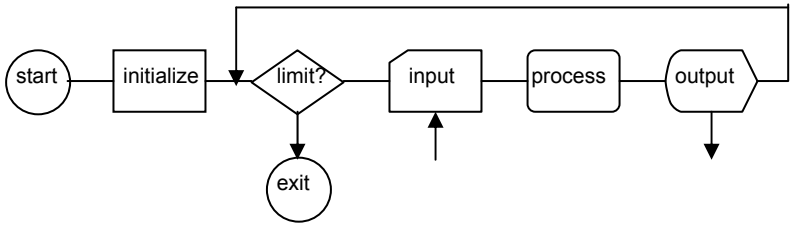
Systems Design
and the 8051

The hardware, firmware, and software design of microprocessor systems

Second Edition

Marcus O. Durham, PhD, PE



TechnoPress
Tulsa

Systems Design and the 8051

Contact:
THEWAY Corp.
P.O. Box 33124
Tulsa, OK 74153

www.ThewayCorp.com
mod@superb.org

Edited by:
Cover Design: Marcus O. Durham, PhD
Printed in United States of America
First printing by Fidlar Doubleday, August 2003
Second edition by Fidlar Doubleday, August 2004

Library of Congress Control Number

ISBN: 0-9719324-6-8

TO

Russ Sanders, a former student who has taken his early training and made a business of it. *Ron Sanders*, a colleague, who has worked on my various designs for many years.

Dan Sossamon, Widjaja Sugiri, Neil Chikode, and Matthew Olson, former graduate students who contributed to the development of the ideas, projects, and designs. *Tuan Huynh*, the first graduate assistant to review the Second Edition.

Rosemary Durham, my wife, best friend, and supporter for all my various ventures.

$\Leftarrow \Uparrow \Rightarrow$

Table of Contents

⇐ ⇑ ⇒

SECTION I - FOUNDATIONS

⇐⇑⇒

1

INTRODUCTION

Thought
Engineering is the tradeoff
between quality, time, and money
You can have two, but you cannot have all three.
MOD

Why this book, now

Why is there a need for another book on system design or on the 8051 microcontroller? Actually, the answer is quite simple. There are numerous books on systems and digital design. There are even several on the 8051. However, we have found no reference that treats the 8051 as the basis of a system.

In addition, most books present substantial theory before introducing projects. The task of this book is to start building projects immediately after discussion of the relevant topics. After all, that is why most people want to be an engineer – it is to build objects and see them work.

Most books and articles on the 8051 treat it as a simple device, suitable for just a few tasks. Most authors have sample projects, but they are not integrated. On the other hand, I have used the microcontroller family as a basis for industrial and commercial systems for over 20 years. I have also used it as the foundation for an upper / graduate level university design class.

Why use the 8051 architecture? The core of the 8051 arguably is used more than any other device. It is an expansion of the i8042 architecture, which is the fundamental processor used in keyboards for personal computers. Phenomenally, the architecture has been around for over 25 years. It is constantly being enhanced by various vendors to perform tremendous tasks.

What is meant by a system design with the microcontroller? One microcontroller forms the kernel for all the electronic and digital functions associated with a system. The power of the 8051 devices makes it very capable of filling the functions of a much larger computer system.

The system approach is to integrate together in one microprocessor every task simultaneously. These include digital in, digital out, analog in, analog out, serial communications with other computers, keypad, seven-segment display, liquid crystal display, local and remote control, data logging, 32-bit math calculations with square root, and all this in a real time environment. That is a system!

An earlier reference, *Microcontrollers in Systems Design* began the task. It was very project oriented with an inclusive reference for the 8051. However, it had one shortcoming. It required the reader to have a good grasp of computer systems and (s)he had to translate the reference material into the projects.

This work, *Systems Design and the 8051,* takes a different tack. It is based on engineering design principles that are elucidated as required. Extreme programming (XP) approach to compatibility allows each new project to be integrated as a separate, but interactive module.

The book begins with the fundamentals and sequentially adds new projects. This is accomplished by integrating a straightforward and concise application explanation for all software commands. That is followed by a circuit representation of the new components. The project requirements are laid out. Finally, an exemplar program shows a similar system that the designer can modify to complete the project.

The procedure can obviously work. For many years, I have had students step-by-step build a functioning, integrated computer system around 10 projects. They start with a parts list, obtain the parts, and build the projects into a working system. This is done in a single semester. What an accomplishment!

Development environments

In most design environments, hardware and software are segmented. Moreover, specialists typically handle the tasks separately. While this may be beneficial for large systems development, it restricts understanding of the overall big picture of computer applications.

This treatise will present the application of both hardware and software to the solution of real problems. The most effective piece of equipment to provide a simultaneous understanding of computer architecture and programming is the microcontroller.

The microcontroller has a complete computer built on a chip. In addition, much of the interface hardware to outside components has been included on the chip. A working computer is obtained by connecting switches to the input, digital displays to the output, and adding a program to the memory.

The prerequisites for being able to successfully build the projects are minimal. A reasonable understanding of TTL digital logic is assumed. Proficiency in a high level programming language is also assumed. Understanding of an assembly language would be very beneficial, but is not necessary.

Book structure

The structure of the treatise is to provide a working tool that can be readily referenced. The first topics are general items and fundamental connections. The next area is a group of projects that can be constructed. Chapters that give the foundation material precede each project. An exemplar program that can be modified to complete the project follows the project description.

A parts list and development board are discussed. The instructions and memory organization comes next. This is followed by communications protocols. Then, a programmable logic device (PLD) representation is used for the combinational logic. The final chapters are documentation techniques. It is a good idea to look over these section to have an idea of the available reference material for use as the projects are developed.

The projects will begin with using a memory device as a table for arithmetic functions. A number of projects will be constructed for performing common control tasks. The epitome of control will entail communications between two computers.

The projects are for demonstration of technology and gaining of experience. Where necessary, a schematic and/or software are provided. Often this is an illustration of a related idea. It is not intended to be a solution to the project. It is intended to provide a framework for tackling the project.

Class structure

It is not necessary to perform all the projects or to do them all in order to obtain a working knowledge of microprocessors.

A proven procedure is to use Project 1, 2, and 3. Project 4 can be included if a development board is used. Then Project 6 and 7 should be completed. These will provide the foundation to do any project. The remaining projects can be used in almost any order, if desired.

Projects 8, 9, and 10 provide human interface devices.

Project 11, 12 and 13 give infrared communications and analog / digital conversion exposure.

Within a semester, I typically use 1, 2, 3, 4, 6, 7, 8, 9, 10 and a choice of 11, 12, or 13.

You can accomplish the tasks. The projects should prove both challenging and enjoyable. Occasionally, you may be frustrated but persevere. As a result, the completion of the task will be most rewarding.

The material comes with a guarantee. If you complete all the tasks, you will have the tools to design a computer system for any purpose. Good Computing!

Credit where credit is due

Everything we know is developed from something we have read, heard, or seen. Therefore, these other thoughts necessarily influence what we write. To the best of our knowledge, we have given specific credit where appropriate.

Rather than footnotes or references, we have listed the works that have provided significant information in one way or another, since this is often in concepts rather than quotes. Some of the information is general and public domain, while some is device specific. The generic information is used where possible.

Statements that are attributed to us are things we have used commonly and do not recall seeing from someone else. Others obviously have similar thoughts. If we have made an oversight in any credits, we apologize and we would appreciate your comments.

Dr. Marcus O. Durham

⇐ ⇑ ⇒

2

FUNDAMENTAL CIRCUITS

Thought
Tell me the input, output, and
what to do in between, then
I can write a computer program.
Capt. Ed Fischel, USAF

Fundamentals

What are the fundamentals? Is it not input, output, and how these are related? A mechanical switch represents every type of input there is in a digital world. Similarly, a lamp or light emitting diode (LED) is a surrogate for every possible output.

These two simple devices can effectively interface to any real world circumstance by placing a buffer between the digital components and the external items. Hence, a transistor or gate can substitute for the switch and LED.

The power for most digital computer devices historically has been 5 volts DC referenced to ground. In many of the newer technologies, this level is dropping below 2 volts. Regardless, the power voltage is called V_{CC} and has a digital value of 1 or True. Similarly, the reference ground is zero volts, is called V_{SS}, and has a digital value of 0 or False.

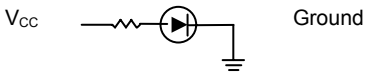
The power is connected to the upper right corner of most chips. Similarly, the ground is connected to the lower left corner.

Output

Since a light emitting diode (LED) is a logic device, it has polarity. The positive connection is called the anode, it is the longest lead, and is the arrow side of the diagram. The negative terminal is the cathode, it has a flat spot on the edge of the LED, and is the bar in the diagram.

A diode has a very low internal resistance. Hence, it has very limited current handling capability. If it were connected directly from power to ground, the current through the low resistance would rise very high. The power dissipation would increase by the square of the current. Then the diode would burn out or blow.

To restrict the current, common practice is to place a resistor in series with the LED. In restricting the current, the brightness is reduced. Therefore, for optimum brightness, the series resistor should be calculated to give the optimum current.



Various light emitting diodes have a voltage drop of 1.4 to 2.5 Volts. The maximum current for different LEDs is 20 up to 750 mA. However, the current must be restricted below the maximum that can be supplied by the gate. To limit the current a resistor is placed in series. Assume a current of 5 mA and 1.5 V drop.

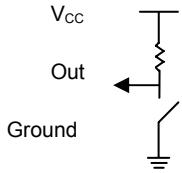
$$R = V / I = 1.5\ V / 5\ mA = 300\ \Omega$$

Typical values are 270 to 330 Ohms. If the lamp is not bright enough, the specifics should be verified, and a different resistor should be calculated.

Input

A switch must represent two states, both off and on. Therefore, it must be connected to both power and ground. To prevent a short circuit when the switch is closed, a resistor is placed between the power and the switch. This is called a pull-up resistor. The other side of the switch is connected to ground.

The output of the switch is taken between the resistor and the switch. The other side of the output is ground.



The resistor must be sized to limit the current in a short-circuit condition. Many devices have a 10 mA limit on the amount of current they can sink. When connecting a switch as a short circuit, the current is often limited to 2.5 mA. Therefore, the resistor is sized based on that value.

For a voltage source of 5 V and a current load of 2.5 mA, the resistance is calculated from Georg Ohm's rather famous law.

$$R = V / I = 5 \text{ V} / 2.5 \text{ mA} = 2000 \text{ Ohm}$$

The common sized pull-up resistor is typically 2,000 to 2,200 Ohms.

Next consider the power rating of the resistor.

$$P = V*V / R = 5*5 / 2200 = 0.011 \text{ W}$$

At this low quantity of power, virtually any resistor network has adequate dissipation capability.

Computer vs. microcontroller

What is the difference in a computer, microprocessor, and microcontroller? A computer is a device with an internal processor, memory, and interface for input and output. The processor consists of a central processing unit, a control unit, and an arithmetic processing unit.

A computer is a complete system, while a microprocessor is basically the core of the unit. The microprocessor will generally require other chips to connect to the input / output interface.

A microcontroller is a microprocessor with additional interface components as part of the chip. In essence, a microcontroller is designed to connect directly to the input and output. In addition, it has some memory as part of the chip. In some cases, these are also called single chip computers.

A microcomputer is typically designed for general-purpose applications. A microcontroller usually has a dedicated purpose as a control or instrumentation device.

History 101

The computer was originally developed during World War II. These machines were physically very large, but had limited memory. Large computers continued to develop until the very powerful mainframe systems of the 1960's. These machines still had discrete magnetic core memory, transistors, and diodes.

At that time, the first integrated circuits (ICs) were being developed. From the earliest days of the IC, microcomputers have been a major component. A brief chronology of the microcomputer shows the rapidity of the development.

1968 Intel formed from Fairchild Semiconductor

1971 Intel 4004 – 4 bit calculator
1972 Intel 8008 – eight bit
1974 Intel 8080 – eight bit TTL compatible
Motorola 6800
MOS Technology 6502
1978 Intel 8086 – sixteen bit basis of PC
1979 Intel 8051 – eight bit microcontroller

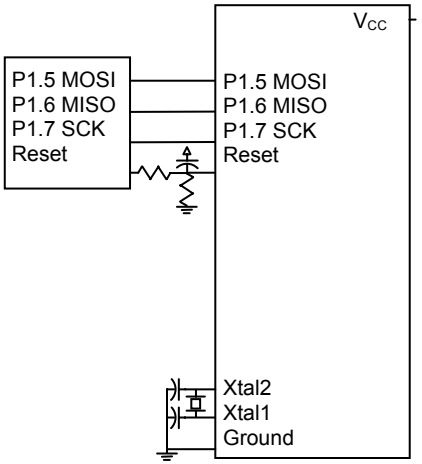
It is fascinating that the architecture and software of the 8051 has continued for over 20 years. It is the core architecture for over half of all microcontrollers.

Microcontroller

The basic computer circuit is actually very simple. First power and ground are connected to the upper right and lower left pins respectively. Next, a crystal is connected to the XTAL pins to provide stimulus for the internal oscillator. An 11.059 MHz crystal will provide good performance and excellent communications. Other choices are available and can be found in later sections. A very small coupling capacitor of 10 – 40 pf is connected from each side of the crystal to ground.

Third, pulling the RESET line high restarts the processor. The RESET line can float low during normal operation. A power-on reset circuit is often connected to the pin to automatically perform the start function.

When using a flash microcontroller, a method of loading the program is required. Program loading involves three lines, an input for instructions, an output for data, and an input for clocking the other two lines. An illustration is given in the figure below. Details for implementing the in-system feature are provided in a chapter on in-system programming in the reference section after the projects. For microcontrollers without in-system programming, these lines are not required.



Microcontroller input/output

The input / output interface is the defining feature of a microcontroller. On the processor, there are four different ports for this function. Each port represents one byte, which is connected in parallel to pins of the chip.

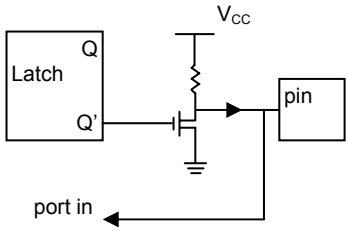
The port is identified as a special function register within the internal RAM. Because of the address of the ports, all are bit addressable. The ports are bi-directional. In default operation, the port will output a byte or bits of data.

Any bit of the port can be converted to input. Simply send a one to the bit of the port. Then that bit can be read or input. If all ones are sent to a port, then the entire byte becomes an input.

The internal circuitry for one bit of a port is shown. A latch holds the output to the pin. A separate line provides the input.

For output, data is written to the port. The result is displayed on the pin. For input, a one is sent to the port. This enables the line and

pulls the pin high. Therefore, with no other connection to the pin, input from the port bit will be one. If the pin is pulled low by an external switch, then the input is a zero.



Sink or source

Connecting external devices to a gate can critically impact the chip. A gate, whether in a microcontroller or a logic device, is limited to the amount of current it can handle. If too much current is drawn by an external connection, the chip may be damaged. Therefore, a resistor or a buffer must limit the current.

A chip may provide V_{CC} or Ground to the external devices. Supplying V_{CC} is the high state and is a source for current. Supplying the ground connection is the low state and is a sink for current.

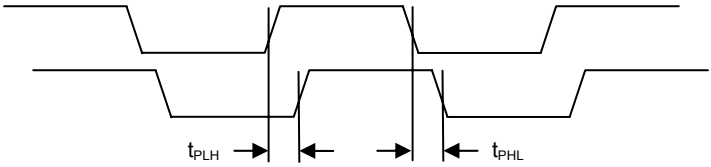
Sink current is the amount of current that will flow into an output, I_{OL}, at low state. *Source* current is the amount of current that will flow out of a gate, I_{OH}, at the high state.

Total source (I_{OH}) = input to load (I_{IH}) * number of loads (m)
Total sink (I_{OL}) = input to load (I_{IL}) * number of loads (n)

The number of loads is called fan-out. The limiting fan-out is the lower of the number of loads (m or n). The maximum current, and resulting fan-out, must be observed to preclude overloading and damaging a gate. When interfacing to different logic families, total the load current for all devices.

Propagation delay and power consumption

Two other parameters determine the performance of a gate. Propagation delay is the delay of low-to-high (t_{PLH})and high-to-low (t_{PHL}) transitions between the input and output of a gate.



For TTL logic, power consumption is an average of power consumption for high and low state output.

$$P_D = V_{CC} * (I_{CCH} + I_{CCL}) /2$$

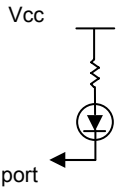
The table below gives the performance for various logic families. A minus sign indicates current is flowing out of the gate.

Description	Range	Unit	Sym	S	LS	AS	ALS	F	CMOS
Propagation delay	typical	Ns	t_{PD}	3	9	1.7	4	3	
P consumption per gate	avg.	mW	P_D	19	2	8	1.2	4	
Speed*power (energy)		pJ	E	57	18	13.6	4.8	12	
Low-level input voltage	max	V	V_{IL}	0.8	0.8	0.8	0.8	0.8	1.5
Low level output voltage	max	V	V_{OL}	0.5	0.5	0.5	0.5	0.5	0.05
High-level input voltage	max	V	V_{IH}	2.0	2.0	2.0	2.0	2.0	3.5
High-level output voltage	max	V	V_{OH}	2.7	2.7	2.7	2.7	2.7	4.95
Low-level input current	max	mA	I_{IL}	-2.0	-0.4	-0.5	-0.2	-0.6	-0.1 µA
Low-level output current	max	mA	I_{IH}	20	20	20	8	20	0.1 µA
High-level input current	max	mA	I_{OL}	50	8	20	20	20	0.88 µA
High-level output current	max	µA	I_{OH}	1000	-400	-2000	-400	-1000	-0.88 µA

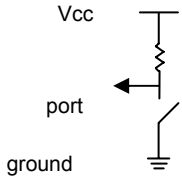
A later chapter on circuit time and phase shift addresses the timing and propagation delay in more detail.

External input/output

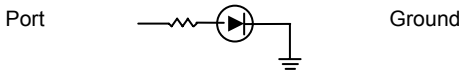
An LED is shown connected with the chip as a sink. In this mode, a zero "0" is sent to the port to illuminate the LED. This is the preferred approach since chips sink more current than they can source.



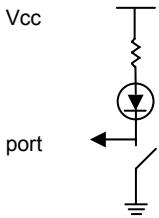
This circuit is very similar to that used for a switch input.



In this schematic, an LED is shown connected with the chip as a source. In this mode a one "1" is sent to the pin to illuminate the LED. This is a poor design since the chip is limited in the amount of current it can source.



One circuit of particular interest combines both an LED and a switch on a single port pin. This permits both a display and an input.



When the switch is closed, the lamp will be on. Therefore, the switch must be open for the lamp to be controlled by the port. When the port sends a zero, the lamp is on. When the port sends a one, the lamp is off, and the switch can be read.

⇐⇑⇒

3

MEMORY DEVICES

Thought
Memory is the second thing to go.
He just does not remember the first.
Old quip

Where do you keep it

Two fundamental types of memory are typically used in a computer system. These are program memory and data memory. Program memory is static and seldom changes. Data memory is dynamic and changes with the conditions. These functions determine the technology that can be used for each purpose.

Program

Program memory is typically completely changed when there is a new program. Program memory can use one time technology. This includes devices such as read only memory (ROM). The basic version is programmable ROM or PROM. Enhanced versions allow the device to be erased by ultraviolet light and programmed by special hardware. These are called erasable programmable ROM or EPROM.

As technology advanced, the chip could be erased and programmed by changing the voltage. These are electrically erased PROM or

EEPROM. A variation of the technology is called flash memory. The erasure and programming capability is built into the chip. This is selected by using one of the pins.

Data

Data memory requires the status to be changed frequently. Only a part of the memory is typically involved. Therefore, random access memory (RAM) is used. Not surprisingly, this technology is more complex.

Some internal memory uses EEPROM technology to accomplish this purpose. Because of the technology, the number of read/ write cycles is often limited.

Dual in-line package

A common version of separate memory chips uses dual inline packaging (DIP). These are typically limited to prototyping or limited quantity projects.

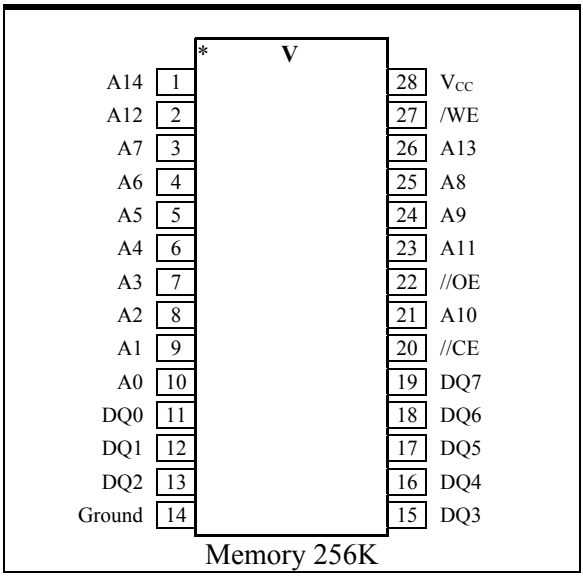
The quantity of memory in the package is constantly increasing as technology improves. To handle the additional pins, other package technologies are often used for production devices.

The pin-out for static RAM, PROM, and EEPROM is almost identical. There are two basic differences. The programming pins are called write pins and the output enable pins are called the read pins.

A typical 32K address X 8-bit byte package is illustrated as a 256K bit device in the following diagram.

Connections

Like most other DIP packages, power is supplied to the upper right (pin 28), and ground is supplied to the lower left (pin14).



In addition, the chip must be selected or enabled. The Chip Enable not (/CE) line must be pulled low for the chip to function. If there are multiple memory chips, this is done by an address decoding chip. If there is only one chip, then the line is simply connected low.

There is one precaution about connecting address lines on the memory chip. Avoid letting any pin float. Connect the line to an address decoder. If the address line is unused, connect it to ground.

When sending data to the chip, the Write Enable not (/WE) line must be pulled low. When obtaining data from the chip, the Output Enable not (/OE) line must be pulled low. Obviously, the two lines should not be pulled low simultaneously.

How it works

Since reading is common to both types of memory, an output or read will be used as a description of how a memory chip works. The combination of address lines is used to select a particular byte of information stored in the memory. The information stored in that location will be imposed on the output pins.

The number of addresses is a power of two function of the address lines. If there are 'm' address lines, then the number of addresses can be calculated easily.

$$\text{Addresses} = 2^m$$

For eight (8) address connections, there are 256 addresses. The first address is zero (0) and the last would be 255.

Ten addresses lines would access 1024 locations. In common usage, this is rounded down and called a 1K device. Larger sizes are proportionally rounded to give common sizes that are powers of two. A few examples are 16K, 32K, 64K, and 512K.

Each address accesses 'n' bits of information stored in the memory and displayed on the output. In microcontroller systems, the number of bits is 8 and is called a byte.

The total number of bits is used to describe memory devices. The number of addresses and the number of bits output determines the size of the memory. For example, a chip with 8 address lines and 8 output lines would be a 2K device.

$$2^m * 8 = 2^8 * 8 = 2048$$

Other uses

Since the chip will have an output based on the status of the address lines, it can be used to represent a conventional logic network. There are 'n' outputs and each output can have 2^m maxterms.

The ROM has fixed values, which makes it well suited for projects that require a table lookup. This is particularly appropriate for mathematics problems that are repeated frequently.

Consider an example.

Given: $y = 2x^2 - 1$

Allowable range: $0 \le x \le 3$

What is the address (input) variable?	x
What is the data (output) variable?	y
How many addresses are there?	4 (0, 1, 2, 3)
How many address lines are required?	$m = 2 \rightarrow 2^m = 4$
What is the largest value output?	17
How many output lines are required?	$n = 5 \rightarrow 2^n = 32$

Create a table of values to implement the function. Negative values are represented by setting the most significant bit (MSB) to 1. Decimal numbers can also be represented by setting another bit.

Input	Address lines	Memory	Output lines
0	00	-1	100001
1	01	1	00001
2	10	7	00111
3	11	17	10001

By connecting switches to the address lines and LED's to the output lines, the special purpose calculator is realized.

⇐ ⇑ ⇒

4

PROJECT 0 - MEMORY

Thought
Skip this project,
if not using external memory.

Project 0: Math using ROM _____

***Purpose*:** Skip this project if you are not using external memory.
To show the function of a ROM.
To program an EPROM.

Preamble:

Every microcomputer has a Read Only Memory (ROM) to store the basic system program. In the case of a development system, a programmable and erasable ROM is preferred. Erasable Programmable Read Only Memory (EPROM) is the most common example. EPROM has lower cost than most other alternatives.

An EPROM contains fuses. These are disconnected or left as is to create a pattern of information called programs. An EPROM has address lines usually called A1, A2...An, on the circuit diagram. The EPROM data lines are accessed for the output. The lines usually are called O1, O2, O3...On.

The function of address and data lines is exactly the same as the ones on a ROM. Placing information in an EPROM is called

programming. Exposing the EPROM to an ultraviolet (UV) light source will clear all locations of their programmed contents. The duration of the exposure depends on the UV intensity. An alternative is electrically erasable memory (EEPROM). The programmer / burner electrically erases this version of storage. The EEPROM is much faster and easier to erase and program.

Plan:

Demonstrate a ROM is a program memory. Program the ROM to act as a table for a certain function.

Preparation:

Prepare a breadboard with a +5V power supply. Observe the EPROM data sheet. Then wire light emitting diodes on the EPROM data lines. Wire five switches on the address lines.

Investigate the use of the EPROM burner program on the personal computer (PC). Become very familiar with these tools since they will be used in burning all programmable chips. Before burning an EPROM, make sure that the device is blank.

Check the EPROM to ensure any lock bits are off.

Procedure:

Implement the function $F = x^3$ on the EPROM. Use the address lines as the inputs (x) and data line as the outputs(F). Only 5 switches are required on the address lines.

Presentation:

Demonstrate the output for x = 0, 1, 2...5. Only selected numbers will be requested for demonstration.

5

MICRO PRIMER

Thought
Prim and proper
is an old Southern saying for
'You have got it together.'

Its all in the family

The 8051 family is the dominant core among all microcontrollers. Intel originally developed the architecture about 1979. Other vendors took the basic device and developed enhanced versions. Various reports are that versions of the device represent over half of all microcontrollers.

The standard package is a 40-pin, dual-in-line package (DIP) with 600-mil spacing between the rows of pins. It is without program memory and has only 128 bytes of data memory. Program and data memory are external chips. The most popular variation has program memory and data memory on board the processor chip.

In this family of computers, program memory is separate from data. This is called Harvard architecture and it is the design used by Intel related machines. Mixed memory is Von Neumann's Princeton architecture. It is the design used by Motorola related machines.

Reduced instruction set computers (RISC) have very few instructions but these can access any register in the computer.

Complex instruction set computers (CISC) have many instructions and each is used to access a limited register or memory. The 8051 has characteristics of both because of its memory and software strategy. However, it is primarily a CISC device.

On-board data memory

The on-board data memory is divided into three memory areas. First, there are 128 bytes of general purpose RAM. A significant portion of this can also be accessed as registers or as bits.

The next 128 bytes are segmented into 21 Special Function Registers (SFR). These bytes are the only ones accessible from the upper 128 locations on a standard machine. The twenty-one memory locations accessed as special registers cover many areas.

A & B accumulators
Timer/counter mode and control
Serial port mode and control
Stack pointer
Data pointer
Program status word for flags
Interrupt enable and priority
Parallel ports 0, 1, 2, and 3 for data I/O

Many machines now make an additional 128 bytes available as an extended internal RAM. Additional data memory is often available in the form of EEPROM or similar technology.

Arithmetic

Arithmetic functions are unsigned, binary integers. Familiar commands include add, add with carry, subtract, and subtract with borrow.

There are hardware multiply and divide instructions. The A register is operated on by the B register. The results are in A with B as supplemental.

Logical operations include AND, OR, and EXCLUSIVE OR. These operate on A with the results placed back into A.

Other packages

In addition to the 40-pin wide DIP package, there are other variations. Although there are surface mount designs, our discussion is oriented to packages that can be used for construction of a small number of units.

The programmable logic chip carrier (PLCC) is a square configuration that can fit in some spaces more effectively.

Projects that do not require external memory and that can be implemented with only two ports use a very cost and size effective arrangement. This is a 20-pin slim package.

Flash microcontroller

One of the variations of the 8051 which has almost everything but the kitchen sink is an Atmel design. Other than part of this section, the section on in-system programming, and the chapter on extended special function registers, all the remaining information in this book is generic and applicable to the entire family of microcontrollers.

Atmel's AT89S8252 flash microcontroller has several significant characteristics. The features are listed in the following table. The standard or core features are shown in the last column of the table.

Program memory locks

Some versions have three lock bits that can be left unprogrammed (U) or can be programmed (P) to obtain security and prevent further programming or fetching of the program. In prototyping, these are left unprogrammed.

The lock bits may be inadvertently set if the burner program is improperly configured. With some in-system programmers, the bits may also be inadvertently set.

Once programmed, the lock bits can only be unprogrammed with the Chip Erase operation in either the parallel or serial modes. Alternately, the chip can be placed in a burner and the bits cleared.

Features

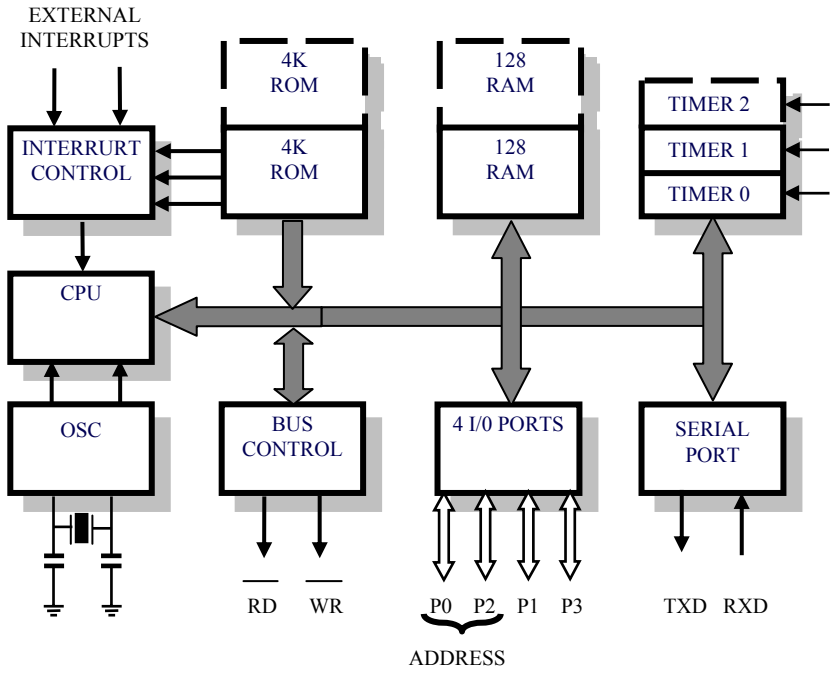
MCS51 Compatibility	**89s8252**	**8051**
Bytes of in-system memory	8k	none
Reprogrammable downloadable flash SPI serial interface for program downloading Endurance: 1,000 write/erase cycles		
Bytes EEPROM	2K	none
Endurance: 100,000 write/erase cycles		
Operating range: volts	4 - 6	4.9-5.1
Fully static operation frequency: MHz	0 - 24	3-50
Three-level program memory lock	yes	
Eight bit internal RAM: bytes	256	128
Programmable I/O lines	32	32
Sixteen bit timer/counters	3	2
Interrupt source addresses	6	5
Programmable UART serial channel	yes	yes
SPI serial interface	yes	none
Low-power idle and power-down modes	yes	none
Interrupt recovery from power-down	yes	none
Programmable watchdog timer	yes	none
Data pointer	two	one
Power-off flag	yes	none

The extended family

The data sheet for the device provides very substantial design information. Much of this is similar to generic controllers.

A primer provides all the details about how to use the features that are special to this machine. Both these data sheets are available from the manufacturers and on the web.

The diagram illustrates the majors features of the machine.



⇐ ⇑ ⇒

6

ADDRESS AND INTERFACE

Thought
Education:
Learn - Do -Teach

What is the connection _____

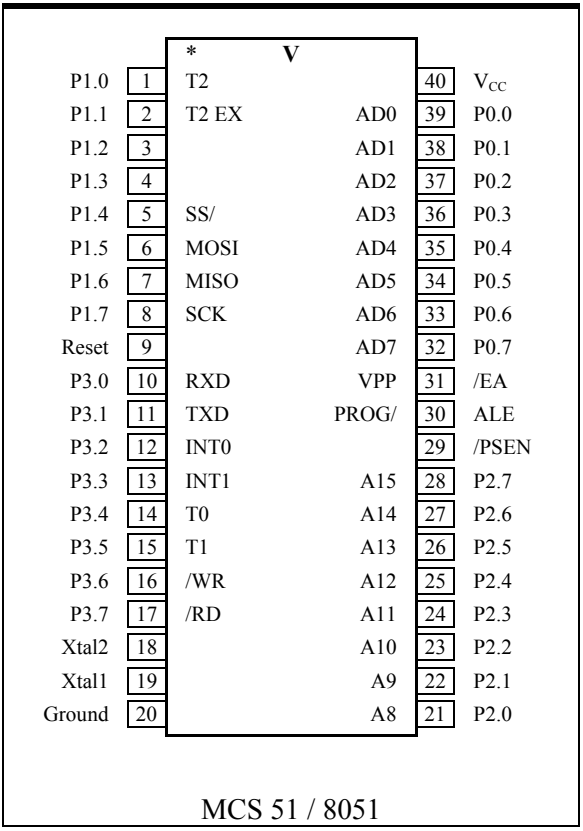
The standard 40-pin package is used for most discussions. It contains all the external connections that will be used on any variation of the microcontroller. Some designs do not need all the features. These will be smaller and will not include some pins.

Power _____

Before anything can work, power must be applied. The upper right (pin 40) is connected to V_{CC}, which is usually 5 volts. The lower left (pin 20) is connected to the ground reference.

Clock _____

The clock or crystal is connected to pins 18 (XTAL2) and 19 (XTAL1). This provides a frequency basis for all the micro operations and timing. Each cycle is divided into 12 phases which trigger internal circuits to create an instruction.



When using a crystal, a buffering capacitor of 10 to 40 pf is connected from each pin to ground. The crystal frequency ranges from 3 to over 50 MHz. However, the choice of frequency is often determined by outside connections. For example, serial communications requires rather unusual frequencies such as 11.059 MHz.

If an oscillator is used, XTAL1 is the input to the oscillator amplifier and the input to the internal clock. XTAL2 is the output from the inverting oscillator amplifier.

Reset

Reset is connected to pin 9. The machine runs when the line is low. Reset occurs if the pin is pulled high for two machine cycles. In the simplest realization, a switch performs this function. The switch should be open for normal operations and momentarily closed for reset.

The most elegant circuit occurs with power on. This circuit will automatically reset the computer whenever power is applied. The circuit consists of a 10 microfarad capacitor connected to Vcc and a 10 K Ohm resistor connected to ground.

When using in-system programming, the pin may be activated by a software command. A 1500 Ohm resistor should be connected between the cable and the reset pin.

Ports

Ports are parallel locations within the processor. Each port includes eight bits that are individually addressable. Three different notations are used to represent a bit on a port, as shown in the diagram. These include pin number, port number followed by the bit number, and function such as address / data.

Limit the number of devices and resulting current connected to any port. The ability to sink or source current is very restricted. A buffer such as a latch or 7406 is highly desirable to protect the port. Current limiting resistors should be used for LEDs connected directly to the port.

The maximum output currents that the chip can handle are shown in the table.

Condition	I_{OL}
Max per port pin	10 mA
Max for eight bit, port 0	26 mA
Max for eight bit, port 1,2,3	15 mA
Max total for all output pins	71 mA

To determine the number of devices that can be connected, it is necessary to identify the current required by each load.

Total source (I_{OH}) = input to load (I_{IH}) * number loads (m)
Total sink (I_{OL}) = input from load (I_{IL}) * number loads (n)

Use lower power devices such as ALS or CMOS technology.

Port 0

Port 0 consists of eight pins (pins 32-39). As a true bi-directional I/O port, it can be used for normal input and output. As an output, it can sink eight TTL loads.

When 1s are written to the pins, they become high-impedance inputs. Therefore, it requires a pull-up or pull-down resistor to operate. Because of other features, it is probably the least used port in normal programming. However, when used as a memory connection, the port has internal pull-up resistors.

Port 2

Port 2 is another 8-pin I/O connection (pins 21-28). It is less frequently used in normal programming.

It is quasi bi-directional, has internal pull-ups, and can sink/source four TTL devices. When 1s are written to the pins, they are pulled high by the internal pull-ups and the pin can be used as an input. As inputs, pins that are externally being pulled low will source current because of the internal pull-ups.

External memory

If external data or program is required, then port 0 takes on a multiplexing task. It has eight bits of the address impressed on the port. Then this is followed by the eight bits of data associated with that address.

A latch is used to trap the address. An eight bit flip/flop such as a 74573 is commonly used. It contains the 8 bus latches connected to the Data (D) or input pins of the latch. The output or Q pins are connected to the low order address pins of the memory device.

Two pins control the data capture and transfer. The chip or latch enable (pin 11) is asserted high, the gates receive and pass information. When the pin is asserted low, the flip/flops trap and hold the information. The output enable not (pin 10) is hardwired low to permanently activate the output.

When the 8 address bits are on the port, the Address Latch Enable line (pin 30) is toggled. This pin is connected to the chip enable pin of the external latch. Therefore, when the address is superimposed, the latch will capture and hold the location.

External program

If external program memory is employed with the microprocessor, then the External Address (pin 31) must be pulled low. This will transfer all program code from external memory.

If there is both internal and external program memory, the internal memory will be accessed when the /EA line is high or not connected. The next address after the top of the internal memory, will go to the external, regardless of the status of /EA.

When a code byte is requested from the program memory, the program storage enable not (/PSEN) is pulled low by the processor. The code will enter on port 0 pins.

The /PSEN line is connected to the external programmable read only memory (PROM) device output enable not (/OE) pin 22. The PROM chip select not (/CS) is connected to ground, if it is the only device. If there are multiple devices, the chip select must be decoded perhaps through a programmable logic device (PLD).

Do not allow input address pins on any memory device to float. Pull the pin low if it is not connected to the processor.

External data

External data memory is in the form of static ram (SRAM), electrically erasable prom (EEPROM) or similar devices. If these are in a DIP package the pins will correspond with those of program memory.

However, the connections to the processor are very different. Data is transferred from the processor to memory when the write not (/WR) pin 16 is pulled low by the processor. The pin is connected to the write enable not (/WE) pin 27 on the memory device.

Data is read from external memory when the read not (/RD) pin 17 is pulled asserted low by the processor. This is connected to the read not (/RD) or output enable not (/OE) pin 22 on the memory device.

External 64K

The eight bits on port 0 can address a maximum of 256 bytes of memory.

$$\text{Address} = 2^8 - 1 = 255$$

By using port 2 as the upper eight bits of an address, 64K of external memory can be accessed.

$$\text{Address} = 2^{16} - 1 = 65{,}535$$

Port 2 is not multiplexed. Therefore, an address latch is not required. The lines of port 2 are wired directly to the corresponding high bit address lines of the memory device.

Since the port is bit accessible, any bits not used for addressing are available for use as general purpose I/O. Often the upper bit (A15) is used to select memory or memory-mapped input/output.

Port 1

Port 1 is the most used general purpose I/O (pins 1-8) on the processor. The internal pull-ups operate like port 2. Most devices are connected through these pins.

If the processor has a serial peripheral interface (SPI), it is connected to the upper bits of port 1 (pins 5-8). These pins are connected to external chips. If the SPI is used for in-system programming, these pins are connected to an external cable.

Port 3

Port 3 is truly a multifunction byte (pins 11-17). The internal pull-ups operate like port 2. Pins not used for other functions can be used for general purpose I/O. The pins provide input for timers and interrupts as well as a bi-directional serial UART. Two pins are shared with the read not and write not functions for external data memory.

7

MINIMUM SYSTEM

Thought
Elegantly simple

Minimalist

The minimum system for a computer is the ability to do input, decisions, and output based on the decision. This will provide an outline for every other project that can be built.

Although the software code is very simple for this type project, the implementation will be in a structure that can support very complex problems. As systems grow, more interaction is needed. A later software illustration identifies issues of compatibility, particularly for variable descriptions.

The hardware required covers several issues. First, appropriate power must be supplied to ensure operation in range. Second, the fundamental computer is wired. This includes the processor, crystal, and reset circuit. Third, a method of transferring the code to memory is required. In this implementation, in-system programming is used. Finally, connections to the external world are made in the form of switches for inputs and lamps for outputs.

When building a circuit for digital processing, several practices will provide results that are more consistent. Lay out the components adjacent to other components that will be interconnected. Use short

wires. Red should be used for power. Black is preferred for ground. A variety of other colors for inter-connections will help construction. Keep the board neat, since this will aid troubleshooting and improve your attitude about the project.

Project

The project is to apply one switch and two LED's to the computer. One LED is on while the other is off. Change the state of the switch and the LEDs will change state.

Minimum software

On reset, program control transfers to address 0000h. The number of digits in the address is based on the maximum amount of memory that is directly accessible. There are sixteen bits for addressing. A hexadecimal number is made up of four bits. Sixteen bits then would be divided into four hexadecimal numbers.

Addresses 0000 – 0002h are the power on reset vector and are reserved to send the program to another location. Addresses 0003 – 002Ah are reserved for interrupt processing in a standard processor. Many enhanced processors reserve the next 8 bytes for expansion interrupts. Additional explanation is given in the architecture section covering memory organization.

Therefore, the lowest code address should be 0033h. However, programmer information and ownership is often embedded in the next code addresses. For that reason, the first address used for program code is usually 0080h.

For normal procedures, the first command at address 0000h is a jump to address 0080h. Several jumps are available. These include absolute short, absolute long, and relative. Any of these can be used for the first command in the computer.

Opcodes, mnemonics, comments ___

The most basic level of programming is using machine language and op codes. This can be done for relatively small routines and for practice in debugging. However, it is not time effective for any program of significance. A later chapter introduces higher level language techniques. The opcode list is provided in the reference chapter on instructions set.

When coding, it is very desirable to use precise spacing for the instructions. This makes the program easier to read and keeps all the information about each instruction on one line of a standard size paper. The comments will be truncated on the narrow paper of this book.

```
;add OPCODES   LABEL:    MNEM  REGISTER       ;COM

0000 020080              ljmp  INITIAL        ;go
0080 D2B5      INITIAL:  setb  0B5h           ;on
```

This is the complete program necessary to turn on an LED connected to port 3, bit number 5.

The address is a hexadecimal value from 0000 – FFFFh. Each program location represents an eight bit byte of data.

The op codes are hexadecimal values that are decoded and processed by the internal central processing unit. A series of micro operations are executed to create the correct response. Mnemonics are used for convenience in recognizing the instruction. The processor only understands the op codes.

In general, an instruction gets data from a source register, operates on the data, and places the results at a destination register. By convention on this particular machine, the destination is noted first. A comma separator is used before the source value.

Classes of instructions

Only three classes of instruction are available to a computer. These are data transfer, mathematics manipulation, and program control. Any instruction is a variation of these classes. In only a few cases, the instruction may be a combination.

Data transfer instructions change the location of data. The major instruction is a move (mov) from a source to a destination. In keeping with the leaning toward a RISC machine, this is the only transfer instruction. Other computers use many instructions to accomplish this same task, including load, store, and transfer.

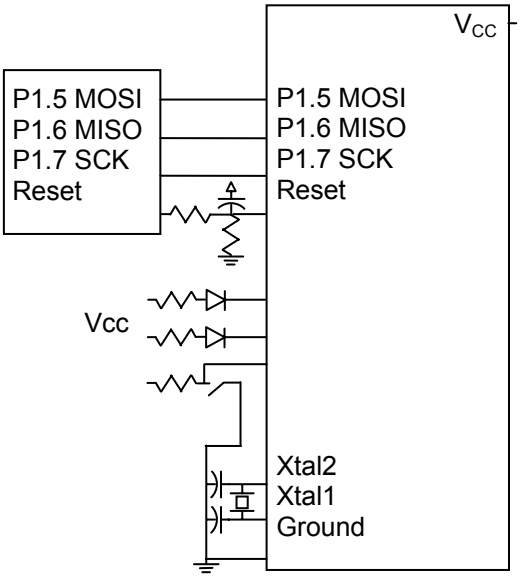
Mathematics manipulation instructions include arithmetic operations such as add, subtract, multiply and divide. More limited operations are increment and decrement. Logical operations are AND, OR, and EXCLUSIVE-OR as well as clear, set, and rotate. Again, the instruction set is reduced by not having a shift command.

Program control instructions change the value of the program counter under certain conditions. Some of these instructions are jumps, calls, and returns.

Schematic

A project should be illustrated with a schematic. Numerous programs can be used to create the drawing. The preferred systems will provide a circuit that can be transferred to a printed circuit board design if desired.

The illustration should show the circuit so that another builder can construct it. This particular example also shows the in-system programming connections, although that is not normally shown. It is provided so the novice can start from scratch and build it.


V_{CC}
P1.5 MOSI
P1.6 MISO
P1.7 SCK
Reset
P1.5 MOSI
P1.6 MISO
P1.7 SCK
Reset
Vcc
Xtal2
Xtal1
Ground

⇐ ⇑ ⇒

8

MACHINE CYCLE TIME

Thought
Engineers do not run the world,
But make the world run.

First computer circuit _____

A microprocessor is the engine of a microcomputer system. Hardware refers to the physical, electronic components. Software includes the application programs that provide user functions. Firmware is the program that is resident with the computer and is not easily changed.

The microprocessor firmware is a basic system program that is usually stored in a ROM. This system program is intended to do basic input / output (I/O). The system program also includes a simple power-on reset routine to run the system when it is started. On a personal computer system, this program is usually called ROM BIOS (Basic Input/Output System).

A microcontroller has the program as an integral part of the system. This makes the program look like both firmware and software. The program memory may be internal or external to the computer chip.

If the program memory is external, an address latch is needed. Since the data and address lines are time-multiplexed, a latch is used to

hold the address while the data/program is fetched from the ROM. If the program memory is internal, then the system is very simple. In the latter case, a connection for programming the in-system memory is needed.

Cycle time

In its simplest implementation, any computer program can be used as a timer. First, determine the speed of each instruction, then add all the instruction cycles.

Each instruction in a computer requires a definite amount of time to execute. A machine cycle is the internal count for executing an instruction. This is based on the oscillator frequency.

The fixed rate of an instruction time delay is calculated as follows. Divide the number of states for the processor by the crystal frequency. There are twelve states.

Time = (sec/11.059E6 cycle)(12 period/mach cy)
= 1.08509 E-6 sec / machine cycle

To obtain a time delay for an instruction, multiply the time per machine cycle by the number of cycles for the instruction. Repeat for all the instructions in the routine, to obtain the total time delay.

A delay with two nested loops is shown. Registers R2 & R3 are used for counting the number of times through the loops. A smaller number on the inside loop gives more precise counting.

```
          mov   R3,#02H       ;Outer loop counter
ZDEL2:    mov   R2,#01H       ;Nested loop counter
ZDEL1:    nop                 ;Delay
          djnz  R2,ZDEL1      ;Nested loop, 256 x
          djnz  R3,ZDEL2      ;Outer loop, 256 x
```

The maximum count is obtained by preloading the registers with zero (0). The first pass through the djnz instruction will decrement before it tests for the jump. Decrementing from zero gives 0FFh.

To obtain less time, remove the outer loop that includes register R3. To obtain greater time, add another loop outside the shown code.

Machine cycles

The machine cycles for each instruction are gathered from the instruction table toward the back of the book.

```
mov     = 1 cycle
nop     = 1 cycle
djnz    = 2 cycle
ret     = 2 cycle
```

The time for each loop is calculated by counting the total number of cycles in the loop. The inner loop includes just three instructions.

```
mov     = 1
nop     = 1* loop count
djnz    = 2* loop count
```

The total cycles depend on going through the inner loop as many times as the outer loop counter.

```
Inner loop  = (3 * inner count) + 1
Outer loop  = [(inner loop + 2) * outer count] + 1
Plus        = 4 for call & return
Total cycles
```

Determine the total cycles for the example given above. The inner count is one (***1***), while the outer loop is two (***2***).

```
Inner loop  = (3 * 1) + 1           = 4
Outer loop  = [(4 + 2) * 2] + 1     = 17
Plus        = 4 for call & return   = 4
```

Total cycles= 25

The elapsed time is the total cycles multiplied by the time for each cycle.

