as open collector lines.

The eight DMA control lines are:

1) DMA0*
2) DMA1*
3) DMA2*
4) DMA3*
5) ADSB*
6) DODSB*
7) SDSB*
8) CDSB*

Detailed specification of the use of these lines is given in 2.8.

2.2.7.1 DMA arbitration

The four lines that arbitrate among simultaneous requests for bus control by temporary masters are DMA0* through DMA3*. The encoded priority of requesters is asserted on these lines and, after settling, they contain the priority number of the highest priority requester.

Detailed specification of this process is given in 2.8.3

2.2.7.2 Bus transfer signals

Four signals are available on the bus to disable the line drivers of the permanent bus master. They are:

1) ADSB*, address disable.
2) DODSB*, data out disable.
3) SDSB*, status disable.
4) CDSB*, control output disable.

Use of these lines is tightly specified during the transfer of the bus from a permanent master to a temporary master, as given in 2.8.2, and any transfer involving the control output lines should follow a similar protocol.

The address, data, and status signals from the permanent master may be disabled and replaced using these signals as long as the contents of these buses is valid for the current bus cycle as though no replacement had occurred.

2.2.8 Vectored interrupt bus

The eight lines of the vectored interrupt bus are used in conjunction with the generalized vectored in-

terrupt request, INT*, to arbitrate among eight levels of interrupt request priorities. They are typically implemented as inputs to a bus slave which masks and prioritizes the requests, asserts the generalized interrupt request to the permanent bus master, and responds to the interrupt acknowledge bus cycle with appropriate vectoring data.

The eight lines of the vectored interrupt bus are VI0* through VI7*, where VI0* is considered the highest priority interrupt.

The vectored interrupt lines should be implemented as levels, that is, they should be held active until service is received.

2.2.9 System utilities

2.2.9.1 System power

Power in S-100 systems is distributed to bus devices as unregulated voltages. A total of nine bus lines are used:

1) +8 volts, 2 lines.
2) +16 volts, 1 line.
3) −16 volts, 1 line.
4) GROUND, 5 lines.

Ground lines are distributed across the edge connector such that low impedance grounds are available on both sides of the edge connector, and on both sides of the circuit cards.

Power lines are subject to the specifications given in 3.2.

2.2.9.2 System clock

The system clock, Φ, is generated by the permanent master. The control timing for all bus cycles, whether they are cycles of the permanent master or cycles of temporary masters in control of the bus, must be derived from this clock.

This signal is never transferred during a bus exchange operation.

2.2.9.3 CLOCK

This clock is specified as a 2-MHz (0.5 percent tolerance) signal with no relationship to any other bus signal. It is to be used by counters, timers, baud-rate generators, etc.

2.2.9.4 System reset functions

System reset functions are divided into three lines:

1) RESET*, resets all bus masters.
2) SLAVE CLR*, resets all bus slaves.
3) POC*, power-on clear is active only on power-on, and asserts SLAVE CLR* and RESET*.

The POC* signal is specified as having a minimum active period of 10 msec.

RESET* and SLAVE CLR* are specified as open collector lines.

2.2.9.5 Memory write strobe

The memory write strobe, MWRT, must be generated somewhere in the system. It is usually generated by front panel type devices, but is optionally generated by permanent masters or mother boards in systems without front panels. Care must be taken that it is generated at only one point in a given system.

Memory write is defined as:

$$MWRT = pWR \cdot \overline{sOUT} \quad \text{(logic equation)}$$

2.2.9.6 Phantom slaves

A line, PHANTOM*, is provided for overlaying bus slaves at a common address location. When this line is activated phantom bus slaves are enabled and normal bus slaves are disabled.

This line is specified as an open collector line.

2.2.9.7 Error

The line ERROR* is a generalized error line that is asserted when an error of some sort (i.e., parity, write to protected memory) is occurring in the current bus cycle.

This line is specified as an open collector line.

2.2.9.8 Manufacturer specified lines

Three lines which can be specified by individual manufacturers are provided on the bus. These lines, termed NDEF (not to be defined), should only be implemented as options, and shall be provided with jumpers so that possible conflicts may be eliminated.

Any manufacturer MUST specify in detail any use of these lines. Signals on these lines are limited to 5 volt logic levels.

2.2.9.9 Power fail (PWRFAIL)*

The power fail line indicates impending power failure, and remains true until power is restored and POC* is true.

2.2.9.10 Reserved lines (RFU)

The five remaining lines are reserved for future use and may not be used for any purpose.

2.2.10 Pin list

Pin connections to the card edge connector shall conform to the list given in Table 3.

2.3 The permanent master interface

2.3.1 General

The permanent master interface provides the capability to transfer device dependent messages to and from all bus slaves. It is responsible for the genera-

Table 3. S-100 bus pin list.

PIN NO.	SIGNAL & TYPE	ACTIVE LEVEL		DESCRIPTION
1	+8 VDLTS (B)			Instantaneous minimum greater than 7 volts, instantaneous maximum less than 25 volts, average maximum less than 11 volts.
2	+16 VOLTS (B)			Instantaneous minimum greater than 14.5 volts, instantaneous maximum less than 35 volts, average maximum less than 21.5 volts.
3	XRDY (S)	H		One of two ready inputs to the current bus master. The bus is ready when both these ready inputs are true. See pin 72.
4	VI0*(S)	L	D.C.	Vectored interrupt line 0.
5	VI1*(S)	L	O.C.	Vectored interrupt line 1.
6	VI2*(S)	L	O.C.	Vectored interrupt line 2.
7	VI3*(S)	L	D.C.	Vectored interrupt line 3.
8	VI4*(S)	L	D.C.	Vectored interrupt line 4.
9	VI5*(S)	L	D.C.	Vectored interrupt line 5.
10	VI6*(S)	L	O.C.	Vectored interrupt line 6.
11	VI7*(S)	L	O.C.	Vectored interrupt line 7.
12	NMI*(S)	L	O.C.	Non-maskable interrupt.
13	PWRFAIL*(B)	L		Power fail bus signal. (See Section 2.10.1 regarding pseudo open-collector nature)
14	DMA3* (M)	L	O.C.	Temporary master priority bit 3.
15	A18 (M)	H		Extended address bit 18.
16	A16 (M)	H		Extended address bit 16.
17	A17 (M)	H		Extended address bit 17.
18	SDSB* (M)	L	O.C.	The control signal to disable the 8 status signals.
19	CDSB* (M)	L	O.C.	The control signal to disable the 5 control output signals.
20	GND (B)			Common with pin 100.
21	NDEF			Not to be defined. Manufacturer must specify any use in detail.
22	ADSB* (M)	L.	O.C.	The control signal to disable the 16 address signals.
23	DODSB* (M)	L	O.C.	The control signal to disable the 8 data output signals.
24	Φ (B)	H		The master timing signal for the bus.
25	pSTVAL*(M)	L		Status valid strobe.
26	pHLDA (M)	H		A control signal used in conjunction with HOLD* to coordinate bus master transfer operations.
27	RFU			Reserved for future use.
28	RFU			Reserved for future use.
29	A5 (M)	H		Address bit 5.
30	A4 (M)	H		Address bit 4.
31	A3 (M)	H		Address bit 3.
32	A15 (M)	H		Address bit 15 (most significant for non-extended addressing.)
33	A12 (M)	H		Address bit 12.
34	A9 (M)	H		Address bit 9.
35	DO1 (M)/DATA1 (M/S)	H		Data out bit 1, bidirectional data bit 1.
36	DO0 (M)/DATA0 (M/S)	H		Data out bit 0, bidirectional data bit 0.
37	A10 (M)	H		Address bit 10.
38	DO4 (M)/DATA4 (M/S)	H		Data out bit 4, bidirectional data bit 4.
39	DO5 (M)/DATA5 (M/S)	H		Data out bit 5, bidirectional data bit 5.
40	DO6 (M)/DATA6 (M/S)	H		Data out bit 6, bidirectional data bit 6.
41	DI2 (S)/DATA10 (M/S)	H		Data in bit 2, bidirectional data bit 10.
42	DI3 (S)/DATA11 (M/S)	H		Data in bit 3, bidirectional data bit 11.
43	DI7 (S)/DATA15 (M/S)	H		Data in bit 7, bidirectional data bit 15.
44	sM1 (M)	H		The status signal which indicates that the current cycle is an op-code fetch.
45	sOUT (M)	H		The status signal identifying the data transfer bus cycle to an output device.
46	sINP (M)	H		The status signal identifying the data transfer bus cycle from an input device.
47	sMEMR (M)	H		The status signal identifying bus cycles which transfer data from memory to a bus master, which are not interrupt acknowledge instruction fetch cycle(s).
48	sHLTA (M)	H		The status signal which acknowledges that a HLT instruction has been executed.
49	CLOCK(B)			2 MHz (0.5%) 40-60% duty cycle. Not required to be synchronous with any other bus signal.
50	GND (B)			Common with pin 100.
51	+8 VOLTS (B)			Common with pin 1.
52	−16 VDLTS (B)			Instantaneous maximum less than −14.5 volts, instantaneous minimum greater than −35 volts, average minimum greater than −21.5 volts.
53	GND (B)			Common with pin 100.
54	SLAVE CLR* (B)	L	O.C.	A reset signal to reset bus slaves. Must be active with POC* and may also be generated by external means.
55	DMA0* (M)	L	O.C.	Temporary master priority bit 0.