Time delay = (1.08509 E-6 sec / machine cycle) * 21
= 27.28 microseconds

Long precise wait

Some routines for display require a longer wait string than typically used. These can be developed by multiple calls to a precise delay. One-tenth second is a good base counter.

```
;-------------------------------------------------
WAIT:
;-------------------------------------------------
;  Create a 0.1 second wait.
;  Count = 0.1/1e-6 = 100000
;  Loop = 192 = C0H

          mov     R3,#0C0H    ;Outer loop counter
ZDEL1:    mov     R2,#00H     ;Nested loop counter
ZDEL2:    nop                 ;Delay
          djnz    R2,ZDEL2    ;Nested loop, 256 x
          djnz    R3,ZDEL1    ;Outer loop, 192 x

          ret                 ;Return to call
```

In from out

The four ports are four bytes of the special function registers. These bytes are used as the input and output connections to the computer. Each of the ports is bit accessible. The operation of the ports was explained in an earlier chapter. This discussion is how to program the ports.

The output is quite simple. Place the value in the register that represents the port. This triggers a latch, which holds the value on

the external pins. A simple mov to the location will implement the transfer of data to the external pins. No other action is required.

An input from a pin requires a two-step process. First, the latch must be pulled high by sending a one (1) to the location. This is exactly like an output. Then the value on the pin is transferred to port register memory. In other words, if the pin is pulled low externally, then the register bit will be a zero (0).

Ports 1, 2, and 3 have an internal pull-up triggered by the FET. So, no external resistor is required. port 0 has a high impedance FET. Therefore, an external resistor must be used to pull the pins high or low.

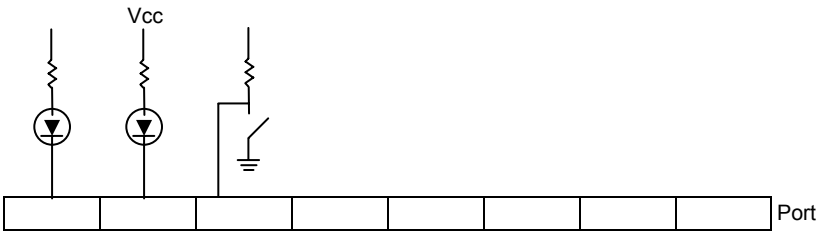
```
                        ;ILLUSTRATE OUT/IN
mov   90h,#0FFh         ;set port1 for input
mov   A,90h             ;read port status
mov   0B0h,#01          ;output port3,bit 1
```

Switch a bit

Port pins can be manipulated as bits as well as bytes. This permits a pin to be used as a switch. A single bit is set for the input. Then the bit is tested for high or low.

```
                        ;ILLUSTRATE OUT/IN
setb  0B0h              ;set port3 for input
jb    0B0h,EXIT         ;bit=1,then go end
                        ;else do this instr.
```

Circuit: led metronome _____



⇐⇑⇒

9

PROJECT 1 – OUTPUT & TIME DELAY

Thought
You see what you seek.
Yakov Smirnoff

Project 1: Metronome _____

Purpose: To build a basic computer system.
To use a microcontroller with program memory.
To write a simple program.
To use the output port.

Preamble:

The microcontroller has I/O ports, which are built-in. In order to transfer a logic value to output ports, treat the ports as memory locations. Only the mov command is needed.

Although the processor uses machine code to execute, the information is converted to Intel Hex format for sending to the memory burner. This is simply placing the addressing on the front of each string with a checksum inserted at the end.

To make a light flash, require the computer to place a signal on the output. Then the computer has an elapsed time delay by counting instructions. Next the processor sends a signal to stop the output.

This makes one flash. The cycle is repeated for multiple flashes. A cpl command is easily used to toggle a bit.

Plan:

For the project, build a basic microcomputer system with memory for the program. If the chip has on-board memory, the program may be stored there. Then, the microcontroller will be used to implement a digital metronome.

Preparation:

Supply a crystal to the microcontroller. The chip already has an on board oscillator circuit. Observe the circuit for a complete board shown in a later chapter. The circuit contains more items than are required. For this project, neglect unnecessary peripherals.

Simply wire the microcontroller, crystal, and power, if using a machine with on-board memory. Select a crystal frequency that is compatible with serial communications. A frequency of 11.059 MHz is a very popular selection. The finished wired circuit is a complete microcomputer system. Add the power-on-reset circuit to make start-up easier.

Procedure:

Check the system that has just been built. Implement a metronome on the computer. Since ports 0 and 2 are used for memory addressing, ports 1 and 3 are free for direct input/output. Put an LED on one of the free I/O port bits. The LED must blink at a specific rate. The cycling output implementation is a metronome. Writing a 1 and 0 sequentially to a bit of the output port is the actual task.

Test a switch to start or stop the display.

Presentation:

Implement the program using only machine code without any assembler or high-level compiler. Developing this ability can aid in debugging future programs.

Demonstrate your circuit. Write an additional short note about your delay time. The delay duration should be equal to 10 divided by the number of characters in your first name. This note should follow the program listings.

Program sample example _____

The following code contains a delay subroutine. The project is implemented by modifying the values that go into R2 and R3, the loop counters, to fit the delay requirements.

Write the program with mnemonics and comments. Then translate to the address and machine code. The op codes and the number of bytes are obtained from the instruction tables in the reference section. This information will then be placed in a file to be programmed into the code memory.

The file information can be placed in an eprom burner program editor. The burner program will convert the output to the appropriate format for loading in the microprocessor program memory.

```
ADDR OP CODE  MNEMONICS & COMMENTS

              ;-----------------------------
              ;Program: MODDelay.asm
              ;Update:  29 January 2003
              ;Initial: 17 October 1991
              ;
```

```
                ;By:  Dr. Marcus O. Durham, PhD, PE
                ;     Tulsa, OK, USA
                ;     mod@superb.org
                ;     www.ThewayCorp.com
                ;     Copyright (c)1991, 2002.
                ;     All rights reserved
                ;
                ;Purpose:
                ;  A routine to demonstrate software
                ;  delay. One LED will come on.
                ;  After a delay, another will.
                ;
                ;Processor: 8031 family
                ;PROM:      8k (2000H) onboard
                ;Crystal:   11.059 MHz
                ;##############################
                ;         PROGRAM
                ;##############################
0000                    org   00H
0000 020080     START:  ljmp  INITIAL

0033                    org   0033H
0033 4D617263       db  'Marcus O. Durham, PhD, PE'
0037 7573204F
003B 2E204475
003F 7268616D
0043 2C205068
0047 442C2050
004B 45

                ;------------------------------
0080                    org   0080H
                INITIAL:
                MAIN:
                ;------------------------------
0080 D2B4               setb  0B4h        ;turn on
0082 12008A             lcall DELAY       ;wait
0085 D2B5               setb  0B5h        ;turn on

0087 020080     MAN9:   ljmp  MAIN        ;Repeat

                ;------------------------------
                DELAY:
                ;------------------------------
```

```
008A 7B00              mov   R3,#00H    ;Outer
008C 7A00     ZDEL2:   mov   R2,#00H    ;Nested
008E 00       ZDEL1:   nop              ;Delay
008F DAFD              djnz  R2,ZDEL1   ;Nested
0091 DBF9              djnz  R3,ZDEL2   ;Outer
0093 22                ret              ;go back

                       end
```

10

SOFTWARE DEVELOPMENT

Thought
God is in the details.

The here and now _____

Earlier discussions introduced software in terms of machine language and op codes. Although that is great for understanding how the machine works, it is atrocious for effective use of time with large projects.

There are three fundamental language levels of programming. Machine language is the most basic. It involves use of zeroes and ones to form op codes. The op codes are used directly by the computer.

The intermediate level is assembly language, which uses symbols and mnemonics to represent the basic op codes. This includes Assembler. Although it has many capabilities and is somewhat transportable, "C" is very similar to assembly because of its terseness and difficulty in comprehending the meaning of the code.

High level languages are the most sophisticated and involve conceptual statements similar to a spoken language. This includes Basic and very advanced languages. Because of its advanced features, most authors include "C" as a high level language. Both

Basic and "C" were developed in conjunction with operating systems. Basic is to DOS as "C" is to Unix.

The software languages commonly used for programming the microcontroller family are assembly, Basic, or C+. By far, the most frequent is assembly language. Regardless of the source code, it must be converted to an Intel Hex format for programming the memory.

There is great flexibility in how program code is structured. There is at least as one more way than the number of programmers. Therefore, there is no 'right' way. Nevertheless, there are formats that provide certain advantages.

The development of effective software using an assembler is discussed in the next sections.

1. First comes the formatting of instructions.
2. Next is the description of directives.
3. A step-by-step process which goes from text to hex files follows.
4. Then a program shows the source, assembler listing, and hex code.
5. This is followed by practical suggestions for commenting.
6. Finally, structured programming is argued.

Substantially more information and illustrations are provided in the extreme program harmonization example.

Instructions

The source program can be created using any word processor or editor. Save the program unformatted in a flat text file. The extension for the source depends on the particular software package. *.txt and *.asm are the most common.

An earlier illustration is shown as an introduction to using an assembler. This is a working program. It is the result of a listing after the assembler completes working on the program.

```
ADDR OPCODES   LABEL:     MNEM  REGISTER        ;COM
0000 020080                ljmp  INITIAL         ;go
0080 D2B5      INITIAL:    setb  0B5h            ;on
```

The source code consists of the information on each line that starts with the label. The assembler adds the address and op codes to each line.

Label is an identifier for the address of that line of code. It is restricted to 8 characters. Some assemblers are case sensitive; therefore, it is a good idea to use the same capitalization in all references. The label is followed by a colon (:). However, the reference to the label does not have a colon. The assembler simply translates the label to an address.

Comments must begin with a semicolon (;). Everything on a line after the semicolon is ignored by the assembler. Neither does the computer recognize or use the comments. Comments are for the programmer. Comments should explain the purpose for the instruction.

Mnemonics are abbreviations for an instruction. These are for convenience rather than memorizing numbers for each instruction. These are converted to op codes by the assembler. The op codes are hexadecimal values that are decoded and processed by the internal central processing unit.

The registers are listed as the destination with a comma separator before the source. These registers are the information that is manipulated by the instruction.

Names or variables starting with a letter are translated as a value. Numbers must begin with 0 – 9. A hexadecimal number that begins with a letter must be preceded by a 0. An example is 0Fh for the number 15.

Numbers are identified by their base. Decimal numbers are followed by a “d” or nothing. Hex numbers are followed by a “h”. Binary numbers are followed by “b”.

When coding, it is very desirable to use precise spacing for the instructions. The table lists the various components of a line of code. It shows the preferred column for starting that code. This will make the program easier to read and will assure all the information fits on one page after the assembler processes the information.

Name	Column	Length
Label:	1	8 + colon
Mnemonic	11	5
Register	17	14
Comment	31	19 max
Heading comment	1	50

Assembler directives

In addition to program code, the assembler must be informed about certain operations. The directives are not comprehended by or translated to the microcomputer.

Each assembler has it own format for the directives. There is no standard among the various vendors. The ones illustrated are based on the original Intel ASM51 assembler.

The first directive is org. This organizes the program to a specific place in memory. The instruction simply gives the address for the next instruction.

The second is equ. The equate directive provides a numeric equivalence for a variable. This can simply be a number or a place in memory.

The third is db and its related types. This defines byte values in code memory. It often involves table values and ASCII messages. Include a line with a copyright message. If it is not copyrighted, use

'author', then include your name. That provides a permanent record of who did the programming.

Every program must terminate with the directive end.

Several assemblers are available for the 8051. The major differences are in the syntax of the directives. Three commonly used formats are compared.

TASM.EXE is a table lookup. It is accessed by the instruction:

tasm -51 -p -l Program2.asm

```
          .org  0050h          ;Next inst @ 0050H
Value     .equ  12h            ;predefined value
Table     .byte 34h            ;define code byte
          .text "123"          ;define ASCII
          .end                 ;last thing
```

A51.EXE is the student version of PseudoCodes. It is accessed by the instruction:

A51 -s Program2.asm

```
          .org  0050h          ;Next inst @ 0050H
          .equ  label,12h
 Table:   .db   34h            ;define code byte
          .db   "123"          ;define ASCII
```

ASM51.EXE is the Intel assembler with library. It is the original assembler. It is very powerful. Unfortunately it is DOS based and has not been maintained. It is accessed by the instruction:

ASM51 Program2.asm

```
          org   0050h          ;Next inst @ 0050H
Value     equ   12h            ;predefined value
Table:    db    34h, '123'     ;define code storage
          end                  ;last thing
```

The major directives are listed in the table. Each assembler has a different set, but these are the most common. There is an overlap

between assemblers. In addition there is an overlap in usage. As illustrated above, the combination of equ and db will perform all the other functions.

Dir	Purpose
org	Organize the program memory location
end	End of the program for the assembler
equ	Equate a variable to a value
db	Define a byte of program memory to a value
dw	Define two bytes of program memory to a value
ds	Reserve a byte of program memory for a value
bit	Equate a variable to a bit address
byte	Define a byte of program memory to a value
data	Equate a variable string to a number
text	Equate a variable string to an ASCII value

Step by step

The program can be copied to or created in a text file. Then assemble the file. This example has been successfully assembled using Intel ASM51. The directives may need changing if using an alternate assembler. If the assembler does not output a *.hex file, convert the *.obj to an Intel Hex format.

Download the program in Intel Hex format to the program (code) memory. Download can be either using a burner or the in-system programming pins. When the processor is reset, the program will run.

Program with comments

```
;-------------------------------------------------
;Program: MODIo.asm
;Update:  07 February 2003
;Initial: 01 December 1990
;
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
```

```
;         www.ThewayCorp.com
;Copyright (c)1990, 2003.  All rights reserved
;
;Purpose:
;  The LED on P3.4 will come on.
;  The LED on P3.5 will go off.
;  The LEDs will change state if P3.3 is at ground

;  The program is very simple. Nevertheless,
;  structured programming is used for illustration
;
;Processor: 8051 family
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz

;Assembler: Intel ASM51.exe
;           ASM51 MODIo.asm


;################################################
;
;                     ASSIGNMENTS
;
;################################################

                     ;PORT USE
P33       equ   0B3H  ;Port3.3, switch input
P34       equ   0B4H  ;Port3.4, LED
P35       equ   0B5H  ;Port3.5, LED flashing


;################################################
;
;                     PROGRAM
;
;################################################
START:
;------------------------------------------------
;  When processor is reset, program control comes
;  here. Jump to the first executable address
;  after all interrupts reserved locations, etc.

          org   00H
          ljmp  INITIAL
```

```
          org   33H
          db    'Marcus O. Durham, PhD, PE'

;-------------------------------------------------
          org   80H
INITIAL:
;-------------------------------------------------

;  Set the initial conditions. Output to LEDs and
;  make P33 an input.

          setb  P34           ;turn on LED P34
          clr   P35           ;turn off LED P35
          setb  P33           ;make bit an input

;-------------------------------------------------
MAIN:
;-------------------------------------------------

;  The main process loop will test the limit.
;  If switch open, do nothing.
;  If closed, change states.
;  Since there is no delay & no debounce,
;  the output is sensitive to when the switch is
;  released.

          jb    P33,MAIN9     ;skip if off
          cpl   P34
          cpl   P35

MAIN9:    sjmp  MAIN          ;play it again, Sam

;*************************************************
          end
```

Listing

After the assembler runs, it produces several files. Perhaps one of the most useful for debugging coding errors is the listing (*.LST). This includes each line of the original code. The line is shifted to the right. It is preceded by two very important pieces of information.

The first information is the memory location of the first byte of the instruction. The next information is the machine code translation of the instruction.

After the program, the listing shows the value of every variable in the code.

An edited version of the listing is shown to save space and fit within the page size. Comments have been truncated in addition to removal of header information.

```
MCS-51 MACRO ASSEMBLER    MODIO 01/28/:3   PAGE 2
LOC  OBJ      LINE SOURCE

0000          52            org   00H
0000 020080   53            ljmp  INITIAL
              54
              55  ;-----------------------
              56  ;AUTHOR
              57  ;-----------------------
0033          58            org   33H
0033 4D617263 59  db   'Marcus O. Durham, PhD, PE'
0037 7573204F
003B 2E204475
003F 7268616D
0043 2C205068
0047 442C2050
004B 45
              60
              61  ;--------------------------------
0080          62            org   80H
              63  INITIAL:
              64  ;--------------------------------
              65
0080 D2B4     69            setb  P34
0082 C2B5     70            clr   P35
0084 D2B3     71            setb  P33
              72
              73  ;--------------------------------
              74  MAIN:
```

```
                75  ;------------------------------
                83
0086 20B304     84              jb    P33,MAIN9
0089 B2B4       85              cpl   P34
008B B2B5       86              cpl   P35
                87
008D 80F7       88  MAIN9:     sjmp  MAIN
                89
                90  ;******************************
                91              end

SYMBOL TABLE LISTING
------ ----- -------

N A M E   T Y P E  V A L U E   A T T R I B U T E S

INITIAL.  C ADDR   0080H   A
MAIN . .  C ADDR   0086H   A
MAIN9. .  C ADDR   008DH   A
P33. . .    NUMB   00B3H   A
P34. . .    NUMB   00B4H   A
P35. . .    NUMB   00B5H   A
START. .  C ADDR   0000H   A

REGISTER BANK(S) USED: 0

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Intel hex

Data needs to be transferred between the assembled program and the code memory. The machine code is converted to Intel hex format for loading into program memory.

The file is shown in a table simply for explanation. It has been divided into columns to indicate different parts of each data line.

Each line begins with a colon (:). This is followed by the number of bytes on the line, the address of the first byte, a directive, then the machine instructions in that block. The line concludes with the checksum for the line.

Column 1 - Byte, number of byte/addresses on the line
Column 2 - 2 Bytes, hexadecimal address in memory
Column 3 - Byte, 00 if data, 01 if end-of-file
Column 4 - 16 Bytes, op code data for each address
Column 5 - Byte, checksum

:	#	Addr	Dir	Op Codes Data	√
:	03	0000	00	020080	7B
:	10	0033	00	4D6172637573204F2E2044757268616D	34
:	09	0043	00	2C205068442C205045	8B
:	0F	0080	00	D2B4C2B5D2B320B404B2B5B2B680F7	--
:	00	0000	01	FF	

The checksum is calculated from the data. Its purpose is to check data transfers for loss of bits. The checksum is the two's complement of all the other bytes on the line. It is calculated in the following method.

1. Add all the bytes on the line.
2. Subtract the results from FF.
3. Add one to the result.
4. This is the checksum.

Commentary

Good software development is a skill, which is often neglected, and which is best learned by example. Suggestions for good documentation habits include comments, placement of code information, and structure.

The first item that should be used is a header which gives the name of the program and who wrote the document. Remember, there are laws against plagiarism. The use of a version number of a program (e.g. V1.3, V2.2) is strongly recommended. Update or increase the number of the version every time you edit a file. This helps to keep track of each change made to the files.

Following the header, there should be a list of major updates to the program. Especially, note updates after release to the public.

The next major item that should follow is a short description of the program. This should be long enough to give the user a general idea of what the program does. The user should be able to use the software without having to figure it out the hard way. Avoid going into endless detail. Depending on the program, the description may be anywhere from 2-3 paragraphs to about one page. A page and a half may be required for BIG programs.

Any quirks of the program and any hardware dependencies about which the user should be aware need to be included here. Examples are software that supports a UART needing an 11.059 MHz crystal.

Assume that the user does not have your hardware available, but that (s)he is intelligent enough to put together a system to run your software. Therefore, state such things as where to hook-up port-pins and what parts of the software to modify if a different crystal is used. Similarly, the documentation should be good enough that the user can modify the code or add to the program. (S)he should not have to spend weeks trying to figure out whether the additions/changes will affect the old code.

The top placement _____

Place all the equate statements in one place such as at the top of the file. Typical equates include ASCII codes, RAM locations, and external locations in the memory map. This item can save you and the user substantial time when digging for a particular item. Remember to update these addresses when you change locations.

Often the equate list can save substantial amounts of time when determining which items may adversely affect the program. This is especially important if you are writing only one module of a large program.

Document any test pin or port bit assignments. Note the uses for all the registers dedicated to only one or two uses. If a register is used as a general-purpose register, it can be so stated.

The subs

Documenting subroutines is another practice that will help writing compatible procedures. Each routine should have a header, which sets it apart from the rest of the program. The header should include the following items as well as the subroutine name. Make a brief description of what the routine does. List parameters that are passed to the routine via which registers or memory locations. Note what parameters are passed out of the routine and which subroutines are called. In the optimum case, mention which registers, flags and other storage variables are destroyed by the routine.

The destroyed variables may imply variables destroyed either by the routine itself, or by the routine and all subroutines it calls. Noting all these items requires a lot more work. However, it saves an incredible amount of time when debugging, since it becomes unnecessary to chase down each subroutine.

Your comments, please

Furthermore, document the code itself. This should be done as you write the program! It is not necessary to comment every line, but one comment every 3 lines is not enough either. Seldom is a string of code really that obvious.

Comments should describe the underlying process, not the actual command that is being executed. For example, the comment for an instruction like mov A, 45H should say why the value is being moved. Discuss things such as the value should be in a certain range.

If you do not write the comments until after you have finished writing the program, often you will have forgotten what a particular

line of code does. Alternatively, you will run out of time and skip the comments altogether. In that case, the client will not even bother to look at the program.

The bottom placement

Messages, tables and similar items, are often accessed by many different subroutines. These should be placed in one area such as the end of the file. If a message is only used by one subroutine, it is acceptable to put the message at the end of the subroutine. Again, consistency and find-ability are the goal.

Structure

Finally, it is worthwhile to stress the utility of structured programming and extreme programming. Look at the main program section of the code in the following pages. Notice the main program consists almost entirely of CALL statements. Therefore, the program is written in modular chunks.

First, this type of program is much easier to read and understand than pages and pages of straight-line code with no subroutines.

Second, the structure forces you to think about what you are trying to accomplish with the program. Hence, the program can be broken down into distinct sub-tasks, which can be more easily tackled. Often these subroutines can be used in several different parts of the program, thus saving ROM space. If subroutines are well written, they can also be used in other programs calling for similar functions.

Third, it is easier to make changes in a modular program since the effects of any modifications are more obvious.

Structured programming is logical. Its use reduces the likelihood of becoming lost in reams of straight-line code. Long code segments encourage forgetting what the program is trying to do. Structured

programming will not run quite as fast as straight-line code. Nevertheless, it has the advantages of readability and it encourages intelligent programming. The programs written for most projects will not be time constrained.

⇐⇑⇒

11

DESIGN PRACTICES

Thought
One small step for man
One giant leap for mankind.
Commander Neal Armstrong

Top down

Engineering tasks can be divided into three functional descriptions.

1. *Design* involves developing a procedure or technique to do a task that has not been done.

2. *Application* involves taking existing pieces or components and combining them to perform a task.

3. *Analysis* involves observing an existing system and determining how it works.

Since design must come first, it has top priority. However, it is useless if it cannot be applied.

To be effective, the design must first consist of an overview or big picture perspective. The broad project is then broken into individual subsystems, which represent major components of the project. The subsystems are loosely coupled. The subsystems are then segmented

into tasks that may be developed virtually independent of the other tasks. This process is called top down design.

The procedure is called structured programming when applied to computer code development. Niklaus Wirth developed Pascal as a high level language that forces structure. He gave a definition: "Structured programming is the formulation of programs as hierarchal, nested structures of statements and objects of computation."

Projects should use a top down approach to the interfacing and computer hardware as well as the program.

The project is the overall chore. It represents the complete program. The subsystems are tied together with a main or control program. It should primarily consist of calls to subroutines. An occasional decision may be required to determine which subroutine to call.

Each subroutine should be independent of other subroutines. Data arguments that must be transferred should be left in common registers that can be accessed by any routine. The registers should be defined prior to the main program. Then the main should call the subroutines.

Extreme programming (XP)

Extreme programming (XP) is a practice used by large teams of programmers. It is obviously based on top down design and structured practices. It is primarily a method of communicating with the design team. There are 12 core practices.

1. Customers define application features with user "stories".

2. XP teams put small code released into production early.

3. XP teams use a common system of names and descriptions.

4. Teams emphasize simply written, object-oriented code that meets requirements.

5. Designers write automated unit tests upfront and run them throughout the project.

6. XP teams frequently revise the overall code design, a process called refactoring.

7. Programmers work side by side in pairs, continually seeing and discussing each other's code.

8. All programmers have collective ownership of the code and the ability to change it.

9. XP teams integrate code and release it to repository every few hours and never hold on to it longer than a day.

10. Programmers work at a sustainable pace, with no extended overtime.

11. A customer representative remains onsite throughout the development project.

12. Programmers must follow a common coding standard so all the code in the system looks as if a single individual wrote it.

In keeping with extreme programming practices, a chapter on software compatibility is included. This lists variable names, locations, and formatting structure.

Steps for success

A consistent process is used in the development of any project. A seven-step pattern has been developed that works for software, hardware, and all other technical procedures. This is a universal model for engineering projects.

1. Entry
   The point where the procedure starts arises from a reset or from a decision.

2. Initialize
   A sequence of events or conditions are involved in the planning and set-up.

3. Input
   The loop first must gather data from other sources.

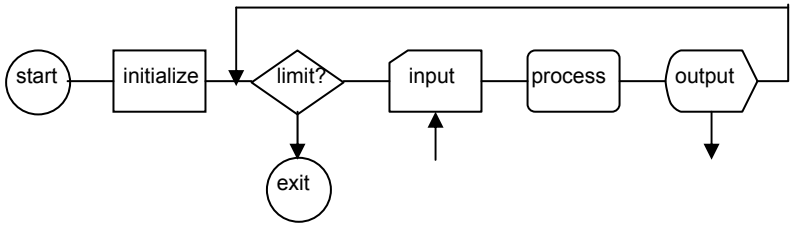
4. Process
   The core of the loop is manipulation of the data until a final value is obtained.

5. Output
   The defined results are displayed or become input to another procedure.

6. Limit
   A test or decision is performed to determine if the loop is completed. The test may occur at three different places.
   a. The test may be implemented before the input if there is a predefined loop range. This is a If-Then-Else or For-Next structure.
   b. A decision is made after the input based on the input. This is a data comparison structure.
   c. A decision is made after the process is complete. This is a Do-While some condition exists.

7. Exit
   The point to leave the project causes transfer to another procedure or task.

Process diagram



⇐ ⇑ ⇒

SECTION II - SYSTEMS

⇐ ⇑ ⇒

12

SWITCH, LOGIC, AND SUBS

Thought
People are where they are,
because of choices they make.
MOD

Switch hitter

A switch converts mechanical movement to electrical signals. The normal circuit for wiring switches was offered in the chapter on fundamental circuits.

One of the problems with mechanical switches is contact bounce. When the switch is depressed, the contact will vibrate for some time. Because of its speed, the microprocessor will count these as multiple switch closures.

Hardware devices can catch the switch as a single bounce. This could be a flip/flop or a one shot. However, software is much less expensive in terms of real estate. The traditional software scheme places a time delay on the switch input. Although this is effective, it delays the processing and sampling rate of the system. A better choice is to use a software solution that detects the first time the switch is stable.

Debounce

Using a logical exclusive-or provides a very clever switch debounce. The routine is entered multiple times until the last state of the input is equal to the present state. This provides response the first time the signal is stable.

```
;-------------------------------------------------
DEBOUNCE:
;-------------------------------------------------
                                ;CHECK CHANGE BY xrl
          mov   B,A             ;hold the input
          xrl   A,CharP         ;exclusive or
          jnz   DEBN1           ;<>0, so a change

                                ;KEEP DEBOUNCED
          mov   CharD,B         ;0=no change,debounc
DEBN1:    mov   CharP,B         ;not debounce
          ret
```

The routine compares bytes. The value enters in the A register. The previous value is kept in CharP. By using an EXCLUSIVE-OR, bits that have changed are easily found. If bits have not changed, keep the debounced value in CharD.

Bit manipulation

The ability to manipulate bit addresses is a powerful feature that separates this microprocessor from other computers. The internal memory is byte addressable for both the data RAM and the special function registers (SFR). Certain locations may be addressed as bits.

Any SFR whose address is divisible by 8 can be bit addressable. This includes any hexadecimal address that ends with a 0 or 8. The low order bit has the same address as the byte. For example, port 1 is located at address 90h. The least significant bit would also be bit 90h, while the most significant bit would be 97h.

In addition the internal data locations 20 – 2Fh (32 – 47d) are bit addressable. The bit address is sequential. Address 20h, bit 0 has a bit address of 00h. Bit 7 has an address of 07h. Address 2Fh, bit 7 has a bit address of 7Fh.

A number of instructions are available to manipulate the bits. The two instructions that change individual bits are setb to make a one and clr to make a zero.

```
                                ;bit MANIPULATION
CLEAR:    clr   93h             ;clear port1,bit3
SETCARRY: setb  C               ;1→Carry
```

Masking logic

Not all the values that exist in a byte are always necessary for evaluation. If only some of the bits are used, the others can be removed from the byte. The process of selectively modifying some of the bits is called masking.

A mask is a register that is similar to a template. It has a defined value that prevents other values from being observed. For example, a template or mask that is placed over a paper permits only a certain item to be painted.

Masking requires the use of logical operators. The three operators used with a template are AND, OR, and EXCLUSIVE-OR. The instructions are logical duals.

The logical AND (anl) of a bit with a 0 clears the bit, while using a 1 leaves the bit unchanged.

The logical OR (orl) of a bit with 1 sets the bit, while using a 0 leaves the bit unchanged.

The logical EXCLUSIVE-OR (xrl) of a bit with 1 inverts (complements) the bit, while using a 0 leaves the bit unchanged.

The instruction takes the value of the destination register, performs the logical operation with the source register, then places the results back in the destination.

```
                                    ;ILLUSTRATE LOGIC
ZERO:      anl   A,#11110111b  ;A bit3 becomes 0
ONE:       orl   A,#00001000b  ;A bit3 becomes 1
INVERT:    xrl   A,#00001000b  ;A bit3 complemented
```

Logical orl and anl should be used to selectively set bits on output ports. This will change the status of bits without impacting the other functions. If there are multiple functions on the port, avoid mov instructions.

Rotate and exchange

Data bits can be moved one position to an adjacent bit location. The data must be in the accumulator or A register.

If the register is rotated to the right, the least significant bit (LSB) is rotated to the most significant bit (MSB). Conversely, if the register is rotated to the left, the MSB is rotated to the LSB. No data is lost in the transaction.

The carry bit (C) can be considered to be bit 8. It is treated as if it were the next bit after the MSB. A rotate right with carry will cause the LSB to go to the C and the C will move to the MSB. A rotate left with carry will move the MSB to C and the C will go to the LSB.

The carry can be individually set or cleared from other operations. Therefore, it can be used to load values serially into a register.

The machine does not have a shift instruction since it has hardware multiply and divide. A shift is similar to a rotate, except a bit falls off the end with each shift. If a shift is required the rotate with carry can be used.

Several different commands are available for moving and changing the values of bits. In order to exchange the upper four bits (nibble) with the lower four bits (nibble) in a byte, the swap command is available.

A very powerful instruction permits the exchange of a byte between the Accumulator and a register or internal data location. The value in A goes to the register and the value in the register goes to A. This very quick operation replaces three mov instructions.

```
                                    ;ROTATE & EXCHANGE
RIGHT:     RRA    A                 ;bit 0→7, 7→6, 1→0
LEFT:      RLA    A                 ;bit 0→1, 1→2, 7→0
RC:        rrc    A                 ;bit 0→C, C→7, 1→0
LC:        rlc    A                 ;bit C→0, 0→1, 7→C
UPLOW:     swap   A                 ;0↔4,1↔5,2↔6,3↔7
EXCHANGE: xch    A,Hold            ;A↔Hold
```

Conditional branch

Often it is necessary to make a decision in a program. Three components are necessary for a decision: a value to evaluate, a reference value, and the result of the comparison.

The process is called a conditional branch. All these functions are included in a single instruction, compare and jump if not equal (cjne). The conditional branch is much like the IF-THEN-ELSE procedure.

The value to be evaluated is stored in a register. The register is either the accumulator, register zero or register one. The reference value for comparison may be an immediate number or the value may be in a RAM location. If the two values are equal, the next instruction is executed. If the two values do not match, the program counter is changed to the specified code address. Neither the evaluation nor the reference value is changed by the instruction.

The conditional branch is a single very efficient instruction. Most computers use a compare or test instruction, which sets a zero flag in the flag register. Then the next instruction does a branch based on the value of the flag.

However, since all decisions are essentially a comparison for zero results, the compare and branch are very effectively combined in one instruction.

Another conditional branch is the DO-WHILE type. The decrement and jump if not zero (djnz) instruction comes in this class.

A register is preset to the limit. It is decremented each time through the loop. When the register is cleared, the loop is exited.

```
                                ;CONDITIONAL BRANCH
BEFORE:                         ;an entry

COMPARE:  cjne  A,#02,BEFORE    ;A<>2, then BEFORE
DECREASE: djnz  D,BEFORE        ;D=D-1, D<>0,then BE
                                ;else, next command
```

Subroutines

Some routines of a program are used more than one time in a sequence. Since several instructions are involved in each routine, it is beneficial to reuse the same code if possible. In addition, it is easier to read a program that is segmented, rather than straight line code.

The instruction lcall invokes a subroutine any place in code memory. It uses a sixteen bit absolute address to reach the entire 64K available memory.

The main program calls a subroutine located at a label such as SUB. The program counter is pointing to the next instruction after the lcall. The value of the program counter is pushed onto the stack. Two bytes are stored. The program counter is then loaded with the address of the subroutine SUB.

A return, ret, instruction is used to indicate the end of a subroutine. Once the subroutine is completed, the ret pops the stack and then places that address into the program counter. This is the next instruction after the lcall.

```
                                ;SUBROUTINE
MAIN:     lcall SUB             ;SUB=label
ONE:

SUB:      mov   A,#00           ;do something
          ret                   ;go back to ONE
```

Only four bytes and four clock cycles are required for the two instructions that create a subroutine. The lcall uses 3 bytes and 2 clock cycles, while ret uses 1 byte and 2 clock cycles.

A structured program is realized by making the main program a sequence of call statements. If adequate room is provided, changes require only the addition or moving of a CALL.

Each subroutine should be stand-alone. All subroutines should be as consistent as possible with arguments passed and returned in common registers. The preferred registers are A or R0 – R7. This permits easy linking of multiple routines.

Stack

One of the unique features of the machine is a stack. A stack allows information to be pushed into a temporary location. The information is removed in the reverse sequence from the order it was filled. This is called a Last In, First Out (LIFO) sequence.

New data is added to memory on top of earlier data. The removal of a data item makes the item below it appear to be the new top of the stack.

A stack can be located anywhere in the internal data RAM. On reset, the stack pointer is preset to data address 07h. This should be changed since the stack will overwrite Bank 1.

Setting the address at which the stack is to begin initializes the stack. The stack pointer (SP) contains the address for the top of the stack. The stack grows up toward a higher address. The next byte is stored at the address above the value placed in the stack pointer register. Once the SP is initialized, it is seldom explicitly changed again.

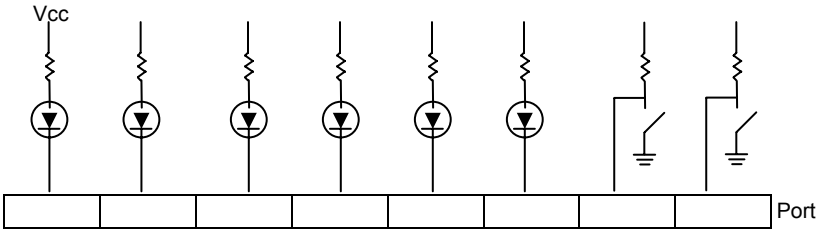
The subroutine instructions, lcall and ret, places and retrieves the two program counter bytes on the stack.

There are two instructions that move a byte between direct internal memory and the stack. push places the direct value into the next memory address above the location in the stack pointer. pop moves the value at the stack pointer location to the direct internal memory. Although other instructions may be between them, always pair stack operations.

```
                                ;STACK
INITIAL:  mov   SP,#5Fh         ;move stack to 5Fh
                                ;other commands
SUB:      push  Direct          ;direct to stack
                                ;other commands
          pop   Direct          ;stack to direct
```

The stack can use the entire internal data memory. The large stack makes the machine capable of pseudo-reverse polish operations. In many ways the machine appears to be a stack processor.

Circuit: led and switch _____



⇐⇑⇒

13

PROJECT 2 – INPUT & DECISIONS

Thought
Sell to people that are buying.
Jim Stovall

Project 2: T-bird taillights _____

Purpose: To learn the basic I/O system.
To implement a T-Bird tail light system.
To make use of subroutine calls.
To use an assembler.

Preamble:

A microprocessor has several advantages over a hardwired sequential logic circuit. This will become apparent in this project. Other implementations of this project have been with discrete logic and with programmable logic devices.

In order to communicate with the real world, a microprocessor must have an input-output system. This is called an I/O port. The computer treats the I/O port as a register and as internal data memory. Therefore, data can be written to the port without initialization. A one must be written to each bit before it is read.

The use of subroutine calls is preferable in large programs. Use of subroutines will make debugging the program much easier. Even in

a small size program, this method is still a winner when similar chunks of codes are repeated several times in the program.

Plan I: Basic input-output

Build a microcomputer system. Then, the computer will input bits from switches and output the bits to an LED.

Preparation I:

Connect the basic operating microcomputer, crystal and a power supply. Observe how switches must be connected between power, ground, and the computer port. Review the logic instructions.

Procedure I:

Input four bits from switches on port 1 or 3. Output the bits to LED's on the same port. When the switch is operated, the appropriate LED should be energized. This should be a continuous scan process.

Presentation I:

Demonstrate your circuit. Write a note about the differences in set up for input and output. Discuss your procedure for moving (mov) the values from one bit location to another.

Plan II: T-bird Taillight System

In this project, implement a display reminiscent of an old Thunderbird tail light system.

Preparation II:

Wire 6 LEDs on one port and 2 switches on the other bits of the port. The input port switches will represent brake, idle, turn left, and turn right signals. A two-bit input will be sufficient. The LEDs will represent the taillights. When the switch combination is selected, the LEDs should have the following display:

Brake turns on all LEDs.
Idle turns off all LEDs.
Turn left turns on the right LED of the left bank of 3. Next turn on the adjacent, then the left LED simultaneously.
Turn right. This will be the opposite of turn left. It will turn on the left LED of the right bank of 3. Then it will turn on the adjacent, then the next.

The sequence repeats until another kind of input is applied.

Procedure II:

There are no restrictions in choosing the port for your I/O system. Remember to use proper software documentation. Then, the port used will be obvious to the customer.

Use a delay routine to implement turn right and turn left.

Presentation II:

Explain your observations of how a microprocessor has advantages over a hardwired logic circuit. Please write your explanation following the program listing.

⇐ ⇑ ⇒

Program sample example

The exemplar program is similar to the project specifications. However, it contains elements that should be modified to complete the project as required.

```
;Program: ModTbird.asm
;Initial: 7 February 2003
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)2003.  All rights reserved

;Purpose:
;  Check switches for status and display
;  results in pattern of old T-Bird car
;  P3.0-P3.1 used as direction input
;    00-idle, 01-right, 10-left, 11-brake
;  P3.2-P3.4  left lights
;  P3.5-P3.7  right lights
;  P1.2       start switch
;
;Processor: 8031 family
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz
;Assembler: Intel ASM51
;
;################################################

;                     ASSIGNMENTS

;################################################
Pio       equ   0B0h  ;port for input output
Switch    equ   092h  ;switch input


;################################################

;                     PROGRAM

;################################################
          org   00H
```

```
START:    ljmp  INITIAL

          org   0033h
          db    'Marcus O. Durham, PhD, PE'

;-------------------------------------------------
          org   0080H         ;get past interrupt
INITIAL:
;-------------------------------------------------
          setb  Switch        ;make switch input
          mov   Pio,#03h      ;make bits 0&1 input

;-------------------------------------------------
MAIN:
;-------------------------------------------------
          jb    Switch,MAIN   ;use as start,1=open
          lcall INPUT         ;get switch input

          cjne  A,#03h,TWO    ;<>3 brake
          lcall BRAKE         ;else, =3, go brake
          sjmp  ZERO          ;continue scan

TWO:      cjne  A,#02h,ONE    ;<>2 left
          lcall LEFT          ;else, =2, go left
          sjmp  ZERO          ;continue scan

ONE:      cjne  A,#01h,ZERO   ;<>1 right
          lcall RIGHT         ;else, =1, go right

ZERO:     lcall IDLE          ;else, =0, idle
MAN9:     ljmp  MAIN          ;Repeat

;-------------------------------------------------
INPUT:
;-------------------------------------------------
;  Check bits for input.

          mov   A,Pio         ;input port
          anl   A,#03h        ;mask bits 2-7
          ret                 ;exit

;-------------------------------------------------
OUT:
;-------------------------------------------------
```

```
;  Show the results

          mov   Pio,A          ;display results
          ret                  ;exit

;-------------------------------------------------
DELAY:
;-------------------------------------------------
;  Create a short time delay

          mov   R3,#00         ;Outer loop counter
ZDEL2:    mov   R2,#00         ;Nested loop counter
ZDEL1:    nop                  ;Delay
          djnz  R2,ZDEL1       ;Nested loop, 256 x
          djnz  R3,ZDEL2       ;Outer loop, 256 x
          ret                  ;exit

;-------------------------------------------------
RIGHT:
;-------------------------------------------------
;  Flash right lights.

          mov   A,#10000000b   ;first lamp
          lcall OUT            ;show it
          lcall DELAY          ;wait

          mov   A,#01000000b   ;second lamp
          lcall OUT            ;show it
          lcall DELAY          ;wait

          mov   A,#00100000b   ;third lamp
          lcall OUT            ;show it
          lcall DELAY          ;wait

          ret                  ;exit

;-------------------------------------------------
LEFT:
;-------------------------------------------------
;  Flash left lights.

          ret                  ;exit

;-------------------------------------------------
```

```
BRAKE:
;-------------------------------------------------
;  Turn all lights on.
;  Ensure port input is not cleared.

          ret                 ;exit

;-------------------------------------------------
IDLE:
;-------------------------------------------------
;  Turn all lights off.
;  Ensure port input is not cleared.

          lcall DELAY         ;
          ret                 ;exit

;*************************************************
          end                 ;Program end
```

⇐⇑⇒

14

REGISTER, TIMERS, AND INTERRUPTS

Thought
Everyone is given the same 24-hours.
What you do with it makes the difference.
MOD

Timer registers

Special function registers (SFR) include tools to configure timers and interrupts, as well as numerous other chores. The details of each register are shown in a reference chapter. This discussion is how to implement the register to perform a program task.

One of the more useful features is the built-in timer counters. These can be used to count events or to count machine cycles. The machine cycles then translate into time.

Timer 1 is used for serial communications control. So, for our counting purposes, we will restrict operations to Timer 0. Many chips have a third timer, but it will not be discussed since we want to remain as generic as possible.

The timer control (TCON) register determines when the timer operates. The timer is started by setting the run control bit (TR0), and is stopped by clearing the bit under software control.

The timer mode (TMOD) register determines whether it is operating as a timer or counter.

The counting register is sixteen bits. The low byte is TL0, while the high byte is TH0.

During operation, the counting register increments on a 1 to 0 transition of the port3 pin (0B4h). This transition is checked every machine cycle. During operation as a timer, the register is simply incremented by the hardware. In essence, the timer is counting machine cycles.

A preset value can be placed in the counting register. The counter will increment from this value. A carry bit from the most significant counter stage signals when the register has finished a complete count.

The overflow causes a flag (TF0) to be set for the counter. By monitoring the overflow flag, the program is aware of a completed count cycle. The flag can be monitored by normal polling software. Alternately, it can be used as an interrupt.

Timer

A timer can be used to indicate when a particular amount of time has elapsed. This is accomplished by loading a preset value into the timer register (TH0,TL0). The timer increments at each machine cycle. Therefore, the preset is a negative value. The value is counted backward (negative) from zero. As the count progresses, an overflow occurs at count zero.

One machine cycle consists of six states. Each state lasts for two oscillator periods. Hence one machine cycle takes 12 oscillator periods.

The preset value is equal to the machine cycles subtracted from zero.

$$\text{Preset} = - \frac{\text{time (seconds)} * \text{oscillator frequency (period/sec)}}{\text{12 (oscillator periods / machine cycle)}}$$

Consider a time of 1/30 of a second from a crystal of 11.059 MHz.

$$\text{Preset} = - \frac{\text{1/30 (seconds)} * \text{11,059,000 (period/sec)}}{\text{12 (oscillator periods / machine cycle)}} = -30719$$

The decimal number must be converted to hexadecimal. This permits the upper byte to be stuffed in TH0 and the lower byte to be shoved into TL0.

Since the timer counts in machine cycles, the elapsed time in seconds can be calculated from the value in TH0,TL0.

$$\text{Time} = \frac{\text{TH0,TL0} * \text{12 (oscillator periods / machine cycle)}}{\text{oscillator frequency (period/sec)}}$$

Interrupts

Some events are important enough that other tasks should be suspended. These events are interrupts.

Processing interrupts is a common technique used for real-time computer applications. The mechanism is usually found in data acquisition systems, since timing is a critical problem.

The basic processor has five different sources of interrupts. These are external 0, external 1, timer 0, timer 1, and serial.

Interrupts can arise on two pins of port3. These are INT0 (0B2h) and INT1 (0B3h). The interrupts are initialized in the interrupt enable (IE) register. The precedence can be set in the interrupt priority (IP) register. Detailed explanations are provided in the section on special function registers.

When an interrupt occurs, the next program counter address is pushed to the stack. The program counter is set to the interrupt handling address in low code memory. There are separate addresses for INT0 (0003h) and INT1 (0013h). Each of these has 8 bytes reserved to process the interrupt. If the interrupt service routine requires more room, an ljmp can transfer to another location.

The interrupt processing routine is terminated by a reti instruction. The instruction pops the stack for the location to reset the program counter. The interrupt procedure is very similar to a subroutine that has its own assigned memory location.

Counter & interrupt examples

The first task is to setup timer 0 to count. TH0,TL0 have results of the count.

```
;-------------------------------------------------
COUNT0:
;-------------------------------------------------
          setb  0B4h           ;set T0 for input
          mov   TMOD,#00000101b ;T0,Mode1,counter
          setb  TR0            ;start the count
          ret
```

The first illustration is to use counter 0 to register 5 events that occur on the timer 0 pin of port 3 (0B4h).

```
;-------------------------------------------------
COUNTS:
;-------------------------------------------------
                               ;COUNT 5 EVENTS
          lcall COUNT0         ;set up timer
          mov   A,#5           ;preload the limit
HERE:     cjne  A,TL0,HERE     ;T0 count low byte
```

An interrupt can be used when count gets to a preset value stored in TH0,TL0. For example, to interrupt after counting 5 events, preload with -5 = 0FFFBh.

When count overflows to zero, the interrupt will transfer program control to address 000Bh for processing. This requires the interrupt to be enabled. If an interrupt is not desired, the overflow bit, TF0, can be tested with software polling.

```
;-------------------------------------------------
COUNTINT:
;-------------------------------------------------
                                ;INITIAL
          setb  0B4h            ;count 1 as input
          mov   TMOD,#00000101b ;counter=0, mode1
          mov   TH0,#0FFh       ;minus leading ones
          mov   TL0,#0FBh       ;-5
          setb  TR0             ;start counting

                                ;OPTIONAL INTERRUPT
          orl   IE,#10000010B   ;enable all & T0
;HERE:    jnb   TF0,HERE        ;optional polling
          ret                   ;get outta here
```

Timer with interrupt examples

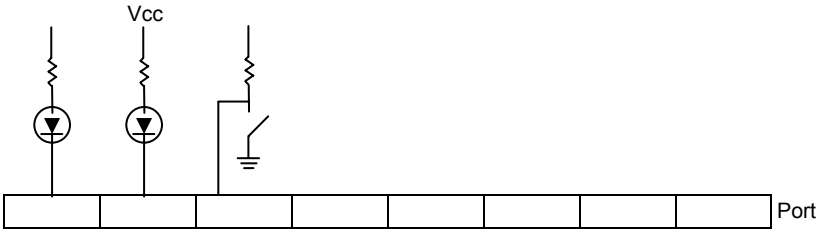
A variation of the counter is to create a timer. The timer simply counts machine cycles. The first routine will initialize timer 0 for use as a clock. The mode will be sixteen bits and a software gate. The routine preloads values and starts the timer.

If the optional interrupt is used, a transfer to address 000Bh is generated when the time has elapsed. Alternately, if an interrupt is not desired, the overflow bit, TF0, can be polled.

```
;-------------------------------------------------
TIMEINIT:
;-------------------------------------------------
;  The preset time is calculated. The crystal osc.
;  frequency is 11.059 MHz.
```

```
;
;     Preset= (Sec * Osc freq)/12 osc/machine cy
;
;  for 1   msec, THxTLx= -921.583= -400h = 0FC66h
;  for 1/20 sec, THxTLx= -45954.167      =  4C7Eh
;  for 1/30 sec, ThxTlx= -30719.444      =  8801h

;                              ;INITIAL
           mov   TMOD,#00000001b ;timer=0, mode=1
TIMERUN:   mov   TH0,#88h       ;4C7Eh = 1/20 sec
           mov   TL0,#01H       ;lo count
           setb  TR0            ;start counting

                                ;OPTIONAL INTERRUPT
           orl   IE,#10000010b  ;enable all & T0
           ret                  ;MrButler to Atlanta
```

Circuit: interrupts _____



⇐⇑⇒

15

PROJECT 3 – CLOCK & INTERRUPT

Thought
Objectivism
is rational self-interest.
Ayn Rand

Project 3: Time to count _____

Purpose: To use the internal registers.
To demonstrate external event triggers.
To calculate very precise times.
To implement interrupt processing.

Preamble:

The basic processor has 5 different sources of interrupts. These are external 0, external 1, timer 0, timer 1, and serial. In this project, only the external interrupts using INT0 or INT1 are required.

To receive an interrupt signal from the pins, make sure that the bits corresponding to each pin are enabled. Set the bit to one, since the ports are bit addressable.

When interrupts are used, an interrupt service routine is placed in the program. The computer program jumps to an address that corresponds to the interrupt.

Several registers are associated with the interrupt locations. Of particular interest are the two counter/timer locations. These can be used to count external events. Alternately they can count clock cycles, which in essence, makes them a timer. The registers include sixteen-bit data and two control registers.

Plan I: The External Interrupt System

Create a project to show that both interrupt pins work properly with either a level or an edge-triggered signal.

Procedure I:

For example, illustrate the project by changing the LED displays every time a trigger is sensed. This means the program is displaying continuously but is interrupted by an external signal. Show at least one interrupt source. The sources can be a combination of level-triggered or edge-triggered with the INT0 / INT1 pins.

A more elegant design will count the number of times that the INTx pin is triggered. Then, after a preset number of events, an LED will illuminate.

Presentation I:

Demonstrate that the interrupt process is working properly.

Plan II: Digital Clock

Implement a real-time digital clock based on the timer register. Timer 1 is used for serial communications; therefore, only Timer 0 should be used for these projects. Use an LED to show seconds and other LEDs to show minutes.

Procedure II:

Toggle the same LED every second. After each minute, illuminate another LED. It is advisable to write the clock routine inside the interrupt program and the display routine as the main program. This will provide a more precise clock.

Presentation II:

Demonstrate the operation of the clock. Ensure all LEDs are off before time start.

⇐⇑⇒

Program sample example _____

The exemplar program is similar to the project specifications. However, it contains elements that should be modified to complete the project as required.

```
;-------------------------------------------------
;Program: MODTime.asm
;Update:  07 February 2003
;Initial: 17 October 1991
;
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)1991, 2002.  All rights reserved
;
;Purpose:
;  A set of routines are provided to perform the
;  basic functions of a clock.
;
;Processor: 8031 family
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz
```

```
;Assembler: Intel ASM51

;################################################
;
;                     ASSIGNMENTS
;
;################################################

TimPer    equ   3CH   ;time periods counter
TimFlag   bit   20H   ;time out

;################################################
;
;                     PROGRAM
;
;################################################
          org   00H
START:    ljmp  INITIAL

;------------------------------------------------
;INTERRUPT-Timer 0
;------------------------------------------------
;  The procedure provides direction when timer
;  completes count.

          org   0BH
          ljmp  TIMECAL       ;interrupt processor

;------------------------------------------------
          org   0033H
          db    'Marcus O. Durham, PhD, PE'

;------------------------------------------------
          org   0080H
INITIAL:
;------------------------------------------------
          mov   SP,#5Fh       ;start stack @ 5f+1
          lcall TIMEINIT      ;start timer0

;------------------------------------------------
MAIN:
;------------------------------------------------
                              ;PROCESS
          jnb   TimFlag,Man9  ;time cycle at end?
```

```
          clr   TimFlag       ;<> 1/2 sec
          cpl   P3.5          ;is 1/2, so flash

MAN9:     ljmp  MAIN          ;Repeat


;-------------------------------------------------
TIMEINIT:
;-------------------------------------------------
                              ;INTERRUPT
          orl   IE,#10000010b ;enable all & T0

;                             ;INITIAL
          mov   TMOD,#00000001b ;timer=0, mode=1
          lcall TIMERUN       ;preset & start T0
          ret                 ;MrButler to Atlanta

;-------------------------------------------------
TIMERUN:
;-------------------------------------------------
;  The routine preloads values and starts the
;  timer.
;
;  The preset time is calculated. The crystal osc.
;  frequency is 11.059 MHz.
;
;    Preset= (Sec * Osc freq)/12 osc/machine cy
;
;  for 1   msec, THxTLx= -921.583= -400h = 0FC66h
;  for 1/20 sec, THxTLx= -45954.167      =  4C7Eh
;  for 1/30 sec, ThxTlx= -30719.444      =  8801h

                              ;INITIAL
          clr   TR0           ;stop counting
          mov   TH0,#88h      ;8801h = 1/30 sec
          mov   TL0,#01H      ;lo count
          setb  TR0           ;start counting

          ret                 ;MrButler to Atlanta

;-------------------------------------------------
TIMECAL:
;-------------------------------------------------
;  This routine is called from the interrupt
```

```
;  processor. First restart the timer.

                                ;INITIAL
          lcall TIMERUN         ;preset & start T0

          inc   TimPer          ;another time period
          mov   A,TimPer        ;period count
          cjne  A,#15,TIMC9     ;periods in 1/2 sec
          mov   TimPer,#0       ;reset period
          setb  TimFlag         ;end of cycle

TIMC9:    reti                  ;leave Scarlet OHara

;***********************************************
          end
```

⇐⇑⇒

16

BOARD CONSTRUCTION

Thought
Moral economics:
Maximizing personal benefit,
without compromising integrity.
Professor Durham

One step. Check!

The procedures used by a board manufacturing or stuffing company are considerably different from a single construction venture by individuals that are building only one or a few projects.

These guidelines are oriented to the prototype construction of a single board. The preferred procedure for populating a board is to do one step at a time. Then check the work as you go. Troubleshooting will be greatly reduced by this procedure.

Show and tell

Insert components carefully to ensure proper polarity. Confirm that the notch on chips is aligned in the same direction as the notch on the board. Other polarity sensitive devices include capacitors, resistor packs, diodes, and transistors. If you are not sure, get help or check data sheets. Incorrect polarity will assuredly destroy semiconductors.

The board has many components that have to be soldered in a relatively close space. Be cognizant of the difference in a pin hole and a via. A via is simply a hole that connects circuit traces on different layers of the board. Verify that your soldering does not flow or connect to a via or another pin. The circuits will short and it is very difficult to find later.

Improper soldering will keep the board from working and can be nearly impossible to find. If you are uncomfortable with soldering, ask for instructions or help. There are a few training videos that are excellent tools. It is better to start correctly than to redo much of your work.

Basics

Build the board the way you are learning the projects. Begin with the basics, get it working, and then move to things that are more complicated.

The first step is to get power to the board. If you do this first, you can test the board without concern for damaging expensive electronics. Install the voltage regulator, filtering capacitor, and the power plug connector.

Then apply power. Verify that there is V_{CC} (5V) and ground where it is required on the board.

Next, install the minimum components necessary for the computer to function. Mount the computer socket, the clock or crystal with capacitors, and the reset resistor and capacitor. Carefully insert the processor in the socket.

Then install the in-system programming circuit. This includes the connection for the external cable. Refer to the section on parts and pins to obtain cable pin-out information. The board can support both serial peripheral programming and serial RS232 communications

through similar jacks. Therefore, it is imperative that proper configurations be used.

At this point, it is prudent to test the status of your work. Download a simple program, like the metronome. Start the computer. Since there are no I/O devices, use a meter or scope to verify the operation of the appropriate pins.

Proceed only if everything is operating at this point. Otherwise, troubleshooting and correcting become a major issue.

Socket to me _____

The board is designed for many functions and applications. It is not necessary to stuff all the components. Only install those items that are needed.

It is highly desirable to use sockets for some of the components. The microprocessor should be in a socket. The programmable logic device must be in a socket and memory devices must be in sockets.

It may be desirable to place in sockets other chips with many pins. This is particularly true if your soldering skills are questionable.

The memory expansion has a special socket that will allow a 0.3" or 0.6" wide memory chip. To install the socket, cut the connecting bars that hook the sides of the socket together. Install the 14-pin SIP header in the center row of the socket.

The programming header (RJ45 or strip) must be installed on top of the board. All other headers should be placed down. This allows the board to be plugged onto a proto board.

Several jumpers are available to expand the flexibility of the board.

- External memory EA': select 5V or Ground
- Seven-segment onboard: select ground or 7406
- Address Latch Enable: select ALE or PLD
- Memory Output Enable: select ground or PLD

- Port 0 resistor pack: select pull-up or pull-down or nothing
- LCD contrast: select auto circuit or jumper to Ground

Initially, connect EA' to 5V, since external memory is not used. The others can be connected if the components are added.

What's left

The remaining items include latches for expansion, analog to digital converter, liquid crystal display, infrared, seven-segment displays, LED's, pushbuttons, and resistor packs.

There is space for a coupling capacitor for each of the major integrated circuits. If the speed is kept low, these generally are not needed. Another alternative is for film capacitors that can be placed under the chips.

For most prototype and project development, only a few of the remaining items are required. It is strongly suggested that you test each circuit system as you add components.

By careful placement and soldering, the board has been proven to work trouble-free.

⇐ ⇑ ⇒

17

PROJECT 4 - DEVELOPMENT BOARD

Thought
A liberal selectively
acts the way he wants without regard for
accountability, absolutes, or consequences.
MOD

Project 4: Build from scratch _____

Purpose: To MOVe from proto-type to system integration.
To have a one-board development system.

Preamble:

Several chapters refer to board construction. One has a parts list and pin-out diagrams. The board schematic and specifications are included. Another has details for construction practices.

The numerous wires used on the proto-board are prone to loose connections and improper operation. Once the fundamentals of design and trouble-shooting are understood, a circuit board can be used.

The board has every tool of a simple development system. The software can be downloaded from a PC to the program memory. By use of the expansion ports, any control circuit can be developed.

For the most basic computer, only the processor, crystal oscillator, memory, and power supply are required. If the program memory is internal, then the system is very simple.

A serial interface is added for troubleshooting. A PEEL is added for combinational logic to decode addresses. Optional memory is available. A liquid crystal display can be connected as well as the circuits for automatic contrast control. An analog to digital converter can be used.

Plan:

A custom-designed board has been developed for this purpose. The board is commercially produced. The schematic is included in the chapter on parts.

The board must be ordered in advance. Delivery time is typically three weeks. The cost is much less if the boards are ordered in quantity, since there is a substantial set-up fee for each order.

Preparation:

All projects to this point should be completed. The board requires the components previously used on the proto-board. These will be soldered on to the circuit board.

It is strongly recommended that sockets be used for the processor, PEEL, and memory. Sockets may be used for other chips, but it runs up the cost.

Procedure:

Once the board has been wired, software must be installed. Continue to program the controller as before.

If using a processor with on-system memory, then connect the cable to the appropriate connector.

If using external memory, program the EPROM. A download program has been written for the development system. If the download program is used, the hex code can be downloaded to the SRAM. The program will transfer control to the microprocessor. The download program has built-in test features. After the system appears to be working, implement any previous project on the completed board. This program is discussed in a later chapter.

Presentation:

Show the system works by downloading a program such as the metronome. Then execute the program.

⇐⇑⇒

18

EXTERNAL MEMORY

Thought
A conservative
is constrained by
accountability, absolutes, or consequences.
MOD

Storage control lines

Harvard architecture separates program and data memory. Moreover, memory can be internal or external. Both external data and program memory are allocated as 64K byte pages. This size is an inherent characteristic of sixteen bit addressing. The program counter (PC) is a sixteen-bit register. Therefore, it can directly access the entire page.

Since only one page of memory is available, any addresses that are used for internal memory are not available in the external memory chip.

Control lines are used to segregate data and program memory and to isolate internal and external operation. On Reset, program control transfers to program memory address 0000h.

If the external address not (/EA) line is low, the address points to external memory. If /EA is high, the program counter points to

internal memory. Regardless of the setting, the next line executed after the highest internal address is at the external device.

When the computer is accessing external program memory, the program storage enable not (/PSEN) line is asserted low. This line is connected to the output enable not (/OE) line on the memory chip. The line is not invoked during internal memory fetches.

When the computer is accessing data memory, one of two lines is asserted. Both are multipurpose lines on port 3. Read not (/RD) is asserted low to get data from external memory. It is connected to output enable not (/OE) on the memory chip. Write not (/WR) is asserted low to send data to external memory. It is connected to the write enable not (/WE) on the memory chip.

Address fetching

The addressing is handled the same for both data and program memory. The sixteen-bit address for external memory is placed on two ports. Port 0 has the low byte and port 2 has the high address byte. The eight bit data is transferred via port 0.

Port 0 multiplexes both address and memory. Therefore, an address latch is required to catch the low memory. The address latch enable (ALE) line is asserted high when addressing is on port 0. An external eight-bit latch is required to trap the address information.

The data for the latch is setup when the latch enable (LE) pin is driven high. When the pin is pulled low, the data is trapped by the latch.

The ALE line is connected to the latch enable pin of the latch. When the ALE line is asserted, the address is setup. The address is trapped on the trailing (falling) edge of ALE. The latch holds the low address. Port 2 holds its high address until the information byte from memory is received.

Port 0 must be set for input to receive data. The CPU automatically sets port 0 for input during a memory fetch. However, it does not retain special function settings. Therefore, that information is destroyed during a program fetch.

Since there is not input on port 2 during a fetch, it does not have to be configured. Therefore, port 2 special function register data is preserved during an external program fetch. The data reappears on the port during cycles, which are not a part of external program fetch.

The program counter (PC) contains the sixteen-bit address for instructions that are fetched from program memory.

The data pointer register is also sixteen bits. The program must fill the register. It will point to program memory when the movc instruction is used. It will point to data memory when the movx instruction is used. This is called indirect addressing

External data memory can be accessed with eight-bit or sixteen-bit addressing. The data pointer low (DPL) contents are placed on port 0. If the address is sixteen bits, the data pointer high (DPH) contents are placed on port 2. If the address is eight bits, the special function register information is retained on port 2.

One other method can be used to address sixteen bits of data memory without modifying the data pointer register. Port 2 is preloaded with the high address byte, as if it were a special function register. Then use eight-bit addressing with the registers R_0 or R_1.

Timing sequence

Although this information is seldom necessary to construct projects, it is of interest to some practitioners. A machine cycle is divided into 6 segments (S1-S6). Each segment will provoke the internal hardware of the CPU to perform a particular function.

Different types of instructions obviously have a different timing sequence. The basic operation is to read a byte at S1 and read the next byte at S4. What happens with each byte depends on the number of bytes and the number of cycles required for the instruction.

The one-byte, one-cycle instruction mov A, R_1 operates with this sequence. Read instruction at S1. Read next byte at S4 and discard. Reread next instruction at next S1.

A two-byte instruction requiring one cycle is mov A,#. It reads byte 1 at S1, then reads byte 2 at S4. The next instruction is read at the following S1.

There are a few one-byte, two-cycle instructions, such as ret and inc. The sequence is read instruction at S1, read and discard next byte at S4, S1, S4.

At S5 of the previous cycle, the address is valid when ALE drops low. A byte is received after PSEN is asserted low during S6. PSEN returns high during S1. Then the program counter is incremented to prepare for the next instruction.

During an external fetch, /PSEN is asserted two times per cycle. An exception is movx. It is not asserted in the second machine cycle.

The external timing diagrams and discussion are shown in a reference chapter near the end of the book.

Virtual memory

In a machine with Harvard architecture, data and program memory are separated. The data is generally stored in random access memory (RAM), while the program is stored in read only memory (ROM). However, permanent or fixed data may be stored in the program memory. Moreover, program code may be stored and executed from a RAM chip.

These practices require hybrid procedures to create virtual memory. Virtual memory is simply memory that appears in application to be different from what it is designated.

Program store enable not (/PSEN) is actuated by the computer to control the output enable not (/OE) on the ROM during a program fetch and a movc instruction. Read not (/RD) is actuated by the computer to control the output enable not (/OE) on the RAM during movx.

If the program memory ROM is to be virtual data memory, either the /PSEN or /RD control lines must be able to control the output enable not (/OE). Since the signals are active low, an AND gate must be used. If either input to the gate is low, the output will be low. The AND is equivalent to an Invert-OR-Not gate.

The program store and the read signals are high-frequency. Therefore, fast gates must be used. Type ALS or faster should be employed.

Alternately a programmable logic device such as a 22CV10, can be used rather than glue logic.

Wiring ROM or RAM

The custom circuit board is discussed in detail in a reference chapter. The board is implemented with an external memory socket. The socket can be used for either ROM or RAM. Therefore, the socket is a model of virtual memory. There are only a few differences in connections. These involve writing information to memory and reading the stored values.

This case will specifically discuss 32K x 8 memory, often referred to as 256K.

ROM is programmed by V_{PP} on pin 1. Address A14 is connected on pin 27. ROM is read via output enable not (/OE) on pin 22. The ROM read is activated by program store enable not (/PSEN).

RAM is written on write enable not (/WE) on pin 27. Address A14 is connected on pin 1. RAM is read via output enable not (/OE) on pin 22. RAM read is activated by read not (/RD) and write not (/WR).

A map clearly illustrates how the connections to the chip socket are interrelated. A programmable logic device (22CV10) can be used as the logic to switch the appropriate lines to the socket.

Direction	Device	Pin 1	Pin 27	Pin 22
to	ROM	V_{PP}	A14	/OE
from	micro		A14	/PSEN
to	RAM	A14	/WE	/OE
from	micro	A14	/WR	/RD

A Boolean equation is required for the line switching on each pin.

```
Mp1  = ucA14 & !ucA15 & (!WRn # !RDn)
Mp27 = (!PSENn & ucA14) # (WRn & !ucA15)
OEn  = PSENn # !(!RDn & ucA15)
```

Address A15 is included in the equations to segment memory. A15 is used as a control line for input / output device expansion. The expansion is mapped into high memory.

There is also a pin required for chip control. Functionally, since there is only one memory device, the chip select can be left active by connection to ground.

⇐⇑⇒

19

BIOS

Thought
An anarchist
is unconstrained by
accountability, absolutes, or consequences.
MOD

Definition

This chapter and the associated project will only be used if your design includes both an external EPROM and SRAM and the program is loaded via the serial port. Otherwise, the topics can be skipped with no loss of ability to build a workable system.

What is bios? Bios is an acronym for basic input output system. The program begins when the computer is initialized. The procedure is typically stored in read-only memory and cannot be changed. Because of the permanent nature of the software, it is often called firmware.

The firmware provides the interface to the keyboard and displays until an operating system or application program takes over more elegant implementation of the functions. It also provides for transfer of control to the operating system.

The bios is a group of subroutines and procedures that are common across the spectrum of projects. These are written so they can be

called from applications programs. In some cases, they become the model for other specialty routines.

The bios for the microprocessor system would include the firmware implementation of the serial, seven-segment or liquid crystal display, and keypad. That is what is commonly used as a backbone for developing applications software.

The bios that will be discussed in this chapter is slightly different. The firmware is loaded in the read-only memory that will be accessed on initialization. However, the function of that code is to use the serial port for loading an application program into static ram. Program control is then transferred to run from the static ram.

The BIOS developed in this chapter is specifically for programming an external EPROM, then using the serial port to load a program into external SRAM. If you are using internal memory, this topic will not apply to your projects. Nevertheless, you may find some of the information beneficial.

Bios main

The firmware is the initial code if serial transfer is used to move a program to external memory. It sets up RS232 loading of the program from a personal computer (PC) to static ram. Then it directs switching the program execution from ROM to RAM.

The code is for use with the most fundamental derivatives of the Intel 8031. Many variations of the microprocessor have this or similar capabilities as an integral part.

The program has several key functions:

1. Test the SRAM.
2. Download a new program from the serial port.
3. Save the program to SRAM
4. Send a control signal to switch from EPROM to SRAM.
5. Execute the next instruction from SRAM.
6. Display error messages.

A light emitting diode connected to port 3.5 (P3.5) is used for messages. This very fundamental display can be implemented without extensive code and additional hardware.

1. on-off = controller is operating.
2. on-off-on = serial download is complete.
3. long on-long off, repeated 32 times = serial fault.
4. fast on-fast off, repeated 128 times = SRAM fault.
5. longer on-fast off, repeated 10 times= not switch eprom

```
;-------------------------------------------------
BIOSMAIN:
;-------------------------------------------------
                                ;INITIALIZE
          mov   SP,#5Fh         ;start stack @ 5f+1
          lcall UART            ;Config & start UART

                                ;PROCESS
          lcall RAMTEST         ;Test RAM & interfac
          mov   DPTR,#SerGreet  ;get start addr
          lcall BIOSSER         ;Send message on ser
          lcall DOWNLOAD        ;Serial DOWNLOAD Ram
          lcall P35DONE         ;Flash P35=done dnld

                                ;TERMINATE
          ljmp  MEMSWIT         ;Switch prog storage
                                ;to RAM from EPROM
```

Static memory test

This subroutine tests the microprocessor to static ram interface. That includes the device, wiring, and control logic for the virtual memory operation. Each RAM address is tested with two complete write/read loops. Each loop involves writing an alternate bit pattern to each address, then reading each address to confirm the value. For the second loop, the starting bit pattern is the complement of the starting bit pattern of the first loop.

If any one of the values read does not equal the value written, then the test fails. A failed test can result from a bad interface or a bad RAM chip. The message LED on P3.5 cycles off/on 128 times.

The message LED at P3.5 turns on at the start of the test and turns off at end of a successful test. The memory subroutine returns to the main calling routine upon test completion.

```
;-------------------------------------------------
RAMTEST:
;-------------------------------------------------
                                ;TURN ON LED & INIT
          setb  P3.5            ;Set LED on/uC Busy
          mov   TmpA,#02        ;two wr/rd passes
          mov   TmpC,#0AAH      ;First wr=10101010

                                ;LOOP
RAMT1:    mov   TmpB,TmpC       ;Start write pass
          mov   DPTR,#0000H     ;First RAM address

                                ;WRITE
RAMT2:    mov   A,TmpB          ;Value to write
          movx  @DPTR,A         ;Write to RAM
          cpl   TmpB            ;Flip bits
          inc   DPTR            ;Increment address
          mov   A,DPL           ;} Continue write
          cjne  A,#00H,RAMT2    ;} loop until each
          mov   A,DPH           ;} RAM address has
          cjne  A,#80H,RAMT2    ;} been written to.

                                ;READ & TEST
          mov   TmpB,TmpC       ;Start read pass
          mov   DPTR,#0000H     ;First RAM address
RAMT3:    movx  A,@DPTR         ;Read value @ addrs
          cjne  A,TmpB,RAMT7    ;Comp to write value
          cpl   TmpB            ;Flip bits
          inc   DPTR            ;Increment address
          mov   A,DPL           ;} Read each RAM
          cjne  A,#00H,RAMT3    ;} address until
          mov   A,DPH           ;} comp fails or
          cjne  A,#80H,RAMT3    ;} all addrs read.

          cpl   TmpC            ;Flip starting bits
```

```
          djnz  TmpA,RAMT1    ;Do second pass

                              ;SUCCESSFUL TEST
          clr   P3.5          ;LED off/uC not Busy
          sjmp  RAMT9         ;Return to call

                              ;FAILED TEST
RAMT7:    mov   TmpA,#0FFH    ;Init loop counter
RAMT8:    cpl   P3.5          ;Toggle LED
          lcall DELAY         ;Delay to see LED
          djnz  TmpA,RAMT8    ;Loop 256 times
          setb  P3.5          ;Set LED on/uC Busy
RAMT9:    ret                 ;Return to call
```

Download

The download routine is used to receive an ASCII file in Intel hex format on the serial port. Each record begins with a colon (:) and ends with a checksum.

The routine waits for a colon byte. Then the next byte determines the number of hex values in one line or record. The stream next has a two-byte value for the beginning program address where the data will be stored. The following byte is to check end-of-file. Finally, the stream of hex contents follows and is transferred to RAM.

```
;-------------------------------------------------
DOWNLOAD:
;-------------------------------------------------
                              ;NO HANDSHAKE
          clr   P3.5          ;Hndshak-uP not BUSY

                              ;RECEIVE READY
          jnb   RI,DOWNLOAD   ;Wait for rec'd byte
          lcall SERIN         ;Input byte
          cjne  A,#3AH,DOWNLOAD ;Wait for : input

                              ;COUNTER BYTES
DOWN1:    lcall DOWNBYTE      ;':' rec'd, get byte
          mov   R5,A          ;Put in R5 for cntr
          mov   R4,A          ;Init CHEKSUM in R4
```

```
                                    ;ADDRESS
DOWN2:    lcall DOWNBYTE            ;Get next byte which
          mov   DPH,A               ;is high order addrs
          lcall CKSUMINC            ;Update checksum
          lcall DOWNBYTE            ;Get next byte which
          mov   DPL,A               ;is low order addrs
          lcall CKSUMINC            ;Update checksum

                                    ;CHECK FOR EOF
DOWN3:    lcall DOWNBYTE            ;Ck next byte, EOF?
          cjne  A,#0H,DOWN6         ;If so, exit, else
          lcall CKSUMINC            ;Update checksum

                                    ;NEXT byte
DOWN4:    lcall DOWNBYTE            ;Input HEX file
          movx  @DPTR,A             ;Output to RAM
          inc   DPTR                ;Next address
          lcall CKSUMINC            ;Update checksum
          djnz  R5,DOWN4            ;Repeat if more byte

                                    ;CHECK IF ERRORS
DOWN5:    mov   A,R4                ;Verify CKSM of recd
          cpl   A                   ;byte & HEX file.
          inc   A                   ;Complement, add 1 &
          mov   TmpA,A              ;compare with in
          lcall DOWNBYTE            ;file.  If OK,
          cjne  A,TmpA,DOWN7        ;continue, else err
          ljmp  DOWNLOAD            ;Continue receiving

                                    ;end-OF-FILE
DOWN6:    lcall DOWNBYTE            ;Input file chksm.
          cjne  A,#0FFH,DOWN7       ;If not FF, error
          ret                       ;Return to call

                                    ;FAILED
DOWN7:    mov   LoopC,#3CH          ;Loop counter
DOWN8:    cpl   P3.5                ;Toggle LED/uP BUSY
          lcall DELAY               ;Delay to see LED
          lcall DELAY               ;Delay to see LED
          lcall DELAY               ;Delay to see LED
          djnz  LoopC,DOWN8         ;Loop 60 times
          setb  P3.5                ;LED on/uP BUSY
          ret                       ;Return to call
```

Downbyte

The downbyte routine is simply protocol for transferring serial information into a byte.

This routine inputs two sequential ASCII values and converts them to a corresponding binary value. The binary format is a nibble. The low order and high order nibble are combined to yield the hexadecimal Intel-hex format. Register A is used for input, R3 for output, with R1 and R2 for processing.

The serial communications routines are discussed with substantial detail in the next chapters.

```
;-------------------------------------------------
DOWNBYTE:
;-------------------------------------------------
                                ;RECEIVE READY
          jnb   RI,DOWNBYTE     ;Wait for full byte
          lcall SERIN           ;Input when rec'd
          mov   R1,A            ;Put byte in R1
          anl   A,#40H          ;Ck lead nibble, if
          jz    DOWB2           ;above 9, convert.
                                ;
                                ;HEX TO DECIMAL
DOWB1:    mov   A,#09D          ;Conv ASCII A to F
          add   A,R1            ;by adding 9 then
          sjmp  DOWB3           ;adjust lead nibble
                                ;
                                ;DECIMAL
DOWB2:    xch   A,R1            ;Xchg ASCII value
                                ;
                                ;ADJUST
DOWB3:    swap  A               ;Swap nibbles then
          anl   A,#0F0H         ;strip unused nibble
          mov   R3,A            ;Put hi nibble in R3
                                ;
                                ;GET NEXT byte
DOWB4:    jnb   RI,DOWB4        ;Wait for next byte
          lcall SERIN           ;Input when rec'd
```

```
          mov   R2,A          ;Put ASCII byte n R2
          anl   A,#40H        ;Strip lead nibble
          jz    DOWB6         ;If ASCII above 'A'
                              ;
                              ;HEX TO DECIMAL
                              ;then adjust by + 9
DOWB5:    xch   A,R2          ;to stripped nibble
          add   A,#09D        ;& save in R2 to
          xch   A,R2          ;combine w/hi nibble
                              ;
                              ;COMBINE
DOWB6:    xch   A,R2          ;Adj ASCII to A
          anl   A,#0FH        ;Strip lead nibble
          orl   A,R3          ;combine nibbles, &
          ret                 ;Output thru A
```

Checksum

The last item in the line record is a checksum. A checksum is calculated and compared to the received check value. The checksum is calculated by adding each byte received over the serial line into a register that holds the total. Each byte added is already converted into binary form.

If a serial communications checksum error is detected, the message LED toggles off/on 30 times. The off/on cycle period is thrice that of the RAM test error condition.

```
;-------------------------------------------------
CKSUMINC:
;-------------------------------------------------
          add    A,R4         ;add new byte to sum
          xch    A,R4         ;put sum into accum
          ret
```

The procedure calls the serial input routine that was discussed in the serial project. It also calls a routine to convert ASCII data to a binary byte.

ASCII to hex conversion

The conversion routine inputs two sequential ASCII values and converts them to a corresponding binary value. The binary format is a nibble. The low order and high order nibble are combined to yield the hexadecimal format. Register A is used for input, R3 for output with R1 and R2 for processing.

```
;-------------------------------------------------
ASCIBYTE:
;-------------------------------------------------

                                ;RECEIVE READY
          jnb   RI,ASCIBYTE     ;Wait for full byte
          lcall SERIN           ;Input when rec'd
          mov   R1,A            ;Put byte in R1
          anl   A,#40H          ;Ck lead nibble, if
          jz    ASCB2           ;above 9, convert.
                                ;
                                ;HEX TO DECIMAL
ASCB1:    mov   A,#09D          ;Conv ASCII A to F
          add   A,R1            ;by adding 9 then
          sjmp  ASCB3           ;adjust lead nibble
                                ;
                                ;DECIMAL
ASCB2:    xch   A,R1            ;Xchg ASCII value
                                ;
                                ;ADJUST
ASCB3:    swap  A               ;Swap nibbles then
          anl   A,#0F0H         ;strip unused nibble
          mov   R3,A            ;Put hi nibble in R3
                                ;
                                ;GET NEXT byte
ASCB4:    jnb   RI,ASCB4        ;Wait for next byte
          lcall SERIN           ;Input when rec'd
          mov   R2,A            ;Put ASCII byte n R2
          anl   A,#40H          ;Strip lead nibble
          jz    ASCB6           ;If ASCII above 'A'
                                ;
                                ;HEX TO DECIMAL
                                ;then adjust by + 9
```

```
ASCB5:    xch   A,R2          ;to stripped nibble
          add   A,#09D        ;& save in R2 to
          xch   A,R2          ;combine w/hi nibble
                              ;
                              ;COMBINE
ASCB6:    xch   A,R2          ;Adj ASCII to A
          anl   A,#0FH        ;Strip lead nibble
          orl   A,R3          ;combine nibbles, &
          ret                 ;Output thru A
```

Memory switch

After the program download to the SRAM is complete, it is necessary to switch control from the ROM to the RAM for program execution. This routine does a memory-mapped output to change the status of the mode latch in the programmable logic device. Mode 0 is the default value and the program will run normally from ROM. Mode 1 is sent as a signal to switch to SRAM

This is a switch of memory devices during program execution. Therefore, the entry point into the program that will run from SRAM will be at the address after the instruction movx @DPTR,A. The routine is six bytes long. For that reason, back up from the desired start location by six bytes to determine the org location for the routine.

All register values retain their value during this switch since no reset occurs. Therefore, the new program must initialize the register values it uses.

The bios program can be in either internal or external memory. The switch program will start at the next address after the switch. Therefore, the SRAM program memory location can be at the same address space as the original bios.

For example, consider the bios starts at address 0000h. It has the requisite jump past the interrupts. It begins operation at address 0080h. Then the download procedure occupies the next few hundred

bytes. If the memory switch routine is org 007Ah (80h – 6h) in the ROM, then the new program will also start at 0080h in the RAM.

```
;-------------------------------------------------
007A         org   007AH        ;After interrupt
MEMSWIT:
;-------------------------------------------------
007A 90FFFF mov    DPTR,#0FFFFH ;Mode latch address
007D 7401   mov    A,#1         ;Mode =1
007F F0     movx   @DPTR,A      ;Make switch to RAM
```

The process just described works fine if the ROM and RAM are both external. However, if the ROM is internal, /EA will not be configured for external operation. As a result, it is necessary to be a little more creative with the software.

Assume there are 8K bytes of on board program memory. Then, the top of the address space will be 1FFFh. The next executed address will be 2000h, but it will be external memory. This is independent of the /EA setting.

Consequently, the memory switch should be org 1FFAh, which is 6 bytes from the top.

```
;-------------------------------------------------
           org   1FFAh          ;After interrupt
MEMSWIT:
;-------------------------------------------------
```

Then the next program code line will execute at 2000h. In order for the new program that has been downloaded to execute properly, it must have the program organized to org 2000h.

```
;-------------------------------------------------
           org   2000h          ;start address
START:
;-------------------------------------------------
```

Use of low memory

One cost of this practice is the lower 8K of SRAM is not usable as program memory. Nevertheless, if the control lines are kept correct by the programmable logic device (PLD) program, there is nothing to keep that section from being used as data memory. Although this is slightly convoluted from a PLD program perspective, it is a true use of virtual memory.

Suppose it is desirable to address a routine in the bios from the new program in SRAM. The programs do not know each other exist. The low memory SRAM overlays the same address space as the program bios. This little feature gives a solution.

Create a routine that will display the location of all the routines that may be called. Use the data to define the location to the SRAM program. The SRAM procedure will require an equ for the name and address of each routine that is called. Alternately, an org directive can be set for the same location.

⇐ ⇑ ⇒

20

PROJECT 5 – BIOS DEVELOPMENT TOOL

Thought
Perfect practice makes perfect.
Proverb

Project 5: Develop operating system

Purpose: This project can be skipped if the program is loaded into internal memory.
The design includes an EPROM for startup, and serial communications to load the running program into SRAM.
To download a new program from a PC computer.

Preamble:

Time required for software development by programming EPROM's is too extensive for industrial applications. As a result, a development system is used to speed-up the software process. A development system usually consists of the target microprocessor, memory, and a monitor program.

The monitor is a dedicated operating system that transfers program code from an external computer to the project computer. In some systems, the monitor is called a basic input / output system (BIOS). The target microprocessor in the development system is sometimes called an emulator.

In this project, a simple development system will be built for faster software development and to minimize the number of times the EPROM is burned.

Plan:

An external static RAM will be added to the microcontroller. The static RAM will be used to store the program that is downloaded from the host computer. The host will be a PC compatible computer.

First, the static RAM is used as ***data memory*** to receive a file, which is program code from the host. Then the control lines will be changed to make the RAM operate as ***program memory.***

In order to download the program, the host computer has a driver or file transfer program, such as HyperTerminal that transmits the data on the serial port. The development system also has a program that will monitor the received data. In this case, the microprocessor will have a BIOS program reside in the ROM.

The interface used to download INTEL HEX files from the host to the microcontroller system is simple RS-232 serial communication without any control (handshake) lines.

Preparation:

Observe the comparison between RAM and ROM pins. For 32 K byte devices, only two pins are different.

EPROM: Mp1=V_{PP}, Mp27=A14
SRAM: Mp1=A14, Mp27=/WE

There are two other pins that are used for chip control. These are the chip select and the output enable. Functionally, since there is only one memory device, the chip select can be left active by connection to ground. The output is determined by the function.

EPROM: /OE=/PSEN
SRAM: /OE=/RD

The RAM initially is used as data memory storage, then as program. Therefore, both the EPROM and SRAM must have their control lines switched. An address latch in the PLD (PEEL) is used to make the change.

The indicator for the download process will use port P3.5. Connect an LED with proper buffering to this port.

Finally, program the code memory with the BIOS program that was discussed in the previous chapter.

Procedure:

The BIOS program will operate as a RAM tester. If P3.5 turns on, then off, the right connections have been made. If the LED flashes 128 times, hardware debugging of the SRAM system is required.

The monitor program constantly scans for the incoming serial data. The host PC computer will execute a download file transfer program. This program transmits an INTEL HEX file from the host to the development board. If P3.5 flashes 30 times, the download process was not successful. Otherwise, the download was completed.

The BIOS changes control to execute the program from RAM. The application program should begin at an address above all interrupts and other reserved spaces.

Using serial communications, download the new program hex code to the SRAM. The program will transfer control to the microprocessor.

Presentation:

Show the system works by downloading a program such as the metronome. Then execute the program from the static RAM.

$\Leftarrow \Uparrow \Rightarrow$

21

SERIAL COMMUNICATIONS

Thought
For lack of a nail…
the kingdom was lost.
Benjamin Franklin in
Poor Richard's Almanac.

Background

Communications between systems takes many forms. For a number of years, parallel communications was regarded as the preferred mode since a full byte of data was available with each transfer. However, the limitation is in the distance that the logic can drive the circuit. Typically, this is less than 20 feet. In addition, many wires are required.

The oldest form of electronic data communications is serial. In 1844, Samuel F. B. Morse developed the first serial communications system with the practical implementation of the telegraph. The first communications was Baltimore to Washington. He developed a code for the on-off bits used to switch a remote relay.

Unfortunately, it used uneven spacing. Shortly after Morse, Emile Baudot invented the 5 digit code. Communications to this day remember his name with Baud.

In 1902, Charles Krum developed an electro-mechanical system that was the predecessor of the Teletype. This technique could literally transmit data around the world.

His circuitry provided the basis of present day serial communications using Electronic Industry Association (EIA) standard RS232. The original standard was developed in 1960. The standard was modified in 1969 and that one is still used in computer systems. The system responds to a sequence of +25 and –25 volt signals. Moreover, the current loop technique of the Teletype spawned the structure of the 4-20 milliamp current loop used in instrumentation systems.

With his system, Krum also developed a code for the sequence of on and off bits. This code is the predecessor of the present day American Standard for Information Interchange (ASCII) characters. That standard was adopted in 1966.

Many different attempts have been made to eliminate RS232, but it persists for several reasons. Just a few of these are noted. First, because of its heritage, it is very simple to construct equipment. Therefore, virtually any device or machine can include its features. Second, because of its long use, there are innumerable devices that are compatible. Therefore, it will persist for some time to come. Third, the limited number of wires, simple configuration of wires, and the long distance capabilities make it very economical.

The theoretical maximum speed of communications using the system over a standard telephone circuitry is about 9600 baud. This is because of the Nyquist frequency limit in the voice range. However, clever compression algorithms have raised this value over an order of magnitude. Nevertheless, simple systems still limit their speed to 9600 baud.

Another major limitation is the number of items that can be connected simultaneously on the network. The system is designed for communications between only two devices on the network.

Microcontroller

The microcontroller has serial communications as part of the basic chip. A universal asynchronous receive transmit (UART) package is included. The data exchange is on port3. Receive is bit zero (0B0h) and transmit is bit one (0B1h).

Separate buffers are used for transmit and receive, therefore they can occur simultaneously as true duplex communications. Since the micro code is structured for communications, software implementation is very simple. The speed is setup, a message is transferred via the buffer, and a flag is checked for completion of the message.

Integration only requires a method to convert the higher voltages of serial RS232 to the V_{CC} of the chip. Although a discrete circuit can be built, commercial interface chips are very cost effective and require little real estate.

Several registers are used in serial communications. These are serial control, power control, timer mode, timer control, and timer 1 value. The details of the bit patterns are illustrated in the chapter on special function registers.

Generating baud rates

The serial control register (SCON) determines the UART mode of operation. This determines whether there is a fixed or variable baud rate.

If the rate is variable, then the numeric value is placed in timer 1. For serial communications, timer 1 would be used in the auto reload mode. In this auto reload timer mode, the serial communications can continuously run without additional software action from the program. Auto reload is configured in the timer mode (TMOD) register as mode 2 for timer 1.

The Baud rate calculation is dependent on the data protocol. The mode may be as a shift register, eight bit UART, or 9-bit UART. The Baud rate for each serial mode is shown in the following equations.

Mode 0

Mode 0 has a fixed baud rate, which is 1/12 of the oscillator frequency. To run the serial port in this mode, none of the Timer/Counters need to be set up. Only the SCON register needs to be defined. This is simply a shift register.

$$\text{Baud Rate} = \frac{\text{Oscillator Frequency}}{12}$$

Mode 1

Mode 1 has a variable baud rate for eight bit data. Either Timer 1 or Timer 2, if available, can generate the baud rate. The generic procedure uses Timer/Counter 1.

$$\text{Baud Rate} = \frac{K * \text{Oscillator Frequency}}{32 * 12 * [256 - TH1]}$$

A multiplier (K) is available to double the calculated baud rate. This is the SMOD bit in the power control (PCON) register. If SMOD = 0, then K = 1. If SMOD = 1, then K = 2. Since the PCON register is not bit addressable, one way to set the bit is logical ORing the PCON register (orl PCON, # 80H). The address of PCON is 87H.

The reload time for the baud rate is placed in the high byte of Timer 1 (TH1). The equation to calculate TH1 can be written as follows.

$$TH1 = 256 - \frac{K * \text{Oscillator Frequency}}{32 * 12 * \text{Baud Rate}}$$

TH1 must be an integer value. Rounding off TH1 to the nearest integer may not produce the desired baud rate. In this case, it may be necessary to choose another crystal frequency.

Mode 2

In Mode 2, the baud rate is fixed for 9-bit data. It is 1/32 or 1/64 of the oscillator frequency, depending on the value of the SMOD bit in the PCON register. In this mode, none of the timers is used, and the clock comes from the internal phase-2 clock.

SMOD = 1, Baud Rate = 1/32 Osc Freq.
SMOD = 0, Baud Rate = 1/64 Osc Freq.

To set the SMOD bit, use orl PCON,# 80H. The address of PCON is 87H.

Mode 3

The baud rate in mode 3 is variable and sets up exactly the same as in mode 1. This allows 9-bit data transfer.

Timer/counter 2 baud rates

Timer 2 can be used in the baud-rate generating mode. If Timer 2 is clocked through pin T2 (P1.0) the baud rate is given by the following equation.

$$\text{Baud Rate} = \frac{\text{Timer 2 Overflow Rate}}{16}$$

If serial communications is being clocked internally, the baud rate is given by the following equation.

$$\text{Baud Rate} = \frac{\text{Oscillator Frequency}}{32 * [65536 - (\text{RCAP2H,RCAP2L})]}$$

To obtain the reload value for RCAP2H and RCAP2L the previous equation can be rewritten as follows.

$$\text{RCAP2H,RCAP2L} = 65536 - \frac{\text{Oscillator Frequency}}{32 * \text{Baud Rate}}$$

Timer baud table

Several rates are commonly used and the choice of acceptable crystal frequencies is rather limited. Therefore, a table can provide the most commonly used values required for Timer 1.

TH1		7.3728 MHz	8.00 MHz	11.0592 MHz	11.0592 MHz	12.00 MHz	12.00 MHz	14.7456 MHz	22.1184 MHz
				smod=0	smod=1	smod=0	smod=1		smod=1
E0		600	651	900		976		1,200	
E6	-26					1,202			
E8	-24			1,200	2,400				4,800
F0	-16	1,200	1,302	1,800		1,953		2,400	
F3	-13					2,404			
F4	-12			2,400	4,800				9,600
F8	-8	2,400	2,604	3,600		3,906		4,800	
F9	-7	2,743	2,976		8,299	4,464	8,923	5,486	
FA	-6	3,200	3,472	4,800	9,600	5,208		6,400	19,200
FD	-3			9,600	19,200				38,400
FF	-1	19,200	20,833	28,800	57.6K		62,500		115.2K

Timer 1 and color burst

Let us imagine that on one of his adventures, Mac MacGyver, from the old television series, gets caught on a remote island. The only communications is via an old instrumentation system operating at 300 Baud. He is challenged to make a computer interface. On searching, he finds an old television that he can strip of its color

burst crystal. Although odd for computers, the frequency is the very common 3.5795 MHz.

MacGyver asks you to determine what is the preload value that he must load into the microcontroller?

$$TH1 = 256 - \left\{\frac{1 * 3579500}{32 * 12 * 300}\right\}$$

$$TH1 = -31.07$$

He must round this value to –31 or 0E1h. Since it is not precise, there may be some loss of data. However, at this slow speed, that should not be a problem.

Interestingly, this slow rate of 300 Baud is still frequently used with long range radio communications such as amateur or "ham" radio.

Serial initialization

The set-up for serial communication has many options that require substantial details. Nevertheless, the actual code used is quite small. A complete serial initialization contains a very limited number of lines.

Register Timer 1 is used in the eight-bit, auto-reload mode. Therefore, TMOD bit M1-1 is set and bits M0-1, C/T'-1 and GATE-1 are clear. If the GATE were set, the external input on port3 would control starting and stopping the timer.

GATE-1	C/T -1	M1 –1	M0 -1	GATE-0	C/T -0	M1 -0	M0 -0

Register PCON, bit SMOD can be used to double the baud rate. If the bit is clear, the rate multiplier is K=1. If the bit is set, then the multiplier is K=2. However, the speed multiplier is not used for these examples.

SMOD	—	—	—	GF1	GF0	PD	IDL

Register SCON is used to define serial communication as mode 1, eight-bit UART, by setting bit SM1. Reception is enabled with bit REN set.

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Set TI to indicate the previous serial transmission is complete. Otherwise, the code that tests for a complete transmission will hang.

Register TCON, bit TR1, is set to enable the timer to begin counting.

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Register TH1 contains the count for the baud rate.

```
TH1 = 256-(K * Osc Freq / 384 * baud rate)
```

For a 11.059 MHz oscillator, and 9600 baud operation, the value placed in TH1 is 0FDHh (-3d).

```
;-----------------------------
UART:
;-----------------------------
          mov   TMOD,#00100000b ;Timer 1 Mode 2
          mov   TH1,#0FDH       ;9600 BAUD @11.05MHz
          mov   SCON,#01010010b ;Set SM1, REN & TI
          setb  TR1             ;Start timer
          ret                   ;Return to call
```

Serial data protocol _____

Many options are available for data protocol. The SCON register is configured for eight-bit or nine-bit data transfer. This will then determine the hardware communications sequence. Eight bits are normal communications. Nine are used when additional control bits are required.

Eight-bit data is transferred in the SBUF register. If a ninth bit is required it is placed in the SCON register.

Asynchronous operation is typical. This implies the device on both ends of the serial communications line has its own clock.

Parity is a very simple method to determine data integrity. A parity flag (P) is located at bit 0 in the Program Status Word (PSW) register. The flag is set if there is an odd number of ones in the accumulator. The flag is clear if there is an even number of ones in the accumulator.

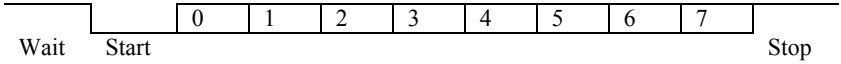
If parity is used, the bit is placed as the last bit in the transmission. For eight-bit exchange, this would be the most significant bit in the accumulator. However, because of the difficulty in managing eight bits of data, plus a parity bit, the parity is often not used.

The standard data format for interchange with most serial devices is ASCII. This is a 7-bit code for characters. The table is shown in the reference chapter on ASCII.

However, it is not necessary to send ASCII. Any protocol structure can be used. Nevertheless, both the receiving and transmitting devices must be using the same structure.

The most common configuration is eight bits of data, no parity, and one stop bit. The protocol is called 8-N-1.

The serial line is normally held high. When data is ready to be sent, the line is pulled low for one bit time, the start bit. Then the eight bit string is sent. This is followed by a high stop bit that allows the receiver to process the character.



Serial buffer

The exchange of data using the internal serial buffers is very simple. First, verify that the last process is complete. Then clear the process interrupt flag. Next, transfer the data between the accumulator and the buffer.

Once the data is in the buffer, the program can go on to other things. The hardware micro code will complete the exercise of shifting the data between the port 3 pins and the buffer.

A simple scan and wait routine is most frequently used. Alternately, a serial interrupt can be invoked. Since there is only one serial interrupt, the handler first must decide if the interrupt was caused by transmit or receive. If interrupt handling is used, the only change in the processing routine would be removal of the first or polling line from the next two routines.

These routines do not have hardware handshaking. If that is required another port bit must be wired to the data cable.

Serial in brings data from an external device into the processor.

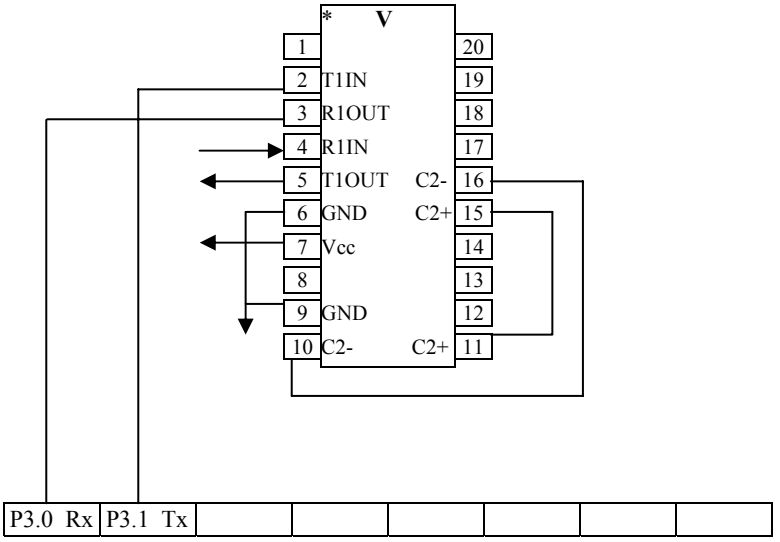
```
;-----------------------------
SERIN:
;-----------------------------
          jnb   RI,$          ;rcv busy so wait.
          mov   A,SBUF        ;Get byte from SBUF
          clr   RI            ;Clear RI
          ret                 ;Return to call
```

Serial out exports data to an external device.