PIN NO.	SIGNAL & TYPE	ACTIVE LEVEL		DESCRIPTION
56	DMA1* (M)	L	O.C.	Temporary master priority bit 1.
57	DMA2* (M)	L	O.C.	Temporary master priority bit 2.
58	sXTRQ* (M)	L		The status signal which requests 16-bit slaves to assert SIXTN*.
59	A19 (M)	H		Extended address bit 19.
60	SIXTN* (S)	L	O.C.	The signal generated by 16-bit slaves in response to the 16-bit request signal sXTRQ*.
61	A20 (M)	H		Extended address bit 20.
62	A21 (M)	H		Extended address bit 21.
63	A22 (M)	H		Extended address bit 22.
64	A23 (M)	H		Extended address bit 23.
65	NDEF			Not to be defined signal.
66	NOEF			Not to be defined signal.
67	PHANTOM*(M/S)	L	O.C.	A bus signal which disables normal slave devices and enables phantom slaves—primarily used for bootstrapping systems without hardware front panels.
68	MWRT (B)	H		pWR• — sOUT (logic equation). This signal must follow pWR* by not more than 30 ns. (See note, Section 2.7.5.3)
69	RFU			Reserved for future use.
70	GND (B)			Common with pin 100.
71	RFU			Reserved for future use.
72	RDY (S)	H	O.C.	See comments for pin 3.
73	INT* (S)	L	O.C.	The primary interrupt request bus signal.
74	HOLD* (M)	L	O.C.	The control signal used in conjunction with pHLDA to coordinate bus master transfer operations.
75	RESET*(B)	L	O.C.	The reset signal to reset bus master devices. This signal must be active with POC* and may also be generated by external means.
76	pSYNC (M)	H		The control signal identifying BS_1.
77	pWR* (M)	L		The control signal signifying the presence of valid data on DO bus or data bus.
78	pOBIN (M)	H		The control signal that requests data on the DI bus or data bus from the currently addressed slave.
79	A0 (M)	H		Address bit 0 (least significant).
80	A1 (M)	H		Address bit 1.
81	A2 (M)	H		Address bit 2.
82	A6 (M)	H		Address bit 6.
83	A7 (M)	H		Address bit 7.
84	A8 (M)	H		Address bit 8.
85	A13 (M)	H		Address bit 13.
86	A14 (M)	H		Address bit 14.
87	A11 (M)	H		Address bit 11.
88	DO2 (M)/DATA2 (M/S)	H		Data out bit 2, bidirectional data bit 2.
89	DO3 (M)/DATA3 (M/S)	H		Data out bit 3, bidirectional data bit 3.
90	DO7 (M)/DATA7 (M/S)	H		Data out bit 7, bidirectional data bit 7.
91	DI4 (S)/DATA12 (M/S)	H		Data in bit 4 and bidirectional data bit 12.
92	DI5 (S)/DATA13 (M/S)	H		Data in bit 5 and bidirectional data bit 13.
93	DI6 (S)/DATA14 (M/S)	H		Data in bit 6 and bidirectional data bit 14.
94	DI1 (S)/DATA9 (M/S)	H		Data in bit 1 and bidirectional data bit 9.
95	DI0 (S)/DATA8 (M/S)	H		Data in bit 0 (least significant for 8-bit data) and bidirectional data bit 8.
96	sINTA (M)	H		The status signal identifying the bus input cycle(s) that may follow an accepted interrupt request presented on INT*.
97	sWO* (M)	L		The status signal identifying a bus cycle which transfers data from a bus master to a slave.
98	ERROR* (S)	L	O.C.	The bus status signal signifying an error condition during present bus cycle.
99	POC* (B)	L		The power-on clear signal for all bus devices; when this signal goes low, it must stay low for at least 10 msecs.
100	GND (B)			System ground.

tion and timing of all bus cycles while it has control of the bus, and is capable of generating all possible bus cycles.

The permanent master normally has control of the bus. It may relinquish bus control to a temporary bus master via a hold operation for an arbitrary number of cycles. Upon completion of the hold operation control of the bus is always returned to the permanent master.

2.3.2 Permanent master state diagram

The permanent master interface shall be implemented so as to conform to the state diagram given in Figure 1.

2.3.3 Permanent master state descriptions

2.3.3.1 Bus state 1

The initial bus state, BS_1, is the state in which the status and address buses are in transition to their values for the new bus cycle. pSYNC goes true in the middle of the BS_1 state, indicating the beginning of a new bus cycle.

2.3.3.2 Bus state 2

Bus state 2, BS_2, is the state during which the address and status lines become stable. When they are guaranteed stable the pSTVAL*, status valid strobe, is activated.

The ready lines and the sixteen acknowledge line are sampled during the BS_2 state.

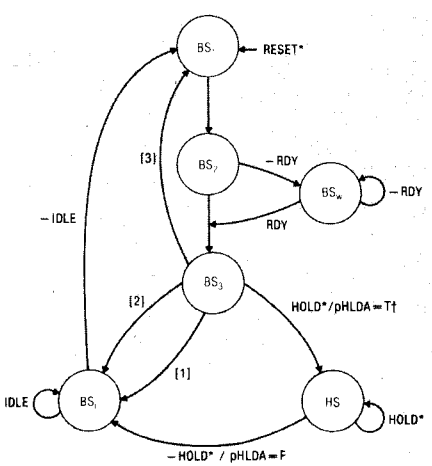
2.3.3.3 Wait state

The wait state, BS_w, is entered if the ready line sampled in BS_2 indicates that the addressed bus slave is not ready for data transfer. The ready line is sampled once every clock cycle until a ready condition is indicated. When the ready condition is indicated the BS_2 state is completed and the BS_3 state entered.

The BS_w state is thus used to synchronize bus cycles generated by bus masters with the response speed of assorted bus slaves.

2.3.3.4 Bus state 3

Bus state 3, BS_3, is the bus state during which the data transfer actually takes place between the master and the addressed slave.



† There is a minimum specified time delay between hold and hold acknowledge.
[1] instruction execution complete • INT enable • INT request / interrupt accept
[2] instruction execution not complete + INT disabled + no interrupt request
[3] (instruction execution complete • –HOLD* • –interrupt accept) + (instruction execution not complete • –IDLE)

Figure 1. Permanent master state diagram.

2.3.3.5 Idle bus states

After completion of the BS_3 state, the master may enter one or more idle bus states.

While in an idle bus state the generalized data strobes, pWR* and pDBIN, must not be active, and pSTVAL* must not be asserted in conjunction with pSYNC active.

2.3.3.6 Hold accept

Permanent masters must be configured to conditionally accept hold operations from temporary masters. This function may be gated off under hardware or software control, to allow indivisible test and set operations. If hold is enabled and active, the permanent master will enter the hold state HS following a BS_3 state, and pHLDA will be asserted.

The permanent master remains in the hold state until the hold request HOLD* becomes false.

Hold operations always take priority over interrupt operations.

2.3.3.7 Interrupt accept

If hold request is not active, if execution of the current instruction is complete, and if interrupts are enabled and an interrupt is being requested, then the permanent master accepts the interrupt request at the end of the BS_3 state. In the case of a vectored interrupt, the next bus cycle may be an interrupt acknowledge bus cycle. In the case of a non-maskable interrupt, the response is usually a transfer to a predetermined location.

2.3.4 Required signals for permanent masters

2.3.4.1 Output signals

The following signals are output signals from permanent masters to bus slaves:

1) A0-A23†.
2) All status signals.
3) All control output signals.
4) Data output signals (8 or 16 depending on processor type).
5) Φ, the system clock.

2.3.4.2 Input signals

The following signals are required input signals to permanent masters:

1) The control input signals, except NMI* and SIXTN*.
2) Data input signals (8 or 16 depending on processor type).
3) The four disable signals ADSB*, DODSB*, SDSB*, CDSB*.
4) RESET*.

† A16 through A23 are optional on permanent masters.

2.3.5 Dummy mastering

In cases where a number of processors co-exist in a single system as temporary masters, it may prove inefficient from a systems point of view to implement a permanent master.

In such a case it is permissible that the permanent master be implemented as a dummy, that is, as a device that conducts no bus cycles, but only supplies an arbitration interval so that the DMA control bus may settle.

The dummy master takes control of the bus between temporary masters, asserting the control output bus in the null state, and passes the bus to the next requester after an arbitration interval of one clock cycle.

Required output signals for dummy masters are the control output signals, and the system clock Φ. Input signals are HOLD* and CDSB*.

2.4 The temporary master interface

2.4.1 General

The temporary master interface provides the capability to transfer device dependent messages to and from a selected set of bus slaves. The temporary master thus differs from the permanent master in that it need not generate all possible bus cycles.

The temporary master requests control of the bus from the permanent master. If the bus is granted, the temporary master is responsible for the generation and timing of all bus cycles until it returns control to the permanent master.

Since up to 16 temporary masters may co-exist in a single system, a protocol has been developed to arbitrate among simultaneous bus requests. Detailed specification of this protocol is given in 2.8.3.

2.4.2 Temporary master state diagram

The temporary master interface shall be implemented so as to conform to the state diagram given in Figure 2.

2.4.3 Temporary master state descriptions

2.4.3.1 Arbitration (ARB)

If more than one temporary master is present in the system, bus requesters must arbitrate for the bus as given in 2.8.3.

During the arbitration sequence, bus requesters try to assert their priorities on the arbitration bus, and the contents of the arbitration bus are compared with each requester's priority.

If the contents of the arbitration bus is of higher priority than the locally attempted priority assertion, then a higher priority requester is present in the system, and the low priority requester removes its low order bits from the arbitration bus. Thus, after some settling time, the priority of the highest priority requester is present on the arbitration bus. This requester is granted the bus on the rising edge of hold acknowledge.

2.4.3.2 Bus transfer states (XS I and XS II)

Since the bus has positive polarity control signals, extreme care must be taken in bus transfer operations to avoid erroneous pulses on the control lines.

In general terms, this is accomplished by specifying that both the permanent master and the temporary master drive the control lines in specified logic states during the bus transfer.

Detailed specification of this operation is given in 2.8.2.

Proposed S-100 bus layout—Quick reference

pin 1	+8 Volts (B)		pin 51	+8 Volts (B)	
pin 2	+16 Volts (B)		pin 52	−16 Volts (B)	
pin 3	XRDY (S)	H	pin 53	GND	
pin 4	VI0* (S)	L	pin 54	SLAVE CLR* (B)	L
pin 5	VI1* (S)	L	pin 55	DMA0* (M)	L
pin 6	VI2* (S)	L	pin 56	DMA1* (M)	L
pin 7	VI3* (S)	L	pin 57	DMA2* (M)	L
pin 8	VI4* (S)	L	pin 58	sXTRQ* (M)	L
pin 9	VI5* (S)	L	pin 59	A19	H
pin 10	VI6* (S)	L	pin 60	SIXTN* (S)	L
pin 11	VI7* (S)	L	pin 61	A20 (M)	H
pin 12	NMI* (S)	L	pin 62	A21 (M)	H
pin 13	PWRFAIL* (B)	L	pin 63	A22 (M)	H
pin 14	DMA3* (M)	L	pin 64	A23 (M)	H
pin 15	A18 (M)	H	pin 65	NDEF	
pin 16	A16 (M)	H	pin 66	NDEF	
pin 17	A17 (M)	H	pin 67	PHANTOM* (M/S)	L
pin 18	SDSB* (M)	L	pin 68	MWRT (B)	H
pin 19	CDSB* (M)	L	pin 69	RFU	
pin 20	GND		pin 70	GND	
pin 21	RFU		pin 71	NDEF	
pin 22	ADSB* (M)	L	pin 72	RDY (S)	H
pin 23	DODSB* (M)	L	pin 73	INT* (S)	L
pin 24	Φ (B)	H	pin 74	HOLD* (M)	L
pin 25	pSTVAL* (M)	L	pin 75	RESET* (B)	L
pin 26	pHLDA (M)	H	pin 76	pSYNC (M)	H
pin 27	RFU		pin 77	pWR* (M)	L
pin 28	RFU		pin 78	pDBIN (M)	H
pin 29	A5 (M)	H	pin 79	A0 (M)	H
pin 30	A4 (M)	H	pin 80	A1 (M)	H
pin 31	A3 (M)	H	pin 81	A2 (M)	H
pin 32	A15 (M)	H	pin 82	A6 (M)	H
pin 33	A12 (M)	H	pin 83	A7 (M)	H
pin 34	A9 (M)	H	pin 84	A8 (M)	H
pin 35	DO1 (M)/DATA1 (M/S)	H	pin 85	A13 (M)	H
pin 36	DO0 (M)/DATA0 (M/S)	H	pin 86	A14 (M)	H
pin 37	A10 (M)	H	pin 87	A11 (M)	H
pin 38	DO4 (M)/DATA4 (M/S)	H	pin 88	DO2 (M)/DATA2 (M/S)	H
pin 39	DO5 (M)/DATA5 (M/S)	H	pin 89	DO3 (M)/DATA3 (M/S)	H
pin 40	DO6 (M)/DATA6 (M/S)	H	pin 90	DO7 (M)/DATA7 (M/S)	H
pin 41	DI2 (S)/DATA10 (M/S)	H	pin 91	DI4 (S)/DATA12 (M/S)	H
pin 42	DI3 (S)/DATA11 (M/S)	H	pin 92	DI5 (S)/DATA13 (M/S)	H
pin 43	DI7 (S)/DATA15 (M/S)	H	pin 93	DI6 (S)/DATA14 (M/S)	H
pin 44	sM1 (M)	H	pin 94	DI1 (S)/DATA9 (M/S)	H
pin 45	sOUT (M)	H	pin 95	DI0 (S)/DATA8 (M/S)	H
pin 46	sINP (M)	H	pin 96	sINTA (M)	H
pin 47	sMEMR (M)	H	pin 97	sWO* (M)	L
pin 48	sHLTA (M)	H	pin 98	ERROR* (S)	L
pin 49	CLOCK (B)		pin 99	POC* (B)	L
pin 50	GND		pin 100	GND	

2.4.3.3 Bus cycle

The definition of bus cycle states is the same as that for the permanent master interface, given in 2.3.3.1 through 2.3.3.5.

An arbitrary number of bus cycles may be performed by the temporary master before returning control to the permanent master.

2.4.4 Required signals for temporary masters

2.4.4.1 Output signals

The following are required output signals for a temporary master interface:

1) Address lines A0-A23†.
2) All status signals.
3) All control output signals ††.
4) Data output lines.
5) DMA arbitration lines DMA0*-DMA3*.
6) Hold request, HOLD*.

2.4.4.2 Input signals

The following are required input signals for a temporary master interface:

† *Note:* Temporary masters must generate A16-A23; they need only generate falses or lows on these 8 lines, however.

†† *Note:* Temporary masters should provide a jumper on the pSTVAL* line, as 8080 CPUs of old design do not transfer this line with the control output lines. In this case all bus masters use the same pSTVAL* signal.

1) The ready lines, RDY and XRDY.
2) Hold acknowledge, pHLDA.
3) Data input lines.
4) The system clock, Φ.

2.5 The slave interface

A slave device responds to a bus cycle initiated by a bus master. Memory and input/output devices are examples of bus slaves.

A slave device may request service by a bus master by generating an interrupt request.

2.5.1 Slave interface state diagram

The slave interface shall conform, in general, to the state diagram given in Figure 3. Slave interfaces need not have both read and write capability.

2.5.2 Slave state definitions

2.5.2.1 Slave idle state

The slave idle state, S_i, is a passive state with respect to the bus.

The slave monitors the stream of bus cycles to determine if it is selected for the current bus cycle.

The slave may be performing internal operations while in the idle state.

The assertion of SLAVE CLR* forces all slaves into the idle state.

2.5.2.2 Slave setup

A slave moves from the slave idle state to the setup state, S_s, when it has been addressed by the current bus cycle. This is an operation internal to the slave which sets up a data transfer with a bus master. If a

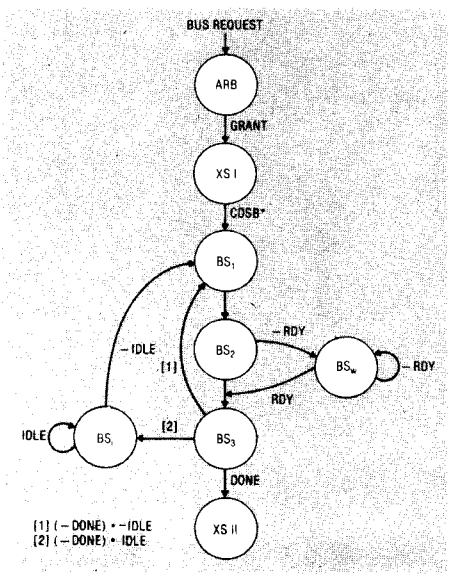


Figure 2. Temporary master state diagram.

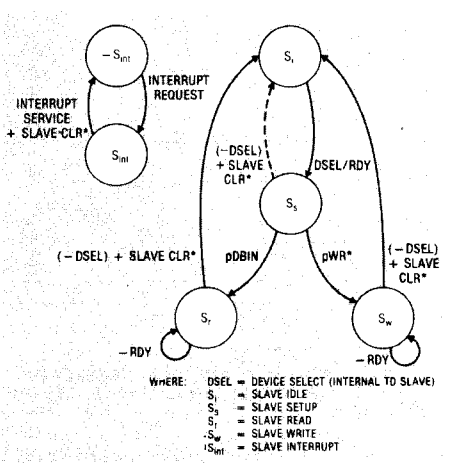


Figure 3. Slave interface state diagram.

slave can tolerate spurious transitions from the idle state to the setup state, then the device select signal may be decoded statically from the address and status buses. If a device cannot tolerate spurious transitions, the device select line should be decoded in conjunction with the status valid strobe, pSTVAL*.

If synchronization is required by the slave before the data transfer may take place, the ready line is asserted false during this state until the device is ready for data transfer.

2.5.2.3 Slave read

Data from the addressed slave is gated onto the data bus during the slave read state, S_r. The generalized read strobe governs the transition to this state.

When device select becomes false the slave returns to the idle state.

2.5.2.4 Slave write

Data from the current bus master is written into the slave during the active period of the generalized write strobe, pWR*.

When device select becomes false the slave returns to the idle state.

2.5.2.5 Interrupt request state

If a slave requires service by a bus master, an interrupt request may be generated by the slave. The interrupt should be held active until the slave is serviced, or until SLAVE CLR* is asserted.

2.5.3 Required signals for slave interfaces

Slave interfaces need only receive and generate that subset of bus signals necessary for communication with masters.

2.6 8/16-bit data transfer protocol

2.6.1 General

Implementation of the 8/16-bit data transfer protocol allows both 8-bit and 16-bit parallel data transfers over the bus, and hence allows both 8-bit masters and 16-bit masters and slaves to co-exist in a single system. For 16-bit transfers the two unidirectional 8-bit data buses are ganged to form a single 16-bit bidirectional data bus.