```
;-----------------------------
SEROUT:
;-----------------------------
```

```
        jnb   TI,$          ;xmit busy so wait.
        clr   TI            ;clr for next xmit
        mov   SBUF,A        ;mov byte to serial
        ret
```

Circuit: serial _____



⇐⇑⇒

22

PROJECT 6 – RS232 COMMUNICATIONS

Thought
Serial is for communicating.
Cereal is for eating.

Project 6: RS 232 to PC exchange ___

Purpose: To use a serial interface system.
To implement a simple network system
To implement a door lock system with a dedicated uC.

Preamble:

Communication is one of the most important parts of a computer system. Computer networks are commonly used for data communications. Digital data communication uses one of two modes.

Serial communications has several advantages over the parallel type. A major benefit is that the number of wires is reduced significantly. This reduces the cost of wiring in a long distance type circuit. Another benefit is serial communications operate at a signal level which can be transmitted much further without degradation. Most major local area networks for computers use serial type communications.

The processor has a built-in serial circuit that can be used for communications. The computer has its own baud rate generator, which uses a timer. Several bits in various registers must be set before opening serial communications. The timer interrupt registers are a necessary component, since Timer 1 determines baud rate. The details of setting registers, interrupts, and the baud rate generator can be observed from the reference section.

The EIA RS-232 is the most widely used standard in serial communication. The primary signals are lines 2, 3, and 7. These are receive, transmit, and ground signals. RS-232 has several handshake lines that may be used. These other handshake lines are optional.

The RS-232 circuitry must be capable of handling +25 to –25 Volts. The minimum voltage level of a line is +9 V for high and -9 V for low. Therefore, the voltage level of the serial pin must be raised, since it is operated in the TTL range of 0 and 5 volts. A voltage doubler can be used.

The easiest way to interface RS-232 to the serial line is with a special purpose IC such as MAX 232. The chip will receive an RS-232 signal and send it to the controller at TTL level. The receive pin (R1IN) will get data from the RS-232 transmit pin #2 (TXD) of the PC. The MAX232 output pin (R1OT) will be connected to the microprocessor pin #10 (RX). A ground (RS-232 pin# 7) must be connected between the computers.

Plan:

Implement a microprocessor as a data communication controller that is coupled to a PC.

Preparation:

Consult the data section for more details about serial communications. The interface circuit for RS-232 communications is given in the following pages. The software for the PC is typically

the Windows based program HyperTerminal in the Accessories > Communications folder. The cable from the PC can be constructed using the diagram in the reference section.

Procedure:

The system works according to the following description. The microprocessor will scan the switches for an input. When the appropriate switch is set, an encoded message is selected. The ASCII characters of the message will be sent serially to a host personal computer. The host will evaluate the message. The PC will send a certain code to the microcontroller. The microcontroller then deciphers the returned code. A decision is made based on that deciphered data.

If the code is valid, turn on or off an LED. This can represent a relay that opens / closes a door. Alternately, it can be a signal to another device.

More importantly, this procedure can also be used as a troubleshooting tool to display messages on the PC at various stages of a project.

Presentation:

Demonstrate the system. The PC with its serial communication program is provided. Each one is encouraged to have a unique implementation.

Program sample example

The exemplar program is similar to the project specifications. However, it contains elements that should be modified to complete the project as required.

```
;-----------------------------
;Program: MODRs232.asm
;Update:  29 January 2003
;Initial: 17 October 1991
;
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)1991, 2002.  All rights reserved
;
;Purpose:
;  A routine to demonstrate serial communication.
;  A single character message will be displayed on
;  the HyperTerminal or PC serial display.
;
;Processor: 8031 family
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz
;Baud:      9600
;Assembler: Intel ASM51


;################################################
;
;                     PROGRAM
;
;################################################

          org    00H
START:    ljmp  INITIAL

          org    0033H
          db    'Marcus O. Durham, PhD, PE'

;-----------------------------
          org   0080H         ;Addres past reserve
```

```
INITIAL:
;-----------------------------
                                ;INITIALIZE
          mov   SP,#5Fh         ;start stack @ 5f+1
          lcall UART            ;config & start UART

;-----------------------------
MAIN:
;-----------------------------
                                ;PROCESS
          mov   A,#3Fh          ;ASCII ?
          lcall SEROUT          ;send character
MAN9:     ljmp  MAIN            ;Repeat


;************************************************
;
;                      SERIAL RS232
;
;************************************************
UART:
;-----------------------------
;
;  Initialize the registers that control the
;  serial communications process.
;
;  Timer 1 is used in  eight bit auto-reload mode.
;  So TMOD bit M1 is set and bits
;  M1, C/T' and GATE are clear.
;  With GATE set, INT1' & INT0' control the timer.
;
;  Bit SMOD in register PCON can be used to double
;  the baud rate when set.  It's clear here, K=1.
;  If set, then K=2.
;
;  SCON is used to define serial communication
;  mode 1, eight bit UART, by setting bit SM1,
;  and enable reception with bit REN set.
;
;  Bit TR1 in register TCON is set to enable
;  timer.
;
;  Set TI to indicate serial transmission is
;  complete.
```

```
;
;  Register TH1 contains the count for baud rate.
;    TH1 = 256-(K * Osc Freq / 384 * baud rate)
;
;  For 11.059 MHz oscillator,
;  TH1 is 0FDH (-3D) for 9600 baud.
;  TH1 is 0E8H (-24D) for 1200 baud.

          mov   TMOD,#20H     ;Timer 1 Mode 2
;         anl   PCON,#7FH     ;SMOD = 0, K=1
          mov   TH1,#0FDH     ;9600 BAUD @11.05MHz
          mov   SCON,#50H     ;Set SM1 & REN
                              ;
          setb  TR1           ;Start timer
          setb  TI            ;last xmission thru
                              ;
          ret                 ;Return to call

;-----------------------------
SERIN:
;-----------------------------
;
;  General purpose serial receiver routine.  It
;  gets a byte from the serial buffer, SBUF,
;  converts to the needed binary form by removing
;  the leading bit, and clears the RI (receive
;  interrupt) flag that is set by the uP after a
;  full byte is received.
;
;  Clearing RI allows next byte to be received.
;  SBUF is the input and register A is the output.

          jnb   RI,$          ;xmit busy so wait.
          mov   A,SBUF        ;Get byte from SBUF
          clr   RI            ;Clear RI
;         anl   A,#7FH        ;Clr MSB for ASCII
          ret                 ;Return to call

;-----------------------------
SEROUT:
;-----------------------------
;
;  Transfer one byte out to the serial port.
;  TI is high while a transmission is happening.
```

```
;  It goes lo, when a new byte can be sent.

          jnb    TI,$          ;xmit busy so wait.
          clr    TI            ;clr for next xmit
          mov    SBUF,A        ;mov byte to serial
          ret

;************************************************
                               ;Program end
          end
```

⇐⇑⇒

23

EXPANSION LATCHES

Thought
The process of success:
Dream – overcome – success.
MOD

I/O expansion port

What do you do when you want to hang more devices on the microprocessor? The standard chip is configured with four parallel ports. However, these registers have many other functions. Consequently, it is very easy to run out of pins for connections to peripherals.

One choice is to place latches on one of the ports and select the latches with control lines. This is very straightforward, but three design constraints must be considered.

First, only four loads can be placed on a port because of fan-out constraints. However, if they are not simultaneously selected, the port sees only those that are active. Second, it takes additional port pins for the control lines. The number of pins can be improved by using a 2:4 or 3:8 decoder. That allows connection to more devices with fewer lines. A similar function can be created in a programmable logic device. Third, port expansion requires several lines of code to invoke the control lines and the resulting latch.

I/O expansion memory

A seemingly infinite number of devices can be added with memory-mapped input / output. In essence, a latch is placed on port 0 and is addressed exactly as if it were data memory. Two design configurations must be considered.

First, only a limited number of latches can be placed on port 0 because of fan-out considerations. Second, the address must be decoded.

The most efficient application of memory-mapped input/output would be to use low memory. However, that requires numerous address lines. The more common approach is to use address A15 as a control line for external input / output. This will give up to 32K addresses. However, it restricts data memory to the lower 32K addresses. Seldom is that a problem.

In addition to the address, a control line is required to activate the latch. The read not (/RD) or write not (/WR) line *must* be ANDed with the address. These lines are pulled low when the data is on the port.

Latch in/out connection

Two latches are inherent for an effective system. The key latch has lines for output of column and input of row lines. The display-out latch has eight data lines for display devices. The routines for these latches are common to many procedures.

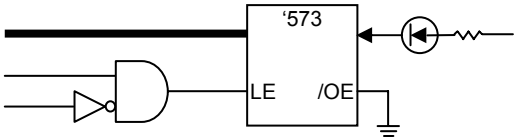
The latch is typically a D-type flip/flop. A common device is a 74573. The chip has eight transparent gates with two control lines. This chip has the input pins on the left, and the output pins directly across on the right side. The latch family should be ALS or CMOS type, since they have less fan-out load.

When the latch enable (LE) is asserted high, the flip/flops are transparent. The data is simply received by the flip/flops and clocked through to the output. When the latch enable (LE) is asserted low, the data on the input is trapped in the latch.

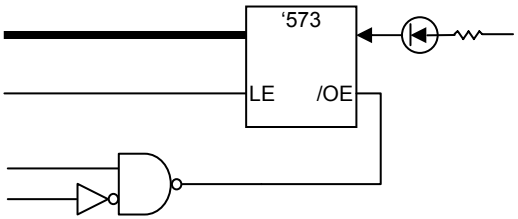
When the output enable not (/OE) is asserted low, the data on the flip/flops is placed on the output terminals. When the /OE line is asserted high, the output pins of the latches are high impedance. Therefore, it does not interfere with other lines connected to the same terminal pins.

When doing an output from the processor through the latch, the output enable not (/OE) can be permanently wired low. When doing an input from the latch to the processor, the latch enable (LE) line can be permanently wired high.

The programmable logic must make the latch enable line high then low. This may be done with hardware, but it is preferably done in software using the select lines. The pins must be set high, then cleared low for the latch to hold.



An input latch uses the output enable not (/OE) line. The line must be asserted low by the processor before the value is read. So that data is always available, the latch enable line is wired high, making the latch transparent.



Latch in/out code

The latches can be connected as expansion registers on any one of the ports. Consider using port 1 for the data. For latch select lines, use 3 pins from Port 3 (0B3-0B2h).

Function	Select B4B3
unused	00
Keypad	01
Display out	10
ADC	11

The unused latch or an output latch is chosen as the default so the latch enable line can be taken high then low. An input latch should not be selected except at the time data is desired. Otherwise, it may bias data on the port.

```
;-------------------------------------------------
LKEY:
;-------------------------------------------------
;  Latch for keys. Select as #01 on B4,B3.

          setb  0B3h          ;key-in latch
          clr   0B4h          ;OE active
          mov   A,P1          ;get row in
          ret


;-------------------------------------------------
LOUT:
;-------------------------------------------------
;  Latch for output. Select as #00 on B4,B3.

          mov   P1,A          ;set-up latch
          clr   0B3h          ; out latch
          clr   0B4h          ;LE hi
          setb  0B4h          ;LE lo
          ret
```

```
;-------------------------------------------------
LDISPLAY:
;-------------------------------------------------
;  Latch for display out. Select as #10 on B4,B3.

                              ;byte OUT
          mov   P1,A          ;data to display
          clr   0B3h          ;display-out latch
          setb  0B4h          ;LE hi
          clr   0B4h          ;LE lo
          ret
```

An alternative addressing technique, memory mapping, is frequently used to select latches. This is illustrated in the next chapter.

⇐⇑⇒

24

MEMORY-MAPPED INPUT AND OUTPUT

Thought
Doing the same thing and
expecting different results is ignorance.
Uncle Albert Einstein

Accessing external data

The external memory can be up to 64K. The addressing is on port 0 for the low byte and port 2 for the upper byte. Two techniques can be used to access the memory.

Eight-bit addressing can be use to access the lower 256 bytes. The address is stored in R0 or R1. There are four banks selected by the process status word (PSW) register. Therefore, up to eight banks can have an address.

The sixteen-bit expansion of that technique simply uses the same low byte bank registers, R0 and R1. The upper byte addressing is placed on port 2 as if it were a special function register. This gives many addresses that can be held without swapping data in the registers.

The more common approach to sixteen-bit addressing is to use the data pointer register (DPTR). The DPTR is the only two-byte latch in the special function registers. It consists of a low byte (DPL) at address 82h and a high byte (DPH) at 83h. These may be

individually loaded or the full address may be stuffed into the DPTR.

The same register is employed for data memory and code memory. Therefore, it must be changed frequently. That requires additional temporary locations to hold the last values.

The instruction

Only one type instruction is available to transfer data with external memory. That is movx. The instruction may be executed with the eight-bit R0 / R1 format or the sixteen-bit DPTR structure. The instruction is only a single byte.

The mode of addressing is referred to as register-indirect addressing. In essence, the address is placed in the register. Then the data at that address is exchanged with the accumulator.

```
E0        movx  A,@DPTR       ;info @address to A
F0        movx  @DPTR,A       ;info @addres from A
E2        movx  A,@R0         ;info @address to A
F3        movx  @R1,A         ;info @addres from A
```

The memory address to be accessed is stored in the register. On execution of the instruction, the computer goes to the register, gets an address for the instruction, and then transfers information at (@) that address. In order to employ the indirect addressing mode, the address must be stored in the register before the movx instruction is executed.

The instruction activates either the read not (/RD) or the write not (/WR) line. These lines are active low when the data is available at the port.

When moving data to a register in a bank, the opcode is a one-byte instruction that contains the register number. The first nibble is the type instruction. The second nibble starts with 1 followed by the binary equivalent of the register, xxxx1rrr.

The setup

When using the bank registers, the bank must be selected first. Next, the address is placed in the register. Then the transfer is made with the external location.

```
          clr   0D4h          ;PSW.4 bank 0
          clr   0D3h          ;PSW.3 bank 0
          mov   R1,#129       ;address= 129
;         mov   P2,#page      ;optional paging
          movx  A,@R1         ;contents of 129 →A
```

The data pointer is considerably easier to setup. As a result, it is the preferred technique.

```
          mov   DPTR,#8000h   ;address= 8000h,A15
          movx  A,@DPTR       ;contents of 8000h→A
```

The generic microprocessor has a single DPTR. A very common and useful enhancement to derivative machines is the addition of a second data pointer register.

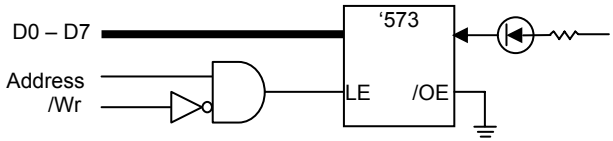
The hook-up

A latch or D-type flip/flop is commonly applied for the memory-mapped input / output interface. A common device is a 74573. Pin-out details are given in the reference section and a detailed operation is exhibited in the expansion latch chapter. This chip has eight transparent gates with two control lines.

When read or write lines are used for the latch enable (LE), the process is very simple. The control line must be high during setup. However, the read or write line is asserted low. Therefore, it must be inverted before connecting to the latch enable. Then the latch

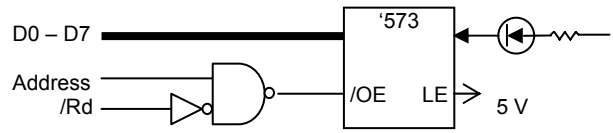
traps the data when the read or write line is released to high, which makes the LE go low.

When doing an output to the latch, the output enable not (/OE) can be permanently wired low. The latch enable (LE) is controlled by the and of the address with the inverted write not (/WR) line.



When doing an input from the latch, the process is slightly more complex. The output of the latch is not permitted on the port except when the port is ready to read. Otherwise, the load of the latch would influence other inputs.

The latch output enable not is selected by the and-invert of the address with the inverted read not (/RD) line. In addition, the latch enable is wired high.



The connection of the addresses with the read not or write not line is accomplished in a programmable logic device (PLD). Therefore, each of the two diagrams simply become an equation line of code in firmware. The firmware latch is illustrated in the chapter on programmable logic devices.

When the control is to the latch enable (LE) line, the device is self latching simply by the transition of the read not (/RD) or write not (/WR) line that is connected to the latch.

Latch in/out memory-mapped

The previous chapter illustrated latch selection by a port. A common alternative is to connect external latches as memory-mapped devices. Address A15 is frequently used as a simple decode address select line. When ANDed with other address lines, 32K locations can be used for expansion.

A couple of the common function latches are selected by these addresses. Other addresses are used to drive particular lines high or low, but these are not connected to drive a latch or flip/flop

Function	Address
Memory map	8000h
Key Oen & Key LE	8001h
Display LE	8002h
	8003h
	8004h

The latch controls circuits can be glue logic. However, to save space and increase flexibility, a programmable logic device is preferred. The latches that require the data to be held can be locked or gated with firmware in the PLD.

```
;-------------------------------------------------
LKEYIN:
;-------------------------------------------------
;  Latch for key in.

          mov   DPTR,#8001h   ;key-in latch
          movx  A,@DPTR       ;get row in
          ret


;-------------------------------------------------
LDISPLAY:
;-------------------------------------------------
;  Latch for display out.

                              ;byte OUT
```

```
mov   DPTR,#8002h   ;display out latch
movx  @DPTR,A       ;send column out
ret
```

It is very apparent that memory-mapped is simple and easily implemented in the software. However, it does require slightly more hardware or firmware to obtain the address combinations. This arises because an address select line and read or write lines must be ANDed with the lower address bits.

⇐ ⇑ ⇒

25

PROJECT 7 – I/O EXPANSION

Thought
Listening is
asking open ended questions.
MOD

Project 7: Unlimited I/O _____

Purpose: To expand I/O system.
To decode an I/O port from memory space.

Preamble:

In a common microprocessor system, the I/O system provides the interface to the real world. In many microcomputer systems, the I/O port is accessed by special instructions such as IN and OUT. However, an I/O port may be treated as external memory. This technique is called memory-mapped I/O. The I/O device is accessed like an external data memory location.

The I/O ports are already built within the chip. Moreover the ports are an internal memory-mapped type.

To obtain additional external ports simply add a latch.

Plan:

For this project, implement the T-Bird Tail Light System at the memory-mapped locations.

Preparation:

In this project, the I/O system will be changed, rather than using an individual I/O port. Decode the input and output system as a memory location on the same lines as the external SRAM. Memory mapping is accomplished by wiring a latch to the address / data lines.

The latch is enabled by an address line combination from the PEEL. Connect the address / data lines of port 0 to the corresponding latch data pins.

Decode the address through the PLD or discrete logic. Generally, the address is selected from the higher order address (Port 2) lines. AND the decoded address with the write not (/WR) line from the processor. Remember that address lines are active high and the latch enable is active low. Therefore, the output must be inverted.

Alternatively, a latch can be used for an input. The only change is the address AND is with read not (/RD) rather than write not.

Some latches are used bi-directionally. These would simply have the decoded address line connected, without either read or write.

Procedure:

Use the 74573 as a latch to hold the data for the display. Rewrite the previous display program so that it will run at the new addresses. Remember, the previous display was handled as a port location. If it were written with appropriate subroutines, only two lines of code will change.

The new display address will be a memory location. Hence the DPTR must be initialized every time anything is written to the display system. The display should be LEDs connected to the output of the latch.

Presentation:

Show that the memory-mapped display system is working properly. Display various values on the relocated output LEDs.

Program sample example

The exemplar program is similar to the project specifications. However, it contains elements that should be modified to complete the project as required.

```
;Program: MODmmio.ASM
;Update:  2 August 2004
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)1991 - 2004.  All rights reserved

;################################################

;                     ASSIGNMENTS

;################################################
AdSeven   equ   8002h;MMIO address for Seven Seg

;################################################

;                     PROGRAM
```

```
;################################################
          org   00H
START:    ljmp  INITIAL

          org   0033h
          db    25, 1,'Marcus O. Durham, PhD, PE'

;------------------------------------------------
          org   0080h         ;get past interrupt
INITIAL:
;------------------------------------------------
          mov   SP,#5Fh       ;start stack @ 5f+1

;------------------------------------------------
MAIN:
;------------------------------------------------
                              ;DISPLAY
MAN1:     mov   A,#55h        ;turn on
MAN2:     lcall OUT           ;output seven seg
          mov   A,#0AAh       ;alternate segments
          lcall OUT

          sjmp  MAIN

;------------------------------------------------
OUT:
;------------------------------------------------
;  Create a loop to display latch.

          mov   DPTR,#AdSeven ;address
          mov   R2,#200       ;persistence loop
ZDEL2:    movx  @DPTR,A       ;show it again
          djnz  R2,ZDEL2
          ret                 ;Return to call


;************************************************
          end
```

26

TABLES

Thought
A project has four seasons-
plan, act, reap, reward
MOD

Data in code memory

Some data is fixed. Therefore, the values can be permanently stored in memory. This characteristic is more like program memory than data memory. Often, fixed data is embedded in the program code.

What are these kinds of data? Message strings are one common example. Tables are another. Tables can be used to quickly translate values. For example, numbers shown on a seven-segment display have a unique pattern. The pattern can be stored in the table to correlate with a number.

Memory	Pattern	Base + Offset	Number
200	3fh	200 + 0	0
201	06h	200 + 1	1
202	5bh	200 + 2	2

The pattern can be easily referenced if the first number is assigned to a memory location called a base value. Then each other number is an offset from the base. The base value can be stored in the data

pointer register (DPTR) or the program counter (PC). The index is stored in the accumulator register. This is called base plus index register addressing.

When information is transferred from data memory, the instruction code is movx. The data table is stored in program memory rather than data memory. Therefore, a unique transfer code is required to load the data pattern.

The only instruction available for obtaining program code is movc. It creates a code memory address by adding the base register, DPTR or PC, and the accumulator as an index register. The contents of the code address are moved to the accumulator.

```
movc   A,@A+DPTR      ;[A + DPTR] →A
movc   A,@A+PC        ;[A + PC] →A
```

Data byte

Fixed data will be stored as code in program memory. Data is stored directly by setting the byte during machine language programming. Rather than an instruction, directives tell the assembler that the information is data. The directive type is Define (D). The directive can reserve bytes (db), words(dw), or space(ds).

```
          org   1000h         ;start of table
RESERVE:  db    3Fh           ;insert list of byte
          dw    1234h         ;insert list of word
          ds                  ;reserve space
```

In reality, the Define Byte (db) has become the directive that all assemblers use and is the only one required for programming. By placing a string of bytes, words are automatically defined. If numbers are used with the byte, they are stored directly. If ASCII characters are stored, they are included in a single quote.

The table that is illustrated above can be placed directly into memory. This is the list of values that represent bits of a seven-segment display.

```
          org   1000h          ;start of table
TABLE:    db    3Fh            ;a '0'
          db    06h            ;a '1'
          db    5bh            ;a '2'
```

To load the data, the corresponding number must be loaded into the accumulator (A) register. The base determined by the org must be loaded into the data pointer register (DPTR).

```
          mov   DPTR,#1000h    ;base of table
          mov   A,#02h         ;number seeking 7seg
          movc  A,@A+DPTR      ;[1002]= 5bh →A
```

Be cautious that program instructions do not infringe into the table area. Since the table is in program memory, the computer would try to execute the byte as an instruction. The 3Fh would execute as addc.

The /EA pin must be connected low to ground in order to access external memory. If the program is strictly internal, then the pin is left high.

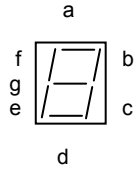
Characters available table

A seven-segment display obviously has only a limited number of elements. These are arranged in a square pattern. Therefore, characters that require a diagonal are not possible. Furthermore, the limited arrangement prevents creation of all possible letters. Some letters can only be shown as lower and some as upper case. Nevertheless, all numbers can be recognized.

The following table illustrates the available letters and numbers that can be built.

Character	Pattern
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D
7	07
8	7F
9	6F
A	77
C	39
E	79
F	71
H	76
I	06
J	1E
L	38
O	3F
P	73
S	6D
U	3E
Y	6E
b	7C
c	58
d	5E
g	6F
h	74
i	04
n	54
o	5C
r	50
t	78
u	1C
?	53
-	40
_	08

The hexadecimal pattern corresponds to the bit pattern for the seven-segment display. Segment a is the top bar on the display. The details are shown in the reference section.



It is assumed that segment a is connected to bit 0 of the output port latch.

bit	7	6	5	4	3	2	1	0
segment	dp	g	f	e	d	c	b	a

Movx vs. movc

Two transfers from external memory are available. They look similar, but the registers are very different. movx moves information to and from data memory. movc moves data from program memory. movx is simply indirect addressing, while movc is index plus indirect addressing.

The string of code provides a parallel comparison to the structure of the instruction.

```
                          ;EXTERNAL CODE
mov    DPTR,#1000h        ;base of table
mov    A,#02h             ;number seeking 7seg
movc   A,@A+DPTR          ;[1002h]= 5bh →A

                          ;EXTERNAL data
mov    DPTR,#08000h       ;memory-mapped I/O
movx   A,@DPTR            ;[8000h] →A
```

Code messages

Another common use of code memory is to store a message string. These messages may be for serial communications or displays. The message is typically stored with ASCII characters rather than hexadecimal numbers. Nevertheless, to the processor they are just bits.

The structure for storing and retrieving the message string is precisely the same as numbers. Since it is ASCII, the data is contained in a single quote. One item that appears different is simply a style issue. That is all bytes of the string are stored in one line.

```
          org   1010h          ;start of table
MESSAGE:  db    'Message to display'
```

The string is read as before. One additional requirement is the string is read with a loop, since only one byte is obtained at a time.

```
          mov   DPTR,#1010h    ;base of table
          mov   A,#0           ;read initial byte
STRING2:  movc  A,@A+DPTR      ;input byte
          lcall SEROUT         ;ship the byte
          djnz  LoopC,STRING2  ;go thru loop
```

The coded messages are a very powerful capability that permits common language communications, rather than terse signals.

By using code memory to contain table for conversions and messages for display, less data memory is required. These defined byte data take very little space compared to the program memory. Nevertheless, they do dramatically improve the interaction with the processor.

Enhanced serial messages

By use of tables, messages can be significantly enhanced. This allows communications to look very normal. The routines can be added to any program for display.

```
;-------------------------------------------------
DISPLAY:

          lcall UART         ;initialize serial
          mov   DPTR,#ONLINE ;serial message
          lcall BIOSSER      ;send serial message
          lcall SERCOMM      ;send to serial

;-------------------------------------------------
BIOSSER:
;-------------------------------------------------
;Send a string to the serial port.

          mov   A,#0          ;read length string
          movc  A,@A+DPTR     ;input byte
          mov   LoopC,A       ;setup loop

BIOL2:    mov   A,#1          ;get 1st message
          movc  A,@A+DPTR     ;input byte
          mov   CharL,A       ;character icd
          lcall SEROUT        ;send out byte

          inc   DPTR          ;restore offset
          djnz   LoopC,BIOL2  ;go thru loop

          ret

;-------------------------------------------------
SERCOMM:
;-------------------------------------------------
;  Send byte on serial line.
;  Convert byte to 2 ASCII characters.
;
                              ;SAVE byte & SEGMENT
          mov   R0,A          ;hold data

          anl   A,#11110000B  ;keep high nibble
```

```
          swap  A              ;move to low
          mov   DPTR,#TABASCII;ASCII table
          movc  A,@A+DPTR      ;convert to ASCII

          lcall SEROUT         ;send first number

                               ;GET LOW NIBBLE
          mov   A,R0           ;restore frm earlier
          anl   A,#00001111B   ;keep low nibble
          movc  A,@A+DPTR      ;convert to ASCII

          lcall SEROUT         ;send second number

          mov   A,#13          ;carriage return
          lcall SEROUT
          mov   A,#10          ;line feed
          lcall SEROUT

          ret                  ;its all over

;###################################################
;             CODE-STORED CONSTANTS
;###################################################
TABASCII:
;---------------------------------------------------
;look-up table for hex to ascii conversion:
          db      '0123456789ABCDEF'

;---------------------------------------------------
;Canned message to confirm serial port activation:
Online:   db      6, 80h ,'Value=', CR, LF, 0

;###################################################
```

27

MULTIPLEXING

Thought
Our perception is an analog world,
the reality is discrete sampling.
Professor Durham

Perception

Perception is what you think. To you it is reality. However, in a multiplexed digital world that is not true.

A phenomenal tool is the eye – brain sensory system. The eye sees an object and translates the image to the brain. Actually, the eye is a sampling system rather than a continuous analog network. The brain processes the samples and gives the impression of continuous, smooth information. Technology takes advantage of this visual perception to reduce the quantity of information that must be provided.

Consider some of the sampling to which the eye is exposed, but the perception considers the data as stable. The tests are performed in frames per second. The duration of the sampling is the reciprocal of the frame rate.

Frame/sec	Image
<18	Movie has flicker
18	Motion appears fluid
24	Movie picture rate
25	Television image rate
60	Fluorescent lamp
75	Computer monitor refresh
100	Cannot detect any flicker
220	Air Force pilots identify a plane
500	No detection, but sense something not as it should be

When there are small changes near 20 frames per second, the changes appear to be fluid motion. Near 100 frames per second, the eye cannot see any transitions. However, trained pilots can detect an airplane when it is flashed before their eyes for 1/220 second. If an image is flashed occasionally, in the order of 500 frames per second, the mind never detects it. However, the brain does sense something is not quite right.

A single white flash has a residue image on the brain for an extended period of time. However, at 24 frames per second, black is imperceptible. Subliminal imagery has been used in movies at rates as low as 24 frames. These messages are overlaid on a black background. The brain perceives the message, but it does not register in the visual pathways.

Multiplex

What does the eye detection have to do with microprocessors? It has everything to do with real world data. Virtually all continuous analog signals that a person can detect operate at a rate less than 24 frames per second.

This phenomenon permits the microprocessor to sample input and output at a rate much slower than the computer is operating. Therefore, it can appear to be doing multiple things at one time.

Multiplexing is transmitting of several signals simultaneously on the same circuit. Each message appears to be individual, but all the signals are combined into one channel.

Consider a keyboard or keypad. How fast does it have to sample in order to detect every keystroke? If a very fast typist can do 120 words per minute and a word consists of 5 letters, then the fastest stroke rate is 10 strokes per second.

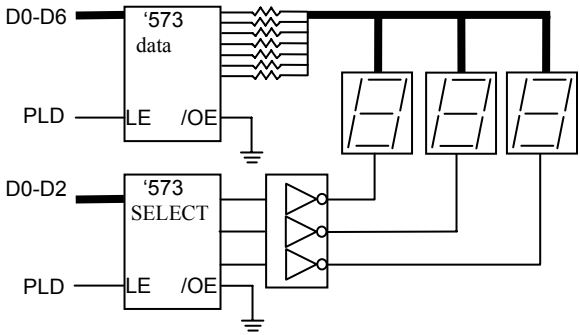
Similarly, displays that are updated more often than 25 times per second will appear to be on continuously.

Our perception is an analog world, the reality is digital sampling.

The remaining projects will tap into this phenomenon.

Circuit: displays

Seven-segment displays are commonly used to display limited data in a large format. These consist of multiple individual elements that all appear to be energized simultaneously. However, in a microprocessor, that would require too many lines and connections. Therefore, all the elements are in parallel on one port channel. Then each display is progressively turned on then off. This digital sampling creates a continuous imagery.



Seven-segment displays are inexpensive and simple to implement. Any number of displays can be used. The relatively large size makes them effective for many situations. Data is output as a byte on a port. Then other individual bits are used to select which display gets the information. These are connected to the common pin.

Each segment of a display is an LED. Therefore, there are seven LEDs in a display. There may also be another LED for a decimal point or other similar symbol.

The current limiting resistors in series with the data seven-segments are typically 270 to 330 Ohm, similar to any other LED.

The select line drivers are open collector inverters, such as 7406. These require a pull-up resistor on the output. This permits the chip to supply much more current. Alternately, field effect transistors (FETs) can be used as drivers. When all seven-segments are on, then substantial current is required through the common pin.

When a one is sent to the inverter, the output is low. If the seven-segment is common cathode, this is proper. However, if the display is common anode, the data to the inverter must be complimented.

Latches are shown on the data lines. The devices should be ALS or CMOS type, since they are less fan-out load. These latches may be connected as memory-mapped I/O. They may also be expansion latches on a port. The only difference is the control logic in the programmable logic device.

For the project, the data latch is connected to MMIO. However, a latch is not used on the select lines. Each select line is connected through a FET directly to port1.

The latches setup the output when the latch enable (LE) line is high. When the line is then asserted low, the data is trapped in the latch. That data is displayed as long as the output enable not pin is low.

Therefore, the programmable logic must make the latch enable line high then low. This may be done with hardware in the case of

memory-mapped I/O controlled by the write not (/WR) line. Alternately it is done in software if using port pins as select lines. *The pins must be set high, then cleared low for the latch to hold.*

Code requirements

Once data is displayed, it must remain energized long enough for it to be perceived by the eye. The period of time that it is maintained energized is called persistence. The delay cycle necessary to hold persistence is time dependent. Therefore, a faster machine will require more delay loops.

The display is a routine that operates in three steps that are repeated for each digit to be displayed.

1. Select the digit display line.
2. Send the data byte to the display.
3. Wait briefly for persistence of display.

The first routine (SEVEN) sets the bit that activates the common line on one of the seven-segment displays. These lines of code would be repeated for as many displays as shown. The output routine gets the byte to display, shows it and waits. The information to be displayed is stored at the location referenced by R0.

For the examples, the select lines are on port 1, but this is easily changed in the equate directives. Display A is bit 0 (P1.0), display B is bit 1 (P1.1), and display C is bit 2 (P1.2). The output is either on a port or memory mapped.

```
;-------------------------------------------------
SegA      equ   90h           ;address
SegB      equ   91h           ;address
SegC      equ   92h           ;address

;-------------------------------------------------
SEVEN:
;-------------------------------------------------
                              ;DIGIT 1
          clr   SegA          ;turnoff select line
```

```
          clr   SegB          ;turnoff select line
          clr   SegC          ;turnoff select line

          setb  SegA          ;Bit is hi, select A
          lcall SEVOUT        ;output & delay
          clr   SegA          ;Bit is hi, select A

                              ;REPEAT NEXT DIGITS
          ret
```

Code segment for port _____

The seven segment can be connected to a port or it can be memory mapped. The first section has the entire code connected to ports.

```
;-------------------------------------------------
SEVOUT:
;-------------------------------------------------
                              ;byte OUT
          mov   A,@R0         ;data byte
          inc   R0            ;next info to show

                              ;PERSISTENCE WAIT
          mov   R2,#200       ;Nested loop counter
ZDEL1:    mov   Pio,A         ;display digit
          djnz  R2,ZDEL1      ;Nested loop, 256 x

          ret
```

Code segment for memory map ____

Alternately, memory mapping can be used. The only change is the substituting the address information for the port. This is an excellent example to see the similarities between the processes. The address is used by the PLD code to select the latch. The PLD code causes the latch to be held until the next write command.

```
;-------------------------------------------------
SEVOUT:
;-------------------------------------------------
```

```
                                ;byte OUT
            mov   A,@R0         ;data byte
            inc   R0            ;next info to show

                                ;PERSISTENCE WAIT
            mov   DPTR,#8002h   ;addr for disp latch

            mov   R2,#200       ;Nested loop counter
ZDEL1:      movx  @DPTR,A       ;display digit
            djnz  R2,ZDEL1      ;for persistence
            ret
```

Binary to binary coded decimal _____

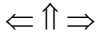
One routine often required is to convert a binary number into individual digits so they can be displayed. The digits are displayed as decimal values. The format is often called binary coded decimal (BCD).

Because of the hardware divide command, the conversion of binary to decimal is actually very straight forward, if the binary value is in a single byte.

Simply do what your third grade teacher told you. Divide by the place value to get the number for that place. The maximum number is 255. So, first divide by 100 to get the number of hundreds. Take the remainder and divide by the next place value of 10 to get the number of tens for that place. Then remainder is the units.

```
;-------------------------------------------------
BINBCD:
;-------------------------------------------------
                                ;byte BINARY TO BCD
            mov    A,GapD       ;count 2
            mov    B,#100       ;divisor
            div    AB           ;A/B, Quo= A, Rem= B
            mov    GapA,A       ;# of 100's
            mov    A,B          ;remainder
            mov    B,#10        ;divisor
            div    AB           ;A/B, Quo= A, Rem= B
            mov    GapB,A       ;# of 10's
```

```
          mov     GapC,B          ;# of 1's
BINB9:    ret                     ;out of here
```



28

PROJECT 8 - SEVEN-SEGMENT DISPLAYS

Thought
You are looking,
but are you seeing?
Dr. Eden Ryl

Project 8: Seeing what is not there __

Purpose: To design an interface which is a display system.
To fetch data from the program ROM.

Preamble:

Utilizing a software design for circuits is usually preferred to a hardware design. A software design provides more flexibility than does hardware. Software is very advantageous if the system is not time constrained.

In this project build a 3-digit, 7-segment display. Only one byte may be used to display LEDs. To display all the digits, use time multiplexing.

The multiplex cycle must be very short, so that the human eyes will not be able to see the blinking. The eye can discern any frequency slower than 17 frames per second. Therefore, refresh the display faster than 20 times a second. This task can be easily achieved, since the uC is fast enough to process the output.

The software description of a digit can minimize the calculations and simplify the system. The patterns of a letter for a 7-segment display can be stored in a table. Use sequential characters (e.g. 0, 1, 2, 3, 4...a, b, c, etc.), so it will be easier to find any item. An easy way to build a table is to program the data into the code memory. Then the program has only to fetch the data from the table to display a character.

The processor has capabilities of separate external memory for data and for program code. To access code memory use the movc command. This triggers the program storage enable not (/PSEN) pin. To access data memory use the movx command. This triggers the read not (/RD) or write not (/WR) pins. In both cases, the address is stored in the data pointer (DPTR) register. The register must be reinitialized every time it is used, if it has been modified since the last access.

Plan:

In this project, implement a 3-digit, 7-segment time multiplexed display system.

Preparation:

To drive the 7-segment displays, use a 74573 latch for the data buffer. Connect the output of the latch to the seven pins of the display. Repeat with a parallel connection to all other 7-segment displays.

The common (cathode or anode) of each display must carry the current of all the segments. Therefore, it should be connected to a driver other than the microcontroller.

One option is to use an open collector driver. The 7406 hex inverter is a popular open collector chip. The open collector must be connected to 5 volts through an external pull-up resistor. Typically

use 2.2 kOhm resistor with adequate power rating to handle the full load current.

Alternatively, a field effect transistor (FET) can be used. These have current capabilities in the 30 A range.

The driver input is a single bit that can come from any port or memory-mapped I/O location. This becomes the digit select line.

Test your display system by illuminating simple characters such as 1 or 7.

Procedure:

Write a simple multiplex display program that has the following abilities. First, turn off all digits. Second, select the first display. Third, send data for the first display. Fourth, create persistence by a very short delay of NOPs. Fifth, turn off all displays. Select the next digit. Send data for the next display. Create persistence by a very short delay of NOPs. Repeat steps five through eight for as many displays as required. As the last step, turn off all displays.

Write an input routine that calculates a hexadecimal number in the range of 00 – 0FFh, based on switch inputs. Caution: be sure that stable data is on the input before starting the calculation. Several techniques can be used. Software debounce is the easiest.

Separate the hex number into two bytes or digits. Do a table lookup to convert the digits to 7-segment values. Send the digits to the multiplex display program. Display the value for about two seconds.

Convert the hex number to a binary coded decimal value in the range of 0 - 255. This will make separate bytes for each digit of the decimal number. Do the same table lookup to convert the digits to 7-segment values. Send the digits to the multiplex display program. Display the value until the input routine calculates a new number.

Presentation:

Show at least 4 hex-number conversions. The numbers may be randomly chosen.

⇐⇑⇒

Program sample example

The exemplar program is similar to the project specifications. However, it contains elements that should be modified to complete the project as required.

The latch key out and latch display routines are given in the chapter on expansion latches.

```
;Program: ModSeven.ASM
;Update:  20 February 2003
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)1991, 2003.  All rights reserved

;Purpose:
;  Data is stored in GapA, GapB, GapC.
;  Data is displayed on 3 7-segment displays.
;  Display-out latch shows the data.
;  The 3 displays are selected by port 3.2, 3.3, &
;  3.4
;
;################################################
;                      ASSIGNMENTS
;################################################
GapD      equ   37H   ;general purpose variables
GapC      equ   36H
GapB      equ   35H
GapA      equ   34H

LoopC     equ   07H   ;loop counter
```

```
;R0        equ   00H   ;destination indirect addr

P35        equ   0B5h  ;switch input

SegA       equ   90h   ;Port 1.0
SegA       equ   91h   ;Port 1.1
SegA       equ   92h   ;Port 1.2

;###################################################
;                      PROGRAM
;###################################################
           org   00h
START:     ljmp  INITIAL

           org   0033h
           db    25, 1,'Marcus O. Durham, PhD, PE'

;---------------------------------------------------
           org   0080H         ;get past interrupt
INITIAL:
;---------------------------------------------------
           mov   SP,#5Fh       ;start stack @ 5f+1
           setb  P35           ;make input

           lcall HELLO         ;start display
           lcall SEVBCD        ;BCD to 7 segment

;---------------------------------------------------
MAIN:
;---------------------------------------------------
           lcall SEVSEG        ;output seven segmen

MAN9:      jnb   P35,MAIN      ;Repeat

;---------------------------------------------------
HELLO:
;---------------------------------------------------
;  Preload a value into the general purpose bytes.

                               ;PRELOAD
           mov   GapA,#0Fh     ;data byte
           mov   GapB,#0Ah
           mov   GapC,#0Bh
           ret
```

```
;-------------------------------------------------
SEVBCD:
;-------------------------------------------------
;  Convert the binary coded decimal to seven seg.
;  Use a table look up.

                              ;INITIAL
          mov   DPTR,#TabSeven ;seven seg table

          mov   R0,#GapA      ;base of digits
          mov   LoopC,#3      ;number of digits

                              ;CONVERT
SBCD1:    mov   A,@R0         ;get BCD
          movc  A,@A+DPTR     ;table offset to A
          mov   @R0,A         ;replace w/ 7 seg

                              ;NEXT DIGIT
          inc   R0            ;next
          djnz  LoopC,SBCD1   ;>=0, repeat process

          ret                 ;else, exit

;-------------------------------------------------
SEVSEG:
;-------------------------------------------------
;  Seven Seg is a routine that operates in 4 steps
;  1. Select the digit stored in GapS
;  2. Select the digit to turn on
;  3. Send the data
;  4. Wait briefly for persistence of the led.
;  5. Repeat

                              ;INITIAL byte LOCATE
          mov   R0,#GapA      ;least sig display

                              ;SELECT LINES
          clr   SegA          ;turnoff all selects
          clr   SegB
          clr   SegC

                              ;DIGIT 1
          setb  SegA          ;Bit is hi, select A
```

```
          lcall SEVOUT        ;output & delay
          clr   SegA

                              ;DIGIT 2
                              ;DIGIT 3

                              ;TERMINATE
          ret                 ;back to Hotlanta

;#################################################
;                    TABLES
;#################################################
TabSeven:
;-------------------------------------------------
;  Seven-segment display
          db    3Fh   ;0
          db    06h   ;1
          db    5Bh   ;2
                      ;complete the table

;*************************************************
          end                       ;Program end
```

29

MATRIX SCANNING

Thought
Integrity is
recognizing someone's short coming
and saying nothing about it to anyone.
Rosemary Durham

Matrix inputs

Although the processor is extremely powerful, one of the limits of any computer is how many things can be connected. Several techniques for expanding those options have been addressed.

An earlier chapter discussed expanding the number of devices by using expansion memory and latches. Another chapter discussed the challenges of getting more devices connected to the microprocessor with the same number of pins. The concept is called multiplexing. This chapter discusses another technique for obtaining more information with limited connections. It employs a matrix network to literally multiply the effect of pins.

Perhaps the most common method of data entry is a keyboard. A personal computer (PC) has over 100 keys. To minimize the number of wires, a microprocessor is embedded in the keyboard. This converts all the keys to just a few lines.

Although the same concept is feasible for control systems, seldom is that much data required. Another consideration is the system is often constrained to a small size. Therefore, a much more common arrangement is to use a keypad similar to a telephone. In some cases, the keypad may only have three or four keys.

Regardless of the number of keys, it is necessary for the processor to decipher which key is pushed. The simplest technique requires no logic. One wire connects to each key and the electrical common provides a return path.

```
# wires = # keys + 1
```

This technique would require way too many wires, a huge cable, and an excessive number of pins on the microprocessor. An alternative is to use a matrix. A wire is connected to each row and a wire is connected to each column.

```
# wires = # rows + # columns
```

Compare the techniques for a 3x4 telephone keypad with three columns and four rows for a total of 12 keys.

```
# wires = 12 +1 = 13        ;no logic network
# wires = 3 + 4 = 7         ;matrix network
```

A 4 x 4 keypad will add four more keys put will require only one more wire. That is a huge gain.

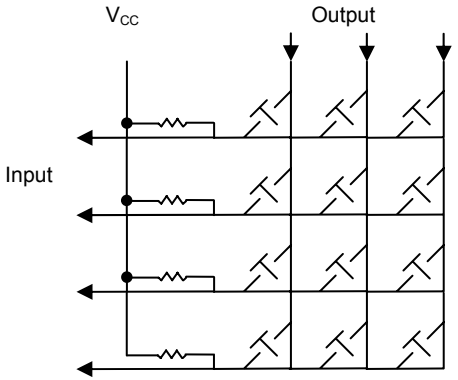
The matrix requires rather extensive logic to decode the network. Software is always preferred to hardware. Less real estate is involved and the coding must only be done one time.

Contact arrangement

The switch contacts for a matrix connect between a row and a column. Whenever a particular switch is depressed, the row and

column become shorted together. There is only one combination of row and column for any switch.

Since the keypad is a switch, it is connected exactly like any other switch. A pull-up resistor is connected to the open side of the switch. That contact becomes the input to the computer. By convention, the rows will be used as inputs. As a result, a pull-up resistor should be connected between each row and V_{CC}. Therefore, when a row is read, the value is '1' if no key is pressed.



The columns are applied as outputs. When a zero or ground is placed on a column, then the switch looks like a traditional connection.

The columns are configured to write through a latch. The output enable not can be permanently active by connecting it to ground. The latch is set up with data when the latch enable pin is asserted high by the processor. Then, when the pin is asserted low, the latch traps the data.

The rows are configured to read through a latch. Since this is an input, the latch enable line can be pulled high. When the output enable not (/OE) line is asserted low by the processor, the data will be input to the computer.

Conflicts

If the project were only that easy, anyone could do it and there would be no need for a design engineer. However, there are several opportunities for improvement.

First, switches will bounce when they are exercised. This will cause the impression of multiple key depressions. Hardware triggers can be used, but they are expensive. Software is the best solution. The logical exclusive-or instruction makes the problem almost trivial. The chapter on switch inputs and logic gave an illustration of the procedure.

Another decision process involves more than one key depressed at the same time. Resolution of the keys can be made in at least four ways.

1. n-key lockout on first. The first key depressed is recognized and all others are locked out. A variation is to only recognize the first key decoded.

2. n-key lockout on last. The last key released is recognized and all others are locked out.

3. n-key rollover. All keys are recognized and placed in a first in-first out (FIFO) memory until the computer can accept the data.

4. alternate key. The combination of keys are recognized as another key. A common example is shift and alternate in combination with the others.

Key debounce

The keys are typically a mechanical switch contact. As such, they will vibrate when depressed. Because the computer is faster than the bounce cycle time, the processor will detect multiple changes and interpret this as multiple keys pressed. Hardware can be used to filter the bouncing. However, software is much cheaper in terms of

real estate and investment. Although the topic was introduced with switches, more detail is investigated now.

Many procedures can be used to eliminate the bounce. The simplest is to use a logical EXCLUSIVE-OR to see if a bit has changed from the last test. If it has changed, the bit is unstable, therefore, invalid. If the bit has the same value on two passes, it is assumed it is stable and has been debounced.

The routine is a very elegant procedure to see if an entire byte is stable. This obviously could be modified for individual bytes. However, for keypad decoding, we need a stable bit.

Three values are used – the input, previous character, and debounced character. The previous character is updated with the input on every pass through the test. However, the debounced character is updated only on a stable input.