Two lines are assigned to control the ganging of the data bus:

1) sXTRQ*, status output from the master, which indicates a request for a 16-bit data transfer.
2) SIXTN*, an acknowledge input to the master, which indicates that a 16-bit data transfer is possible.

Use of the sixteen acknowledge line SIXTN* permits the use of current design 8-bit memory boards without modification. When SIXTN* is false, a 16-bit transfer may be accomplished by two sequential single-byte transfers.

2.6.2 8-bit data paths

The current bus master requests an 8-bit transfer by not asserting sXTRQ*.

Byte data output from the master to the addressed slave is asserted on the data output bus, DO0 through DO7.

Byte data input from the addressed slave to the current bus master is asserted on the data input bus, DI0 through DI7.

2.6.3 16-bit data paths

The current bus master requests a 16-bit transfer by asserting sXTRQ*.

If the addressed slave is capable of a 16-bit parallel data transfer, it asserts SIXTN*, as shown in the timing diagram (see page 52).

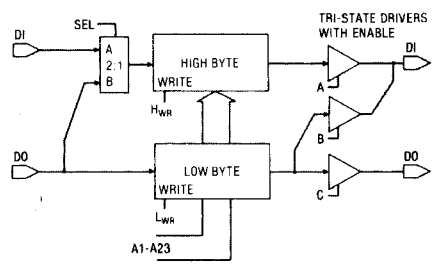
Sixteen-bit data transfer is then conducted via the ganged data buses, where DO0 = DATA0, and DI7 = DATA15.

2.6.4 Memory organization

Memory devices capable of both 8-bit and 16-bit parallel data transfers are organized, as shown in Figure 4, as two banks of 8-bit memory, a high-byte bank and a low-byte bank. These data banks may be activated either together or separately, depending on the condition of the sixteen request status line, sXTRQ*.

2.6.4.1 Byte references

When sXTRQ* is not asserted, memory references are single-byte transfers.



SEL selects A for word references, B for byte references.

Output enables:

$A = 16_{rd} + (8_{rd} \cdot \overline{\ }A0)$
$B = 8_{rd} \cdot A0$
$C = 16_{rd}$
$H_{wr} = 16_{wr} + (8_{wr} \cdot \overline{\ }A0)$
$L_{wr} = 16_{wr} + (8_{wr} \cdot A0)$

Where:

16_{rd} = Device select • sXTRQ* • pDBIN
8_{rd} = Device select • (−sXTRQ*) • pDBIN
16_{wr} = Device select • sXTRQ* • pWR*
8_{wr} = Device select • (−sXTRQ*) • pWR*

Figure 4. 8/16-bit memory organization.

The proper location in memory is selected by the address output on address lines A1 through A15 (A23 for extended addressing systems), while the A0 line selects the high byte or the low byte. A0 equals 0 selects the high byte of the 16-bit word, while A0 equals 1 selects the low byte of the word.

See Figure 5 for address usage.

In the 8-bit mode, data output from the master, on the DO bus, is connected to the data input lines of both memory banks; the low-byte data input lines are connected directly to the DO bus, and the high-byte data input lines are connected to the DO bus via a two-to-one multiplexer controlled by sXTRQ*.

Data output from the memory banks is routed to Tri-State† bus drivers A and B in Figure 4. One of these drivers is enabled when the read strobe is activated, depending on the condition of A0. The selected byte is thus available to the master on the DI bus.

2.6.4.2 Word references

When sXTRQ* is asserted by the master, and SIXTN* is asserted by the slave, memory references are double-byte transfers.

Address lines A1 through A15 (A23 in extended address systems) select the proper word from memory. The condition of the A0 bit does not enter into the decoding or addressing for word references.

See Figure 5 for address usage.

In the 16-bit mode, data output from the bus master is asserted on the 16 signal lines of the DO bus and the DI bus. The multiplexer on the data input lines now routes the high-byte data, on the DI bus, to the data input lines of the high-byte bank. Low-byte data, on the DO bus, is connected to the data input lines of the low-byte bank.

Data output from the memory banks is routed through buffers A and C to their respective data paths. Both A and C will be enabled by the read strobe.

†Tri-State is a trademark of National Semiconductor.

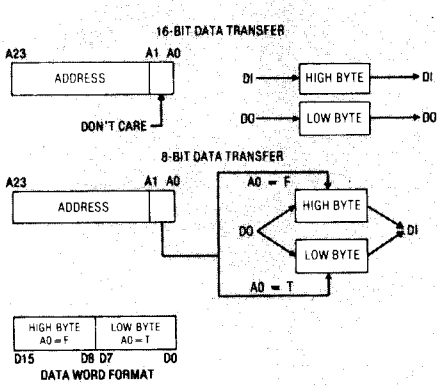


Figure 5. 8/16-bit address and data usage.

2.6.5 Sixteen acknowledge (SIXTN)*

Implementation of the sixteen acknowledge line allows the use of 8-bit memory boards in a 16-bit system without modification, but with a reduction in maximum system bandwidth.

If a 16-bit master requests a 16-bit transfer, but the addressed slave is not capable of such a transfer, the sixteen acknowledge lines will not be asserted.

The master will respond in one of two ways, by generating an error trap or by conducting the transfer in byte-serial fashion.

2.6.5.1 Byte-serial response

If the sixteen acknowledge line is not activated after a specified period, circuitry may be included on bus masters to conduct the requested 16-bit transfer as two consecutive byte operations, thus assembling the requested 16-bit word while holding the master in a wait state.

For this process to occur, the sixteen acknowledge line must meet the timing specifications for the ready line inputs.

2.6.5.2 Error response

If circuitry does not exist on the master to conduct the requested 16-bit transfer as two consecutive byte operations, an error condition shall result immediately, with ERROR* asserted.

2.7 Fundamental bus cycle timing

2.7.1 General

This section deals with the fundamental timing concepts involved in the standard bus cycle. Detailed specification of the timing parameters discussed in this section is given in 3.8 and 3.9.

The standard bus cycle is a pseudo-synchronous cycle, that is, the timing of the control signals bears a specified relationship to the master system clock Φ.

All data transfers, including read or write cycles, 8- or 16-bit transfers, memory or input/output device transfers, and interrupt acknowledge are conducted on the bus as a standard bus cycle.†

Figure 6 shows the fundamental timing for a standard bus cycle, with a single wait state inserted by the addressed slave.

2.7.2 Address and status buses

The beginning of a new bus cycle is indicated by the rising edge of the pSYNC signal, which closely follows the rising edge of the system clock, Φ.

The address and status buses are changing to their values for the new cycle during the beginning of the

† See possible exception, section 2.7.5.3.

pSYNC interval. Shortly after they can be guaranteed stable on the bus, the status valid strobe, pSTVAL*, is asserted. pSTVAL*, decoded in conjunction with pSYNC, indicates to all bus slaves that stable address and status may be sampled from the bus.

The position of the status valid strobe within the pSYNC interval is independent of the system clock, Φ. This affords the designer of bus masters considerable flexibility in interfacing different processors to the bus. The status valid strobe should be positioned within the pSYNC interval such that the delay between guaranteed status on the bus and the activation of the status valid strobe is as close to the minimum specification as possible, thus maximizing memory and device access time.†

In order to prevent false cycle starts in bus slaves, only one negative edge of the status valid strobe may occur while pSYNC is asserted.

Address and status information is thus stable on the bus from the negative transition of the status valid strobe during pSYNC, and is held stable until a specified period after the trailing edge of the data strobe (pDBIN in the read case, and pWR* in the write case). This hold time ensures that false decoding of the address and status information will not occur at the end of the bus cycle.

2.7.3 Ready and sixteen acknowledge lines

The sixteen acknowledge line, since it may be used to place the bus master in a wait state while a requested 16-bit transfer is conducted in byte-serial fashion, is subject to the same timing constraints as the ready lines.

The ready lines are first sampled by the bus master on the rising edge of the system clock during the BS 2 state, and if active, the master enters a wait state, sampling the ready line once every clock cycle on the rising edge of the system clock until the slave is ready for data transfer.

A minimum setup time before the rising edge of the system clock, and a minimum hold time after sampling must be met for the proper operation of the ready lines.

The time between the active edge of the status valid strobe and the sampling of the ready line may be very short. Hence, it is recommended practice not to make assertion of the ready line dependent on pSTVAL*.

Data output, address, and status are held stable during wait states.

2.7.4 Read cycles

2.7.4.1 General

There are four types of read cycles: op-code fetch (M1), memory read, input, and interrupt acknowledge. These cycles are all similar with respect to timing, but make different use of the status bits and the address bus. See Tables 1 and 2.

2.7.4.2 The read strobe

The generalized read strobe pDBIN is used to gate data from an addressed slave onto the data bus during a read operation. The read strobe is asserted true by the bus master after a minimum specified time from the assertion of the status valid strobe.

It is held true during any inserted wait states, and returns to the false state, returning the data bus to the high impedance state, shortly before the address and status buses are allowed to change.

2.7.5 Write cycles

2.7.5.1 General

There are two possible types of write cycles on the bus, a memory write cycle and an output cycle.

These two cycles are similar with respect to timing, but make different use of the status bits and address bus. A special write strobe, MWRT, is generated for memory cycles.

2.7.5.2 The write strobe

The generalized write strobe, pWR*, is used to write data from the data bus into the addressed bus slave. The write strobe may be asserted by the master after the completion of the pSYNC interval.