```
;-------------------------------------------------
DEBOUNCE:
;-------------------------------------------------
                                ;CHECK CHANGE BY xrl
          mov   B,A             ;hold the input
          xrl   A,CharP         ;exclusive or
          jnz   DEBN1           ;<>0, so a change

                                ;KEEP DEBOUNCED
          mov   CharD,B         ;0=no change, CharD
DEBN1:    mov   CharP,B         ;not debounce
          ret
```

Decipher

Jethro Bodine on the old television program *Beverly Hillbillies* often referred to ciphering. He was bragging about arithmetic calculations where he took inputs and determined the results. The deciphering procedure is the opposite. It is taking the results and determining the inputs. The objective of this procedure is to decipher a key based on the inputs.

The row lines are connected through a pull-up to give a one from the 5 volts. Columns are used to output a zero from the processor. These are not connected to a pull-up.

After a zero is sent to a column, the row is read. If the row is one, then a key is not pressed. If the row is zero, then a key has been pushed.

A single port can be used for a 4 x 4 keypad with sixteen keys. Columns are connected to the high nibble and rows to the low nibble. Even when a single port is not used, the same pattern is employed.

Complete solution

The process is a very sophisticated procedure that can be expanded to any number of keys. It solves the problem of multiple keys and uses highly developed decoding techniques. It is very powerful code. Unfortunately, with more power, comes more complexity.

The first step in implementation of the keypad is obviously construction. The next step is a very important tool to minimize wiring problems. Run a KEYTEST program similar to the one shown. The column output and the row input can be easily modified, if memory mapped is not used as shown in the example.

The KEYTEST routine will check for proper wiring and connections. After verification, implement the decipher routine in steps, so it can be properly debugged.

Connections

The rows can be connected to ports or latches that are associated with a port or memory-mapped. The columns can be similarly connected. The only thing that changes with the connection is a few lines of input/output code, which is used to select the control lines

for the latches. The remainder of the process remains the same, regardless of the I/O connection.

For the particular process at hand, the row lines connect to a latch that uses memory mapping. The technique has been discussed extensively in other sections. Columns are connected to a separate latch.

Memory mapped I/O shares space with the upper byte of memory addressing. However, because of the connection configuration, there is not a conflict. Address A15 selects the MMIO, so this line is not connected to memory.

A15	A14	A13	A12	A11	A10	A9	A8
Mmio							

The read and write address can be the same, since the address is ANDed with the read not or write not line from the processor. The addresses are selected using the data pointer register (DPTR). DPTR is a sixteen-bit location that is comprised of a high (DPH) and low (DPL) register. The high is associated with port 2 while the low is allocated to port 0.

The programmable logic device (PLD) firmwire enables the key row read latch at address 8001h. If an alternative is desired, the firmware can be changed. Another technique is to use the MMIO line from the PLD, then AND the line with an address that can be decoded from the microprocessor lines.

Test code

The test code illustrates the basic concepts of deciphering a matrix. The subroutine will determine that some key has been exercised. However, it does not tell which one. That requires considerably more logic and the topic will be addressed later. RUN KEYTEST TO VERIFY THE KEYPAD WIRING!

```
;-----------------------------
KEYTEST:
;-----------------------------
;  Output '0' to columns.
;  Read rows, if any is '0', a key is pressed.

                              ;COLUMN OUT
          mov   DPTR,#8001h   ;keypad address
          mov   A,#0          ;key enable
          movx  @DPTR,A       ;output columns
          movx  A,@DPTR       ;read rows

                              ;DECIPHER push
          anl   A,#0Fh        ;mask hi nibble
          cjne  A,#0Fh,KEYT2  ;<>1, so key pushed
          sjmp  KEYTEST       ;no key

KEYT2:    cpl   P35           ;LED change=key push
          lcall WAIT          ;delay multiple keys
          sjmp  KEYTEST
```

Decode flowchart

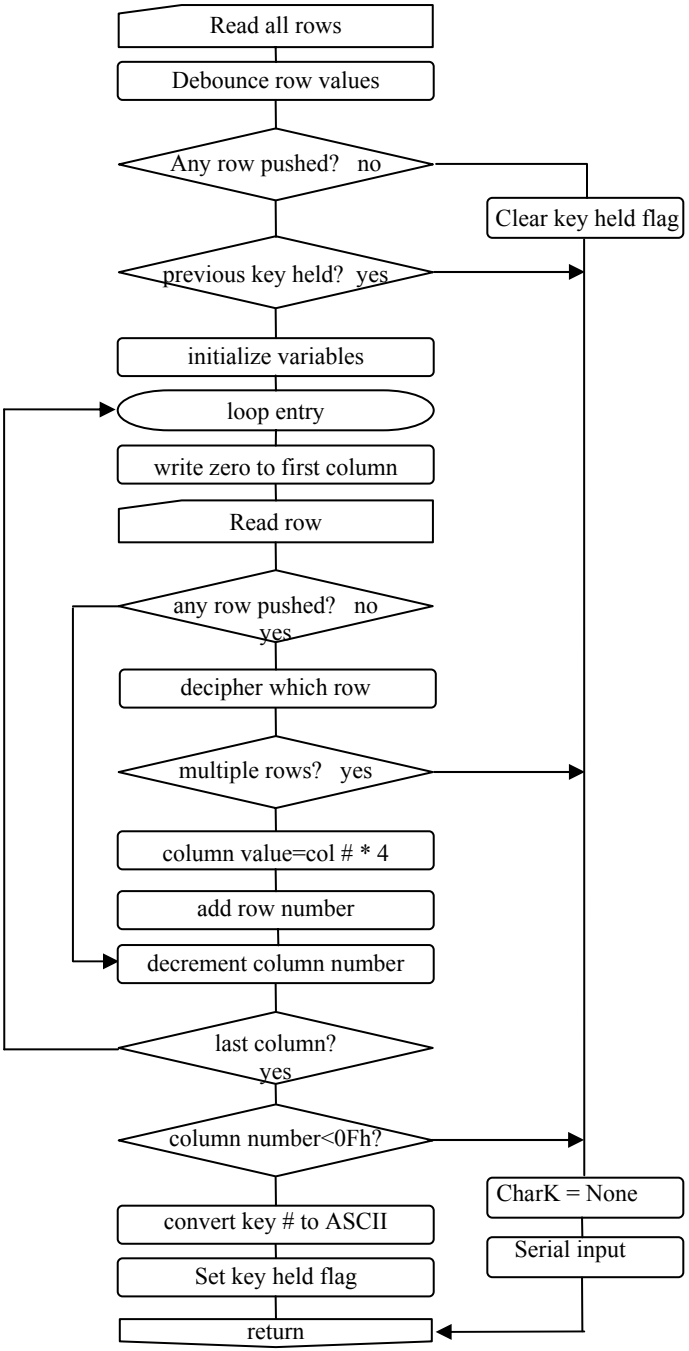
Several steps are required to decode a keypad. The overview gives a sequence of items required.

1. Read all rows simultaneously.
2. Debounce the row values into CharD.
3. Check CharD to see if any row is pushed.
4. If none pushed, clear hold flag, and exit with “None”.
5. If pushed, check if held down from a previous pass.
6. If held, exit.
7. Else, initialize variables.
8. Begin loop based on number of columns.
9. Write a zero to the first column.
10. Read rows.
11. If no row pushed, go to next column in loop at 8.
12. If pushed, decipher which row.
13. Check if multiple rows.
14. If multiple, then exit.
15. Calculate value of column, 0-3 and multiply by 4 for rows.

16. Add row number to get assigned number for key pushed.
17. Continue loop for next column.
18. Check if number is in legitimate range.
19. Convert key number to ASCII value, via table lookup.
20. Set flag for key is held.
21. Return

The list documents that the procedure is as complex as any routine encountered. No specific item is difficult. However, the number of items and the interaction makes a large number of operations.

The list is also shown in a flowchart format below. The information is identical.


Read all rows
Debounce row values
Any row pushed? no
Clear key held flag
previous key held? yes
initialize variables
loop entry
write zero to first column
Read row
any row pushed? no
yes
decipher which row
multiple rows? yes
column value=col # * 4
add row number
decrement column number
last column?
yes
column number<0Fh?
CharK = None
convert key # to ASCII
Serial input
Set key held flag
return

Keys procedure

The Keys routine directs the traffic for determining the key that is pressed. The first task is to check if any key is pressed. To do so, read the row and debounce the value. Enter the routine with the value in the accumulator. Perform an exclusive-or with the previous value, CharP. Exit the task with the debounced value in CharD.

Then use the procedure in the flow chart to decode the row and column value. The routine waits until a key is pushed before returning.

Row lines are connected to a pull-up to give 5V. The routine outputs a zero to the columns. Then read the rows to determine if any key is '0'. If so, it is pressed.

Rows are numbered 0,1,2,3. Columns are numbered 0,1,2,3. However, columns are valued based on the number of rows.

Columns value = column number * number of rows

The normal configuration is four rows.

A = KeyCol * 4

Therefore, columns are valued 0,4,8,12. Then the row number is added to give a location associated with each key. A table is used to decode the value of each key location to ASCII.

For example, the key associated with the number “8” is pushed. This is in column 1, row 2. The deciphering will return a column location of 4 plus a row location of 2 for a key location of 6. The 6 is the offset pointer into a table location that will return andASCII 8.

If no key has been pressed, the procedure contains a serial routine at the end. The routine checks if any external key has been pressed, in lieu of the keypad.

The table look-up is dependent on the wiring of the columns and rows. If an alternate hook-up is used, the table sequence will need to be modified.

Simple solution

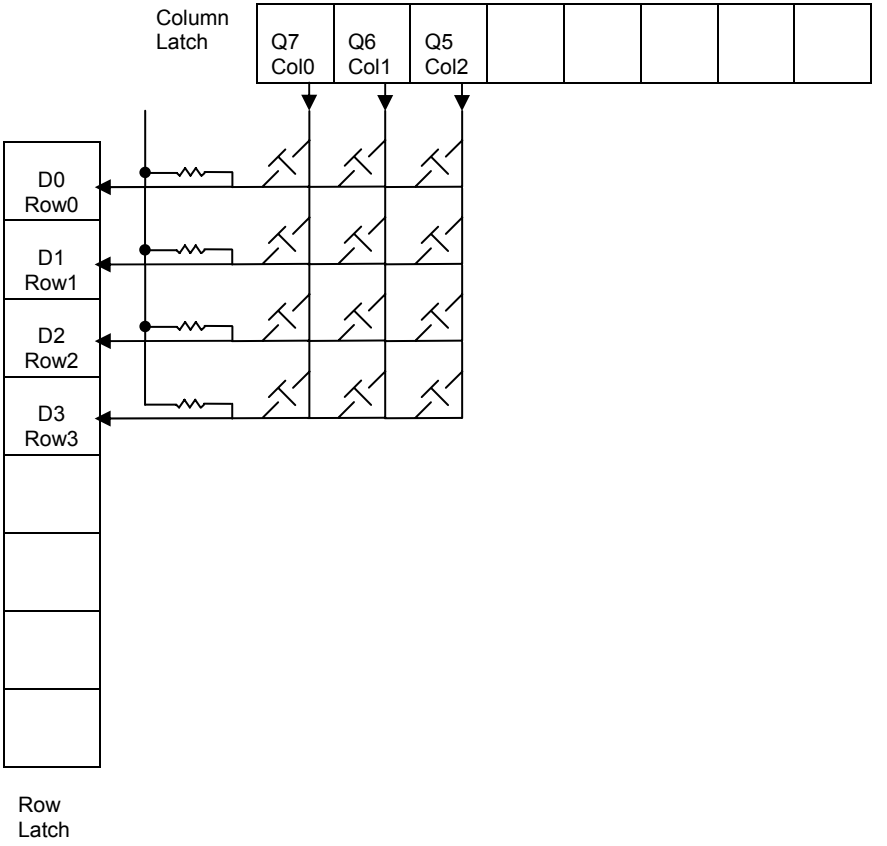
A very simple sequence of code can be used when only a limited number of keys are involved. This routine does not have a debounce routine, keyheld detection, multiple key count, range check, or serial check. These can obviously be added. However, at that point, the routine begins to be very close to the full blown exemplar program referenced in the project.

```
;################################################
;CONSTANTS
;------------------------------------------------
None      equ    0FFh  ;blank key
AdKey     equ    8001h ;mmio latch

;------------------------------------------------
;DEFINED VARIABLES
;------------------------------------------------
                       ;KEYS
KeyCol    equ   17h    ;present column number


;************************************************
;                      KEYPAD
;************************************************
KEYS:
;------------------------------------------------
;  Keypad is used to decode a matrix set of keys.
;  A 4X3 keypad can be used.
;
;  Columns connect to a latch, in the upper bits
;  Rows connect to a latch, in the low bits.
;
;  Column latch bits |7|6|5|4|3|2|1|0|
;  Key column        |0|1|2|3|-|-|-|-|
;
```

```
;  Row latch bits     |7|6|5|4|3|2|1|0|
;  Key row            |-|-|-|-|3|2|1|0|
;
                                ;COL 0
          mov   A,#01111111b    ;Column 0
          mov   KeyCol,#0       ;column location
          lcall KEYROWRD        ;input row
          cjne  A,#0Fh, KEYP7   ;<> F, so pushed

                                ;COL 1
          mov   A,#10111111b    ;Column 1
          mov   KeyCol,#4       ;column location
          lcall KEYROWRD        ;input row
          cjne  A,#0Fh, KEYP7   ;<> F, so pushed

                                ;COL 2
          mov   A,#11011111b    ;Column 2
          mov   KeyCol,#8       ;column location
          lcall KEYROWRD        ;input row
          cjne  A,#0Fh, KEYP7   ;<> F, so pushed

          mov   A,#None         ;null key
          sjmp  KEYP9

                                ;COLUMN VALUE
KEYP7:    add   A,KeyCol        ;A=KeyCol*4 + KeyRow

          mov   DPTR,#TabKey    ;start of table
          movc  A,@A+DPTR       ;conversion

                                ;TERMINATE
KEYP9:    ret                   ;back to message

;-------------------------------------------------
KEYROWRD:
;-------------------------------------------------
;  Output column, read row
;  Determine row number

                                ;COL OUT, ROW IN
KEYROW1:  mov   DPTR,#8001      ;pointer to latches
          movx  @DPTR,A         ;read row
          movx  A,@DPTR         ;read rows
          anl   A,#0Fh          ;mask hi nibble
```

```
          cjne  A,#0Fh, KEYR9 ;<> F, so pushed

                              ;INITIALIZE
          clr   C             ;initialize test bit
          mov   B,#0          ;row number

KEYR4:    rrc   A             ;row bit to C
          jnc   KEYR3         ;C=0,row down
          inc   B             ;next row
          sjmp  KEYR4         ;next bit

KEYR3:    mov   A,B           ;row number

                              ;TERMINATE
KEYR9:    ret                 ;back to message

;-------------------------------------------------
TabKey:
;-------------------------------------------------
;  Three column keyboard

          db    '147*2580369#'
```

Circuit: keypad _____



⇐ ⇑ ⇒

30

PROJECT 9 - KEYPAD

Thought
The key to success?
The Golden Rule

Project 9: Debounce & matrix inputs

Purpose: To implement a matrix scanning mechanism.
To design a keyboard for a computer.

Preamble:

A keyboard is the most common input device for a computer system. The common ASCII keyboard usually uses a dedicated microcontroller to implement the task. The microcontroller reduces significantly the number of ICs and the resulting cost. The decoded value of each key is transferred serially to the main processor.

There are several problems with a mechanical keyboard that the interface must resolve. For this project, the microcontroller is expected to do all these tasks.

Any mechanical switch will bounce when it is pressed. A software debounce may be implemented for most systems. Any actual real time function is generally quite slow. For example, a very fast typist will only type about 120 words per minute or 10 new keys per second.

Second, the priority of the pressed keys must be established. In the dedicated microprocessor system, the priority is determined by a first-come, first-serve basis. Hence, the microprocessor will also act as a buffer. In this project, priority buffering is not a problem, since the system will not be dedicated to the keyboard. Nevertheless, a simple buffering will be implemented for future use.

Plan:

Implement a 16-key keyboard system, using a 4 X 4 keyboard matrix. Only one eight bit I/O bi-directional port is required, if using ports. By simply masking one of the columns, a 3 X 4 keypad can be used.

Preparation:

Use one of the microprocessor ports, if it is available. Alternatively use a latch for the row and a latch for the column at a memory-mapped I/O location.

Procedure:

Send a 4-bit output to the keyboard. Then read another 4-bits from the keyboard to determine which key is pressed.

Implement the key debounce in software. This is easily done by using an exclusive or with the previous key. Use a loop until a stable value is achieved in two successive tests.

Once a stable value is detected, determine the column involved and the row selected. Use the column with row information to decode the key.

Perform a table look-up of the key value to determine the ASCII equivalent. If no key was pushed, return a null value.

Presentation:

Demonstrate the keyboard operation by implementing a HEX keypad (0 - F hex). Display a corresponding key value on the 7-segment display or the serial port to the personal computer.

⇐⇑⇒

Program sample example _____

The exemplar program is similar to the project specifications. However, it contains elements that should be modified to complete the project as required.

Most projects have been presented with an example code that had additional items to be developed. In contrast, this section of code is complete. It is necessary because of the involved process.

```
;-------------------------------------------------
;Program: MODkey.ASM
;Update:  26 July 2004
;Initial: 17 October 1991
;
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.DrMod.com
;Copyright (c)1991, 2004.  All rights reserved
;
;Purpose:
;  A set of routines are provided to perform the
;  keypad input.
;
;Processor: 8031 family
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz
;Baud:      9600
```

```
;Assembler: Intel ASM51


;################################################
;
;                       ASSIGNMENTS
;
;################################################
;CONSTANTS
;------------------------------------------------
                         ;SYMBOLS
None       equ    0FFH  ;blank key
AdKey      equ    8001h ;mmio latch

;------------------------------------------------
;DEFINED VARIABLES
;------------------------------------------------
                         ;KEYS
KeyCol     equ    17H   ;present column number
KeyBit     equ    16H   ;byte moves a bit w/ column
KeyRow     equ    15H   ;row number pushed
KeyMul     equ    14H   ;multiple key count
CharK      equ    13H   ;key input character
CharD      equ    12H   ;debounced

                         ;CHARACTERS, HOLD, COUNT
CharP      equ    0FH   ;undebounced previous input
CharL      equ    0EH   ;character to LCD & Serial
LoopC      equ    07H   ;loop counter
;
;R6        equ    06H   ;size
;R5        equ    05H   ;carry in multiply, GP

;------------------------------------------------
;BITS ASSIGNMENTS
;------------------------------------------------
                         ;AT RAM byte 20H
FgKeyH     bit    00H   ;flag key held down


;################################################
;
;                       PROGRAM
;
```

```
;################################################
          org   00H
START:    ljmp  INITIAL

          org   0033h
          db    25, 1,'Marcus O. Durham, PhD, PE'

;-------------------------------------------------
          org   0080H         ;Addres past reserve
INITIAL:
;-------------------------------------------------
                              ;INITIALIZE
          mov   SP,#5Fh       ;start stack @ 5f+1
          lcall UART          ;config & start UART

;-------------------------------------------------
MAIN:
;-------------------------------------------------
;  The procedures are to input a key, send it on
;  serial, and do a line return.

                              ;PROCESS
          lcall KEYS          ;check keys
          mov   A,CharK
          lcall SEROUT        ;A =message value
MAN9:     ljmp  MAIN          ;Repeat


;************************************************
;                      KEYPAD
;************************************************
KEYS:
;-------------------------------------------------
;  Keypad is used to decode a matrix set of keys.

;  Scan is a routine to check if any key is
;  pressed. To do so, read the row and debounce.
;  Exit with the debounced in CharD and the
;  previous undebounced in CharP
;
;  For a keypad, decode the row and column value.
;
;  The routine waits until a key is pushed before
;  returning.
```

```
                                ;IS ANY KEY PUSHED?
          lcall KEYALLRD        ;all cols=0,read row
          lcall DEBOUNCE        ;CharD=debounced
          lcall KEYPAD          ;decipher key

                                ;LAST KEY
          mov   A,CharK         ;last key
          cjne  A,#None,KEYS9   ;<>none, have a key
          sjmp  KEYS            ;=none, get a key

                                ;TERMINATE
KEYS9:    ret                   ;back to message

;-------------------------------------------------
KEYPAD:
;-------------------------------------------------
;  KEYPAD is a routine to input buttons pressed on
;  keyboard. A 4X4 keypad can be used.
;
;  Columns connect to a latch, in the upper bits
;  Rows connect to a latch, in the low bits.
;
;  Column latch bits |7|6|5|4|3|2|1|0|
;  Key column        |0|1|2|3|-|-|-|-|
;
;  Row latch bits    |7|6|5|4|3|2|1|0|
;  Key row           |-|-|-|-|3|2|1|0|
;
;  Row lines are connected to a pull-up to give 5V
;  Output '0' to columns.
;  Read rows, if any key is '0' it is pressed.
;
;  A table is used to decode the value of
;  each key to ASCII.
;
;  If a key has not been selected, try for a
;  serial input.

                                ;ANY KEY BEEN push?
          mov   A,CharD         ;CharD debounced key
          cjne  A,#0Fh,KEYP3    ;row<>1, one pushed
          clr   FgKeyH          ;fg no key held down
          sjmp  KEYP8           ;return a null
```

```
                                    ;INITIALIZE
KEYP3:    jb    FgKeyH,KEYP8    ;key held, exit
          lcall KEYINIT         ;initialize

                                    ;SEND EACH COL A 0
KEYP4:    lcall KEYROWRD        ;read row
          cjne  A,#0Fh,KEYP5    ;0 in row pushed
          sjmp  KEYP6           ;no 0 in this row

KEYP5:    lcall KEYROWQ         ;query, KeyRow=row
          mov   A,KeyMul        ;check multiple key
          cjne  A,#1,KEYP8      ;<>1, so null

                                    ;COLUMN VALUE
          lcall KEYCOLQ         ;query, KeyCol=col
          add   A,KeyRow        ;A=KeyCol*4 + KeyRow
          mov   CharK,A         ;upgrade the charac

                                    ;NEXT COLUMN
KEYP6:    djnz  KeyCol,KEYP4    ;remain col to write

                                    ;CHECK RANGE ERROR
          mov   A,#0Fh          ;largest value
          clr   C
          subb  A,CharK         ;key
          jc    KEYP8           ;invalid key

                                    ;CONVERT TO ASCII
          mov   A,CharK         ;lookup last value
          mov   DPTR,#TabKey    ;start of table
          movc  A,@A+DPTR       ;conversion
          mov   CharK,A         ;save key

          setb  FgKeyH          ;flag key held down
          sjmp  KEYP9           ;exit

                                    ;NULL RESPONSE
KEYP8:    mov   CharK,#None     ;nothing pushed

                                    ;SO CHECK SERIAL
          jnb   RI,KEYP9        ;no serial either
          clr   RI              ;receive interrupt
          mov   CharK,SBUF      ;serial input
```

```
                                ;TERMINATE
KEYP9:    ret                   ;back to message

;-------------------------------------------------
KEYINIT:
;-------------------------------------------------
;  Initialize is a routine to set variables.
;  A 4X4 keypad can be used with a single byte.
;
;  KeyCol present column number
;  KeyMul multiple key count
;  KeyRow row number pushed
;  KeyBit byte moves a 0 bit for the column move

                                ;ANY KEY BEEN push?
          mov   KeyCol,#3       ;column number 0
          mov   KeyMul,#0       ;multiple key count
          mov   KeyRow,#0       ;row number 0
          mov   KeyBit,#01111111b;first column w/0

                                ;TERMINATE
          ret                   ;back home

;-------------------------------------------------
KEYALLRD:
;-------------------------------------------------
;  Row Read is a routine that sends a 0 to columns
;  It reads the row.
;  For ALLRD, 0 is sent to all columns.
;  For ROWRD, 0 is sent to each column
;  successively
;
                                ;SEND ALL 0
          mov   A,#00001111h    ;all col=0
          sjmp  KEYROW1         ;COL OUT, ROW IN
;
                                ;bit TO SEND
KEYROWRD: mov   A,KeyBit        ;0 bit is column

                                ;SAVE FOR NEXT bit
          rr    A
          mov   KeyBit,A        ;keep it
          rl    A               ;use it
```

```
                                    ;COL OUT, ROW IN
KEYROW1:  mov   DPTR,#AdKey         ;MMIO key address
          movx  @DPTR,A             ;read row
          movx  A,@DPTR             ;read rows
          anl   A,#0Fh              ;mask hi nibble

                                    ;TERMINATE
          ret                       ;back to message

;-------------------------------------------------
KEYROWQ:
;-------------------------------------------------
;  Row Calculate determines the row number that is
;  pressed. RowNum = 0,1,2,3
;
;  A counter is set if multiple rows are pushed.
;
;  The routine can handle 4 rows.
;  If fewer rows are used, simply change the jump
;  to the corresponding number of rows.
;  If a 1 row pad is used, sjmp to ROW1.

                                    ;INITIALIZE
          clr   C                   ;initialize test bit
;;        sjmp  KEYR3               ;pad has only 3 rows

                                    ;IS THIS ROW=0
KEYR4:    rrc   A                   ;row bit to C
          jc    KEYR3               ;C=1,row not down
          inc   KeyMul              ;multiple key count
          mov   KeyRow,#3           ;row number

                                    ;IS THIS ROW=0
KEYR3:    rrc   A                   ;row bit to C
          jc    KEYR2               ;C=1,row not down
          inc   KeyMul              ;multiple key count
          mov   KeyRow,#2           ;row number

                                    ;IS THIS ROW=0
KEYR2:    rrc   A                   ;row bit to C
          jc    KEYR1               ;C=1,row not down
          inc   KeyMul              ;multiple key count
          mov   KeyRow,#1           ;row number
```

```
                                ;IS THIS ROW=0
KEYR1:    rrc   A               ;row bit to C
          jc    KEYR9           ;C=1,row not down
          inc   KeyMul          ;multiple key count
          mov   KeyRow,#0       ;row number
          sjmp  KEYR9

                                ;TERMINATE
KEYR9:    ret                   ;back to message

;-------------------------------------------------
KEYCOLQ:
;-------------------------------------------------
;  Column calculate determines the column that is
;  pressed, KeyCol. Then it calculates the value
;  for the key.
;
;  Rows are numbered 0,1,2,3
;  Columns are numbered 0,1,2,3
;  Columns are valued 0,4,8,12.
;  Columns value = column number * number of rows
;  With four rows A = KeyCol * 4

                                ;CALCULATE
          mov   A,KeyCol        ;column w/ 0
          dec   A               ;column # inc by 1
          mov   B,#4            ;qty of rows
          mul   AB              ;A = column value

                                ;TERMINATE
          ret                   ;back to message


;+++++++++++++++++++++++++++++++++++++++++++++++++
;TABLE SETUP - KEYPAD CONVERSION
;-------------------------------------------------
;  Tables are used to convert between formats.
;  These include keypad & ASCII.
;  The table pointer will show a decimal result
;  that corresponds to a column/row location.

;-------------------------------------------------
TabKey:
```

```
;-------------------------------------------------
;  Three column keyboard

          db     '147*2580369#'

;*************************************************

          end                          ;Program end
```

⇐⇑⇒

31

LIQUID CRYSTAL DISPLAY

Thought
3C procedures:
Command, control, communications.

Different display systems _____

Numerous ways have been used to display information. The first project started with a single light emitting diode (LED) for information display. This is very basic and simple to use. As a result, it has very simple information. Progressively, more LEDs were added to give more info.

Serial communications were added. This permitted very clear, detailed messages, However, it requires another computer or display device.

Then seven-segment displays were implemented. The hardware and software require multiplexing to get very many characters. These characters are easy to read; however, the number of letters are limited. All numbers can be seen, but some letters are not possible and others can only be a lower or an upper case.

A complete text message and limited graphics can be shown using liquid crystal displays. These are the most comprehensive imaging devices for their size. As a result, they are very popular. One slight drawback is the cost. The units are several times more expensive

than using a limited seven-segment display. As a result, very cost competitive items will skip the displays.

High end systems will use a cathode ray tube (CRT) for imaging. The size makes them very useful for showing large amounts of data. However, the size and power consumption is a serious drawback for many projects.

A compromise device for some specialty projects is a liquid crystal display with a comparatively large screen. The layout is a large matrix, therefore more imaging can be shown in addition to text. These can be developed into quite sophisticated devices, so they are infrequently used for simple control projects.

LCD variations

Liquid crystal displays commonly come in one, two, or four line versions. The length of line is typically sixteen or twenty characters. Virtually all models are based on a common design with the same microcontroller driver. As a result, the pin-outs are identical. A few models simply renumber the pins from the opposite direction.

The instruction set to all liquid crystal displays is common between manufacturers. The standard character set is built into the onboard processor. Graphics can be created for the display, but these typically require bit mapping.

Connections

Because of the control sequence, the LCD appears somewhat tedious. There are three groups of routines.

1. LCD control lines.
2. LCD commands: initialize, instruction, data, and busy.
3. LCD communicate: display message.

The LCD has fourteen pins for connecting power, controls, and data.

Pin	Function
1	Vss, ground
2	Vdd, 5 V
3	Vo, power for contrast
4	Reg. Select (RS) 0= instruction, 1= data
5	RW, 0= Write, 1= Read
6	Enable
7-14	DB0-DB7 data bits

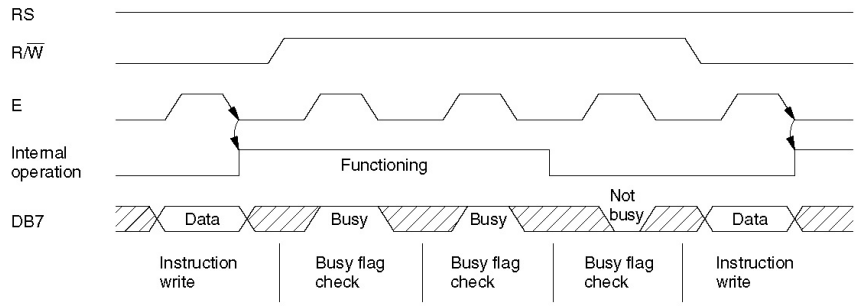
Connect contrast to ground for maximum viewing intensity. The board has an elegant circuit for contrast control based on the temperature sensed by a thermistor. For many applications, the circuit can be bypassed with a grounding jumper.

Control

The liquid crystal display has three control lines- enable (pin 6), read/write not (pin5), and register select (pin 7). It is wise to have the enable line asserted low before the other control lines are asserted. The register select and the read/write are activated by software control. The commands are executed on the low to high transition of the enable line. The high also allows data to be set-up. Data is transferred with the enable line high.

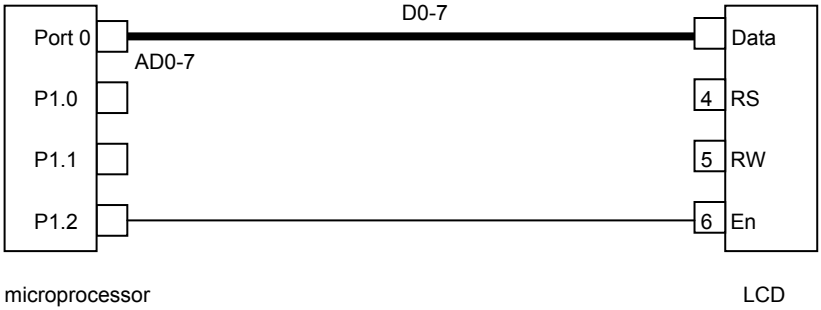
Display of information requires initiating a control sequence. First the enable line is pulled low. Then the control operation is implemented. Then the enable line is pulled high. Then the data is sent. Then the enable line is pulled low.

The timing diagram illustrates the relationship between the lines. A detailed discussion of timing is included in a chapter of the reference section.



Control via port

The LCD has a microcontroller on board. It expects to communicate with another computer. The simplest implementation uses port pins for the control lines. This is very effective if there are available pins. Any delay necessary is created in software.



Regardless of the control line connection technique, the same sequence must be exercised for the control to activate properly. The timing diagram illustrates that the enable line must be taken through multiple transitions for control and data transfer.

The enable line is first pulled low before the control lines are asserted. Then the register select line is asserted high or low, depending on the function for the LCD. Next the write line is pulled

low. On some versions, these two lines can be asserted simultaneously.

The controls are activated when the enable line transitions to high. The data is transferred between the microprocessor and the LCD on the high to low transition of the enable line.

Control via latch

The module is often physically wired through an expansion latch as discussed in an earlier chapter. Therefore, this section will address one procedure that has been used successfully. It will not be used for project implementation.

The register has 8 lines that are configurable. The lines may be shared with other output controls One arrangement that has been used is illustrated. The upper four bits are for columns on a keypad. The lower 3 bits are for the display select. Bit three is for RS485 select.

Flip/flop	Function	Control	Bit
Q1	LCD 4	RS	0
Q2	LCD5	RW’	1
Q3	LCD6	Enable	2
Q4	RS485	RE’/DE	3
Q5	Keypad	Column	4
Q6	Keypad	Column	5
Q7	Keypad	Column	6
Q8	Keypad	Column	7

Alternately, it can look like bit assignments.

[C C C C 3 E W S]

Other bits are on the latch (register) which will not be part of the present operation. The value to the latch must be changed with anl/orl or setb/clr to prevent these other bits from changing.

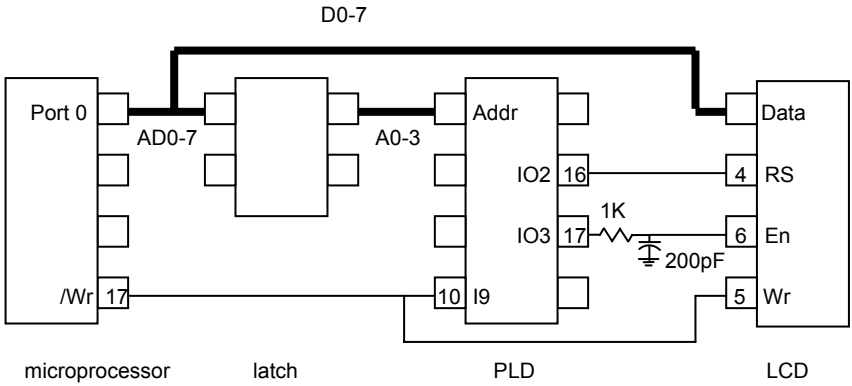
To ensure that other lines are not affected, the value placed on the register can be maintained in temporary storage. However, since no other output on the latch is processed while the LCD is being driven, seldom is it necessary to hold the values. The exception would be if other tasks were performed during the delays for the LCD.

So look at a review of the process. The bits for the LCD control lines are selected. The value may be saved in temporary storage. The value is presented to the latch/register. The latch enable is made active. The LCD then responds.

The next section illustrates the procedure that will be used for the project implementation.

Control via PLD

The LCD can be wired directly from a PLD a latch for the control lines. The enable and register select lines are encoded in the PLD. These are simply chosen based on a memory-mapped address.



For most LCDs there must be a delay of at least 140 ns after the controls are selected before the enable can be taken high. A RC circuit is placed in the enable line to delay the response to the LCD enable. Typical component values are 1 K Ohm and 200 pF.

Command

The LCD has a number of commands and instruction codes that are required for display. The LCD can receive instructions for set-up, send status, receive display data, or send the data currently in the display back to the microcontroller. Two address lines (XX) select the function.

The first table provides the instruction codes to the LCD.

Code	Function
01	clear and home
02	home
04	cursor increment w/ data display
06	cursor decrement w/ data display
0A	cursor on, flash off
0B	cursor on, flash on
10	move cursor left
14	move cursor right
38	function set -8bit, 2line, 5x7dot
8X	cursor position= DDRAM address
81	1st line, 1=home
C1	2nd line home
A1	3rd line home
E1	4th line home

The next table has the codes for the status response from the LCD to the microprocessor.

Code	Function
8X	busy flag, X=current address
0X	clear, ready to accept data

Two other commands are used for moving data.

Write Data: 8 bits of data to display

Read Data: 8 bits of data from display

All instructions for the LCD have a typical execution time of 40 microseconds in the module, except for two commands. Clear Display and Return Home have a typical execution time of 1.64 milliseconds.

The LCD busy flag (BF) is clear when the module is ready to accept another instruction. However, the busy flag (BF) cannot be read until the first 3 LCD initialization bytes have been processed by the module. Thus BF cannot be tested in the initialization subroutine.

The timing sequence of the module is slower than that of most microprocessors. A memory-mapped interface limits the microprocessor frequency to values less than about 9 MHz because of timing. However, timing can be extended by using NOPs if a faster clock is installed. Interfacing the LCD to a port allows the processor to use any crystal frequency available, since additional time is required for the instructions.

In addition, installation of different crystal frequencies dictates changing the software counter used to create a 1 millisecond delay. Insertion of additional NOPs may be necessary for crystal frequencies above 40MHz.

Initialization

A rather extensive procedure is necessary to initialize the microcontroller that is on-board the liquid crystal display module. To take care of timing, several delays must be built into the procedure. The busy flag cannot be used until after initialization. Therefore, the delays must be created by software.

Moreover, as a practical matter, delay routines will be used in all procedures rather than test for the busy flag. If the LCD module is not plugged-in, then there is only a small delay. If busy were checked, the process would hang, waiting for the response from the LCD.

The initialization process is based on a Hitachi 44780 on-board processor. Even if newer processors are used, the codes remain the same to insure compatibility.

The references are Standish document A93531B page 2 and Hitachi document CD-E613P 0591 page 90. The series of initialization instructions are those required by the Philips data sheet.

First, after V_{CC} is applied, allow 15ms for the module internal initialization to be completed. The instruction mode is used for all initializing. The instruction is sent, then there is a delay before the next instruction.

Step	Code	Delay	
1	wait	15 ms	power on
2	30h	5 ms	8-bit
3	30h	100 us	8-bit
4	30h	100 us	8-bit
5	38h	200 us	cursor
6	0Bh	100 us	blink
7	01h	3 ms	clear display

Cursor position

The instruction codes contain items that are used for cursor positioning. The first three bits of the message define the cursor line that will be used. The remainder of the line contains the character location on the line.

Line	Hex	Bits 765	43210
1	80	100	yxxxx
2	C0	110	yxxxx
3	A0	101	yxxxx
4	E0	111	yxxxx

The cursor position requires the most significant bit to be set. In conjunction with the next two bits, the line is described. Then the location is added to the line hex value.

The first position, 1, is home. Any location greater than fifteen would require bit four to be set. When this is added to the line hex value, it will be incremented by one. A few examples will clarify the operations. Line one, home position would be 81h (80 + 1h). Line one position 15 would be 8fh (80 + 0fh). Line one, position sixteen would be 90h (80 +10h).

If using a 2 line display, send 80h to access the first space. Then follow with the 16 or 20 characters for the length of the line. The second line is accessed by sending 0C0h. Then the characters for the length of the line.

A single line display is access exactly as a two line display. Send 80h to get to the first space. Follow that with the characters for one-half of the line length. Then send 0C0h to access the second half of the line. Follow that with the characters for the remaining half of the line.

Message display

The LCD is structured to display a string of characters. One technique for sending a string is to define the characters with a define byte ‘db’ directive. The procedure used in the project includes 3 types of information with the line. First is the number of characters to be sent. Second is the location for the starting character, third is the string.

An alternative to counting the number of characters is to include a termination character at the end of each string. Then test for the termination character as the string is sent. An example is ‘0’. This could be tested with a jz instruction.

32

PROJECT 10 - TEXT DISPLAY

Thought
Confidence is
the mental assurance
that something is true.
MOD

Project 10: Text message screens ___

Purpose: To display text in a friendly format.
To use a control register to expand functions.

Preamble:

A liquid crystal display (LCD) is the most common display mechanism for simple text messages. The devices can have one, two, or four lines of display. The length is typically 20 or 40 characters.

The device has an on-board processor and memory. Therefore, it has extensive control capabilities. Data is sent to the display as an eight bit byte. Instructions are also sent to control the machine. There are three control lines - enable, read / write not, and data / instruction not.

An extensive sequence of controls is required to initialize the display. These need precise time delays before the next step.

Display of information requires initiating a control sequence. First the enable line is pulled low. Then the control operation is implemented. Then the enable line is pulled high. Then the data is sent. Then the enable line is pulled low.

To prevent interference from other operations, interrupts are typically disabled after the enable is high. The interrupts are enabled after the enable is later pulled low.

The data pin outs and the control pin outs are standard among most manufacturers. Some have a different power pin arrangement. There are also lines for controlling the contrast. However, these are often fixed. The board has an automatic circuit for contrast control. No software is required for the contrast control. If the automatic feature is not desired, it may be bypassed by grounding the contrast control pin.

Plan:

Implement a multiple character liquid crystal display.

Preparation:

Connect the data lines through one latch. Connect the control lines through another.

Procedure:

First, use an initialization routine to activate the LCD. Other routines typically are clear screen, display data, advance cursor, backup cursor, and flash cursor.

Next send data to the LCD. The information can come from the keypad, serial line, or from internal tables.

Presentation:

Demonstrate the LCD operation by implementing a greeting message. Then display a message from the keypad or from the serial input.

⇐⇑⇒

Program sample example

The exemplar program is similar to the project specifications. However, it contains elements that should be modified to complete the project as required.

```
;-------------------------------------------------
;Program: BiosLcd.ASM
;Update:  27 July 2004
;Initial: 17 October 1991
;
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)1991-2004.  All rights reserved
;
;Purpose:
;  A set of routines are provided to initialize
;  and operate a liquid crystal display.
;
;Processor: 8031 family
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz
;Baud:      9600
;Handshake: not used at this speed
;Assembler: Intel ASM51
;

;#################################################
;
;                      ASSIGNMENTS
```

```
;
;################################################
;CONSTANTS
;------------------------------------------------
;
                        ;CHARACTERS, HOLD, COUNT
CharL     equ    0EH    ;character to LCD & Serial
LoopC     equ    07H    ;loop counter

;R2       equ    02H    ;loop, destination size,mat

Pio       equ    0A0h   ;Port 0
LcdRS     equ    090h   ;port 1.0
LcdRW     equ    091h   ;port 1.1, write not line
LcdEn     equ    092h   ;port 1.2

;################################################
;
;                     PROGRAM
;
;################################################
          org    00H
START:
;------------------------------------------------
          ljmp   INITIAL

          org    0080H        ;Addres past reserve
;------------------------------------------------
INITIAL:
;------------------------------------------------
;
;  The procedure initializes all common routines.
;  It directs traffic for initiation messages.

                               ;LCD INITIALIZE
          lcall  SCRINIT       ;initialize screen

;------------------------------------------------
MAIN:
;------------------------------------------------
;
;  The main procedure directs traffic.
;  The main orchestrates execution of the
;  subroutines.
```

```
                                ;PROCESS
            mov     DPTR,#SerGreet ;get address
            lcall   BIOSLCD      ;message headr RS232
            lcall   BIOSLCD      ;second line

MAN9:       ljmp    MAN9         ;Halt


;-------------------------------------------------
BIOSLCD:
;-------------------------------------------------
;
;  Send LCD message
;  The first byte is the # of characters to send.

            mov     A,#0         ;read initial byte
            movc    A,@A+DPTR    ;input byte
            mov     LoopC,A      ;set up loop


            mov     A,#1         ;length of message
            movc    A,@A+DPTR
            mov     CharL,A
            lcall   SCRINST
            lcall   DLYMIL

BIOL2:      mov     A,#2         ;get 1st message
            movc    A,@A+DPTR    ;input byte
            mov     CharL,A      ;character lcd

                                 ;LCD data
            lcall   DLYMIC       ;wait for next byte
            lcall   SCRDATA      ;send out byte

            inc     DPTR         ;restore offset
            djnz    LoopC,BIOL2  ;go thru loop

            inc     DPTR
            inc     DPTR
            ret
```

```
;************************************************
;
;                     LCD SETUP
;
;************************************************
;
;  Because of the control sequence, the LCD
;  appears somewhat tedious.
;
;  There are three groups of routines.
;  1. LCD control lines
;  2. LCD initialize, instruction, data, & busy
;  3. Message to display info.
;
;  The LCD has 14 pins.
;    1  = Vss, ground
;    2  = Vdd, 5 V
;    3  = Vo, power for liquid crystal drive
;    4  = RS, 0= instruction, 1= data Reg. Select
;    5  = RW, 0= Write, 1= Read
;    6  = Enable
;    7-14 = DB0-DB7 data bits
;
;  The LCD control lines are programmed as bits
;    LcdRS = bit
;    LcdRW = bit
;    LcdEn = bit

;
;-------------------------------------------------
;SCREEN INSTRUCTION CODES
;-------------------------------------------------
;  The following codes are required by the LCD
;  display.  The LCD can receive instructions for
;  set-up, send status, receive display data, or
;  send the data currently in the display back to
;  the microcontroller.
;  Two address lines (XX)select the function.
;
;  Instruction Codes
;         01     ;clear and home
;         02     ;home
;         04     ;cursor increment w/ data display
```

```
;         06     ;cursor decrement w/ data display
;         0A     ;cursor on, flash off
;         0B     ;cursor on, flash on
;         10     ;move cursor left
;         14     ;move cursor right
;         38     ;function set-8bit, 2line, 5x7dot
;         8X     ;cursor position=  DDRAM address
;         81     ;each line has 40 chars, 1=home
;         C1     ;41=2nd line home, 80+41=C1
;
;  Status Response
;         8X     ;busy flag, X=current address
;         0X     ;clear, ready to accept data
;
;  Write Data
;                ;8 bits of data to display
;
;  Read Data
;                ;8 bits of data from display
;
;  All instructions have a TYPICAL execution time
;  of "40us" in the LCD module except instructions
;  Clear Diplay and Return Home.  These two
;  have a TYPICAL execution time of 1.64ms.
;
;  The LCD busy flag (BF) is clear when the LCD
;  module is ready to accept another instruction.
;
;  The LCD busy flag (BF) cannot be read until
;  the first 3 LCD initialization bytes have been
;  processed by the LCD module.  Thus BF is not
;  tested in this subroutine.
;
;  Interfacing the LCD to port 1 allows the uC to
;  use any crystal frequency available.  A
;  memory-mapped interface limits uC frequency to
;  values less than about 9 MHz.
;
;  Installation of different crystal frequencies
;  dictates changing the software counter used to
;  create a 1ms delay.  Insertion of NOPs may be
;  necessary crystal frequencies above 40MHz.
;  See subroutines for control line values.
```

```
;-------------------------------------------------
SCRINIT:
;-------------------------------------------------
;--SUBS CALL -
;  The routine initializes the Liquid Crystal
;  Display.  The busy flag stays busy until
;  initialization is complete.  The time is 15ms.
;  Therefore delays must be built into the init
;  routine before Busy can be used.
;
;  Allow 15ms for Hitachi 44780 internal initial
;  to complete after Vcc is applied.
;  (Re: Standish document A93531B p. 2 &
;  Hitachi document CD-E613P 0591 p. 90).
;
;  The series of instructions are as required by
;  Philips mfg data sheet.
;
;  Instruction mode is used for all initializing.
;
;  232 must be on for LCD contrast control
;  to get 10 volts.
;
;  Delay is used rather than test for Busy line.
;  If LCD is not plugged-in, then there is only
;  a small delay. If Busy were checked, the
;  process would hang.