Data out on the data bus must be guaranteed valid for a specified period both before and after the activation of the write strobe. Hence, either the leading or the trailing edge of the write strobe may be used to strobe data into the addressed slave.

Address and status information must be held valid for a specified period of time from the trailing edge of the write strobe.

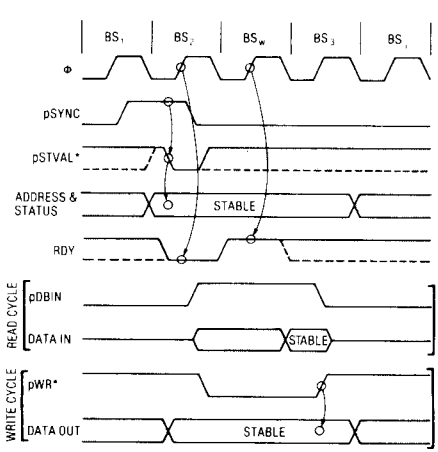


Figure 6. Bus cycle fundamental timing relationships.

†Note: The Φ1 signal output from current 8080 processor boards meets all the specifications as a status valid strobe. Hence, these boards meet the bus cycle specification without modification.

2.7.5.3 Memory write strobe

While the generalized write strobe is activated for all write cycles, the memory write strobe is activated for memory write cycles only. The memory write strobe is usually generated by front-panel devices, if they exist in the system, as a function of bus memory write or a front-panel deposit. If front-panel devices do not exist the memory write strobe must be generated somewhere in the system, but at only one point. This circuit should be designed such that it generates the memory write strobe for all bus masters. Jumpers shall be provided to allow extra circuits to be disabled.

The memory write strobe, MWRT, is defined as:

$$MWRT = pWR \cdot \overline{sOUT}, \text{ (logic equation)}$$

that is, memory write is true when pWR is true and sOUT is false.

The memory write strobe must follow the pWR* strobe by not more than a specified period.†

2.8 Special bus operations

2.8.1 General

This section describes two special bus operations related to DMA operations, that is, the transfer of bus control from the permanent bus master to a temporary bus master for an arbitrary number of bus cycles, and the return of control to the permanent bus master.

These two operations are:

1) The bus transfer protocol.
2) The arbitration protocol among simultaneous bus requesters.

2.8.2 Bus transfer protocol

2.8.2.1 General

When a temporary bus master has been granted the bus by the permanent bus master, control must be transferred to the temporary master in such a way that spurious signals are not generated on the control output lines, causing false bus cycles. Since some of the control output signals are of positive polarity, extreme care must be taken in this operation. In general, the specified bus transfer protocol accomplishes this by having the permanent master and the temporary master drive the control output lines simultaneously at specified levels during the bus transfer.

2.8.2.2 Bus transfer state diagram

The bus transfer operation shall be implemented so as to conform to the bus transfer state diagram given in Figure 7.

2.8.2.3 Bus transfer state definitions

2.8.2.3.1 Idle

The idle state signifies that the temporary master is either involved in internal operations, and does not require the bus, or that it is waiting for the bus to become free so that it may assert its bus request.

2.8.2.3.2 Arbitration

If a temporary master desires the bus, and HOLD is false and pHLDA is false, the temporary master enters the arbitration sequence, where it contests with other bus requesters for control of the bus.

Detailed specification of this process is given in 2.8.3.

2.8.2.3.3 Bus grant

Priority assertions on the arbitration bus settle in the interval between the assertion of a hold request and a hold acknowledge. At the rising edge of the hold acknowledge signal the bus is granted to the highest priority requester, enabling the bus transfer operation for that requester.

If the bus is not granted to a requester, that requester returns to the idle state.

The bus grant state is termed MINE.

†*Note:* Historically the MWRT strobe has been generated by front-panel devices to accomplish the deposit function. In such a case, the MWRT strobe is asserted while the front-panel holds the CPU in a wait state during a memory read cycle. Note that the status will indicate a read cycle and the pDBIN strobe will be active.

While this is acceptable procedure for 8-bit systems, and 8-bit memories should respond to MWRT as well as pWR*, it is not permissible in 16-bit systems as a conflict will exist on the bidirectional data bus. A front-panel could be implemented either as a temporary master, or integrated into the CPU.

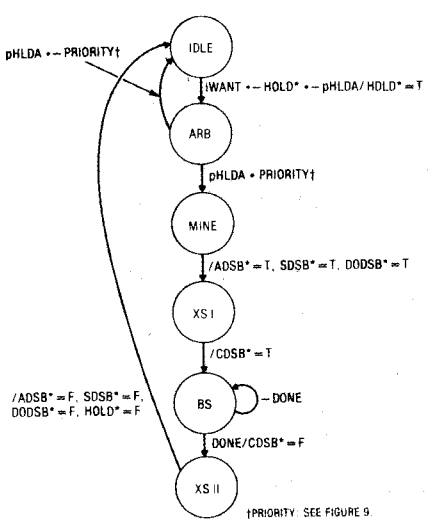


Figure 7. Bus transfer state diagram.

2.8.2.3.4 Transfer state one, XS I

The bus transfer sequence begins with transfer state one, XS I. The bus transfer control circuit asserts the following signals together:

1) ADSB*
2) SDSB*
3) DODSB*

disabling the address, status, and data output drivers of the permanent bus master and enabling the control output drivers of the temporary master. Both the permanent master and the temporary master are now driving the control output lines. These lines are required to have the following levels during this time.

	Signal	*Logic state*	*Electrical level*
1)	pSYNC	F	L
2)	pSTVAL*	F	H†
3)	pDBIN	F	L
4)	pWR*	F	H
5)	pHLDA	T	H

The transfer state is terminated by the assertion (by the bus transfer control circuit) of the CDSB* line, disabling the control drivers of the permanent master and enabling the address, status, and data out drivers of the temporary master. The temporary master now has complete control of the bus and begins its first bus cycle.

2.8.2.3.5 Bus cycles

Any number of standard bus cycles are then conducted by the temporary bus master. Bus control is never transferred between cycles. When the temporary master is done, the process proceeds to XS II, transfer state two.

2.8.2.3.6 Transfer state two

Transfer state two, XS II, is the mirror image of the sequence in XS I. The state begins with the release of the CDSB* signal, enabling the control output drivers of the permanent master and disabling the address, status, and data output drivers of the temporary master.

Both the temporary master and the permanent master drive the control output lines for the remainder of XS II at the levels prescribed for XS I.

The state is ended by the release of other disable signals and HOLD*, enabling the address, status, and data out drivers on the permanent master, and disabling the control output drivers of the temporary master. The permanent master now has complete control of the bus and the temporary master returns to the idle state.

2.8.2.4 Bus transfer timing relationships

2.8.2.4.1 General

The fundamental timing relationships for a bus transfer and a single DMA bus cycle are given in Figure 8.

Relationship to the bus transfer states is shown in boxes at the bottom of the figure.

Detailed specification of these times is given in 3.10 and Table 5.

2.8.2.4.2 T_{set}

A minimum time between the rising edge of the hold acknowledge signal and the assertion of the disable signals in XS I allows time for completion of the preceding bus cycle.

2.8.2.4.3 T_{ov}

The time that both the temporary master and the permanent master must drive the control output signals has a specified minimum to assure a smooth bus transfer.

Assertion of the XFER II signal, or CDSB*, is specified relative to the rising edge of the system clock, Φ, so that the assertion of this signal may be used by the temporary master as a cycle start signal.

2.8.2.4.4 T_{dh}

The "done" signal is a signal internal to the temporary master. This signal should not be asserted until the hold time for data output, status, and address signals in the standard bus cycle has been met.

2.8.2.4.5 T_{rel}

Completion of the reverse bus transfer XFER II shall precede the release of the pHLDA signal by a minimum specified time.

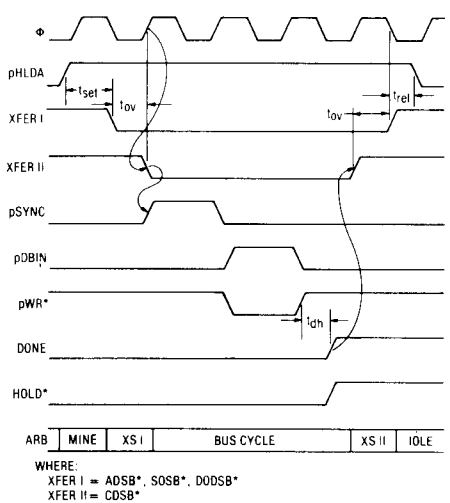


Figure 8. Bus transfer and single bus cycle.

†See note in section 2.4.4.1.

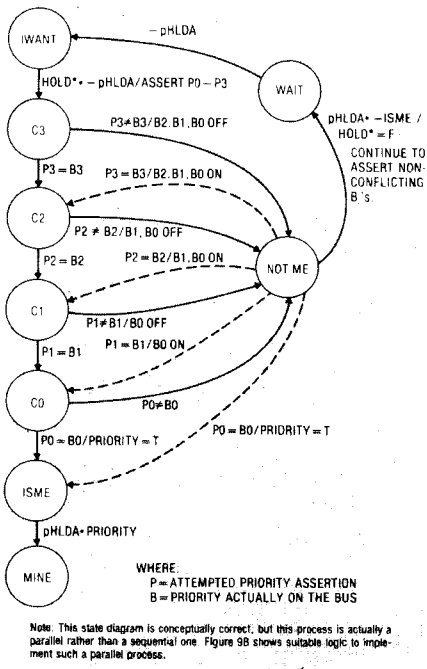


Figure 9a. Bus arbitration diagram.