                                ;POWER ON RESET
          mov    LoopC,#20      ;20ms delay
SCRNI1:   lcall  DLYMIL         ;1 ms routine
          djnz   LoopC,SCRNI1   ;loop

                                ;CLEAR REGISTERS
          mov    CharL,#30H     ;#1function set=8bit
          lcall  SCRINST        ;send command

          mov    LoopC,#5       ;5ms delay
SCRNI2:   lcall  DLYMIL         ;1 ms routine
          djnz   LoopC,SCRNI2   ;loop

          mov    CharL,#30H     ;#2function set=8bit
          lcall  SCRINST        ;send command
```

```
          lcall  DLYMIC       ;>100 microsec

          mov    CharL,#30H   ;#3function set=8bit
          lcall  SCRINST      ;send command
          lcall  DLYMIC       ;>100 microsec

                              ;SET CURSOR
          mov    CharL,#38H   ;#4funct set=8bit,2l
          lcall  SCRINST      ;send command
          lcall  DLYMIC       ;>100 microsec

          mov    CharL,#0FH   ;cursor on, blink
;         mov    CharL,#0CH   ;cursor on,no blink
          lcall  SCRINST      ;send command
          lcall  DLYMIC       ;>100 microsec

          mov    CharL,#01H   ;display clear
          lcall  SCRINST      ;send command
          lcall  DLYMIL
          lcall  DLYMIL
          lcall  DLYMIL

                              ;CHANGE MODE
          mov    CharL,#06H   ;entry mode set
          lcall  SCRINST      ;send command

          ret

;-------------------------------------------------
DLYMIL:
;-------------------------------------------------
;  Delays shorter than the interrupt cycle are
;  required.  Since this is outside the normal
;  program polling, hard calculated delays are
;  used.
;
;  The delay is based on clock cycles.
;    mov=1, nop=1, djnz=2.
;
;  The cycles in the loop are calculated.
;    (1)          for mov
;    (2* count  ) for both nop's
;    (2* count-1) for djnz
;    (2)          for last jump
```

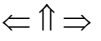
```
;    =            Total cycles
;
;  Old crystal frequency was 7.3728Mhz
;
;  Time = #states(12) * Total cycles/crys freq
;  11.059 MHz = 1.085 microsecond
;
;  The inside loop has 925 cycles.
;  At 11.059MHz  this represents 1.003 ms

          mov    R2,#231      ;max loop=0
DLYM1:    nop
          nop
          djnz   R2,DLYM1     ;inside loop
          ret

;-------------------------------------------------
DLYMIC:
;-------------------------------------------------
;  Delay = 50 microsecond
;
;  Delays shorter than the interrupt cycle are
;  required.  Since this is outside the normal
;  program polling, hard calculated delays are
;  used.
;
;  The inside loop has 49 cycles.
;  At 11.059Mhz,
;  1.085 microsec *  this represents 53 microsec

          mov    R2,#12       ;max loop=0
DLYC1:    nop
          nop
          djnz   R2,DLYC1     ;inside loop
          ret

;-------------------------------------------------
SCRINST:
;-------------------------------------------------
;  The routine writes instructions to LCD.
;  Write an instruction to the LCD requires
;    control line RS= 0
;    control line RW= 0
;  Enable must be low before change R/W' or RS.
```

```
;
;  CAUTION: A delay after data, before deselect
;  can be interrupted and cause erratic data.
;  disable interrupts until complete.

                              ;CONTRL INSTRUCTION
          clr    LcdEn        ;clr enable lo
          clr    LcdRW        ;instruction& write
          clr    LcdRS
          setb   LcdEn        ;set enable

                              ;DISABLE INTERRUPTS
;         anl    IE,#7Fh      ;disable main inter

                              ;SEND INSTRUCTION
          mov    A,CharL      ;select LCD data
          mov    DPTR,#8002h  ;display latch
          mov    @DPTR,A      ;send out

                              ;CONTROL INSTRUCTION
          clr    LcdEn        ;clr enable lo

          lcall  DLYMIL       ;longer delay

                              ;ENABLE INTERRUPTS
;         orl    IE,#80h      ;enable all

          ret

;-------------------------------------------------
SCRDATA:
;-------------------------------------------------
;  The routine writes output data to LCD.
;
;  Write data to the LCD requires
;    control line RS= 1, data display on HD44780
;    control line RW= 0, to write
;  Enable must be low before change R/W' or RS.
;
;  CAUTION: A delay after data, before deselect
;  can be interrupted and cause erratic data.

                              ;CONTRL INSTRUCTION
          clr    LcdEn        ;clr enable lo
```

```
          clr    LcdRW         ;instruction& write
          setb   LcdRS
          setb   LcdEn         ;set enable

                               ;DISABLE INTERRUPTS
;         anl    IE,#7Fh       ;disable main inter

                               ;SEND INSTRUCTION
          mov    A,CharL       ;select LCD data
          mov    DPTR,#8002h   ;display latch
          mov    @DPTR,A       ;send out

                               ;CONTROL INSTRUCTION
          clr    LcdEn         ;clr enable lo

          lcall  DLYMIC        ;short delay

                               ;ENABLE INTERRUPTS
;         orl    IE,#80h       ;enable all

          ret


;################################################
;
;                     TABLES
;
;################################################
;
;  Tables are used to convert between formats.
;  These include kepad & ASCII.
;
;  Tables are also used to carry display messages.
;
;
;++++++++++++++++++++++++++++++++++++++++++++++++
;TABLE SETUP - MESSAGES
;------------------------------------------------
;  Predefined messages are in code memory.
;  These are used in BIOSSER.
;  Place at end of program code or org out of way
;
;  The first byte is the number of characters.
;  The second byte is the cursor location for LCD.
```

```
;  The next bytes are the ASCII message.
;
;  The format for intel assembler is illustrated.
;SerGreet: db    14,   0, 'uC BIOS 11V3- ';intel
;
;  The format for Tasm is illustrated.
;SerGreet .byte  14
;         .byte  0
;         .text  "uC BIOS 11V3- ";


;-------------------------------------------------
TABMESSG:
;-------------------------------------------------

SerGreet: db      8, 80h ,'uC Lcd--'
          db      2,0C0h ,'hi'


;*************************************************

          end
```

SECTION III - APPLICATIONS



33

INFRARED COMMUNICATIONS

Thought
Think excellent, do right,
obtain peace.
MOD

Local wireless

Wireless communications is highly desirable for many control projects. Radio frequency communications is very effective, but circuit design is tedious and requires additional technology.

Infrared communications is a very cost effective alternative. Infrared is light that is slightly lower in frequency than the visible range. It is a thermal energy that is easily generated and detected. It does not suffer from electromagnetic interference. However, its range is limited to line of sight. The receiver can experience interference from other light sources. Pulsing light sources such as fluorescent fixtures are a particular problem.

The technology is mature and widely used in consumer electronics. The devices are used for video, audio, and control devices. Transmitters are extensively available from numerous sources. As a result, the processor project will be a receiver which can control other devices.

There are four basic technologies that are used. Sony uses a bit width technology. The four - NEC, Apex, Hitachi, and Pioneer - all use the same technology, which is another version of bit width. JVC uses a complementary system which is a space width technology. The major departure is the European model by Philips. It has a fixed bit width and a fixed quantity of bits.

In its most basic implementation, an infrared signal is little different from a red LED that is flashed on and off at some precise interval. The receiver, then, is essentially a LED sensor that has a detector to strip off extraneous carrier signals.

Philips protocol

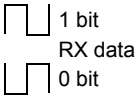
The discussion will focus on the Philips RC5 technology. The circuit uses an infrared receiver with an integrated detector.

The infrared circuit consists of a carrier with a nominal 36-40 kHz square wave. For a 40 kHz carrier, each cycle is 27 microseconds. Then this is modulated with data bits. Therefore, a 40 kHz signal is seen during on(0) and dc is seen during off (1). Because of the variations in carrier frequency, it may be necessary to adjust the cycle time slightly.

The protocol defines each bit as 1.728 ms long. Each IR string is 14 bits long. The bit stream is repeated every 130 ms, if the key is held. This gives a long interval to resynchronize if there is a receive error. Bit 1 is the most significant bit (MSB).

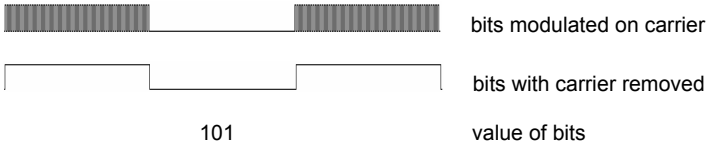
Bits	Function	Operation
1-2	AGC	Automatic gate control, 10=good
3	check	Bit flips with each new transmission
4-8	address	00= television
9-14	command	Instructions out to latch

A data bit is both positive and negative. The first half cycle is setup, then the second half is the state. On is 0 in the second half, while off is 1 in the second half.



When the first falling edge is detected, this is used as the timer frame. Wait 3/ 4 bit time to sync in the middle of the next bit.

The transmitter carrier frequency is turned on/off by the digital signal. The receiver gets this modulated signal, which is a series of carrier frequencies that are turned on/off. The detector removes the carrier to leave the on/off pulses. The bit stream is fed into a single bit input of the microprocessor.



The carrier frequency causes 32 pulses for each half cycle or 64 pulses per bit. The computer does not see these pulses, since they are filtered by the detector.

This program has a wait cycle at the start. It is looking for the first transition in the bit stream. A better approach would be to activate the interrupt on INT1.

The first or start bit is always on (0). The second is also on (0). These are the automatic gate control (AGC).

The third bit is a check (CHK). It toggles each time a new key is pushed.

Next, follows 5 address and 6 command bits. The output should remain on the latch for a short time to prevent glitches in data.

Check the first two bits for on. If not, wait more than 15 bit times to begin a new sample.

Detected string

The address is decoded to determine the equipment type. The television remote is address 00. However, that is a code 10 at the universal programmable remote, because of the detection circuit waiting for a falling edge. There are addresses associated with other devices as shown in the table.

Address	Equipment
0	Television
2	Tele text
5	Video recorder
7	Experimental
16	Preamplifier
17	Receiver / tuner
18	Tape / cassette recorder
19	experimental

A command is associated with each piece of equipment. A table lookup is used to convert the command to a value. The decode command can be used to direct different operations. The table shows a correlation for each number. It also gives an 8-bit value that can be output on a latch.

Command bit 7 is 0 when a valid key is pressed. Otherwise it is a 1 for undefined values. That bit can be used to recognize values. When key zero is pressed, all other bits are high. The protocol has not defined commands that have FF as their description. Those commands can be used for whatever task is required.

```
;-----------------------------------------------
IRRC5TAB:
```

```
;-------------------------------------------------
;  The command code determines the function.
;
;                VALUE TO P1  REMOTE KEY  COMMAND
;                -----------  ----------  -------
          db    01111111b     ; 0         ; 0
          db    01111110b     ; 1         ; 1
          db    01111101b     ; 2         ; 2
          db    01111100b     ; 3         ; 3
          db    01111011b     ; 4         ; 4
          db    01111010b     ; 5         ; 5
          db    01111001b     ; 6         ; 6
          db    01111000b     ; 7         ; 7
          db    01110111b     ; 8         ; 8
          db    01110110b     ; 9         ; 9
          db    11111111b     ;           ; A
          db    11111111b     ;           ; B
          db    01110011b     ; ON/OFF    ; C
          db    01110010b     ; MUTE      ; D
          db    01110001b     ; PP        ; E
          db    01110000b     ; OSD       ; F
          db    01101111b     ; Volume+   ; 10
          db    01101110b     ; Volume-   ; 11
          db    01101101b     ; Bright+   ; 12
          db    01101100b     ; Bright-   ; 13
          db    01101011b     ; Color+    ; 14
          db    01111010b     ; Color-    ; 15
          db    11111111b     ;           ; 16
          db    11111111b     ;           ; 17
          db    11111111b     ;           ; 18
          db    11111111b     ;           ; 19
          db    11111111b     ;           ; 1A
          db    11111111b     ;           ; 1B
          db    01100011b     ; Contrast+ ; 1C
          db    01100010b     ; Contrast- ; 1D
          db    11111111b     ;           ; 1E
          db    11111111b     ;           ; 1F
          db    01011111b     ; Program+  ; 20
          db    01011110b     ; Program-  ; 21
          db    11111111b     ;           ; 22
          db    11111111b     ;           ; 23
          db    01011011b     ; Timer     ; 24
          db    01010111b     ; Special 1 ; 25
          db    01000001b     ; Special 2 ; 26
```

```
db    01000111b     ; Special 3 ; 27
db    01001110b     ; Special 4 ; 28
db    01000101b     ; Special 5 ; 29
db    01010010b     ; Special 6 ; 2A
```

Connections

Only three pins are required by the software. The data is input on one pin.

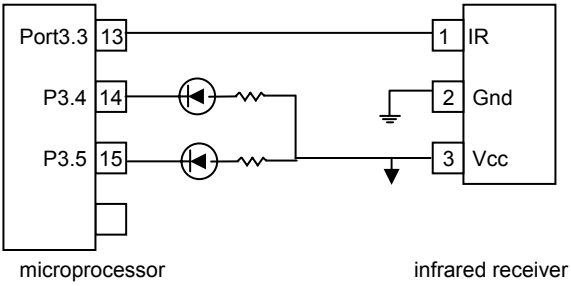
Another pin is used to trigger an LED to show infrared is being received.

The third pin shows a pulse generated by the software. This triggers with each new bit. Therefore, a scope can look at the incoming timing. This will ensure the code is sensing at mid bit.

```
___ ___ ___ ______ _____ ___ ___ ___ ___ IrInput
_|_____|_____|_____|_____|_____|_____|_ IrScope
```

After testing, this code can be removed with no loss of functionality.

Circuit: infrared receiver



⇐ ⇑ ⇒

34

PROJECT 11 - WIRELESS

Thought
Peripheral vision or tunnel vision,
its your choice.
MOD

Project 11: Communicate with IR __

Purpose: To detect an infrared remote.
To control some device based on the remote command.

Preamble:

An infrared transmitter is used to control many consumer electronic devices without physical wires. An oscillator of about 40 kHz carrier is modulated with pulses for off and on states. When the signal passes through a detector, the carrier is removed, leaving only the pulses. These pulses represent digital data.

The transmit and receive procedures are typically performed by a microcontroller. This is easier than making a custom chip.

For this project, connect an infrared photo detector. This is basically a photo diode that will respond to the carrier. Connect the output of the photo detector to an input port or mmio bit.

Basically three protocols are used by most remotes. The Philips, Sony, and Eastern formats are most common. This project will use the Philips protocol with the microcontroller operating as a television. Therefore, any universal remote should be programmable to operate with the microcontroller.

Plan:

Add an infrared detector to the microcontroller. Use Philips protocol. Program the remote for a television.

Preparation:

Determine the address and command structure for the infrared transmission. Create a table to decode the message.

Procedure:

First, write a subroutine to detect the pulses that are incoming. Detect the first falling edge and use this as the start bit. Create appropriate time delay before testing for the next pulse. Shift the pulses in serially until all the data is received. If there is an error, create an adequate delay to ensure the next message has not started.

If the data is valid, decode the pulses according to a table. Use the results to initiate a control action. As a minimum, turn an LED on and off, based on the infrared remote signal.

Presentation:

Demonstrate the infrared operation by displaying the results in LED, seven-segment, or LCD format.

Program sample example

The exemplar program is similar to the project specifications. However, it contains elements that should be modified to complete the project as required.

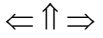
```
;-----------------------------------------------
;Program: MODInfra.ASM
;Update:  16 February 2003
;Date:    17 August 2002
;
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)2002, 2003.  All rights reserved
;
;Purpose:
;  A set of routines are provided to perform
;  infrared receive functions. The code can be
;  easily modified.
;
;  This code has timing based on 11.059 MHz
;  crystal. If another crystal is used, the delay
;  routines must be modified.

;################################################
;                     PROGRAM
;################################################
          org    00H
START:    ljmp   INITIAL

          org   0033H ;Address past vectors
          db    'Marcus O. Durham, PhD, PE'

;************************************************
;                     INITIAL & MAIN
;************************************************
          org   0080H         ;Addres past reserve
INITIAL:
;-----------------------------------------------
```

```
                                ;INITIALIZE
          mov   SP,#5Fh         ;start stack @ 5f+1

;-------------------------------------------------
MAIN:
;-------------------------------------------------
                                ;PROCESS
          lcall IRRECV          ;infrared:A=IrComd
          lcall SEROUT          ;display IrComd
MAN9:     ljmp  MAIN            ;Repeat

;*************************************************
;                      INFRARED REMOTE
;*************************************************
;  PHILIPS RC5 remote receiver.
;  Adopted from code by Wagner Lipnharski 11/99

;-------------------------------------------------
;Ir Receiver Assignments
;-------------------------------------------------
IrOut   equ 0B5h ;LED out
IrInp   bit 0B3h ;IR serial input stream
IrScope bit 0B4h ;soft generate pulse for each bit
IrComd  data 41h ;IR code received

;-------------------------------------------------
IRRECV:
;-------------------------------------------------
;  Receive IR string of 14 bits.

                                ;INITIAL
IRRE0:    setb  IrOut           ;Turnoff IR Indicate
          setb  IrInp           ;Input bit

                                ;WAIT FOR FIRST bit
IRRE2:    jb    IrInp,IRRE0     ;Incoming bit is low

;-------------------------------------------------
;  Interrupt entry on first falling edge.
;  The code must be activated and the processor
;  set up to jump to here.
;         anl   IE,#11111011b  ;turn off ExtInt1
;-------------------------------------------------
```

```
                                ;START BITS (AGC)
IRRE3:    clr   IrOut           ;IR Indicator on 1st
          lcall DLYPHI34        ;3/4 bittime=1.296ms

          setb  IrScope         ;*scope pulse
          clr   IrScope         ;*scope pulse
          jb    IrInp,IRRE31    ;2nd AGC,1st half hi
          sjmp  IRRE8           ;error, resync

                                ;ADDRESS STREAM
IRRE31:   lcall DLYPHI          ;1 bit time delay
          clr   A               ;IR Rx first low lev
          mov   B,#6            ;6 more bits

IRRE4:    setb  IrScope         ;*scope pulse
          mov   C,IrInp         ;IR state to Carry
          clr   IrScope         ;*scope pulse
          rlc   A               ;Insert IR into A

          lcall DLYPHI          ;1 bit time delay
          djnz  B,IRRE4         ;Rotate 6 bits in A
                                ;CHKbit & 5 ADDRESS

                                ;DECODE ADDRESS
          anl   A,#00011111b    ;mask CHK (flip) bit
          cjne  A,#0h,IRRE8     ;<> Address 00

                                ;COMMAND STREAM
          mov   B,#6            ;6bit command stream
IRRE5:    setb  IrScope         ;*scope pulse
          mov   C,IrInp         ;IR state to Carry
          clr   IrScope         ;*scope pulse
          rlc   A               ;Insert IR into A

          lcall DLYPHI          ;1 bit delay
          djnz  B,IRRE5         ;Rotate command to A

                                ;SAVE COMMAND
          mov   IrComd,A        ;Save Command
          sjmp  IRRE9           ;good stuff

                                ;WAIT TO SYNC NEXT
IRRE8:    lcall DLYPHI15        ;invalid, wait 15
          ljmp  IRRE0           ;bit time, restart
```

```
IRRE9:    ret                  ;IRRE8 for Interrupt

;************************************************
;                      DELAY
;************************************************
;  Delay routine using NOPS and a nested loop.
;  Registers R2 & R3 are used for counting loops.
;  Count the machine cycles for each instruction.
;
;  This code has timing based on 11.059 MHz
;  crystal. If another crystal is used, the delay
;  routines must be modified.

;------------------------------------------------
DLYPHI:
;------------------------------------------------
;  For time interval at 1 bit cycle, need
;  time interval of 1.728 ms, so need
;     .001728 * 11059000/12 = 1592.5 machine cycles
;  Use 1.72, need 1585 machine cycles.

          mov    R3,#70       ;Outer loop counter
ZDLR2:    mov    R2,#10       ;Nested loop counter
ZDLR1:    djnz   R2,ZDLR1     ;Nested loop, 256 x
          djnz   R3,ZDLR2     ;Outer loop
          ret                 ;Return to call

;------------------------------------------------
DLYPHI34:
;------------------------------------------------
;  For interval at 3/ 4 of a bit cycle, need
;  time interval of 1.728 * .75= 1.296 ms, so need
;     .001296 * 11059000/12 = 1194.3 machine cycles
;  Use .75 * 1.72 gives 1188 machine cycles.

          mov    R3,#110      ;Outer loop counter
ZDLR4:    mov    R2,#4        ;Nested loop counter
ZDLR3:    djnz   R2,ZDLR3     ;Nested loop, 256 x
          djnz   R3,ZDLR4     ;Outer loop
          nop
          ret                 ;Return to call

;------------------------------------------------
```

```
DLYPHI15:
;-------------------------------------------------
;  For time interval at 15 bit cycle, need
;  time interval of 1.728 * 15= 25.92 ms, so need
;    .02592 * 11059000/12= 23,887.44 machine cycle
;

          mov    R3,#220      ;Outer loop counter
ZDLR6:    mov    R2,#50       ;Nested loop counter
ZDLR5:    djnz   R2,ZDLR5     ;Nested loop, 256 x
          djnz   R3,ZDLR6     ;Outer loop
          ret                 ;Return to call

;-------------------------------------------------
          end                 ;Program end
```



35

SERIAL CHIPS – IIC

Thought
You can not serve two masters.
Jesus of Nazareth, ~AD 30

Other chip interfaces

One of the limits of any computer is how many things can be connected. Several techniques have been discussed for expanding those options. These include expansion memory and latches, multiplexing, and matrix networks. All these methods have used parallel connections.

Because of real estate considerations, another method is frequently used for accessing special purpose chips. Serial communications with devices permits reducing the number of pins and the resulting space. As with every engineering problem, there are trade-offs. Because of the sequential bits, this method is slower and requires considerably more software.

There are several competing technologies. The two dominant protocols are serial peripheral interface (SPI) and inter integrated circuit (IIC). These two have become de facto standards for expansion of microprocessor capabilities.

The major differences are in the connections. SPI has a common bus of three wires, but it requires a separate select line for each device.

IIC has only two wires for all communications. Although the wiring is simpler for the IIC, the software is considerably more complex than that for the SPI.

There are several other protocols that are very powerful and will be mentioned only briefly. Since they are not used for on-board connections, they will not be addressed in detail.

External serial communications such as Firewire and Ethernet are accomplished using specialty chips or embedded technology in the microprocessor. Universal serial bus (USB) is used for connection to external computer peripherals. The controller area network (CAN) is used in the automotive industry.

Inter integrated circuit

During the 1980s Philips Semiconductor developed a bi-directional serial bus for inter integrated circuit (IIC) communications. Two common wires plus ground are used by all devices that are connected on the bus. The common lines are serial data line (SDA) and serial clock line (SCL).

Newer enhancements to the protocol allow 10-bit addressing and communications speeds to 400 Kbits/second.

Every device connected to the lines necessarily has a unique address. The chip can be a transmitter and/or receiver. A display is only a receiver, while a memory chip is both.

The IIC is a multiple master bus. More than one integrated circuit can initiate data transfer. The chip initiating the transfer is the bus master, during which the others are bus slaves. A microprocessor is typically the master.

The sequence of communications follows a set protocol.

1. The master sends a *Start* to get the attention of the slaves.

2. Then the master sends the *Address* of the slave along with a read or write flag.
3. All slaves compare the address with their address. If the address does not match, the slave waits for the *Stop* message.
4. If there is a match, the slave sends an *Acknowledge*.
5. When the master detects the acknowledge, it will write or read the *Data*.
6. After completion, the master will send a *Stop*. This is a signal the bus is released and a different transmission can take place.

Several versions of the microcontroller have IIC capability as an internal design configuration. The pins are located on Port 1.

P1.7	P1.6						
SDA	SCL						

Notice that these occupy the same space as the SPI protocol. Therefore, both will not exist on the same chip. If both are required, it is relatively easy to select another pin location for either protocol. Then bit bang the pins for data transfer. Several bit bang examples are provided in this and the next chapter.

IIC details

The interface is used as one de facto standard for serial communications to peripherals. The system requires the following two lines. SDA contains the I/O data. SCL is the clock and controls data availability. SCL clocks data into the device on a rising edge. It clocks out from a device on the falling edge.

The device addresses are listed. Since the microprocessor does not have the capability for IIC internally, the data must be bit-banged.

```
IicSda    bit 97H    ;port serial I/O data bit
IicScl    bit 96H    ;port clock line
```

In the most common configuration, the system will be set up as Master-Slave. The microprocessor is the master, so it controls the clock. External devices are open collector, so the lines are high when not used.

Data is sent MSB first. Any number of bits can be sent. The slave address is a unique seven-bit number. This is followed by a direction bit. Acknowledge (Ack) is sent by the receiver after each byte.

The slave address has a device number as the four most significant bits. Then three bits allow the address to select a particular device. The final bit is 0 for write and 1 for read.

For example, an eprom slave device address is 0A0h. One device has address of 0H. Follow with the direction bit. For greater than 256 bytes, the address is ORed with the device number.

Since all devices share a common line, one method of assuring the bus is not busy is to check the clock. The problem is this causes a hang-up until the lines are clear.

```
setb   IicScl        ;Scl=1
jnb    IicScl,$      ;wait til not busy
```

A good practice is to leave the start and data transfer routines with the clock low so the next transfer is ready.

There are numerous possible bit operations. These are identified in the table. The device abbreviations are M=master, R=receiver, T=transmitter

Operation	Who	Hold	Change	Clock
Free	M/M	Read-Scl=1	Read-Sda=1	
Start	M/M	Scl=1	Sda=1 to 0	
Slave Address	M/M	Scl=0	Sda=1 or 0	Scl=1
Direct Read	M/M	Scl=0	Sda=1	Scl=0
Direct Write	M/M	Scl=0	Sda=0	Scl=0
Acknowledge	R/M	Sda=0	Scl=0 to 1	Scl=0
NotAck	R/M	Sda=1	Scl=0 to 1	Scl=0
Data bits	M/T	Scl=0	Sda=1 or 0	Scl=1
Stop	M/M	Scl=1	Sda=0 to 1	

IIC sequence

The sequence of subroutines has two layers of calls. The calls are read, write, and acknowledge. Then these call other routines.

```
IICREAD:
           lcall IICSTART   ;start
           lcall IICMSBOT   ;MSB out
           lcall IICMSBIN   ;MSB in
           lcall IICACK     ;acknowledge
           lcall IICNACK    ;not acknowledge
           lcall IICSTOP    ;stop

IICWRITE:
           lcall IICSTART   ;start
           lcall IICMSBOT   ;MSB out
           lcall IICSTOP    ;stop

IICDACK:
           lcall IICSTART   ;start
           lcall IICMSBOT   ;MSB out
           lcall IICSTOP    ;stop
```

A short time delay is required for the IIC bus setup time. Four NOPs are all that is required with 7 MHz crystal. Five are adequate with the 11 MHz crystal.

IIC bit bang

The bit-banging routines are shown below. These include start, stop, acknowledge, and not-acknowledge. These routines can be used with any two bits on the processor, since they do not rely on the IIC registers of special processors.

```
;-----------------------------------------------
IICSTART:
;  Send IIC start sequence.
;  Start      M/M  Hold-Scl=1  Change-Sda=1 to 0
;
;                               ;START SEQUENCE
          setb   IicSda         ;Sda=1
          setb   IicScl         ;Scl=1
         %iicdly                ;mac setup time wait
          clr    IicSda         ;transition 1 to 0
         %iicdly                ;mac setup time wait
          clr    IicScl         ;end clock pulse
          ret
;
;-----------------------------------------------
IICSTOP:
;  Send IIC stop sequence.
;  Stop       M/M  Hold-Scl=1  Change-Sda=0 to 1
;  Leave with clock HI so the line is released.
;
;                               ;STOP SEQUENCE
          clr    IicSda         ;Sda=0
          setb   IicScl         ;Scl=1
         %iicdly                ;mac setup time wait
          setb   IicSda         ;transition 0 to 1
          ret
;
;-----------------------------------------------
IICACK:
;  Send IIC Acknowledge sequence.
;  Acknow  R/M  Hold-Sda=0  Change-Scl=0 to 1
;  Scl=0
;    This is to follow each byte received.
;
;                               ;ACK SEQUENCE
```

```
          clr    IicSda       ;Sda=0
          setb   IicScl       ;start clock pulse
         %iicdly              ;mac setup time wait
          clr    IicScl       ;end clock pulse
          setb   IicSda       ;release sda
          ret
;
;-------------------------------------------------
IICNAK:
;  Send IIC not-acknowledge sequence.
;  NotAck     R/M  Hold-Sda=1  Change-Scl=0 to 1
;  Scl=0
;    This is to follow each byte received.
;
;                             ;ACK SEQUENCE
          setb   IicSda       ;Sda=1
          setb   IicScl       ;start clock pulse
         %iicdly              ;mac setup time wait
          clr    IicScl       ;end clock pulse
          ret
;
;-------------------------------------------------
IICMSBIN:
;  Data is input serially on the IIC
;  with MSB arriving first.
;  Data bit   M/T  Hold-Scl=0  Change-Sda=1 or 0
;  Scl=1
;
;  One byte is handled in the routine. This is
;  rlc for C to go to the LSB of the byte.
;  The data is transferred on a level clock.
;  The SClk must be toggled.
;  The data port bit must be set for input.
;
          setb   IicSda       ;set port for input
          mov    B,#8         ;counter for 1 byte
;                             ;INPUT A byte
IICN1:    setb   IicScl       ;strobe clock
         %iicdly              ;mac setup time wait
          mov    C,IicSda     ;MSB to ser data
          clr    IicScl       ;
         %iicdly              ;mac setup time wait
          rlc    A            ;MSB to transmit
          djnz   B,IICN1      ;8 bits input
```

```
          clr    C             ;error flag=clr
          ret
;
;-----------------------------------------------
IICMSBOT:
;  Data is output serially on the IIC
;  with MSB first.
;  Data bit   M/T  Hold-Scl=0  Change-Sda=1 or 0
;  Scl=1

;  One byte is handled in the routine. This is
;  rlc for MSB to go to C.
;  The data is transferred on a level clock.
;
;  The SClk must be toggled.
;  Leave with data/error code in C.
;
          mov    B,#8          ;counter for 1 byte
;                               ;SHIFT OUT byte
IICT1:    rlc    A             ;MSB to transmit
          mov    IicSda,C      ;MSB to ser data
;                               ;CLOCK
          setb   IicScl        ;strobe clock
         %iicdly               ;mac setup time wait
          clr    IicScl        ;
         %iicdly               ;mac setup time wait
          djnz   B,IICT1       ;8 bits input
;                               ;CHECK ACK FRM SLAVE
          setb   IicSda        ;Sda=1, make input
          setb   IicScl        ;strobe clock
         %iicdly               ;mac setup time wait
          mov    C,IicSda      ;Sda=0 is ack
IICT2:    clr    IicScl        ;strobe clock
          ret
```

36

SERIAL CHIPS – SPI

Thought
Don't walk around with a chip
on your shoulder.
Grandpa

Serial peripheral interface

SPI was originally named and promoted by Motorola. It is also called Microwire by National Semiconductor. Enhancements include queued SPI.

SPI is a full duplex, synchronous, serial data transfer system. One machine is selected as the master, the remainder are designated as slaves. Three common wires plus ground are used by all devices that are connected on the bus. The common lines are master out slave in (MOSI), master in slave out (MISO), and clock (SCK). A select line is unique to each slave device.

Communications speeds can exceed 1 Mbits/second.

The master creates the clock by asserting the strobe pin low and high. The SPI standard allows either positive or negative clock polarity. Two different protocols can be used for clocking 8-bit data.

The MOSI pin is the data output from the master, so it is the input to all the slaves. Conversely, the MISO pin is the data input to the master, so it is the data output from one of the slaves.

The slave select is a chip select for each slave. Therefore, several lines are required for multiple slave integrated circuits. The slave selected is the only one that responds to activity on the bus. The others are high impedance, so they do not interfere.

If a device is trying to act as a master, it will assert the chip select line. Therefore, this line can be read as an input by the master to determine if there is a multiple master conflict. This will automatically disable the outputs to prevent two masters.

Details for implementation are given in a sample program in the next chapter.

Several versions of the microcontroller have SPI capability as an internal design configuration. The pins are located on Port 1.

P1.7	P1.6	P1.5	P1.4				
SCK	MISO	MOSI	/SS				

These lines are shared with the in system programming lines. To assure there are no problems with the SPI chips, disconnect the in system programming lines after the program is downloaded. This can be accomplished by an isolation chip as illustrated on the board schematic.

Analog to digital sensitivity

Analog to digital converters (ADC) are a very common type device that are connected via the serial peripheral interface (SPI). Analog values are converted to binary equivalent values and are input from the converter to the microprocessor.

An analog to digital converter (ADC) chip senses continuous signals in the range of 0-5 volt DC on its input. The device then samples the

analog data, holds the value, and converts it to a digital signal. The number of bits used for the conversion determines the sensitivity. Sensitivity is voltage range divided by digital range. An eight-bit converter has a range of 0 to 255. Therefore, each bit has a sensitivity of 5/256 = 0.0195 volts per bit.

If finer resolution is required, a larger number of bits must be used. Common sizes are 8, 10, 12, 16, and 24 bit converters. Obviously, the trade-off for more sensitivity performance is slower response and more cost.

Analog to digital noise

Noise on analog to digital converters can be a problem. This is partially a characteristic of the chips. However, there are two bigger issues. The wiring layout and the surrounding ambient noise tend to be larger challenges. Analog ground and digital ground should be totally isolated, except for a single-point connection. A ground plane may be necessary below the analog section. Alternately a shield may be necessary to cover the analog components.

Even with good design, often the low order bits of the conversion results are noisy and unstable. On an eight bit converter, this may be one or two bits. On twenty-four bit converters, it may be 4 bits. These bits represent the noise threshold. Several techniques are used to manage the unstable variations.

The simplest is to truncate the noise bits. Rotate the value right, then left with zero fill for the number of unstable bits. The disadvantage is this effectively reduces the sensitivity by a power of two.

Another techniques is using a smoothing routine. The running average is the simplest. Add the previous value and the present value. Then divide the result by two. This has the effective of slowing down response time.

In the smoothing routine, weighting can be placed on either of the values. Multiply the value by a factor before the addition. Then the result is divided by the sum of the weighting factors.

LTC 1098 clocking

The LTC1098 will be used as an application example. The techniques can be easily adapted to other components.

The master triggers the clock bit. Data comes into a device on rising edge and out from the device on a falling edge.

The sequence for reading the ADC is very dependent on the clock.

1. Start: Assert SCK high.
2. Select: Chip select high, chip select low.
3. Command: SCK low, assert data bit, wait, SCK high, wait, repeat the cycle for each bit.
4. Ready: SCK low
5. Input: SCK high, SCK low, read data bit, repeat the cycle for each bit.
6. Stop: SCK high
7. Deselect: Chip select high

LTC 1098 operation

The LTC1098 is a two-channel, 12-bit analog-to-digital converter. The master must configure the converter by writing a four bit message.

1. MSB is the start bit, which is 1.
2. Bit1,2 is the single or differential ended mux address.
   00 = differential channel 0 referenced to channel 1.
   01 = differential channel 1 referenced to channel 0.
   10 = single ended channel 0 referenced to ground.
   11 = single ended channel 1 referenced to ground
3. Bit3 determines which bit is sent first.
   1 = Most significant bit (MSB) is first with zero fill at end.
   0 = Least significant bit (LSB) first.

After the four bits are output on the serial peripheral interface, the master is configured to read twelve bits of data. If the system is using a 10-bit or 8-bit device, the extra bits read will return a 0.

Data is written to the chip on a rising clock edge. Set the data on the pin, then pulse the clock high followed by a low to complete the cycle.

Read data from a chip on a trailing edge. Pulse the clock high followed by a low, then read the data.

The LTC 1098 needs at least 10us after enabling, before the first data bit is output. This is a relatively slow device for use as a physical sensor. It has a 40Hz serial clock limit.

The twelve bits of data are read MSB first. When this is placed directly in a register, it appears as sixteen bits with zero fill. This appears to be a multiply by 2^4 or sixteen. Therefore, it is necessary to shift the data to get the desired precision. This is a bit bang procedure that can be used with any version of the processor.

Program: LTC 1098 bit-bang

```
;-----------------------------------------------
;Program: MODspi.ASM
;Update:  01 March, 2003
;Initial: May 23, 1994 @ Foxfire, MO
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)1994, 2003.  All rights reserved
;
;Purpose:
;  A set of routines are provided to read from
;  a serial peripheral interface. The device is
;  a 12-bit analog to digital converter.
;
;Processor: 8031 family
```

```
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz
;Assembler: Intel ASM51

;################################################
;                      ASSIGNMENTS
;################################################
;CONSTANTS
;------------------------------------------------
                       ;LATCH & SPI
Mosi       equ   95h   ;SPI Mosi from uC to slave
Miso       equ   96h   ;SPI Miso from slave to uC
Sck        equ   97h   ;SPI Clk

;------------------------------------------------
;DEFINED VARIABLES
;------------------------------------------------
QikB       equ    19H   ;Interrupt HEX value, msb
QikA       equ    18H   ;Interrupt HEX value, lsb

;################################################
;                      PROGRAM
;################################################
           org   00h
START:     ljmp  INITIAL

           org   0033h ;Address past vectors
           db    'Marcus O. Durham, PhD, PE'

;------------------------------------------------
           org   0080h        ;Address past reserve
INITIAL:
MAIN:
;------------------------------------------------
                               ;PROCESS
           lcall ADCIN         ;read adc on spi
           mov   A,QikB        ;MSB, channel 1
           lcall SEROUT        ;send to serial
           mov   A,QikA        ;LSB, channel 1
           lcall SEROUT        ;send to serial

MAN9:      ljmp  MAN9          ;Halt

------------------------------------------------
```

```
ADCIN:
;-------------------------------------------------
;  LTC1098, 2channel, 12-bit analog-digital convt
;  Write 4 bit control message. Then do input.
;
;  Write data to chip on rising edge:
;  Set data, pulse hi, pulse low.
;  Read data from chip on trailing edge:
;  Pulse hi, pulse lo, read data.
;
;  The LTC needs at least 10us after enable
;  before first data bit is output.
;  There is a 40Hz serial clock limit.

;  The timing is for a 7.5 MHz crystal. So the
;  delays will increase for 11 MHz.
;

                                ;STOP CLOCK
          setb  SpiClk          ;clock low for null

                                ;ENABLE CHIP SELECT
;         lcall ADCDES          ;disable & shutdown
;         lcall ADCSEL          ;enabl falling edge

                                ;READ 12 BITS, CH0
          mov   A,#0D0H         ;wr cmd1101xxxx,ch0
;         mov   A,#0F0H         ;wr cmd1111xxxx,ch1
          mov   B,#4            ;count bits shifted
          lcall SPIMSBOT        ;byte,MSB 1st

          clr   SpiClk          ;clock low for null
          mov   B,#8            ;count bits shifted
          lcall SPIMSBIN        ;byte,MSB 1st
          mov   QIkB,A          ;high byte in

          mov   B,#4            ;4 bits in
          lcall SPIMSBIN        ;byte,MSB 1st
          mov   QIkA,A          ;byte w/ 0 fill LSB

                                ;DESELECT
          setb  SpiClk          ;hi before deselect
;         lcall ADCDES          ;disable & shutdown
```

```
                                ;NEXT CHANNEL
;         lcall ADCSEL          ;enabl falling edge

ADCI9:    ret

;-------------------------------------------------
SPIMSBIN:
;-------------------------------------------------
;  Read serially from the SPI starting
;  with the most significant bit (MSB). Data is
;  clocked from the device on falling clock edge.

          setb  Miso            ;Make Sdat an input

SPMI1:    setb  Sck             ;Clk high
          nop                   ;AD7714 t6 time
          clr   Sck             ;Clock bit from ADC
          nop                   ;AD7714 t5 time
          mov   C,Miso          ;Sdat bit to C
          rlc   A               ;Rotate bit to C
          djnz  B,SPMI1         ;Get all 8 bits

          clr   Miso            ;undo Sdat input
          ret

;-------------------------------------------------
SPIMSBOT:
;-------------------------------------------------
;  Send B-bits. Most significant bit(MSB) first
;  Data is clocked in device on rising clock edge.

                                ;OUT & WAIT
SPMO1:    rlc   A               ;Rotate MSB to C
          clr   Sck             ;Clock bit to ADC
          mov   Miso,C          ;Bit to port pin
          nop                   ;Delay for ç & 40Hz
          nop                   ;Delay for ç & 40Hz
          nop                   ;Delay for ç & 40Hz
          nop                   ;Delay for ç & 40Hz
          nop                   ;Delay for ç & 40Hz

          setb  Sck             ;CLK line high
          nop                   ;Delay for ç & 40Hz
          nop                   ;Delay for ç & 40Hz
```

```
          nop                 ;Delay for ç & 40Hz
          nop                 ;Delay for ç & 40Hz
          nop                 ;Delay for ç & 40Hz

          djnz  B,SPMO1       ;Send all bits
          ret

;-------------------------------------------------
          end
```

Onboard SPI control register

Many microprocessor version now have a serial peripheral interface built onboard the chip. When using the internal connections, it is unnecessary to develop some of the timing details for their operation.

The operation is explained in detail in the section of special function registers. The SPI control register is configured when using the upper bits of port 1 for serial peripheral interface operations. For typical operations the following bits should be selected.

SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
0	1	0	1	0	1	0	1

This turns off interrupts (SPIE) and enables the SPI channel (SPE). The order of bits is MSB first (DORD). The processor is the master (MSTR). The serial clock polarity is low when idle (CPOL). The slave may remain selected between samples (CPHA). The microprocessor frequency is divide by 16 to provide a clock frequency of less than 1 MHz.

Rather than using an interrupt, the transmission complete flag (SPIF) is polled. The bit is set by the processor when transmission is complete.

Program: EEPROM SPI register ___

Because of the internal SPI capabilities of some version of the microcontroller, registers can be used to control SPI activity. This example uses two-way communications with an external EEPROM.

```
;-------------------------------------------------
;Program: MODonspi.ASM
;Initial: July 28, 2003
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c) 2003.  All rights reserved
;  Original adapted from Atmel.
;
;Purpose:
;  A set of routines are provided to write and
;  read from a serial peripheral interface.
;  The device is an eeprom.
;
;Processor: 8031 family
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz
;Assembler: Intel ASM51

;################################################
;                      ASSIGNMENTS
;################################################
;CONSTANTS
;-------------------------------------------------
                       ;SPI REGISTERS
Spcr      data  0d5h   ;SPI control register
Spsr      data  0aah   ;SPI status register
Spdr      data  86h    ;SPI data register
Spif      equ  10000000b ;interrupt flag

                       ;SPI ASSIGNMENTS
Mosi      bit   95h    ;SPI Mosi from uC to slave
Miso      bit   96h    ;SPI Miso from slave to uC
Sck       bit   97h    ;SPI Clk
CSn       bit   94h    ;device slave select
```

```
                        ;DEVICE COMMANDS
Rdsr      equ   05h     ;Read Status Register
Wrsr      equ   01h     ;Write Status Register
Read      equ   03h     ;Read Data from Memory
Write     equ   02h     ;Write Data to Memory
Wren      equ   06h     ;Write Enable
Wrdi      equ   04h     ;Write Disable

                        ;bit DEFINITION
A8        bit   acc.3 ;MSB of address
NRDY      bit   acc.0 ;hi= write cycle in progress

;################################################
;                       PROGRAM
;################################################
          org   00h
START:    ljmp  INITIAL

          org   0033h ;Address past vectors
          db    'Marcus O. Durham, PhD, PE'

;------------------------------------------------
          org   0080h        ;Address past reserve
INITIAL:
;------------------------------------------------
;  SPI master mode initialization code.
;  SPCR is setup as interrupt disable,
;  pin enable, MSB first, polarity 0, phase 1
;  clock rate /16

          setb  CSn           ;deselect AT25040
          setb  Mosi          ;initialize SPI pins
          setb  Miso
          setb  Sck
          mov   Spcr,#01010101b ;init SPI master

;------------------------------------------------
MAIN:
;------------------------------------------------
          lcall SPIEEROM      ;write/read eeprom

MAN9:     sjmp  MAIN
```

```
;-------------------------------------------------
SPIEEROM:
;-------------------------------------------------
;  Write/Read AT25C040 EEPROM via the Serial
;  Peripheral Interface (SPI).
;  Completion of programming is checked by polling
;  SPI interrupt is not used.
;  Works w/ microcontroller clk of 24 MHz or less.
;
;  Write one byte to AT25040 and verify
;  (read and compare).
;  Code to handle verification failure not shown.
;  Needs timeout to prevent write error from
;  causing an infinite loop.
;
;  Information to write has value of Data.
;  Address to write has 16-bit value of Address.
;  if not valid write & read go to

          lcall SWRENAB       ;precede ea byte wr
          mov   A,#Data       ;data
          mov   DPTR,#Address ;address

          lcall SWRBYTE       ;write
SPEE1:    lcall SRDSTAT       ;check write status
          jb    Nrdy,SPEE1    ;loop until done

          mov   DPTR,#Address ;address
          lcall SRDBYTE       ;read

          cjne  A,#Data,SPEE8 ;jump data chk fail
          sjmp  SPEE9

SPEE8:    setb  0B5h          :turn on led
SPEE9:    ret                 ;valid write & read

;-------------------------------------------------
SRDSTAT:
;-------------------------------------------------
;  Read device status. Returns status byte in A.

          clr   CSn           ;select device

          mov   A,#Rdsr       ;get command
```

```
          lcall SPIIO          ;send command

          setb  CSn            ;deselect device

          ret

;-------------------------------------------------
SWRENAB:
;-------------------------------------------------
;  Enable write.
;  Does not check for device ready before sending
;  command. Returns nothing. Destroys A.

          clr   CSn            ;select device

          mov   A,#Wren        ;get command
          lcall SPIIO          ;send command

          setb  CSn            ;deselect device

          ret


;-------------------------------------------------
SRDBYTE:
;-------------------------------------------------
;  Read one byte of data from specified address.
;  Does not check for device ready before sending
;  command. Called with address in DPTR.
;  Returns data in A.

          clr   CSn            ;select device

          mov   A,DPH          ;get high byte addr
          rrc   A              ;move LSB to carry

          mov   A,#Read        ;get command
          mov   A8,C           ;combine command
                               ;& hi bit of addr
          lcall SPIIO          ;send com&hi bit add

          mov   A,DPL          ;get low byte of add
          lcall SPIIO          ;send low byte addr
          lcall SPIIO          ;get data
```

```
          setb  CSn           ;deselect device

          ret

;-------------------------------------------------
SWRBYTE:
;-------------------------------------------------
;  Write one byte of data to specified address.
;  Does not check for device ready or write enable
;  before sending command. Does not wait for write
;  cycle to complete before returning.
;  Called with address in DPTR, data in A.
;  Returns nothing.

          clr   CSn           ;select device

          push  Acc           ;save data
          mov   A,DPH         ;get high byte of ad
          rrc   A             ;move LSB to carry

          mov   A,#Write      ;get command
          mov   A8,C          ;combine command &
                              ;high bit of address
          lcall SPIIO         ;send com& hi bit ad

          mov   A,DPL         ;get low byte of add
          lcall SPIIO         ;send low byte of
address

          pop   ACC           ;restore data
          lcall SPIIO         ;send data

          setb  CSn           ;deselect device

          ret

;-------------------------------------------------
SPIIO:
;-------------------------------------------------
;  Send/receive data through the SPI port.
;  A byte is shifted in as a byte is shifted out,
;  receiving and sending simultaneously.
;  Waits for shift out/in complete before return.
```

```
;  Expects slave already selected. Called with
;  data to send in A. Returns data received in A.

           mov   Spdr,A         ;write output data

SALL1:     mov   A,Spsr         ;get status
           anl   A,#Spif        ;check for done
           jz    SALL1          ;loop until done

           mov   A,Spdr         ;read input data
           ret
```

TLC549 clocking

The TLC549 is a simple 8-pin a-to-d converters that can work on the SPI bus. The device has an on-chip system clock that allows it to have exception rates of conversions independent of the clock line (SCK).

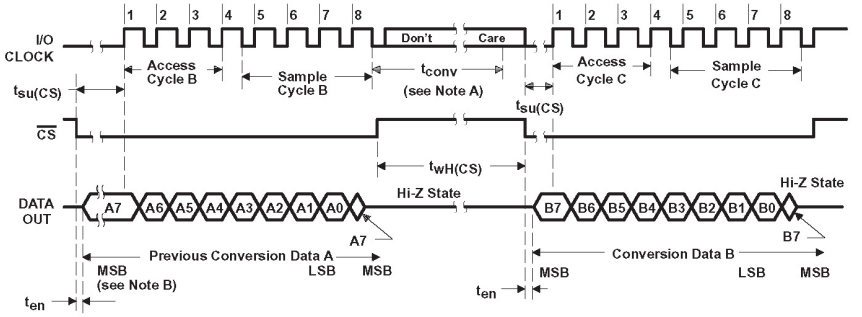
When the chip select line is asserted low, the most significant bit (MSB) is placed on the data line. The next bits (A6-A0) are placed on the data line with the falling edge of the clock line.

The hold function begins with the eighth clock cycle. After the eight clock cycle, chip select must go high, or the clock line must remain low for at least 36 internal system clock cycles to allow completion of the conversion.

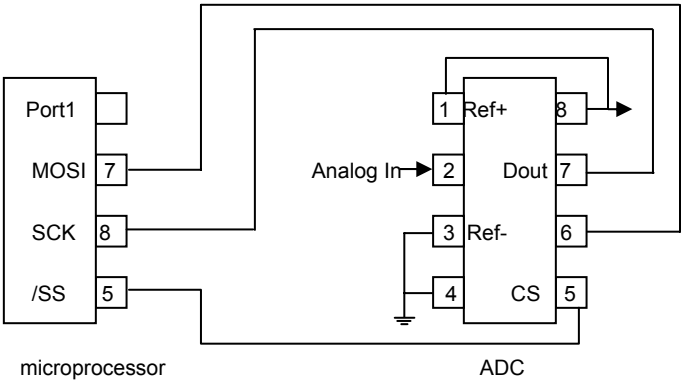
If the chip select is kept low for multiple conversion, the clock line (SCK) must remain glitch free or the microprocessor and the device will lose synchronization.

The clock line must maintain each state for more than 404 ns. The conversion time takes a maximum of 17 microseconds. The maximum clock frequency is 1.1 MHz.

The timing diagram illustrates the process


1 2 3 4 5 6 7 8
I/O CLOCK
Don't Care
Access Cycle B
Sample Cycle B
tconv (see Note A)
Access Cycle C
Sample Cycle C
tsu(CS)
CS
twH(CS)
DATA OUT
Hi-Z State
A7 A6 A5 A4 A3 A2 A1 A0
B7 B6 B5 B4 B3 B2 B1 B0
Previous Conversion Data A
Conversion Data B
MSB (see Note B)
LSB
MSB
ten

Circuit: SPI


Port1
MOSI 7
SCK 8
/SS 5
1 Ref+ 8
Analog In 2 Dout 7
3 Ref- 6
4 CS 5
microprocessor
ADC

⇐ ⇑ ⇒

37

PROJECT 12 - A TO D CONVERTER

Thought
The whole world is an analog stage,
digital only plays bit parts.

Project 12: Analog / digital converter

Purpose: To develop peripheral expansion with serial devices.
To evaluate continuous or analog inputs.

Preamble:

Numerous peripheral devices are often desired for a computer system. These may be memory, analog interfaces, or sensors. All these devices require data input and output. If parallel communications is used for the devices, they have a large footprint that takes occupies a large amount of board space. An alternative is to perform serial communications between the processor and the peripheral device. This requires many fewer wires and typically the device has a size of only 8 pins. As with every engineering design, there is a tradeoff. The speed is slower, but the area is much smaller. This is less costly and has better space performance.

Several protocols are used for serial interfacing. One of the simplest to implement is the serial peripheral interface (SPI). It requires four lines – a data out line, a data in line, a clock line, and a select line.

The data and clock lines can be common with other SPI devices. However, each requires its own select.

An alternative is the inter-integrated circuit (IIC) format, which requires only two lines – data and clock. The select is made by addresses transferred on the data line. The tradeoff is the hardware requirements are less, but the software to implement the addressing is more complex.

Some microprocessors have SPI protocol as part of their system. Others require it to be implemented in software through three or four pins. For example, some versions use SPI to load the on-board memory.

Many converters have the capability of sampling more than one channel of analog input. A control must be sent to the device to select which channel to convert. With an SPI chip, the control would be an address sent to the ADC.

The ADC also requires a reference voltage as an input. The internal circuitry is actually a comparator that evaluates the analog input in relation to the reference voltage. Therefore, the converted input often cannot quite reach the reference value.

The number of bits used for the conversion determines the sensitivity. Sensitivity is voltage range divided by digital range. An eight-bit converter has a range of 0 to 255. Therefore, each bit has a sensitivity of 5/256 = 0.0195 volts per bit.

Disconnect the in-system programming cable to eliminate any noise on the SPI lines.

Plan:

Add an analog to digital converter to the microcontroller. Use SPI connections.

Preparation:

Use the power supply as the reference voltage. Connect the three control lines to a port or memory addressed latch on the processor. Connect a variable voltage source for the input signal.

Procedure:

First, write a subroutine to perform SPI communications. This will include selecting the device, setting up data, and clocking the data out of the chip.

Next, write a routine to select the ADC chip and the appropriate channel. Call the SPI routine to perform the transfer.

Finally convert the data from a digital number to a number corresponding to the voltage. Display the results on the serial line, LCD, or LEDs.

Presentation:

Demonstrate the ADC operation by reading a DC voltage of a known value. Display the digital and voltage values.

Program sample example

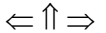
The exemplar program is similar to the project specifications. However, it contains elements that should be modified to complete the project as required.