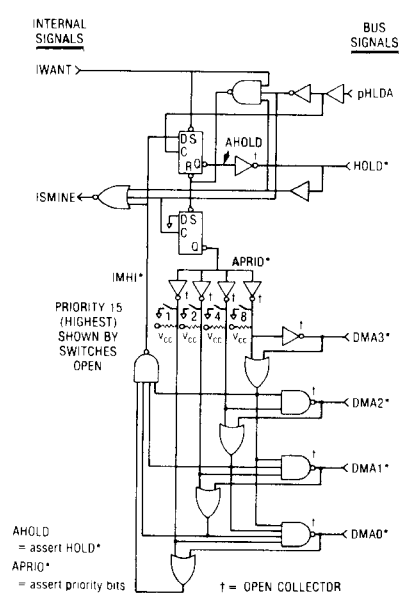


Figure 9b. Bus arbitration example.

2.8.3 Bus arbitration protocol

In a system which allows more than one master to use the system bus, for example a CPU permanent master and several temporary masters such as DMA controllers or multiple CPUs, some means must be provided to determine which device will be allowed to control the bus at any given time.

The bus arbitration system uses four bus lines for arbitrating among 16 temporary masters. These lines are driven by open collector drivers, and are pulled high by pullup resistors. Each temporary master has a unique priority number which it asserts on the arbitration bus at an appropriate time. A higher binary number indicates a higher priority.

The temporary masters compare the priority appearing on the active-low open-collector bus with the priority they are asserting, starting with the most significant bit. If disagreement is detected by any temporary master at any given bit position, then another temporary master must be asserting that priority bit and thus must have a higher priority. In that case all less significant bits are removed by the detecting temporary master. All more significant bits agree, and thus need not be removed, and the bit which disagreed must have been a 0 and thus was not asserted. Leaving the agreeing bits asserted reduces system noise caused by the redistribution of driving currents in the bus, and speeds settling of the correct priority on the arbitration bus. This process is a continuous asynchronous parallel process, not a sequential bit-by-bit process as it may seem from the above description. Incorrect comparisons will occur and be removed as the bus lines settle for as long as four bus delays (not related to the choice of four bus lines) plus logic delays.

The four lines which comprise the arbitration bus are DMA0* through DMA3*, where DMA3* is the most significant bit. These lines, in conjunction with HOLD* and pHLDA, control the bus arbitration process.

2.8.3.1 Bus arbitration implementation

An implementation of the bus arbitration protocol is shown in Figures 9a and 9b.

Any implementation shall obey the rules summarized in section 2.8.4.

2.8.3.2 Bus arbitration state definitions

2.8.3.2.1 IWANT

The IWANT state is an internal state for a temporary master which has determined that a bus access is necessary and thus wishes to arbitrate for bus control.

Temporary masters may not assert their priorities nor remove them at arbitrary times, or the arbitration bus may be in transition when the result is needed. A temporary master may assert its priority and the HOLD* bus request only if (1) pHLDA is not asserted (the permanent master has the bus), and (2)

HOLD* is not already asserted. This guarantees an ample time to settle the arbitration bus before the granting of the bus on the rising edge of pHLDA.

This scheme usually results in the first requester winning the bus. Only if simultaneous bus requests occur will the arbitration have any effect. This, however, is not improbable, since multiple unsuccessful requesters will become synchronized by waiting for the falling edge of pHLDA.

2.8.3.2.2 Priority compare states

The priority comparison states, C3 through C0, are the states where each requester compares the priority it is attempting to assert on the arbitration bus with the priority actually on the arbitration bus. Though C3 through C0 are shown and described as sequential, they are actually parallel processes. While disagreement occurs at any bit position, less significant bits are removed from the arbitration bus. If no disagreement persists after the settling time, the requester has the highest priority and will be granted the bus on the rising edge of pHLDA, proceeding to the state "MINE", where the bus transfer begins. All requesters continue to assert their priorities on the arbitration bus until the falling edge of pHLDA. Thus the priority number of the current bus master is available on the DMA bus while pHLDA is true. If the permanent master has the bus, pHLDA will be false.

A temporary master that wins the bus continues to assert its priority and HOLD* until its bus cycles are complete. A temporary master that loses the bus continues to assert its priority bits not turned off by the arbitration process, but must remove its assertion of the HOLD* line, so that the winner may indicate that it is finished by releasing HOLD*. A losing requester in this state is said to be in the "WAIT" state.

2.8.3.3 Bus arbitration timing relationships

Figure 10 shows two possible cases of the bus arbitration procedure. The first of these is a case where the requester has no competition; it requests the bus and the bus is granted. The second case shows the requester waiting for the bus to be free, arbitrating for the bus and losing, and arbitrating for the bus and winning.

2.8.3.3.1 No competition

When the temporary master determines that it requires the bus, it raises the internal signal IWANT. In this case, the rising edge of IWANT finds the pHLDA signal unasserted, meaning the permanent master has the bus, and the HOLD* signal unasserted, meaning that no other devices are requesting the bus. The temporary master may then assert the HOLD* signal and assert its priority on the arbitration bus. The ISME signal is the result of the arbitration process, and is asserted if none of the bit-wise comparisons on the arbitration bus fail. This arbitration result is clocked by the rising edge of the pHLDA signal, creating the bus grant signal MINE.

When the temporary master is finished with the bus, the IWANT signal is released, releasing the HOLD* signal and resetting the bus grant signal, MINE. The permanent master releases the pHLDA signal, and all assertions are removed from the arbitration bus.

2.8.3.3.2 Wait-lose-win

In this example the requester raises its IWANT signal, but finds the bus already busy and must wait to assert its bus request and priority until the falling edge of pHLDA.

The requester arbitrates for the bus during try 1, but another requester has a higher priority and the arbitration result ISME is low at the rising edge of pHLDA, indicating a loss in the arbitration process. The losing requester removes its assertion of the HOLD* signal, but continues to assert the non-conflicting high-order bits of its losing priority until the falling edge of pHLDA. At the falling edge of pHLDA, the process repeats, but this time results in a win for the requester.

2.8.4 Summary of arbitration protocol

Figures 9A and 9B represent an example, not a required implementation. Any implementation which obeys the rules may be used. The rules which must be obeyed by a temporary master are:

1) HOLD* may be asserted only when it is not already asserted and pHLDA is low.
2) HOLD* must be removed when pHLDA rises if another controller has asserted higher priority.
3) HOLD* must be removed when the controller no longer needs the bus.
4) Priority must be asserted whenever HOLD* is asserted, and must remain asserted until the next falling edge of pHLDA.

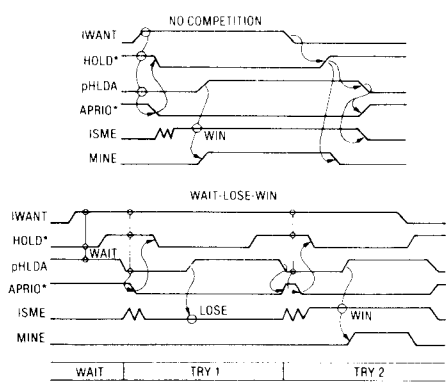


Figure 10. Bus arbitration timing diagrams.

5) The priority level must be user-selectable by switches and asserted by open-collector drivers on bus lines DMA3*-DMA0*.
6) The most significant bit of the priority level (appearing on DMA3*) must be compared with the priority asserted. If the line is asserted low but not by this temporary master, all less significant priority bit assertions must be removed. Similarly, bits DMA2*, DMA1*, and DMA0* must be examined and possible less significant conflicting bits removed.
7) If no lines are asserted low except those asserted by this temporary master after sufficient settling time, this temporary master has highest priority and may take the bus when pHLDA rises.
8) Logic implementations must be such that settling of the arbitration circuitry and bus will be completed between the assertion of HOLD* and the rise of pHLDA.

2.9 Interrupt protocol

The purpose of an interrupt system is to allow peripheral devices to suspend the operation of a bus master in an orderly way and to request that the master service the requesting peripheral. When service is complete, the bus master returns to the operation from which it was interrupted.

The interrupt protocol is comprised of an 8-level vectored interrupt system and a non-maskable interrupt. A complying master need only implement INT*.

2.9.1 Vectored interrupts

2.9.1.1 Vectored interrupt requests

Eight levels of vectored interrupt requests are issued on the vectored interrupt lines, VI0* thru VI7*, where VI0* is the most significant interrupt priority level. Vectored interrupt requests, however, may be rotated, masked individually, or "fenced out" by the interrupt control slave, and hence the priority levels are not fixed. Requests on the VI lines should be asserted as levels, that is, they should be held active until service is received. A slave which asserts a VI line need take no further action to generate an interrupt. It is assumed that if interrupt acknowledge cycles occur, an interrupt controller somewhere in the system will respond appropriately.

The generalized interrupt request line, INT*, is implemented as a communication line between the interrupt controller and an interruptable master. Any slave or interrupt controller, using the INT* line, must respond appropriately to any interrupt acknowledge cycles. The interrupt controller is not required to use INT*. A vectored interrupt may occur without INT* ever being asserted.

2.9.1.2 Interrupt acknowledge

The interrupt acknowledge cycle is a standard bus read cycle. The interrupt acknowledge cycle requests vectoring information from the interrupt controller to be asserted on the data bus during pDBIN.

Since no address information is asserted during an interrupt acknowlege cycle, only one interrupt controller may exist on the bus. If multiple interrupt controllers exist, they must either be "daisy chained" to avoid possible bus conflicts, or polled by the bus master.

2.9.2 Non-maskable interrupt (NMI)*

The non-maskable interrupt is an optional control input to bus masters. This interrupt is not maskable by a software instruction, and takes priority over other interrupt requests. The NMI* line may be used in the implementation of the special condition lines, ERROR* and PWRFAIL*.

NMI* is an open collector line. The bus master shall respond to negative going transitions on the NMI* line.

2.10 Special condition lines

Two special condition lines, PWRFAIL* and ERROR*, are available on the bus. Their use is optional.

2.10.1 Power-fail pending (PWRFAIL)*

This line indicates an impending system power failure. It is specified that this line shall be activated at least 50 msecs before the local voltage regulators drift out of specification.

The line stays low until the power-on clear signal is activated. This implies that either a normally closed relay or a battery powered circuit drive the power fail line. The circuit driving this line must meet the electrical specifications for an open collector line.

*2.10.2 ERROR**

This is a generalized error line that indicates that the current bus operation is producing an error of some sort (i.e., memory parity error, write to protected memory, inability to accommodate 8-bit slaves, etc.)

The ERROR* line should be implemented as a trap. All relevant information about the error-causing cycle—address, data, status, device number (for temporary masters)—should be latched on the falling edge of ERROR*.

ERROR* is implemented as an open collector line.

3.0 Electrical specifications

3.1 Application

This section defines the electrical specifications for interface devices to be used in S-100 bus systems. Proper operation of these devices also depends on two other factors:

1) Short physical distance between devices.
2) Relatively low electrical noise.

The electrical specifications for the bus driver and receiver circuits do not imply a particular technology, unless otherwise noted.

All specifications apply over the temperature range $T_a = 0°C$ to $70°C$.

3.2 Power distribution

Power in S-100 systems is distributed as unregulated DC power at three voltages, +8 volts, +16 volts, and −16 volts. Because these voltages are on adjacent lines it is relatively easy to short these lines on card removal. Therefore, bleeder resistors or other constant loads sufficient to discharge all three supplies rapidly are recommended.

3.2.1 +8 volt specification

Instantaneous minimum must be greater than +7 volts, instantaneous maximum less than 25 volts, and average maximum less than 11 volts.

3.2.2 +16 volt specification

Instantaneous minimum must be greater than 14.5 volts, instantaneous maximum less than 35 volts, and average maximum less than 21.5 volts.

3.2.3 −16 volt specification

Instantaneous maximum must be less than −14.5 volts, instantaneous minimum greater than −35 volts, and average minimum greater than −21.5 volts.

3.3 General signal discipline

Other than the power lines noted above, all signals on the bus are limited to positive signal levels between 0 volts and +5 volts, and may not have loaded rise or fall times less than 5 nsecs.

3.4 Driver requirements

3.4.1 Driver types

Three types of bus drivers are defined:

1) An active driver, either in the high state or in the low state or in transition, which has the capability to accept current in the low state and to provide current in the high state.
2) An open collector driver, which will not accept or provide current in the high state. A 1000Ω ± 5% pullup resistor to +5 volt or equivalent must be provided somewhere in the system for open collector lines. It is recommended that these pullup resistors be provided on the bus. However, implementation on the permanent master is also acceptable.
3) A Tri-State driver, which has the capability to be in the high-impedance state as well as in the high and low states.

3.4.2 Driver specifications

Specifications for bus drivers shall be as follows:

Low state (V_{OL}):	Output voltage less than or equal to +0.5 volts at 24 mA sink current.
High state (V_{OH}):	Output voltage (for active and Tri-State drivers) greater than or equal to +2.4 volts at 2 mA.

The leakage current for Tri-State drivers in the high-impedance state is specified as not greater than ±25 μA.

The internal capacitive load of a driver shall not exceed 15 pF at 25 °C whether in the active or the high-impedance state.

The rise and fall times of bus drivers should be minimized, subject to 3.3. In no case should the rise or fall times exceed 50 nsec at rated capacitive load.

3.5 Receiver specifications

The specifications for receivers on the bus shall be as follows:

Low state:	A voltage less than or equal to +0.8 volts shall be recognized as a low state.
High state:	A voltage greater than or equal to +2.0 volts shall be recognized as a high state.

Bus receivers shall source no more than 0.5 mA at 0.5 volts and sink no more than 50 μA at 2.4 volts.

Bus receivers shall have diode clamp circuits to prevent excessive negative voltage excursions.

Additional noise immunity is afforded by the use of Schmitt-type receiver circuits. Recommended hysteresis for such receivers should be greater than or equal to 0.4 volts.

3.6 Bidirectional signals

Some interface signals, such as the data bus, are combined Tri-State drivers and receivers. For each function these devices must meet the same specifications as separate drivers and receivers.

The total internal capacitive load for a line transceiver shall not exceed 20 pF at 25 °C.

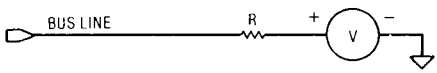
3.7 Card-level bus loading

At the card level, the following specifications apply:

1) The total capacitive load on any bus input shall not exceed 25 pF.
2) A card may not source more than 0.5 mA at 0.5 volts nor sink more than 80 μA at 2.4 volts on any signal line except for DMA0*, DMA1*, DMA2*, DMA3*, PHANTOM*, and PWRFAIL*. On these lines a card may not source more than 0.4 mA at 0.5 volts.

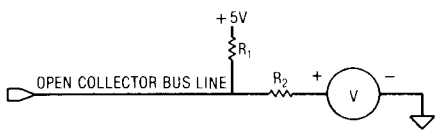
3.7.1 Bus termination

All bus lines except the power and ground lines may be terminated to reduce bus noise using a circuit equivalent to



where V=2.6 volts ± 0.2 volts and R is no less than 180Ω(± 5%).

Open collector lines may have a combination pullup and termination scheme using a circuit equivalent to



where V=2.6 volts ± 0.2 volts, R_1 = 1.5KΩ± 5%, and R_2 should be no less than 180Ω (±5%).

3.8 Read cycle timing specification

Figure 11a depicts the read cycle timing waveforms with the pertinent timing parameters shown. Table 4 specifies these parameters.

3.9 Write cycle timing specification

Figure 11b depicts the write cycle timing waveforms with the pertinent timing parameters shown. Table 4 specifies these parameters.

3.10 Ready and sixteen request timing specification

Figure 12 depicts RDY, XRDY, and SIXTN* timing waveforms during read and write cycles, with pertinent timing parameters shown. Table 4 specifies these parameters.

3.11 Bus transfer timing specification

Figure 8 depicts bus transfer timing waveforms with the pertinent timing parameters shown. Table 5 specifies these parameters.

4.0 Mechanical specifications

4.1 Application

This section defines the mechanical specifications for standard interface systems.

4.2 Connector type

The card edge connector is a 100-pin (dual 50) connector with contacts spaced on 0.125" centers. It is nominally designed for printed circuit boards 0.062" thick.

The connector is subject to the specifications in 4.2.1 and 4.2.2.

4.2.1 Electrical considerations

1) Voltage rating: 200 volts DC, minimum pin to pin.
2) Current rating: 2.5A per contact.
3) Contact resistance: 50 mΩ maximum at rated current after 100 insertions.
4) Insulation resistance: 1000 MΩ minimum.

4.2.2 Connector spacing

Connectors should be spaced 0.75 inches −0.01 inches center to center.

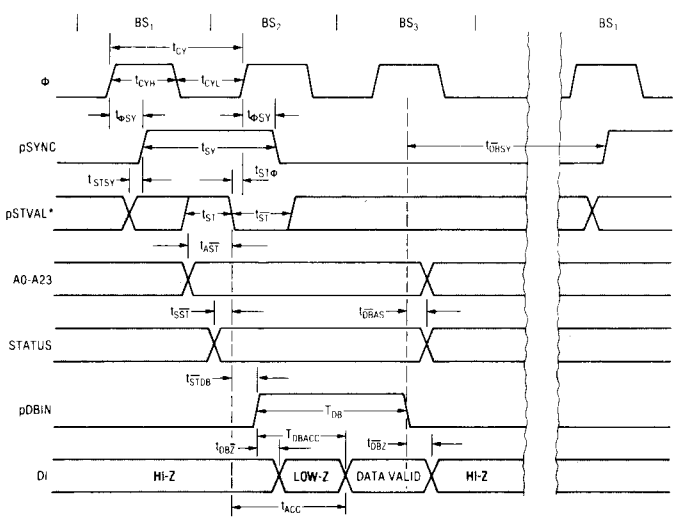


Figure 11a. Read cycle timing diagram.

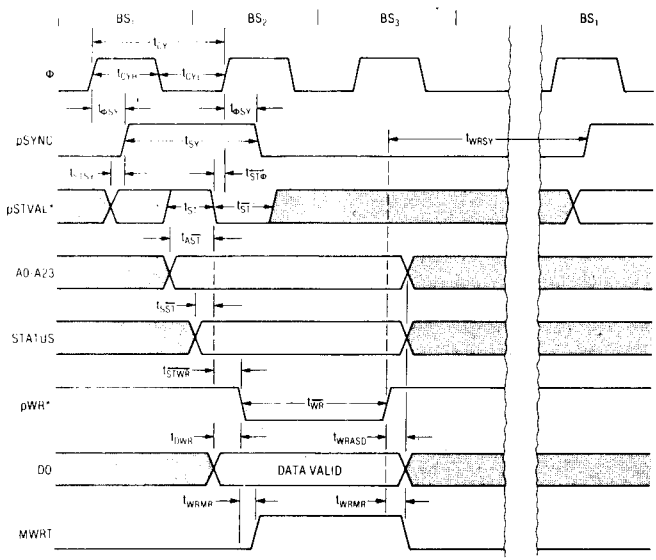


Figure 11b. Write cycle timing diagram.