```
;-------------------------------------------------
;Program: MODadc.ASM
;Initial: 13 November 2003
;By:      Dr. Marcus O. Durham, PhD, PE
```

```
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)2003.  All rights reserved
;
;  Modified code from original by Matt Olson.

;Purpose:
;  A set of routines are provided to read from
;  a serial peripheral interface. The device is
;  an 8-bit analog to digital converter. This uses
;  the SPI registers
;
;Processor: 8031 family
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz
;Assembler: Intel ASM51

;################################################
;                      ASSIGNMENTS
;################################################
;CONSTANTS
;------------------------------------------------
                       ;SPI
SpCr      equ   0D5h  ;SPI control register
SpDr      equ   86h   ;SPI data register
SpSr      equ   0AAh  ;SPI status register
AdCs      equ   94h   ;adc chip select

;################################################
;                      PROGRAM
;################################################
          org   00h
START:    ljmp  INITIAL

          org   0033h         ;Address past vector
          db    'Marcus O. Durham, PhD, PE'
;------------------------------------------------
          org   0080h        ;Address past reserve
INITIAL:
          lcall SPIINIT      ;init spi registers

;------------------------------------------------
MAIN:
```

```
;-------------------------------------------------
                              ;PROCESS

          lcall SPIREAD       ;read adc on spi
          lcall DISPLAY       ;show value someplace

MAN9:     ljmp  MAN9          ;repeat

;-------------------------------------------------
SPIINIT:
;-------------------------------------------------
;  Setup SPI:
;  Disable interrupts; enable spi; msb first;
;  master; clk low when idle; cpha=1; f=osc/64

          mov   SPCR,#01010110B
          ret

;-------------------------------------------------
SPIREAD:
;-------------------------------------------------
;  Read byte from ADC.
;
                              ;CHIP SELECT LOW
          clr   AdCs          ;enable chip select

          nop                 ;delay > 1.4usec
          nop
          nop
          nop
          nop
          nop

;                             ;INIT data REGISTER
          mov   Spdr,#0AAH    ;write anything

                              ;CHECK STAUS
SPR1:     mov   A,Spsr        ;read status reg
          rlc   A             ;move spif to carry
          jnc   SPR1          ;<>1, so no data

;                             ;READ data REG
          mov   A,Spdr        ;read data from reg
          setb  AdCs          ;disable select line
```

```
                                ;RESET STATUS REG
          mov   SPSR,#0         ;clear spsr

          ret                   ;got byte, get out

;-----------------------------------------------
          end                   ;Program end
```



38

WAVEFORM SYNTHESIS

Thought
Quality = excellence.
Professor Durham

Real world output

A computer operates in a discrete environment where everything is either true or false, on or off, high or low, one or zero. The perception is that the real world is an analog environment. In reality, physical activities change very slowly and can be sampled easily. Most natural physical phenomenon change slower than 30 times a second.

The slowness of perception can be used to make an apparent analog signal. A digital to analog converter (D/A) is a chip that takes a binary input and converts it to a voltage.

A microprocessor can be used to generate a variety of waveforms. This is accomplished by outputting a digital value for the wave. The output is used as a signal for a digital to analog converter. The D/A converter creates a continuous output from the previous digital value.

Each wave can be described by a magnitude at some time. By accessing of magnitudes at regular intervals of time, a representation of the wave can be created.

The magnitudes can be held constant and the time interval between outputs can be adjusted to vary the frequency. Alternately, the time interval can be held constant and the table value multiplied by a gain or scaling value to provide varying magnitude for the signal.

Multiple port latches or sequential bits can be sent for greater precision. A full range capability of 8-bits is available on a port. Therefore, the range of values in the table should be 0 to 255 to obtain full sensitivity. The eight bits can include negative numbers. The range of positive values is 0 to 127 (00 - 3fh). The negative values are –1 to –128 (0FFh - 40h).

Negative numbers are acceptable for double-sided digital to analog converters. However, a single-sided or monolithic device will require all positive numbers.

By clever programming a single half-wave table can be used for two-sided waves. The complement of the table is used for the negative half-cycle. The values are directly output to a double-sided converter.

To obtain values for a single-sided converter, 80h must be added before the value is output. The 80h causes a positive shift in magnitude so that all the values are greater than zero.

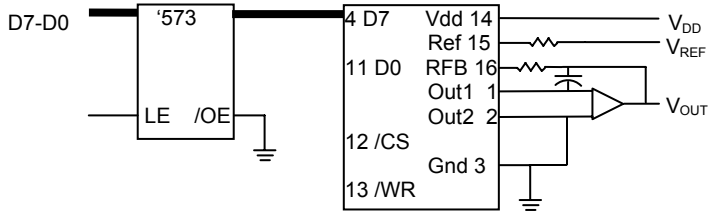
Sensitivity

The sensitivity is determined by the ratio of the volts to bits. When using an 8-bit device, the range of values is 0 to 255. A common power system is 5 volts. Therefore this device sensitivity is 5/256 = 0.0195 V/bit.

Circuit: digital to analog

A typical 8-bit D/A converter has a relatively low level output power. To drive a five-volt signal, an op-amp should be connected. A 5-volt digital to analog converter with a parallel input is the simplest design to implement with a microprocessor. The particular layout shown is for a Max DAC5480. It is for reference only. Any other design will work similarly.

Alternate designs use a serial input and many have multiple converters on board.



The converter circuit shows only three discrete elements. The resistors are not required unless an adjustable gain is desired.

Element	Value	Location
R ref	2 K pot	REF
R rfb	1 K	RFB
C1	10 pF	Out

Software

The software to implement the digital to analog converter is very straightforward. It is simply a table lookup with the results display on a port.

Options to the code are to complement negative numbers. Another option is to add 80h for a range shift for a single sided DAC. Then the data can be complemented if the waveform is mirrored.

By creating a series of waveforms, combining them, and shifting the frequency of output, a wide variety of signals can be generated. An amplifier on the op amp will provide audio tone levels.

The table that is created contains the wave value. Take the waveform and divide into as many entries as necessary to adequately describe the wave. Calculate the value at precise intervals, then place these entries in the table.

Consider a sine wave. It can be describe by one-quarter wave. A six table entry is likely adequate. Calculate the sine at 15 degree intervals to get six entries in 90 degrees. For the second quarter wave, the table is read in reverse. Then for the second half, the values are complemented.

A sawtooth wave can be defined with just two points. Other wave forms are similarly calculated.

⇐⇑⇒

39

PROJECT 13 – D TO A CONVERTER

Thought
Aphorism:
A concise formulation of a principle, truth, or sentiment.
Definition

Project 13: Analog output _____

Purpose: To interface a D/A converter.
To use the system to create waveforms.

Preamble:

The D/A converter allows the output of a precise voltage level from a digital signal. Coupling with a microprocessor allows the output of this voltage to appear at a precise time. Thus the system allows the output of voltage as a function of time.

The D/A converter used is similar to a DAC0806, which is equivalent to the popular 1408. This is a device with eight-bit inputs, which have a range of 0-255.

Use an op-amp on the output. Set the range of the op-amp to yield 0 volts when a digital 00 is applied. Adjust the output to +5 volts when 255 is sent to the DAC. The +5 level can be adjusted by a potentiometer. Use an oscilloscope to make voltage measurements.

The sensitivity is determined by the ratio of the volts to bits. Therefore this device sensitivity is 5/256 = 0.0195 V/bit.

The integrated circuit (IC) will be interfaced to an I/O port or a memory-mapped latch.

Plan:

Write programs to output saw-tooth and sine waveforms.

Procedure:

Write a routine to generate a saw-tooth waveform by continually incrementing a register and sending the register contents to the D/A converter. The frequency is controlled by a DELAY subroutine.

Write a routine to generate a sine waveform. To do this, determine a table of sine values from 0 to 90 degree in 10-degree increments. Convert the sine values to the appropriate HEX code for the D/A converter. Store the values in a lookup table. Only store values once. Use two's complement arithmetic to calculate the negative values for quadrants 3 and 4. The frequency should be variable as in the saw tooth program.

Notice that the values of the sine need only be known for one quadrant of the waveform. The other segments may be formed based on these values.

Presentation:

The routine should be such that it can be easily modified to vary the frequency. Vary the frequency by changing the delay time between successive outputs to the D/A converter.

⇐ ⇑ ⇒

Program sample example

The exemplar program is similar to the project specifications. However, it contains elements that should be modified to complete the project as required.

```
;-------------------------------------------------
;Program: MODDac.ASM
;Update:  28 February 2003
;Initial: 17 October 1988
;
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)1988, 2003.  All rights reserved
;
;Purpose:
;  Use digital to analog converter for waveform
;  synthesis.
;
;Processor: 8031 family
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz
;Assembler: Intel ASM51

;################################################
;CONSTANTS
;################################################
                        ;OUTPUT
Pio       equ   90h     ;port for i/o

;################################################
;                       PROGRAM
;################################################
          org   00H
START:    ljmp  INITIAL

          org   0033h
          db    'Marcus O. Durham, PhD, PE'
```

```
;-------------------------------------------------
          org   0080H         ;Addres past reserve
INITIAL:
;-------------------------------------------------
                              ;INITIALIZE
          lcall DACINIT       ;initialize counters

;-------------------------------------------------
MAIN:
;-------------------------------------------------
;  Display a wave form.

                              ;PROCESS
          lcall DACWAVE       ;display a wave
MAN9:     ljmp  MAIN          ;Repeat

;*************************************************
;                      DIGITAL ANALOG CONVERTER
;*************************************************
DACINIT:
;-------------------------------------------------
;  Initialize waveform pointers.

                              ;INIT
          mov   R2,#0         ;table entries pntr
          mov   R3,#0         ;+half wave=0,neg=1
          ret

;-------------------------------------------------
DACWAVE:
;-------------------------------------------------
;  Produce a wave.

                              ;PROCESS
          mov   DPTR,#TabWave1;table with wave
DACW1:    mov   A,R2          ;table pointer
          movc  A,A+DPTR      ;get wave value

          cjne  R3,#1,DACW2   ;<>1,then pos
          cpl   A             ;else,neg:comp table

DACW2:    add   A,#80h        ;rang shift monolith
```

```
          mov   Pio,A          ;out to DAC

          inc   R2             ;incr table pointer
          cjne  R2,#entry,DACW3 ;if<entry, next
          sjmp  DACW5

DACW3:    lcall DELAY          ;frequency delay
          sjmp  DACW1          ;next value
                               ;COMPLEMENT CYCLE
DACW5:    mov   R2#0           ;reinit data pointer
          mov   A,R3           ;to neg half cycle
          cpl   A
          mov   R3,A
          sjmp  DACW1          ;repeat cycle

DACW9:    ret

;-------------------------------------------------
TabWave1:
;-------------------------------------------------
;  Make entries for half cycle.
          db    01h            ;first entry

;-------------------------------------------------
          end
```

40

PROJECT 14 - PHOTOSENSOR

Thought
Principle 1:
Never criticize, ever.
Dale Carnegie

Project 14: Barcode reader _____

Purpose: To interface a uC with a common sensor system.
To use another interpretation of time pulse.

Preamble:

Sensors are the most important part of data acquisition systems used in real-time computing. The photosensor is a widely used detector. Interfacing a sensor system to the uC depends on the application. As with any engineering problem, each method has its advantages.

Bar code readers are commonly used in department stores, on food items, and on rail cars. Hence bar code readers are an important system for microcomputer applications.

Plan:

Write a program that will utilize a software design for the reader. The sensor uses a simple photo-detector device.

Preparation:

The sensor is provided, but an interface circuit is required.

Procedure:

The sensor input is a bit from the port of your choice. The software will count the width of dark and bright bars in the bar codes. The width is determined by counting the number of samples while the input bit is at a one-state. The program must calibrate the width of the entire bar code. Next, group the number of dark & bright bars into wide & narrow categories. There will be four groups. These are dark wide, dark narrow, bright wide, and bright narrow. Use this data with an encryption algorithm.

Presentation:

Print the bar code to the corresponding number on the display or serial line.

⇐⇑⇒

41

PROJECT 15 – ANALOG CONTROL

Thought
Principle 2:
Give positive affirmation.
Dale Carnegie

Project 15: Pulse width modulation _

Purpose: To use a timer interrupt.
To interface with a pulse width modulation type device.

Preamble:

There are many analog devices that are controlled by a digital based system. Variable speed motor control is one popular example. The digital control of a physical variable such as speed can be implemented in several ways.

In one approach, the digital signal is used with a D/A converter. The converter is followed by a linear power amplifier to produce voltage with a variable amplitude. The voltage is applied to the motor. The cost of the D/A converter and linear amplifier are significant disadvantages of this scheme.

An alternative approach is often used in digital control. The digital signal is used to modulate the on-off application of constant

amplitude voltage to the motor. The modulation can take two forms: (1) on-off pulses of constant duration but with a variable number of pulses per second, and (2) on-off pulses at a constant repetition rate but with a variable width for each pulses.

The microprocessor has a powerful feature that includes two built-in timers. Each timer can be set to different modes to suit the application. Programs should use Timer 0, since Timer1 is used for serial communications.

Using an internal timer interrupt is strongly recommended for this project. The timer interrupt operates much like the external interrupt. Consult the reference information for further details.

The device used to modulate the currents is essentially a solid-state switch. A transistor is used in the DC case and an optically coupled silicon controlled rectifier (SCR) in the AC case. Transistor operation should be familiar, but SCR operation may not be as well understood.

A simplified model of an SCR is as follows.

1. With no gate current, the SCR will act like an open circuit in a "blocking state".
2. With a small gate current, the SCR "snaps on" and conducts like a high current diode in a "conducting state".
3. The SCR remains in the conducting state until the current through the SCR goes to zero. Then the SCR returns to the "blocking state".

Silicon controlled rectifiers are well suited for modulating AC current. They can be triggered (turned on) at any given time in the AC positive half-cycle. The device will turn off when the current goes to zero at the beginning of the negative half-cycle. Thus a variable width pulse derived from a half cycle may be produced.

In order to time the trigger pulse, the beginning of the cycle must be known. An additional circuit called a zero crossing detector (ZCD) takes care of this task. The ZCD output is a TTL level square wave whose transitions indicate a zero crossing of the AC line. Note that

the trigger pulse to the SCR must be turned-off before the next cycle begins or the SCR will turn-on again at the beginning of the cycle.

Plan:

This project uses both types of modulation for the control of light intensity to 6 V DC and 120 V AC lamps.

Preparation:

Wire the circuit to contain a modulating circuit and lamp. Each circuit has a TTL level control line, which is to be connected to an I/O port of the microprocessor.

The control lines in the circuit may source up to 15 mA of current. This is more than the I/O ports can sink so hardware buffering is needed. A 7404 may be used as the buffer.

The keyboard system on the microprocessor is used to select the desired light intensity for each of the control schemes. The keys should be priority encoded as shown in the following table. The keys settings should also be echoed to the display system.

Keypad	light intensity
8	full on
7	
6	.
.	.
.	.
.	.
1	.
0	off

Procedure:

I. 6 VDC Lamp Control

Write a program to control the DC lamp. The program should generate a constant pulse with a 1 ms "on" time. The intensity is controlled by varying the number of pulses per second from 0 pulses per 8.0 ms to 8 pulses per 8.0 ms as selected from the keyboard. Remember that the gate of the PNP transistor must be at 0 V for the transistor to turn-on. An internal timer interrupt will be handy for the delay routine.

II. 120 VAC lamp control

Write a program to control an AC lamp. The program should poll the ZCD for the beginning of the AC cycle, delay for the selected amount of time, and then send a trigger pulse to the SCR. The half cycle should be divided into 8 parts corresponding to the 8 intensity levels. Note that 8 equally spaced turn-on times for the SCR will not give 8 equally spaced intensity levels

Presentation:

Change the intensity of the light when a keypad key is selected for each procedure.

⇐⇑⇒

42

PROJECT 16 - DIGITAL FEEDBACK

Thought
Principle 3:
Find out what the other person wants
and help them get it.
Then you will get what you want.
Dale Carnegie

Project 16: DC motor speed control _

Purpose: To utilize a pulse width modulation mechanism.
To implement feedback theory in a digital system.

Preamble:

A robot uses a motor as the engine. Hence, controlling the motor will be the most crucial task in the robotic system. Most robots are more concerned with position rather than with the speed, but a high performance robot will deal with both speed and position.

A feedback control system is a critical component of speed motor control. Several techniques may be applied to solve the feedback problem. One solution is a phase-lock-loop (PLL) technique. However, a simple feedback system may be used which can neglect several factors necessary for a more sophisticated control scheme.

The speed is detected by a photo-sensor. Since the output signal of the sensor is very small, an amplifier is needed to obtain an adequate output level. The system will maintain the speed of the motor by comparing the detected speed to the desired (setting) speed.

This mechanism is the fundamental feedback control system from basic electrical circuit theory. From machine theory, it is apparent that the DC motor speed will increase at a rate which is proportional to the total current in a period of time.

A simple driver for the motor can establish the total current applied in a certain period of time. This driver is easily turned on and off. A driver that can be interfaced to a digital system will be able to detect digital signals (hi-lo or on-off).

Use the microprocessor as a controller. Maintain the system stability by comparing the output value of the system (motor speed) to the input value (desired speed). The task should use only digital values.

Plan:

Drive the motor from the microcomputer using a pulse-width-modulation technique. The pulse width will be the length of time when the motor driver is turned-on.

Preparation:

To control the DC motor speed, build the circuits to convert the speed to a digital input signal and the digital output signal to a current. Using an optoisolator is strongly recommended. The isolator protects the computer system from high current/voltage transients.

Procedure:

Implement the simple feedback system for DC motor speed control. The preset speed will be input from the keyboard. Operate the system with at least 4 different speeds. Choose the appropriate speeds for your circuit and motor. Use an internal timer interrupt added convenience

Presentation:

The report should include a brief explanation about the following:

1. Why you chose the speeds used.
2. How you calculated the speed of the DC motor.
3. How the system performs at various speeds in term of stability & response.

⇐⇑⇒

43

MATH FUNCTIONS

Thought
1+1 = 10
Binary thinking

Arithmetic

It has been argued that all arithmetic is simply addition. Subtraction is addition with negative numbers. Multiplication is successive addition. Division is a contorted combination of subtraction.

The processor actually has an elegant arithmetic set of operations. The fundamental operations are add (add) and subtract with borrow (subb). Many other machine use shift for multiply and divide by 2. A very powerful hardware multiply (mul) and divide (div) are available.

All the operations are structured for eight bits. In most calculations that is too limited. The following routines were developed to expand to a larger quantity of bits. These are based on routines in very early versions of Intel applications notes. These routines were modified to fit my applications, so the original copy is no longer available to identify as a reference.

Extended precision

```
;************************************************
;                      MATH
;************************************************
;
;  Math routines are used in virtually every
;  process. The range is addition to division and
;  square root.
;
;  I have used a variety of math structures over
;  the years. My objective here is threefold.
;  1. minimize the number of special routines
;  2. minimize the number of lines of code
;  3. minimize the number of ram variables.
;
;  Interestingly, all these criteria are
;  synergistic. The one possible downside is the
;  routines may be slightly longer in time.
;
;  The key was to reduce the routines to the most
;  basic concepts. This is alternative to doing
;  every possible combination in detail.
;
;  Indirect addressing is used extensively.
;  To minimize ram space, this means variables
;  are redefined in every procedure.
;  Nevertheless, there is a common structure.
;
;  Ram memory is precious. The following variables
;  are used throughout. Locations are allocated
;  for data. Some address use two different names,
;  just for convenience.
;
;FraD   equ 3BH  ;4 bytes for fraction
;FraA   equ 38H  ;same space as upper Gap
;RemD   equ 33H  ;4 bytes for remainder
;RemA   equ 30H  ;same space as upper Tmp
;GapH   equ 3BH  ;8 bytes
;GapA   equ 34H  ;BCD digits for display & RESULTS
;TmpH   equ 33H  ;8 byte
;TmpA   equ 2CH  ;temporary or scratch
;HexD   equ 2BH  ;double word
;HexA   equ 28H  ;hexidecimal for all calculations
```

```
;  R6  =    size, mul
;  R5  =    carry, mul
;  R4  =    mul
;  R3  =    loop control iteration
;  R2  =    loop control & size number of bytes
;  R1  =    @ source
;  R0  =    @ destination
;
;  To enter the basic routines, place variables
;  and size to count info in R0, R1, R2
;  using something like the JHEX4 subroutine.

;-------------------------------------------------
JHEX4:
;-------------------------------------------------
;  Standard arrangement for most math.
;  Do this or a similar procedure before calling
;  the math manipulation process.

          mov   R0,#HexA      ;destination
          mov   R1,#TmpA      ;source
          mov   R2,#4         ;4 bytes
          ret

;-------------------------------------------------
JCLEAR:
;-------------------------------------------------
;  Clear variables @ R0.
;  R2 = # bytes
                              ;ZERO GOES IN
          mov   @R0,#0        ;clear register
          inc   R0            ;next bit
          djnz  R2,JCLEAR     ;continue loop

          ret

;-------------------------------------------------
JCOPY:
;-------------------------------------------------
;  Copy source to destination.
;  Do not change source.

          mov   A, @R1        ;source
```

```
          inc   R1
          mov   @R0, A        ;move to destination
          inc   R0            ;next
          djnz  R2, JCOPY     ;loop
          ret

;-------------------------------------------------
JCPL:
;-------------------------------------------------
;  Complement the destination.
;  Simply change all bits.

          mov   A, @R0        ;destination
          cpl   A             ;complement
          mov   @R0, A        ;save at same locate
          inc   R0            ;next
          djnz  R2, JCPL      ;loop
          ret

;-------------------------------------------------
JNEG:
;-------------------------------------------------
;  Negate the destination
;  It is complement and add 1
;  That is called twos complement.

          setb  C             ;set c
JNEG1:    mov   A, @R0        ;destination
          cpl   A             ;complement
          addc  A, #0         ;add 1
          mov   @R0, A        ;save
          inc   R0            ;next
          djnz  R2, JNEG1     ;loop
          ret

;-------------------------------------------------
JINCSC:
;-------------------------------------------------
;  Increment the destination.
;  SC: set carry, signed
;  WC: with carry that comes in.

          setb  C             ;set
;---------
```

```
JINCWC:
;---------
          mov   A, @R0         ;destination
          addc  A, #0          ;add C to byte
          mov   @R0, A         ;save
          inc   R0             ;next
          djnz  R2,JINCWC      ;loop
          ret

;------------------------------------------------
JDECSC:
;------------------------------------------------
;  Decrement the destination
;  SC: set carry
;  WC: with carry that comes in.

          setb  C
;---------                     ;WITH CARRY
JDECWC:
;---------
          mov   A, @R0         ;destination
          subb  A, #0          ;take away the carry
          mov   @R0, A         ;save
          inc   R0             ;next
          djnz  R2,JDECWC      ;loop
          ret

;------------------------------------------------
JINC10:
;------------------------------------------------
;  Increment the value by 10.
;  This is a decimal increment

          inc   @R0            ;destination
          mov   A, @R0
          cjne  A,#10,JINC19   ;<>10, so exit
          mov   @R0, #0        ;=10,so set digit= 0
          inc   R0             ;next
          djnz  R2,JINC10      ;loop
JINC19:   ret

;------------------------------------------------
JSHIFTL:
;------------------------------------------------
```

```
;  Rotate left carry by 1 bit at time.
;  SHIFT  clears the C
;  ROTATE leaves C bit as it comes in.
;  R2 = # bits
                                ;ROTATE C=0
          clr   C
;---------                      ;ROTATE CARRY
JROTATEL:
;---------
          mov   A,@R0           ;point to HexS etc
          rlc   A
          mov   @R0,A           ;update variable
          inc   R0              ;next bit
          djnz  R2,JROTATEL     ;continue loop

          ret

;-------------------------------------------------
JIFZERO:
;-------------------------------------------------
;  Test if the value is zero.
;  C=1, if zero
;  C=0, in not zero

          clr   C
JZER1:    mov   A, @R0          ;get value
          jnz   JZER9           ;notZero
          inc   R0              ;next
          djnz  R2, JZER1       ;loop
          setb  C
JZER9:    ret

;-------------------------------------------------
JIFLESS:
;-------------------------------------------------
;  If @R0 < @R1, then C=1.
;  Subtract but do not save results.
;  R2 = # bytes
                                ;subb BUT NOT SAVE
          clr   C

JLES1:    mov   A,@R0           ;point to HexS etc
          subb  A,@R1           ;subtract but not
save
```

```
          inc   R0            ;next bit
          inc   R1
          djnz  R2,JLES1      ;continue loop

          ret

;-------------------------------------------------
JSUBSC:
;-------------------------------------------------
;  Subtract  @R0 = @R0 - @R1.
;  SC: set carry, Signed
;  CC: clr carry, Unsigned
;  R2 = # bytes
                              ;SIGNED
          setb  C             ;set borrow
          sjmp  JSUB1
;---------                    ;UNSIGNED
JSUBCC:
;---------
          clr   C
JSUB1:    mov   A,@R0         ;point to dest
          subb  A,@R1         ;@R0 = @R0 - @R1
          mov   @R0,A
          inc   R1
          inc   R0            ;next bit
          djnz  R2,JSUB1      ;loop

          ret

;-------------------------------------------------
JADDCC:
;-------------------------------------------------
;  @R0 = @R0 + @R1
;  R2 = # bytes
                              ;add & C=0
          clr   C

JADD1:    mov   A, @R0
          addc  A, @R1
          mov   @R0, A
          inc   R0
          inc   R1
          djnz  R2, JADD1     ;loop
```

```
          ret

;-------------------------------------------------
JMULX16:
;-------------------------------------------------

          mov   R2,#16
          mov   R0,#HexA
          ljmp  JMULXR1

;-------------------------------------------------
JMULXR1:
;-------------------------------------------------
;  Multiply @R0 by one byte in R1.
;  R2 = number of bytes in R0
;  R3:(@R0) = R1*(@R0)

                              ;
          mov   R3, #0        ;clear carry byte
                              ;LOOP
JMXR1:    mov   A, @R0        ;LSByte
          mov   B, R1         ;1 byte multiplier
          mul   AB            ;1 byte multiply

          add   A, R3         ;add Carry byte
          mov   @R0, A        ;save LSB of mult

          inc   R0            ;next higher byte
          mov   A, B          ;get MSB of mult
          addc  A, #0         ;add Carry bit
          mov   R3, A         ;save as Carry byte
          djnz  R2, JMXR1     ;loop on size

                              ;TERMINATE
          ret


;+++++++++++++++++++++++++++++++++++++++++++++++++
JDIV32:
;-------------------------------------------------
;  Standard arrangement for multiply.
;  Do this or a similar procedure before calling
;  the math manipulation process.
```

```
          mov   Size,#4       ;number of bytes
          ljmp  JDIVIDE

;-------------------------------------------------
JDIV16:
;-------------------------------------------------
;  Standard arrangement for multiply.
;  Do this or a similar procedure before calling
;  the math manipulation process.

          mov   Size,#2       ;number of bytes
          ljmp  JDIVIDE

;-------------------------------------------------
JDIVIDE:
;-------------------------------------------------
;--Entry: HexA, TmpA
;--Exit:  GapA, FraA
;  Divide routine for up to Size=4 bytes.
;
;  Operator / Divisor = Quotient.Fraction + Remain
;  HexA / TmpA = GapA . FraA  + RemA
;
;  R3 =          ;counter for # units
;  R2 =          ;number of bytes
;  R1 =          ;source
;  R0 =          ;destination
;
;  Divide and Fraction loop have the following.
;  Loop times: R3= 8*SizeX; R3>0; R3 is decrement
;  Results: Rem:Op<<=1; C=Rem/Divisor
;  Rem -=Divisor

                                ;ZERO VARIABLES
          mov   R0, #RemA       ;remain
          mov   R2, #4          ;all
          lcall JCLEAR          ;Rem = 0

          mov   R0, #GapA       ;quotient
          mov   R2, #4          ;all
          lcall JCLEAR          ;Quo = 0

                                ;DIVIDE LOOP SIZE
          mov   A, Size         ;Size
```

```
          rl    A
          rl    A
          rl    A
          mov   R3, A          ;R3=8*Size =loop

                               ;DIVIDE LOOP
JDIV1:    lcall JDIV           ;divide
          mov   R0, #GapA      ;quotient
          mov   R2, Size
          lcall JROTATEL       ;Quo= Quo<<1 | C
          djnz  R3, JDIV1      ;loop

                               ;CLEAR FRACTION
          mov   R0, #FraA      ;fraction
          mov   R2, Size
          lcall JCLEAR         ;Fract = 0

                               ;FRACTION LOOP SIZE
          mov   A, Size        ;size
          rl    A
          rl    A
          rl    A
          mov   R3, A          ;R3 = 8*SizeX

                               ;FRACTION LOOP
JFRAC:    lcall JDIV           ;divide
          Mov   R0, #FraA      ;fraction
          Mov   R2, Size       ;size
          lcall JROTATEL       ;Frac= Fract<< 1 | C
          djnz  R3, JFRAC      ;loop

          ret

;-------------------------------------------------
JDIV:
;-------------------------------------------------
;  Rem:Op<<=1; C=Rem/Divisor; Rem -= Divisor

                               ;DIVIDE
          clr   C
          mov   R0, #HexA      ;operator
          mov   R2, Size       ;# bytes
          lcall JROTATEL       ;C:Op = Op*2
```

```
          mov   R0, #RemA      ;remainder
          mov   R2, Size       ;# bytes
          lcall JROTATEL       ;Rem= Rem*2 + C

          mov   R0, #RemA      ;remainder
          mov   R1, #TmpA      ;divisor
          mov   R2, Size       ;# bytes
          lcall JIFLESS        ;C=1 if @R0<@R1
          jc    JDIV9          ;if Rem>= Divisor

          mov   R0, #RemA      ;remainder
          mov   R1, #TmpA      ;divisor
          mov   R2, Size       ;# bytes
          lcall JSUBCC         ;Rem -=Divisor,unsgn

JDIV9:    cpl   C              ;C = !C
          ret

;-------------------------------------------------
JMUL32:
;-------------------------------------------------
;  Standard arrangement for multiply.
;  Do this or a similar procedure before calling
;  the math manipulation process.

          mov   ArgH,#HexA     ;destination
          mov   ArgT,#TmpA     ;source
          mov   ArgG,#GapA     ;results
          mov   SizeT,#4       ;4 bytes in Tmp
          mov   Size,#4        ;4 bytes in Hex

          ljmp  JMULTIPLY

;-------------------------------------------------
JMUL16:
;-------------------------------------------------
;  Standard arrangement for multiply.
;  Do this or a similar procedure before calling
;  the math manipulation process.

          mov   ArgH,#HexA     ;destination
          mov   ArgT,#TmpA     ;source
          mov   ArgG,#GapA     ;results
          mov   SizeT,#2       ;2 bytes in Tmp
```

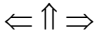
```
          mov   Size,#2       ;2 bytes in Hex

          ljmp  JMULTIPLY

;-------------------------------------------------
JMULTIPLY:
;-------------------------------------------------
;  Multiply multiple bytes. Variables are used
;  as pointers to the values.
;  @ArgG = @ArgH * @ArgT
;  Gap = Hex * Tmp
;
;SizeT  equ 33H  ;same space as Tmp upper bytes
;ArgG   equ 32H  ;used in multiply
;ArgT   equ 31H  ;
;ArgH   equ 30H  ;same space as Tmp upper bytes
;GapH   equ 3BH  ;8 digit
;GapA   equ 34H  ;result
;TmpD   equ 2FH  ;4 byte
;TmpA   equ 2CH  ;math low byte
;HexD   equ 2BH  ;double word
;HexA   equ 28H  ;low byte of variable
;
;  R6  =    size, mul
;  R5  =    carry, mul
;  R4  =    mul
;  R3  =    loop control iteration
;  R2  =    loop control & size number of bytes
;  R1  =    @ source
;  R0  =    @ destination
;
;  Move the location of the source and destination
;  to the ArgS variables.
;  Move the size of the Arguments to SizeS.
;  The Arg and Size will be incremented as the
;  process goes through each byte.
;
;  Xcand= multiplicand, Xer= multiplier

          mov   R5, #0        ;clear carry byte

                              ;LOOP
JMUX1:    mov   R0, ArgH      ;Xcand addr
          mov   A, @R0        ;Xcand data
```

```
            mov   R2, A          ;Xcand data
            inc   ArgH           ;R2= *ArgX++

            mov   R0, ArgG       ;Res addr
            inc   ArgG           ;R0= ArgZ++

            mov   R1, ArgT       ;Xer addr
            mov   R3, SizeT      ;Xer loop
            mov   R4, #0
            jz    JMUX9          ;result=0, exit

                                 ;NIBBLE MULTIPLY
JMUX2:      mov   A, R2          ;Xcand data
            mov   B, @R1         ;Xer data
            inc   R1             ;next byte
            mul   AB             ;B:A= *R1++ * R2

            add   A, R4          ;add carry byte
            xch   A, B
            addc  A, #0          ;add carry bit
            xch   A, B           ;B:A += R4;
            add   A, @R0         ;add Xcand data

            mov   @R0, A         ;save Xer
            inc   R0             ;next Xer
            mov   A, B
            addc  A, #0          ;add carry bit
            mov   R4, A          ;R4:*R0++ =B:A+ *R0
            djnz  R3, JMUX2

            mov   R3, Size       ;bytes in Xcand
            mov   A, R4          ;carry byte

                                 ;CARRY BY 1
JMUX3:      add   A, @R0         ;Xcand data
            mov   @R0, A         ;save new Xcand
            inc   R0             ;C:*R0++ += *R0 + A
            jnc   JMUX9          ;C<>0, next loop

            mov   A, #1
            djnz  R3, JMUX3      ;next Xcand bit

                                 ;
            inc   R5             ;inc carry byte
```

```
JMUX9:    djnz  Size,JMUX1    ;next Xcand

                              ;TERMINATE
          ret

;-------------------------------------------------
JSQROOT:
;-------------------------------------------------
;  QuoS = square root of HexS
;  Enter with a value.
;  Rotate the 2 MSB into a new variable.
;  Compare comparison with new.
;  If comparison < new, then C=1.
;  Then new is greater than nearest multiple of 2.
;  So shift another bit into answer.
;  Calculate new = new-compare-C = new-(compare+1)
;  This removes last compare from re consideratio.
;  Multiply comparison by 2.
;  If Comparison < new then orl #4
;  This sets bit equiv to nest multiple.
;  Continue.
;
;  SHL 2 bits into D  If X<D, C=1, C->Q, D=D-X-C
;                     If X>D, C=0. C->Q
;  SHL 1 bits in X    If X>D, X=X orl #4
;
;  HexA = Input 4 bytes
;  TmpA = comparison 4 bytes
;  RemA = remaining 4 bytes
;  GapA = result 2 bytes
;
;  R0, R1, R2, R3
                              ;INITIALIZE
          mov   Size,#4       ;#bytes of source

                              ;CLEAR VARIABLES
          mov   R0,#GapA      ;result
          mov   A,Size        ;# bytes
          rr    A             ;only need 1/2
          mov   R2,A          ;2 bytes
          lcall JCLEAR        ;clear

          mov   R0,#RemA      ;remaining
          mov   R2, Size      ;4 bytes
```

```
            lcall JCLEAR        ;clear

            mov   R0,#TmpA      ;comparison
            mov   R2, Size      ;4 bytes
            lcall JCLEAR        ;clear

            mov   A,Size        ;# bytes
            rl    A             ;mul by 2
            rl    A             ;mul by 2
            mov   R3,A          ;bits shift to answ

                                ;SHIFT 2BITS=mul BY4
JSQR1:      mov   R0,#HexA      ;input
            mov   R2, Size      ;4 bytes
            lcall JSHIFTL       ;msb to C

            mov   R0,#RemA      ;remaining
            mov   R2, Size      ;4 bytes
            lcall JROTATEL      ;c to lsb

            mov   R0,#HexA      ;input
            mov   R2, Size      ;4 bytes
            lcall JSHIFTL       ;msb to C

            mov   R0,#RemA      ;remaining
            mov   R2, Size      ;4 bytes
            lcall JROTATEL      ;c to lsb

                                ;COMP REMAIN& INTERM
            mov   R0,#TmpA      ;interim comparison
            mov   R1,#RemA      ;remaining
            mov   R2, Size      ;4 bytes
            lcall JIFLESS       ;TmpA<TmpE, C=1

            mov   FgC,C         ;c=resul,abit>2*bbit
            mov   R0,#GapA      ;result
            mov   A,Size        ;# bytes
            rr    A             ;divide by 2
            mov   R2,A          ;2 bytes
            lcall JSHIFTL       ;C to lsb
            jnb   FgC,JSQR2     ;c=0,no change reman

                                ;UPD REMAIN& INTERIM
            mov   R0,#RemA      ;remaining
```

```
          mov   R1,#TmpA      ;interim
          mov   R2, Size      ;4 bytes
          lcall JSUBSC        ;sign,rem=rem-intr-1

JSQR2:    mov   R0,#TmpA      ;interim
          mov   R2, Size      ;4 bytes
          lcall JSHIFTL       ;MSB to C
          jnb   FgC,JSQR3     ;<>sq,no change intm

                              ;IF 2**2 PUT 4 INTRM
          mov   A, TmpA       ;first interim
          orl   A, Size       ;sq,put 2**2 in intm
JSQR3:    djnz  R3,JSQR1      ;loop in 16 bit answ