Table 4. Read/write cycle timing parameters.

		MIN.(NSECS)	MAX.(NSECS)
t_{CY}	Φ PERIOD	166	2000
t_{CYH}	Φ PULSE WIDTH HIGH	$0.4t_{CY}$	
t_{CYL}	Φ PULSE WIDTH LOW	$0.4t_{CY}$	
$t_{\Phi SY}$	DELAY Φ HIGH TO pSYNC HIGH; DELAY Φ LOW TO pSYNC LOW	10	$0.4t_{CY}$
t_{SY}	pSYNC PULSE WIDTH HIGH	$0.7t_{CY}$	
$t_{\overline{ST}\Phi}$	pSTVAL* LOW PRIOR TO Φ LOW DURING pSYNC	0	
t_{ST}	pSTVAL* PULSE WIDTH HIGH	50	
$t_{\overline{ST}}$	pSTVAL* PULSE WIDTH LOW	50	
t_{STSY}	pSTVAL* FALLING EDGE PRIOR TO pSYNC HIGH	0	
$t_{\overline{AST}}$	ADDRESSES STABLE PRIOR TO pSTVAL* LOW DURING pSYNC HIGH	70	
$t_{\overline{SST}}$	STATUS STABLE PRIOR TO pSTVAL* LOW OURING pSYNC HIGH	40	
t_{DB}	pDBIN PULSE WIDTH HIGH	$0.9t_{CY}$	
$t_{\overline{STDB}}$	DELAY pSTVAL* LOW TO pDBIN HIGH	20	
$t_{\overline{DBSY}}$	DELAY pDBIN LOW TO pSYNC HIGH	0	
$t_{\overline{DBAS}}$	HOLD TIME FOR ADDRESSES AND STATUS AFTER pDBIN LOW	50	
$t_{\overline{DBZ}}$	DELAY pDBIN LOW TO SLAVE DI ORIVERS Hi-Z		$25+0.1t_{CY}$
$t_{DB\overline{Z}}$	DELAY pDBIN HIGH TO SLAVE DI DRIVERS ACTIVE	10	$25+0.1t_{CY}$
t_{ACC}	DELAY pSTVAL* LOW TO DATA VALID	SPECIFIED BY MANUFACTURER. WORST CASE MAXIMUM FOR ALL SLAVES AND WORST CASE MINIMUM FOR ALL MASTERS.	
$t_{S\overline{DB}}$	DATA VALID SETUP TIME TO pDBIN LOW		
$t_{\overline{WR}}$	pWR* PULSE WIDTH LOW	$0.9t_{CY}$	
$t_{\overline{STWR}}$	DELAY pSTVAL* LOW TO pWR* LOW	30	
t_{WRSY}	DELAY pWR* HIGH TO pSYNC HIGH	0	
$t_{D\overline{WR}}$	SETUP TIME DO VALID TO pWR* LOW	$0.1t_{CY}$	
t_{WRASD}	HOLD TIME ADDRESSES, STATUS, AND DO FROM pWR* HIGH	$0.2t_{CY}$	
t_{WRMR}	DELAY pWR* LOW TO MWRT HIGH; DELAY pWR* HIGH TO MWRT LOW		30
$t_{\overline{RDY}\Phi}$	SETUP TIME RDY, XRDY, SIXTN* TO Φ RISING	80	
$t_{\Phi RDY}$	HOLD TIME RDY, XRDY, SIXTN* AFTER Φ RISING	70	

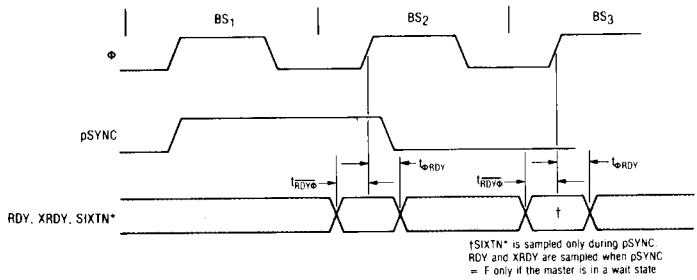


Figure 12. Timing of RDY, XRDY, and SIXTN* during read and write cycles.

Table 5.
Bus transfer timing parameters.

t_{SFT}	DELAY pHLDA TO ADSB*, SDSB*, DODSB*	30
t_{OV}	TIME BOTH TEMPORARY AND PERMANENT MASTER DRIVE THE CONTROL OUTPUT LINES	$0.5t_{CY}$
t_{DH}	HOLD TIME ADDRESS, STATUS, AND DATA OUT DURING DMA CYCLE	$0.2t_{CY}$
t_{REL}	SETUP TIME, END OF BUS TRANSFER TO pHLDA RISING EDGE	20

4.3 Board size specification

Circuit boards shall conform to the board size specifications given in Figure 13. The edge connector pin shown in the figure is pin 50. Pin 100 opposes pin 50 on the back side of the board.

Total board depth shall not exceed 0.65".

Nominal board thickness is 0.062".

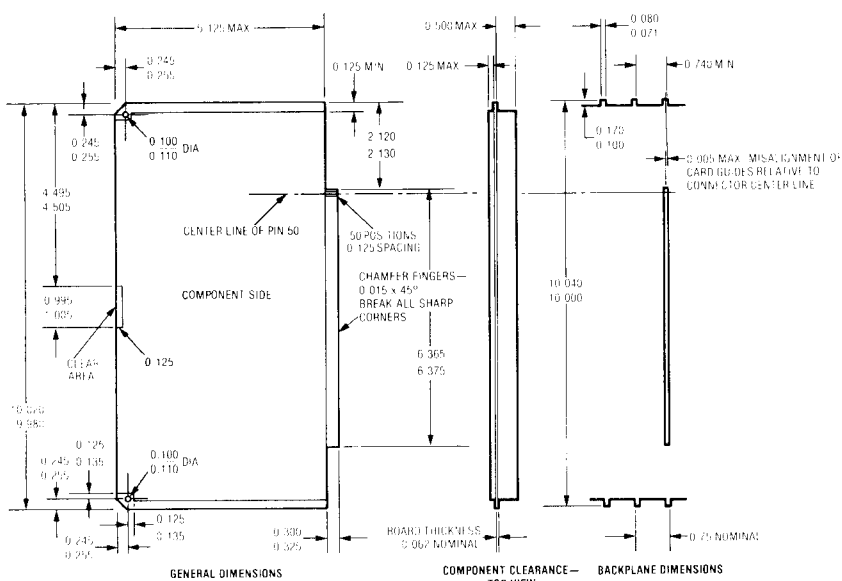


Figure 13. S-100 board mechanical parameters.

index

Other OSBORNE/McGraw-Hill Publications

An Introduction to Microcomputers: Volume 0 — The Beginner's Book
An Introduction to Microcomputers: Volume 1 — Basic Concepts, 2nd Edition
An Introduction to Microcomputers: Volume 2 — Some Real Microprocessors
An Introduction to Microcomputers: Volume 3 — Some Real Support Devices
Osborne 4 & 8-Bit Microprocessor Handbook
Osborne 16-Bit Microprocessor Handbook
8089 I/O Processor Handbook
CRT Controller Handbook
68000 Microprocessor Handbook
8080A/8085 Assembly Language Programming
6800 Assembly Language Programming
Z80 Assembly Language Programming
6502 Assembly Language Programming
Z8000 Assembly Language Programming
6809 Assembly Language Programming
Running Wild — The Next Industrial Revolution
The 8086 Book
PET and the IEEE 488 Bus(GPIB)
PET/CBM Personal Computer Guide, 2nd Edition
Business System Buyer's Guide
OSBORNE CP/M® User Guide
Apple II® User's Guide
Microprocessors for Measurement & Control
Some Common BASIC Programs
Some Common BASIC Programs — PET/CBM Edition
Practical BASIC Programs
Payroll with Cost Accounting
Accounts Payable and Accounts Receivable
General Ledger
8080 Programming for Logic Design
6800 Programming for Logic Design
Z80 Programming for Logic Design

interfacing to S-100/IEEE 696 microcomputers

by Sol Libes and Mark Garetz

The S-100/IEEE 696 is the most widely used bus today. It is supported by over 100 different manufacturers and is compatible with a variety of CPUs. There are several times more languages, operating systems, and application packages available for S-100 systems than for any other system.

This book helps S-100 Bus users expand the utility and power of their systems. It describes the S-100 Bus with unmatched precision. Various chapters describe its mechanical and functional design, logical and electrical relationships, bus interconnections, and busing techniques. Both parallel and serial interfacing are described, as well as interfacing to RAM, ROM, and the real world. Additional chapters discuss A/D and D/A conversion, interrupts, timers, and direct memory access.

Sol Libes is a teacher, author and member of the IEEE Standards Committee on the S-100 Bus. He has written and spoken extensively on the S-100 Bus, and he is the founder of *S-100* magazine. His authorship, along with that of prominent columnist Mark Garetz, makes this book a truly authoritative reference.

ing to
EEE 696
omputers interfacing to
acing to S-100/IEEE 696
IEEE 696 microcomputers
omputers

interfacing to S-10
S-100/IEEE 696 mi
microcomputers

ISBN 0-931988-37-3