          mov   HexA,GapA     ;answer
          mov   HexB,GapB     ;answer

          ret
```



SECTION IV – HARDWARE

⇐ ⇑ ⇒

44

PARTS AND PIN-OUTS

Thought
The universal engineering unit = $
Dr. D

Watch your money _____

Parts are separated into categories. The *Proto then uC* group is the basic components to make a working computer. All these will be later used on a development board. This also includes cable for the SPI download of programs

The *uC only* list is the remaining components for the development board. These will provide interface to most projects. This also includes cables and connectors for RS 232 communications with a PC.

The *optional* list includes liquid crystal display, analog to digital converter, static ram, and expansion headers. Not all these will always be needed for every project.

The *project* list is components that will be used with a proto-board to design various projects. This list will change depending on the particular application.

Proto then uC board _____

Part	Device	Package	Where	Vendor	Number	Qty	Cost
89S8252	uP	Dip40	U1			1	
11.059MHz	Xtal	2 pin	Y1			1	
10 K Ω	Res	Axial 0.4				1	
1.5 K Ω	Res	Axial 0.4	R3			1	
330 Ω	Res pak	Sip8	Rp8			1	
10 uF	Cap	RB.2/.4	2			2	
39 pF	Cap	Axial 0.3	C1,2			2	
7 Segment	Display CC		DS1			1	
	V Reg		U12			1	
	Quad latch	Dip14	U3			1	
5 V	Pwr Sup	Wall wart	JP16			1	
RJ45	Jack		ISP			2	
RJ45	Adapter to	DB25	ISP			1	
RJ45	Adapter to	DB9	RS232			1	
5’	Cat5E cable	RJ45	ISP			2	

uC board only_____

Part	Device	Package	Where	Vendor	Number	Qty	Cost
Develop	Board	2 sided	B1	TUEE		1	
Socket	uP	Dip40	U1			1	
74573	Octal lat	Dip20	U4, 5, 6			4	
233	Rs232	Dip16	U9			1	
P22v10	Peel	Dip24	U8			1	
Socket	Peel	Dip24	U8			1	
2.2 K Ω	Res	Axial 0.4				1	
2.2 K Ω	Res pak	Sip10	BR1			1	
R Y R	LED	Diode 0.2	DS2			1	
	Pushbutton		S1, 2			2	
	Diode	Diode 0.2	D1				
0.1 uF	Cap	Axial 0.3	C4				
4.7 uF	Cap polar	Axial 0.3	C6				
mm	Jack		Power in			1	

uC board optional

Part	Device	Package	Where	Vendor	Number	Qty	Cost
6.8V	Zener LCD		D2			1	n/o
2N3906	Xistor LCD	To-92A	Q4,5			2	n/o
50K	Thermistor	Axial 0.4	TH1			1	n/o
300K	Res LCD	Axial 0.4	R4			1	n/o
12K	Res LCD	Axial 0.4	R7			1	n/o
150K	Res LCD	Axial 0.4	R6			1	n/o
24K	Res LCD	Axial 0.4	R5			1	n/o
3.3K	Res LCD	Axial 0.4	R9			1	n/o
2x40	LCD	Sip14	JP2			1	
Socket	LCD	Sip14	JP2			1	
Cable	LCD	3” ribbon	JP2			1	
32Kx8	Sram	Dip28	U2			1	n/o
Socket	Sram	Dip28	U2			1	n/o
	ADC	Dip8	U10			1	
	FET	Sip3	Q1,2,3			3	
1K	Resistor	Axial 0.4	R4				
485	Rs485	Dip8	U10				
InfraRed	Detector	Sip3	U11				

Projects

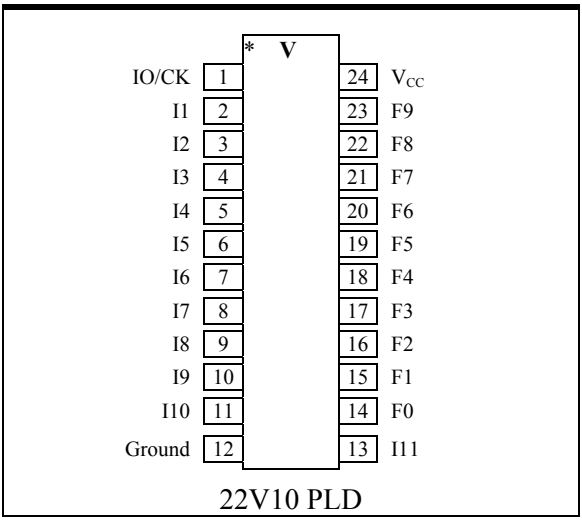
Part	Device	Package	Where	Vendor	Number	Qty	Cost
7 Segment	Display CC					2	
3x4	keypad					1	
330 Ohm	Res	Axial 0.4				6	n/o
R Y G	LED	Diode 0.2				6	n/o
2	Switches	Dip 4				1	
7406	Hex Inverter					1	
Breadboard		3 Column				1	

uC board headers & jumpers

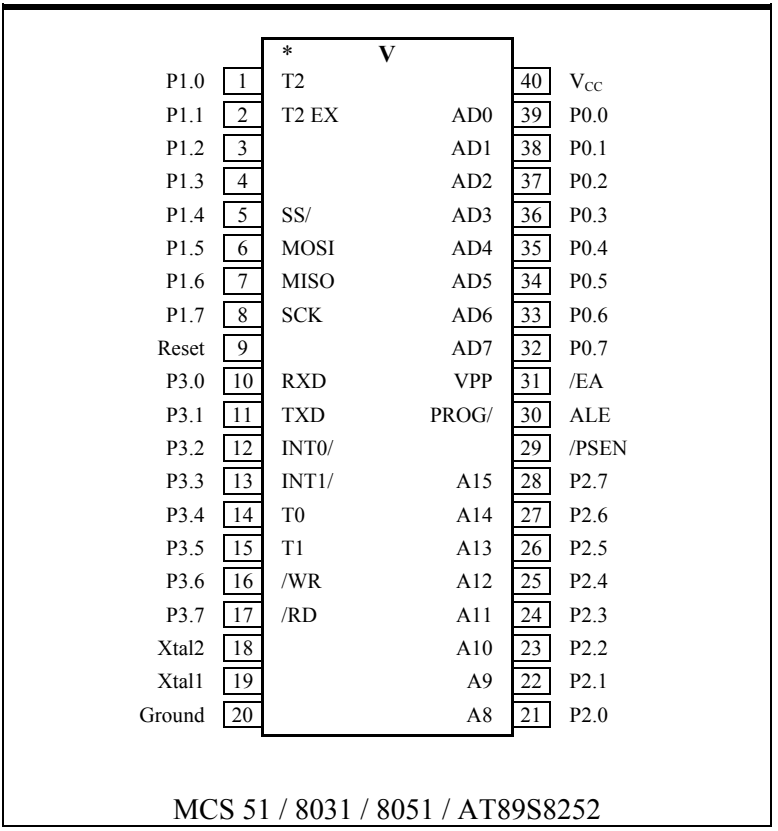
Part	Device	Package	Where	Vendor	Number	Qty	Cost
8 pin	Expansion	Sip 8				4	
16 pin	LCD	Sip16	Jp2			1	
2 pin	Analog in	Sip2	Jp3			1	
11 pin	7-Segment	Sip11	Jp4			1	
4 pin	mmio	Sip4	Jp6			1	
8 pin	Key in	Sip8	Jp12			1	
2 pin	Power in	Sip2	Jp13			1	
8 pin	Address	Sip8	Jp14			1	
2 pin	Power out	Sip2	Jp17			1	
2	EA’	Jumper	J1			1	
2	PLD bypass	Jumper	J15			1	
2	Serial hand	Jumper	Jp5			1	
2	7seg select	Jumper	Jp9			1	

⇐ ⇑ ⇒

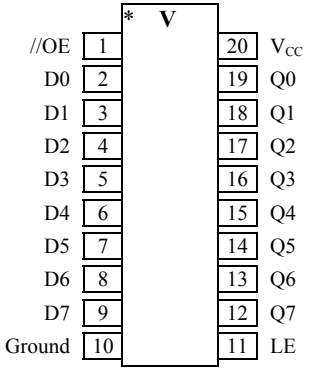
PLD / PEEL pin-out



Microprocessor pin-out



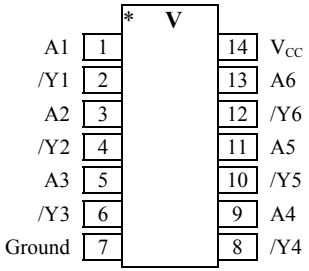
Buffer pin-out

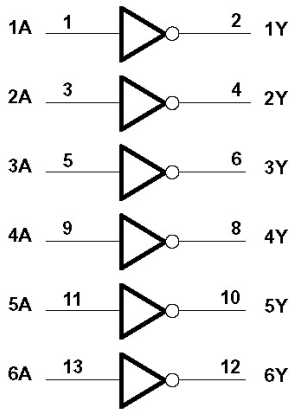


Operating Modes	INP			OUT
	OE	LE	Dn	Qn
enable & read	L	H	L	L
register transp	L	H	H	H
latch & read	L	L	L	L
register	L	L	H	H
latch register&	H	L	L	Z
disable outputs	H	L	H	Z

The data for the latch is setup when the latch enable (LE) pin is driven high. When the pin is pulled low, the latch traps the data.

74573 Octal D latch

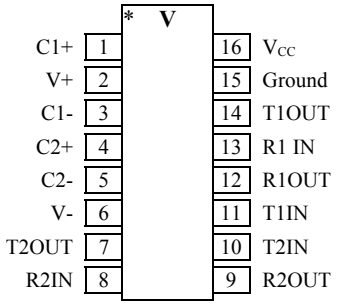




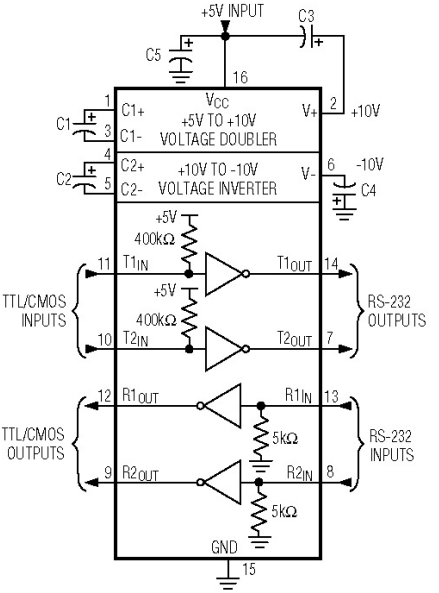
Open-collector requires connection of 2.2K pull-up resistor between each output and V_{CC}.

7406 Hex inverter,open-collector buffer driver

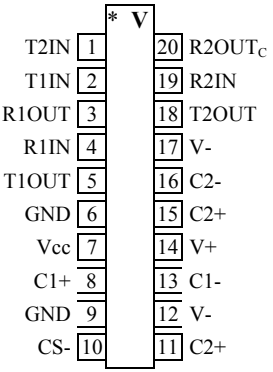
RS232 & RS233 pin-out

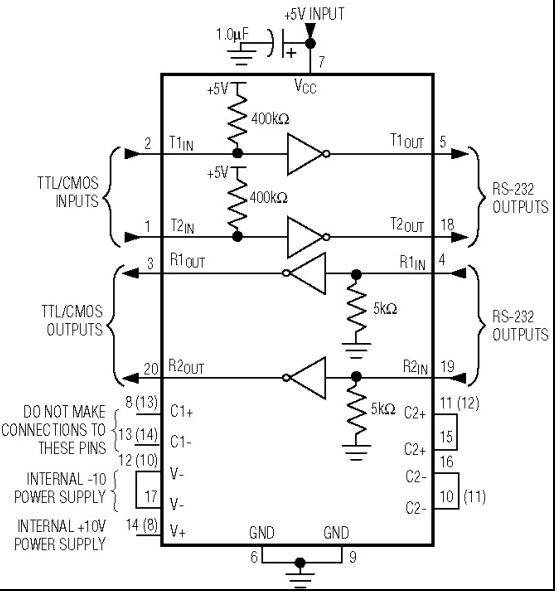


Max	C1 – C5
232	1 uF
232A	0.1 uF



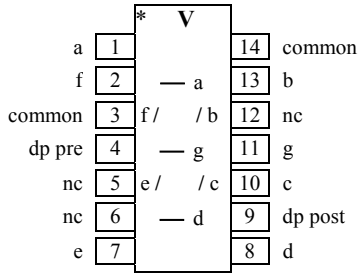
RS 232 Interface





RS 233 Interface

7-Segment & LCD pin-out



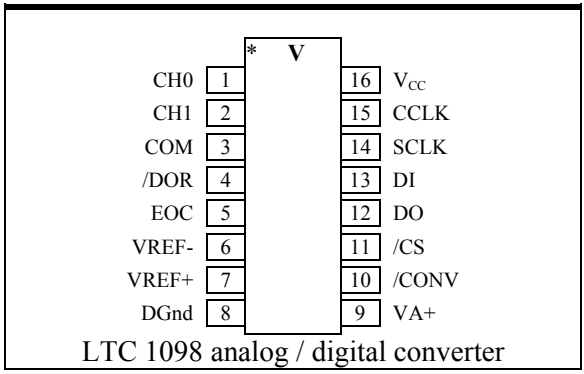
Pin	Function
1	a
13	b
10	c
8	d
7	e
2	f
11	g
4	dp pre
9	dp post
3	common
14	common

7 Segment display

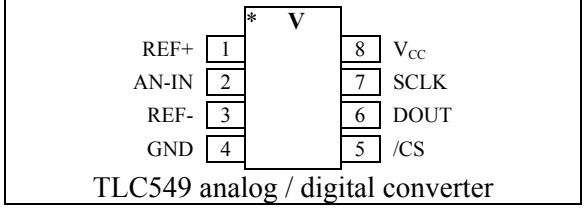
Pin	Symbol	Level	Function
1	DB7	H / L	data bus line
2	DB6	H / L	data bus line
3	DB5	H / L	data bus line
4	DB4	H / L	data bus line
5	DB3	H / L	data bus line / no-connection for 4-bit operation
6	DB2	H / L	data bus line / no-connection for 4-bit operation
7	DB1	H / L	data bus line / no-connection for 4-bit operation
8	DB0	H / L	data bus line / no-connection for 4-bit operation
9	E1	H, H->L	enable signal (no pull-up resistor)
10	R-/W	H / L	read/write select signal, h : read l : write
11	RS	H / L	register select signal
12	VEE	–	power supply for LCD drive
13	VSS	–	power supply (0v, ground)
14	VCC	–	power supply for logic
15	E2	H, H->L	enable signal (no pull-up resistor)
16	NC	–	no-connection
17	LED K	–	led cathode terminal
18	LED A	–	led anode terminal

Liquid crystal display

A/D Converter pin-out

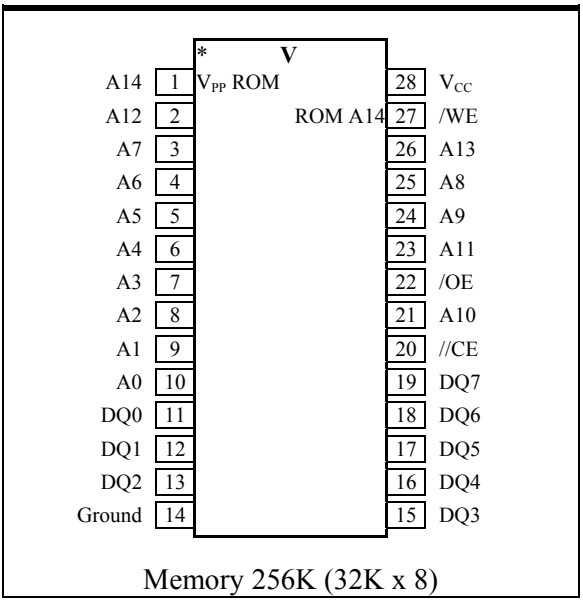


LTC 1098 analog / digital converter

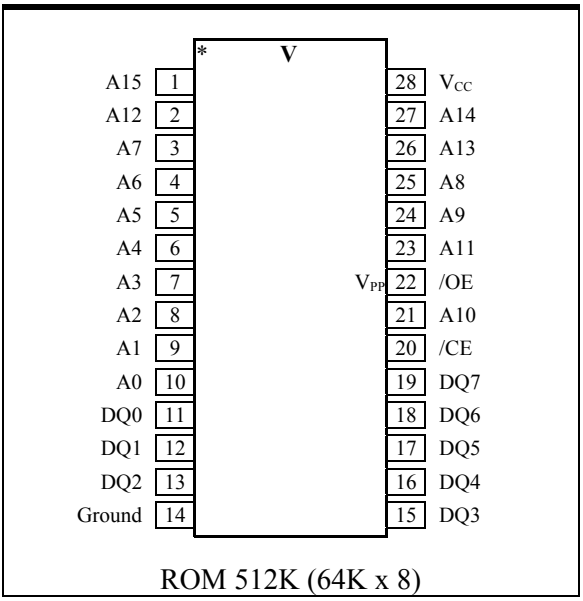


TLC549 analog / digital converter

Memory pin-out



Memory 256K (32K x 8)



ROM 512K (64K x 8)

⇐ ⇑ ⇒

Cable pin-out, SPI & serial

Two connectors are used on the board. One is required for the RS232 communications which will connect by cable to the PC serial port. The other is required for in-system-programming (ISP) which will connect by cable to the PC parallel port.

Note, connect a 1.5K resistor between the cable and the reset.

Funct	**Printer**	**To**	**Cable**	**Cable**
	db 25	*RJ45*	*TIA568B*	*Telco*
	Pin	*Pin*	*Color*	*Alt*
MOSI	7	1	W/Orange	Black
MISO	10	2	Orange	Yellow
		3	W/Green	White
		4	Blue	Red
Gnd	18	5	W/Blue	Green
		6	Green	Blue
Sck	8	7	W/Brown	Brown
Reset	6	8	Brown	Orange

SPI - Parallel RJ45 Cable

Funct	**Serial**	**To**	**Board**	**Cable**	**Cable**
	db 9	*RJ11*	*RJ45*	*TIA568B*	*Telco*
	Pin	*Pin*	*Pin*	*Color*	*Alt*
Jump	1		1	W/Orange	Black
Jump	6		2	Orange	Yellow
TXD	3	1	3	W/Green	White
RXD	2	2	4	Blue	Red
Gnd	5	3	5	W/Blue	Green
nc	4	4	6	Green	Blue
Jump	7		7	W/Brown	Brown
Jump	8		8	Brown	Orange

RS232 – Serial RJ45 Cable

The RJ45 connectors have multiple functions. The following table illustrates the relationship between Ethernet, telephone, and the connections used with the processor schematic.o

Funct	**Printer**	**To**	**Cable**	**Cable**	**Board**	**Funct**	**To**	**Serial**	**Funct**
	db 25	*RJ45*	*TIA568B*	*Telco*	*RJ45*		*RJ11*	*db 9*	
	Pin	*Pin*	*Color*	*Alt*	*Pin*		*Pin*	*Pin*	
MOSI	7	1	W/Orange	Black	1	P1.5		1	Jump
MISO	10	2	Orange	Yellow	2	P1.6		6	Jump
		3	W/Green	White	3	T1X	1	3	TXD
		4	Blue	Red	4	R1X	2	2	RXD
Gnd	18	5	W/Blue	Green	5	Gnd	3	5	Gnd
		6	Green	Blue	6	R2x	4	4	nc
Sck	8	7	W/Brown	Brown	7	P1.7		7	Jump
Reset	6	8	Brown	Orange	8	Reset		8	Jump

RJ45 Cables

⇐⇑⇒

45

DEVELOPMENT BOARD

Thought
A design – build contract
means you do it all.

Design

The development board is a general-purpose microcontroller system. It has a multipurpose design. First, it allows addition of substantial hardware without the complexity of wires on a proto-board. It can be used as a prototype for various devices. Finally, the configuration may be installed in standard electrical enclosures, so it can be used as a production controller. The schematic follows.

Options

Obviously, numerous options can be implemented by simply not installing various devices. However, this does give substantial flexibility in uses and applications.

Perhaps the most significant option is the memory. Either EPROM or SRAM can be installed in the socket. The pin functions are different on these two devices. There are three pins that control the variations. Chip enable is listed simply for reference.

Memory	EPROM	SRAM
/CE	Ground	Ground
/OE	/PSEN jumper	/RD or Gnd jumper
Pin 1	V_{PP}	A14
Pin 27	A14	/WE

To change between memory chips only two things are required. First place the correct chip in the socket. Next, change the jumper for output enable, /OE. The pin conversions are made inside the PEEL, and the chip is permanently enabled.

HyperTerminal

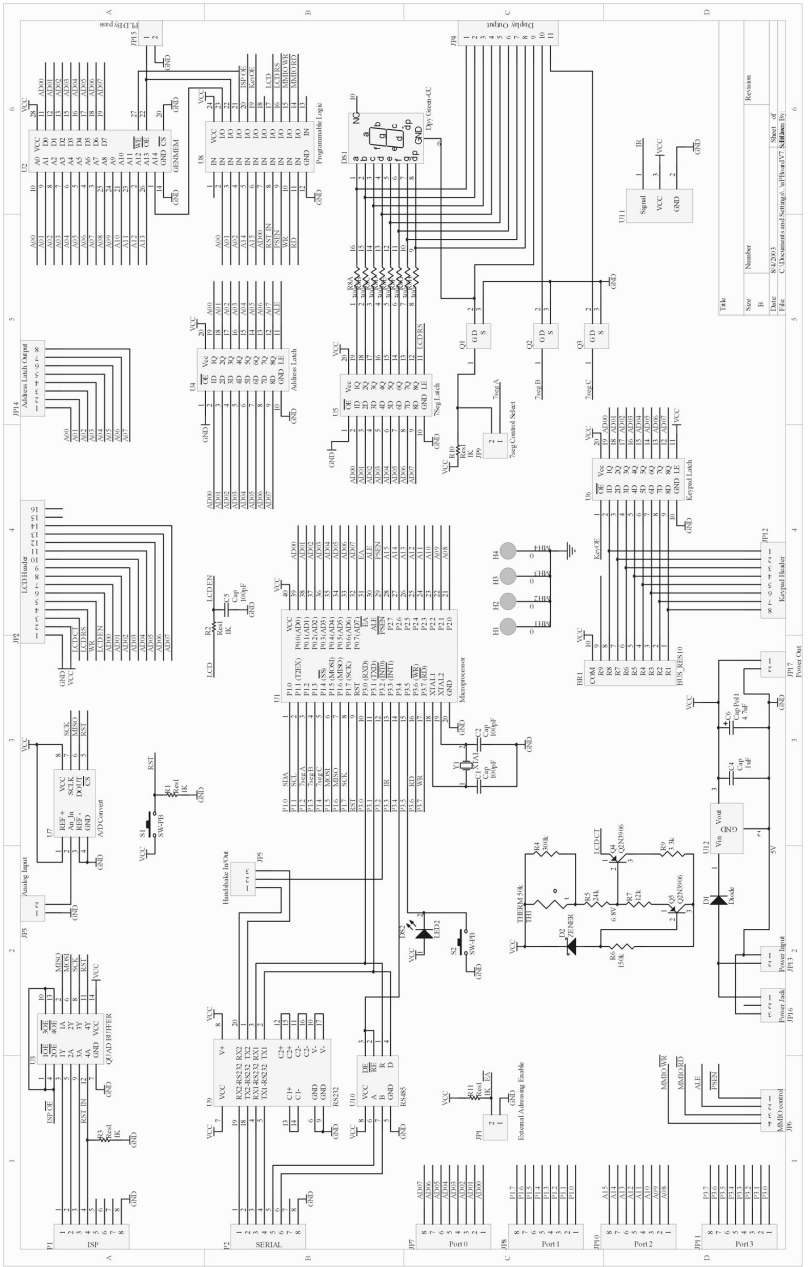
Several programs can be loaded to assist in using the microprocessor development system. HyperTerminal is a Windows accessory communications program. It is setup to use the serial port for communications with the microcontroller. Baud rate is 9600. Programs can be dropped onto the serial port. The program will also display any information arriving on the serial port.

Test Program

The program is run from the development board. Use the sample test program MODTest shown as a simple software illustration. Run the program through an assembler. Take the *.hex file results and download to the board memory. This program is very simple, but illustrative.

Schematic

This board is custom designed for system projects. It incorporates numerous features and characteristics, including selecting various processors and devices. The design is protected by international copyright.



Board specifications

The board design is based on numerous performance specifications. These specs are identified by the device. Therefore, most criteria will be listed twice, once as an output from one device, then as a input to another.

PROCESSOR
1. 89S8252 (8051 core) w/ in-system programming, internal program & data memory
2. Crystal 11.059 MHz
3. Reset can be from auto circuit, pushbutton, and ISP.
4. Package is 40 pin, board to fit electrical gang-box.

EXTERNAL MEMORY
5. Expansion memory socket in a common socket
   a. 32K eeprom
   b. 32K sram w/ socket for DIP or wide
   c. Memory pin 1, 27, and OE from PLD

POWER
6. Wall-wart mini-jack power connector pin
7. LP2954 voltage regulator enlarge hole size
8. Add pins for 5V & Ground
9. Place filter capacitor on line.

ISP
10. Quad buffer on in-system-programming lines for port 1.
11. Enable buffer with uc reset complemented from PLD.
12. Connect in and out lines as required.

RESET
13. Place capacitor and resistor to auto reset on pin 9.
14. Place pushbutton for manual reset.

PUSHBUTTON / LED ON P35
15. Pushbutton as input on P35 (T1).
16. Add LED to P35 (T1) as output.
17. Connect pull-up to led so can work together with switch.

PORT 1

18. Connect IIC to pins 6 & 7, share with SPI
19. Connect 7-segment select lines to pins 2-4.
20. Connect SPI on pins 5-7.

PORT 3

21. Connect RS232 on pins 0&1
22. Connect handshake on pin 2 (INT0).
23. Connect infrared receive/ transmit module to P33 (Int1).
24. If Int1 is used separately, the infrared must be covered with black tape.
25. Connect RS485 select to P34 (T0).
26. Connect LED & pushbutton to P35 (T1).

I/O LATCHES

27. All external I/O uses memory-mapped connections.
28. Display-Out Latch for 7-segment data.
    a. Latch Enable from PLD.
    b. Output Enable is connected to ground.
    c. Take data to on-board 7-segment display through current limiting resistors.
    d. Take data out after resistor to expansion header.
29. Key-Out Latch for keypad column output.
    a. Latch Enable from PLD. Enable when write to column.
    b. Output Enable is connected to ground.
    c. Use upper nibble for columns out to keypad.
    d. Use other bits for digital out.
30. Key-In Latch for keypad row input.
    a. Output Enable from PLD. Enable when read from row.
    b. Latch Enable connected to 5 volts.
    c. Use lower nibble for rows in from keypad..\
    d. Connect pull-up resistors on keypad inputs.
    e. Other bits are for digital input.
31. Address Latch to separate address from data on Port 0.
    a. LE is connected to ALE.
    b. OE is connected to ground.

DISPLAY CONTROL

32. Assign three bits to select three different 7-segment displays.
    a. Add drivers for select lines, 7406 open-collector with pull-ups or 30 A MosFets.
    b. On-board 7-segment has header jumper to select ground or a bit to control.
    c. Connect control lines to P12, P13, P14.

LCD

33. Bypass auto-contrast circuit with a jumper to ground.
34. Add backlight connector adjacent to LCD header
35. LCD has 3 control lines, RW', RS, En. Shared with 7-segment.
    a. Take controls to LCD header.
    b. Connect Enable to P1.2.
    c. Connect Write enable to P1.1.
    d. Connect data/instruction register select to P1.0.

A/D CONVERT

36. Use with serial peripheral interface.
    a. Connect serial I/O to port 1 SPI bus using MISO, SCK
    b. Connect /CS to P1.4, /SS.
37. Make reference voltage 0-5 volts for rail to rail operation.
38. Make header for A/D input with analog ground.
39. Connect analog ground to digital ground at 1 point via a jumper that can be isolated.

INFRARED

40. Add infrared receiver / transmitter
41. Connect to P33 (Int1) pin

SERIAL

42. Use Max 233 since it does not require external caps.
    a. Because of limited port space, only 1 handshake is used..
    b. Connect Int0 as handshake control.
    c. Header selects in or out line for connection to RJ45
43. Add Max 485 chip as option.
    a. Connect control lines RE and DE together to P34 (T0).
    b. Connect serial differential lines to header.
    c. These will parallel the serial lines for the Max233.

IIC

44. Connect IIC to Port 1.
    a. SDA connect to P17.
    b. SCL connect to P16

RJ45 CONNECTORS

45. Connect 3 in-system-programming lines plus common to ISP connector.
46. Connect serial TX, RX, and handshake to serial connector.
    a. Connect handshake to a jumper between out and in handshake
    b. Configurable for computer or modem

PLD (22V10 PEEL)

47. Connect input lines
    a. PSEN
    b. WR'
    c. RD'
    d. Reset
48. Connect input address lines (08001h)
    a. A15, A14, A2, A1, A0.
49. Connect input data line, D0
50. Leave pin 1 open for clock
51. One spare input & output line.
52. Connect output lines
    a. Key OE
    b. Key LE
    c. Display LE
    d. ISP chip select
53. Connect memory control output
    a. Memory OEn
    b. Memory pin MP1
    c. Memory pin MP27
    d. MmWr
    e. MmRd

OFF-BOARD HEADERS

54. In-system-program port through 4-bit isolation latch
    a. Connect in and out lines as required.

55. Seven-segment
    a. Data from Display-Out latch
    b. Seven-segment control lines from Port 1 through 7406.
56. LCD
    a. Data from P0
    b. Control from microprocessor.
    c. Add backlight connector. It can have separate power source.
57. Address latch output connect for expansion MMIO
58. Keypad
    a. Column from latch upper nibble.
    b. Row to latch lower nibble with pull-up resistors.
59. Address latch
    a. Output address for MMIO or expansion
    b. Have pins including ALE, PSEN, MMIO RD, MMIO WR.
60. Analog input
    a. Connect to ADC
    b. Analog ground
61. RS485 needs only 2 differential data lines
62. IIC needs 2 lines.
63. Power pins for 5V & Ground

SELECT JUMPERS

64. External memory EA' select 5V or Ground
65. Seven-segment onboard select ground or 7406 / MosFet
66. Memory Output Enable select ground or PLD
67. LCD contrast jumper to Ground
68. Handshake select for RS232

GENERAL

69. Be cautious that fan out is not a problem on port 0.

PORT CONNECTIONS

70. Port 1
    a. 10 – SegA / RS
    b. 11 – SegB / RW'
    c. 12 – SegC / En
    d. 13 –
    e. 14 – /SS
    f. 15 – SPI MOSI

g. 16 – SPI MISO
h. 17 – SPI SCK

71. Port 3
    a. 30 – serial RXD
    b. 31 – serial TXD
    c. 32 – Int1 serial handshake
    d. 33 – Int0 infrared input
    e. 34 – T0 RS485 select
    f. 35 – pushbutton and LED
    g. 36 – write'
    h. 37 – read'

⇐⇑⇒

46

IN SYSTEM PROGRAMMING

Thought
What is the tradeoff between hardware and software?

Serial downloading

Atmel's AT89S8252 flash microcontroller offers 8K bytes of in-system re-programmable flash code memory and 2K bytes of EEPROM data memory. This information and much more is available from the Atmel data sheet.

Both the Code and Data memory arrays can be programmed using the serial SPI bus while RST is pulled to V_{CC}. The serial interface consists of pins SCK, MOSI (input) and MISO (output).

P1.7	P1.6	P1.5	P1.4				
SCK	MISO	MOSI	/SS				

After RST is set high, the Programming Enable instruction must be executed first before program/erase operations can be executed.

An auto-erase cycle is built into the self-timed programming operation (in the serial mode ONLY). It is unnecessary to first execute the Chip Erase instruction, unless the lock bits have been programmed. The Chip Erase operation turns the contents of every memory location in both the Code and Data arrays into 0FFH.

The Code and Data memory arrays have separate address spaces: 0000H to 1FFFH for Code memory and 000H to 7FFH for Data memory.

Either an external system clock is supplied at pin XTAL1 or a crystal needs to be connected across pins XTAL1 and XTAL2. The maximum serial clock (SCK) frequency should be less than 1/40 of the crystal frequency. With a 24 MHz oscillator clock, the maximum SCK frequency is 600 kHz.

Programming algorithm

To program and verify the flash controller is in the serial programming mode, the following sequence is recommended:

1. Power-up sequence: Apply power between V_{CC} and GROUND pins. Set RST pin to Hi. If a crystal is not connected across pins XTAL1 and XTAL2, apply a 3 MHz to 24 MHz clock to XTAL1 pin and wait for at least 10 milliseconds.

2. Enable serial programming by sending the Programming Enable serial instruction to pin MOSI/P1.5. The frequency of the shift clock supplied at pin SCK/P1.7 needs to be less than the CPU clock at XTAL1 divided by 40.

3. The Code or Data array is programmed one byte at a time by supplying the address and data together with the appropriate Write instruction. The selected memory location is first automatically erased before new data is written. The write cycle is self-timed and typically takes less than 2.5 ms at 5V.

4. Any memory location can be verified by using the Read instruction, which returns the content at the selected address at serial output MISO/P1.6.

5. At the end of a programming session, RST can be set low to commence normal operation.

If a power-off sequence is required, use only three tasks. Set XTAL1 to Low, if a crystal is not used. Set RST to Low. Turn off V_{CC} power.

Data polling is used to indicate the end of a write cycle, which typically takes less than 2.5 ms at 5V.

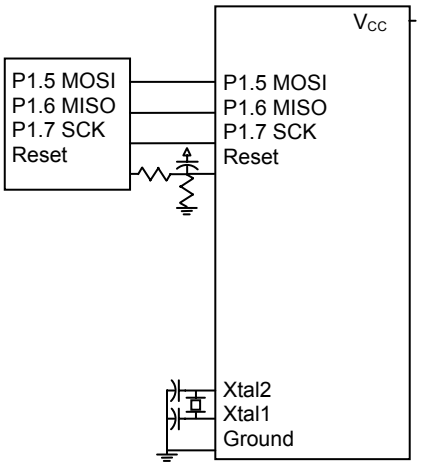
Programming instruction

The Instruction Set for serial programming follows a 3-byte protocol and is shown in the following table:

Operation	Byte 1	Byte 2	Byte 3	
Programming Enable	1010 1100	0101 0011	xxxx xxxx	Enable serial programming interface after RST goes high.
Chip Erase	1010 1100	xxxx x100	xxxx xxxx	Chip erase both 8K & 2K memory arrays.
Read Code Memory	aaaa a001	low addr	xxxx xxxx	Read data from Code memory array at the selected address. The 5 MSBs of the first byte are the high order address bits. The low order address bits are in the second byte. Data are available at pin MISO during the third byte.
Write Code Memory	aaaa a010	low addr	data in	Write data to Code memory location at selected address. The address bits are the 5 MSBs of the first byte together with the second byte.
Read Data Memory	00aa a101	low addr	xxxx xxxx	Read data from Data memory array at selected address. Data are available at pin MISO during the third byte.
Write Data Memory	00aa a110	low addr	data in	Write data to Data memory location at selected address.
Write Lock Bits	1010 1100	x x111	xxxx xxxx	Write lock bits. Set LB1, LB2 or LB3 = “0” to program lock bits.

Programming schematic

The connections to the microcontroller are very simple as shown in the schematic. The standard connections are power, ground, crystal, and reset. Only three additional lines are required for loading a program into the memory.

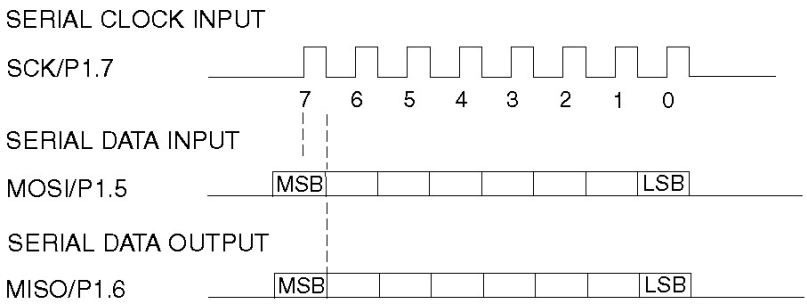


Peripheral timing

Serial peripheral interface (SPI) is a protocol to connect several devices on a common data input and a common data output lines. Both input and output happen simultaneously, as illustrated in the diagram.

The SCLK line shifts data into the device (MOSI) on a rising edge. So SCLK must be made LO then HI to input to the device from the controller.

The SCLK line shifts data out from the device (MISO) on a falling edge. So SCLK must be made HI then LO to output from the device to the controller.



Programming and printer

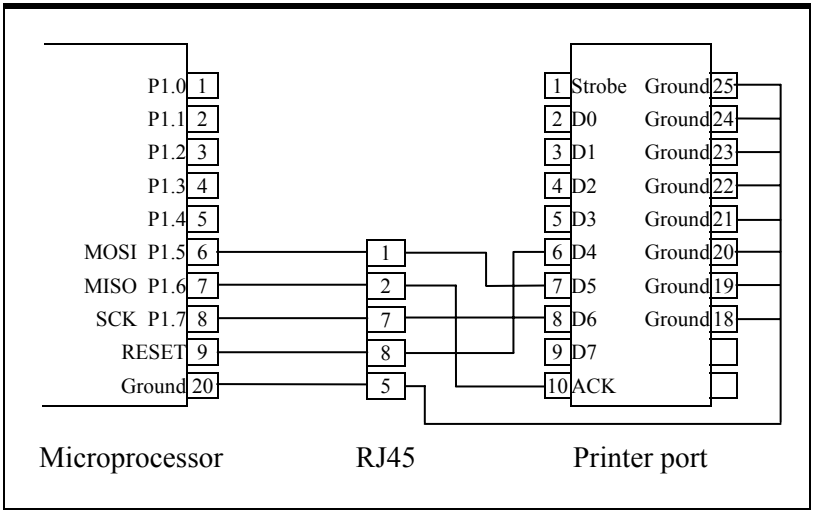
There are several software packages for performing the program code download from the personal computer to the internal PROM on the microcontroller. One of the easier to use programs is a small executable file called AEC_ISP.EXE. This file can be downloaded.

Connections are needed on the personal computer for the three data lines and a reset line. The only access with that many control wires is the parallel port for the printer.

Note place a 1.5K resistor between the connector and the reset line.

The AEC_ISP program uses the pattern of connection shown in the schematic below to make the tie between the printer port and the corresponding SPI lines on port 1 of the microcontroller.

Once the program code has been downloaded, the microcontroller must be reset. The SPI connection to the computer printer port can be removed. It is important that the first few lines of executable code in the program must not change port 1. This can cause a conflict and lock-up. If that happens, the buffer must be cleaned and the program downloaded again.



Connectors

Making a viable, custom termination between a printer connector and a circuit board or proto board can be a tedious process. The best connection technique uses available technology.

One very dependable technology is RJ45 Ethernet connectors. First, obtain a printer port (DB25P) to Ethernet (RJ45S) adapter. Wire the adapter to the appropriate pins according to the following table.

Use the required length of Ethernet cable with RJ45P connectors on both ends. Typically, 5 feet is adequate. The standard pin arrangement and color coding is shown in the chapter on Networking.

Obtain a RJ45S connector that can be configured with pigtails to mount to the proto board.

These three components provide a dependable, flexible, and relatively inexpensive interconnection between the PC printer and the microcontroller.

Funct	**Printer**	**To**	**Cable**	**Cable**	**Board**	**Funct**
	db 25	*RJ45*	*TIA568B*	*Telco*	*RJ45*	
	Pin	*Pin*	*Color*	*Alt*	*Pin*	
MOSI	7	1	W/Orange	Black	1	P1.5
MISO	10	2	Orange	Yellow	2	P1.6
		3	W/Green	White	3	
		4	Blue	Red	4	
Ground	18	5	W/Blue	Green	5	Ground
		6	Green	Blue	6	
Sck	8	7	W/Brown	Brown	7	P1.7
Reset	6	8	Brown	Orange	8	Reset

⇐⇑⇒

SECTION V – ARCHITECTURE

⇐ ⇑ ⇒

47

INSTRUCTION SET

Thought
Triad Principle:
Any item that can be uniquely identified,
can be further explained by three components
MOD

Microcontroller instruction set_____

The instruction set is very powerful for a microprocessor. As a result, this core is one of the most common used in microcontroller applications.

Instructions can be divided into categories consisting of data transfer, arithmetic, logic, program branch, and bit manipulation.

Instructions can be demonstrated many ways. The first group of tables represents the categories of instructions. Then, two tables are provided that are a matrix of the numeric instruction codes.

There are many ways data can be obtained. These are called addressing modes. The machine has seven unique modes - immediate, direct, register, indirect, relative, absolute, and bit.

Addressing modes

R	Register R7-R0 of the currently selected Register Bank.
Direct	eight bit internal data location's address. This could be an Internal Data RAM location (0-127) or a SFR [i.e., I/O port, control register, status register, etc. (128-255)].
@R	eight bit internal data RAM location (0-255) addressed indirectly through register R1or R0.
#data	eight bit constant included in instruction.
#data 16	sixteen bit constant included in instruction.
addr 16	sixteen bit destination address. Used by lcall and ljmp. A branch can be anywhere within the 64K byte Program Memory address space.
addr 11	11-bit destination address. Used by acall and ajmp. The branch will be within the same 2K byte page of program memory as the first byte of the following instruction.
Rel	Signed (two's complement) eight bit offset byte. Used by sjmp and all conditional jumps. Range is -128 to +127 bytes relative to first byte of the following instruction.
Bit	Direct Addressed bit in Internal Data RAM or Special Function Register

Data transfer

Op	Mnem	Register	Description	Byt	Cyc
74	mov	A,#data	Move immediate data to Accumulator	2	1
75	mov	direct,#data	Move immediate data to direct byte	3	2
76	mov	@R0,#data	Move immediate data to indirect RAM	2	1
77	mov	@R1,#data	Move immediate data to indirect RAM	2	1
7*	mov	Rn,#data	Move immediate data to register	2	1
85	mov	direct,direct	Move direct byte to direct	3	2
86	mov	direct,@R0	Move indirect RAM to direct byte	2	2
87	mov	direct,@R1	Move indirect RAM to direct byte	2	2
8*	mov	direct,Rn	Move register to direct byte	2	2
A6	mov	@R0,direct	Move direct byte to indirect RAM	2	2
A7	mov	@R1,direct	Move direct byte to indirect RAM	2	2
A*	mov	Rn,direct	Move direct byte to register	2	2
E5	mov	A,direct	Move direct byte to Accumulator	2	1
E6	mov	A,@R0	Move indirect RAM to Accumulator	1	1
E7	mov	A,@R1	Move indirect RAM to Accumulator	1	1
E*	mov	A,Rn	Move register to Accumulator	1	1
F2	mov	@R0,A	Move Accumulator to indirect RAM	1	1
F3	mov	@R1,A	Move Accumulator to indirect RAM	1	1
F5	mov	direct,A	Move Accumulator to direct byte	2	1
F*	mov	Rn,A	Move Accumulator to register	1	1
90	mov	DPTR,#datal6	Load Data Pointer w/ a 16-bit constant	3	2
83	movc	A,@A+PC	Move Code byte relative to PC to Acc	1	2
93	movc	A,@A+DPTR	Move Code byte relat to DPTR to Acc	1	2
E0	movx	A,@DPTR	Move Exter RAM (16-bit addr) to Acc	1	2
E2	movx	A,@R0	Move Exter RAM (8- bit addr) to Acc	1	2
E3	movx	A,@R1	Move Exter RAM (8- bit addr) to Acc	1	2
F0	movx	@DPTR,A	Move Acc to Exter RAM (1 6-bit addr)	1	2
F2	movx	@R0,A	Move Acc to Exter RAM (eight bit addr)	1	2
F3	movx	@R1,A	Move Acc to Exter RAM (eight bit addr)	1	2
C0	push	direct	Push direct byte onto stack	2	2
D0	pop	direct	Pop direct byte from stack	2	2
C5	xch	A,direct	Exchange direct byte w/ Accumulator	2	1
C6	xch	A,@R0	Exchange indirect RAM w/ Accum	1	1
C7	xch	A,@R1	Exchange indirect RAM w/ Accum	1	1
C*	xch	A,Rn	Exchange register with Accumulator	1	1
D6	xchd	A,@R0	Exc low-order digit indirect RAM w/ A	1	1
D7	xchd	A,@R1	Exc low-order digit indirect RAM w/ A	1	1

Arithmetic operations

Op	Mnem	Register	Description	Byt	Cyc
04	inc	A	Increment Accumulator	1	1
05	inc	Direct	Increment direct byte	2	1
06	inc	@R0	Increment direct RAM	1	1
07	inc	@R1	Increment direct RAM	1	1
0*	inc	Rn	Increment register	1	1
14	dec	A	Decrement Accumulator	1	1
15	dec	direct	Decrement direct byte	2	1
16	dec	@R0	Decrement indirect RAM	1	1
17	dec	@R1	Decrement indirect RAM	1	1
1*	dec	Rn	Decrement Register	1	1
24	add	A,#data	Add immediate data to Accumulator	2	1
25	add	A,direct	Add direct byte to Accumulator	2	1
26	add	A,@R0	Add indirect RAM to Accumulator	1	1
27	add	A,@R1	Add indirect RAM to Accumulator	1	1
2*	add	A,Rn	Add register to Accumulator	1	1
34	addc	A,#data	Add immediate data to Acc with Carry	2	1
35	addc	A,direct	Add direct byte to Acc with Carry	2	1
36	addc	A,@R0	Add indirect RAM to Acc with Carry	1	1
37	addc	A,@R1	Add indirect RAM to Acc with Carry	1	1
3*	addc	A,Rn	Add register to Accumulator w/ Carry	1	1
84	div	AB	Divide A by B	1	4
94	subb	A,#data	Subt immediate data from A w/borrow	2	1
95	subb	A,direct	Subt direct byte from Acc w/ borrow	2	1
96	subb	A,@R0	Subt indirect RAM from A w/ borrow	1	1
97	subb	A,@R1	Subt indirect RAM from A w/ borrow	1	1
9*	subb	A,Rn	Subtract Register from Acc w/ borrow	1	1
A3	inc	DPTR	Increment Data Pointer	1	2
A4	mul	AB	Multiply A & B	1	4
D4	da	A	Decimal Adjust Accumulator	1	1

Program branching

Op	Mnem	Register	Description	Byt	Cyc
00	nop		No Operation	1	1
02	ljmp	addr16	Long Jump	3	2
*1	ajmp	addr11	Absolute Jump	2	2
12	lcall	addr16	Long Subroutine Call	3	2
*1	acall	addrl 1	Absolute Subroutine Call	2	2
22	ret		Return from Subroutine	1	2
32	reti		Return from interrupt	1	2
60	jz	rel	Jump if Accumulator is Zero	2	2
70	jnz	rel	Jump if Accumulator is Not Zero	2	2
73	jmp	@A+DPTR	Jump indirect relative to the DPTR	1	2
80	sjmp	rel	Short Jump (relative addr)	2	2
B4	cjne	A,#data,rel	Compare immediate to Acc & Jump if Not Equal	3	2
B5	cjne	A,direct,rel	Compare direct byte to Acc & Jump if Not Equal	3	2
B6	cjne	@R0,#data,rel	Compare immediate to indirect & Jump if Not Equal	3	2
B7	cjne	@R1,#data,rel	Compare immediate to indirect & Jump if Not Equal	3	2
B*	cjne	Rn,#data,rel	Compare immediate to register & Jump if Not Equal	3	2
D5	djnz	direct,rel	Decrement direct byte & Jump if Not Zero	3	2
D*	djnz	Rn,rel	Decrement register & Jump if Not Zero	2	2

Logical operations

Op	Mnem	Register	Description	Byt	Cyc
03	rr	A	Rotate Accumulator Right	1	1
13	rrc	A	Rotate Accumulator Right through C	1	1
23	rl	A	Rotate Accumulator Left	1	1
33	rlc	A	Rotate Accumulator Left through C	1	1
42	orl	direct,A	OR Accumulator to direct byte	2	1
43	orl	direct,#data	OR immediate data to direct byte	3	2
44	orl	A,#data	OR immediate data to Accumulator	2	1
45	orl	A,direct	OR direct byte to Accumulator	2	1
46	orl	A,@R0	OR indirect RAM to Accumulator	1	1
47	orl	A,@R1	OR indirect RAM to Accumulator	1	1
4*	orl	A,Rn	OR register to Accumulator	1	1
52	anl	direct,A	AND Accumulator to direct byte	2	1
53	anl	direct,#data	AND immediate data to direct byte	3	2
54	anl	A,#data	AND immediate data to Accumulator	2	1
55	anl	A,direct	AND direct byte to Accumulator	2	1
56	anl	A,@R0	AND indirect RAM to Accumulator	1	1
57	anl	A,@Rl	AND indirect RAM to Accumulator	1	1
5*	anl	A,Rn	AND Register to Accumulator	1	1
62	xrl	direct,A	Exclusive-OR Accum to direct byte	2	1
63	xrl	direct,#data	Exclusive-OR immed data to direct by	3	2
64	xrl	A,#data	Exclusive-OR immed data to Accum	2	1
65	xrl	A,direct	Exclusive-OR direct byte to Accum	2	1
66	xrl	A,@R0	Exclusive-OR indirect RAM to Accum	1	1
67	xrl	A,@R1	Exclusive-OR indirect RAM to Accum	1	1
6*	xrl	A,Rn	Exclusive-OR register to Accumulator	1	1
C4	swap	A	Swap nibbles within the Accumulator	1	1
E4	clr	A	Clear Accumulator	1	1
F4	cpl	A	Complement Accumulator	1	1

Bit manipulation

Op	Mnem	Register	Description	Byt	Cyc
10	jbc	bit,rel	Jump if direct Bit is set & clear bit	3	2
20	jb	bit,rel	Jump if direct Bit is set	3	2
30	jnb	bit,rel	Jump if direct Bit is Not set	3	2
40	jc	rel	Jump if Carry is set	2	2
50	jnc	rel	Jump if Carry not set	2	2
72	orl	C,bit	OR direct bit to Carry	2	2
A0	orl	C,/bit	OR complement of direct bit to Carry	2	2
82	anl	C, bit	AND direct bit to Carry	2	2
B0	anl	C./bit	AND complement of direct bit to Carry	2	2
92	mov	bit,C	Move Carry to direct bit	2	2
A2	mov	C,bit	Move direct bit to Carry	2	1
B2	cpl	bit	Complement direct bit	2	1
B3	cpl	C	Complement Carry	1	1
C2	clr	bit	Clear direct bit	2	1
C3	clr	C	Clear Carry	1	1
D2	setb	bit	Set direct bit	2	1
D3	setb	C	Set Carry	1	1

Instructions that affect flags

In addition to impacting data, the instructions can influence control bits called flags. These flags represent the results of a completed operation. The flags are carry (C), overflow (OV), and auxiliary carry (AC), which can be used for binary coded decimal arithmetic.

Instruction	Flag				Bit Instruction	Flag
	C	OV	AC			C
add	X	X	X		clr C	O
addc	X	X	X		cpl C	X
subb	X	X	X		anl C,bit	X
mul	O	X			anl C,/bit	X
div	O	X			orl C,bit	X
da	X				orl C,/bit	X
rrc	X				mov C,bit	X
rlc	X				setb C	1
cjne	X					

Instruction set

The op code is obtained by using the number for each column across the top as the first character. The number down the side for each row is the second character.

	0	1	2	3	4	5	6	7
0	nop	jbc bit,rel	jb bit, rel	jnb bit, rel	jc rel	jnc rel	jz rel	jnz rel
I	ajmp (P0)	acall (P0)	ajmp (P 1)	acall (P 1)	ajmp (P2)	acall (P2)	ajmp (P3)	acall (P3)
2	ljmp addr16	lcall addr16	ret [2C]	reti [2C]	orl dir, A	anl dir, A	xrl dir, a	orl C, bit
3	rr A	rrc A	rl A	rlc A	orl dir, #data	anl dir, #data	xrl dir, #data	jmp @A+DPTR
4	inc A	dec A	add A, #data	addc A, #data	orl A, #data	anl A, #data	xrl A, #data	mov A, #data
5	inc dir	dec dir	add A, dir	addc A, dir	orl A, dir	anl A, dir	xrl A, dir	mov dir, #data
6	inc @R0	dec @R0	add A, @R0	addc A, @R0	orl A, @R0	anl A, @R0	xrl A, @R0	mov @R0,@dat
7	inc @R1	dec @R1	add A, @R1	addc A, @R1	orl A, @R1	anl A, @R1	xrl A, @R1	mov @R1,#dat
8	inc R0	dec R0	add A,R0	addc A.R0	orl A,R0	anl A,R0	xrl A,R0	mov R0, #data
9	inc R1	dec R1	add A, R1	addc A, R1	orl A, R1	anl A, R1	xrl A, R1	mov R1,#data
A	inc R2	dec R2	add A,R2	addc A,R2	orl A, R2	anl A,R2	xrl A,R2	mov R2,#data
8	inc R3	dec R3	add A,R3	addc A,R3	orl A,R3	anl A,R3	xrl A,R3	mov R3,#data
C	inc R4	dec R4	add A,R4	addc A,R4	orl A, R4	anl A,R4	xrl A,R4	mov R4,#data
D	inc R5	dec R5	add A,R5	addc A,R5	orl A,R5	anl A,R5	xrl A,R5	mov R5,#data
E	inc R6	dec R6	add A,R6	addc A,R6	orl A,R6	anl A,R6	xrl A,R6	mov R6,#data
F	inc R7	dec R7	add A,R7	addc A,R7	orl A,R7	anl A,R7	xrl A,R7	mov R7,#data

	8	9	A	B	C	D	E	F
0	sjmp REL	mov DPTR,#d16	orl C, /bit	anl C, /bit	push dir	pop dir	movx A,@DPTR	movx @DPTR, A
1	ajmp (P4)	acall (P4)	ajmp (P5)	acall (P5)	ajmp (P6)	acall (P6)	ajmp (P7)	acall (P7)
2	anl C, bit	mov bit, C	mov C, bit	cpl bit	clr bit	setb bit	movx A, @R0	movx @R0, A
3	movc A, @A+PC	movc A, @A+DPTR	inc DPTR	cpl C	clr C	setb C	MOW A, @RI	movx @RI, A
4	div AB	subb A, #data	mul AB	cjne A, #data, rel	swap A	da A	clr A	cpl A
5	mov dir, dir	subb A, dir		cjne A, dir, rel	xch A, dir	djnz dir, rel	mov A, dir	mov dir, A
6	mov dir,@R0	subb A, @R0	mov @R0, dir	cjne @R0, #data, rel	xch A, @R0	xchd A, @R0	mov A, @R0	mov @R0, A
7	mov dir,@R1	subb A, @R1	mov @R1, dir	cjne @R1, #data, rel	xch A, @R1	xchd A, @R1	mov A, @R1	mov @R1, A
8	mov dir, R0	subb A, R0	mov R0, dir	cjne R0, #data, rel	xch A, R0	djnz R0, rel	mov A, R0	mov R0, A
9	mov dir, R1	subb A, R1	mov R1, dir	cjne R1, #data, rel	xch A, R1	djnz RI, rel	mov A, R1	mov R1, A
A	mov dir, R2	subb A. R2	mov R2 dir	cjne R2, #data, rel	xch A, R2	D jnz R2, rel	mov A, R2	mov R2, A
B	mov dir, R3	subb A, R3	mov R3, dir	cjne R3, #data, rel	xch A, R3	djnz R3, rel	mov A. R3	mov R3, A
C	mov dir, R4	subb A. R4	mov R4, dir	cjne R4, #data, rel	xch A, R4	djnz R4, rel	mov A, R4	mov R4, A
D	mov dir, R5	subb A, R5	mov R5, dir	cjne R5, #data, rel	xch A, R5	djnz R5, rel	mov A, R5	mov R5, A
E	mov dir, R6	subb A, R6	mov R6, dir	cjne R6, #data, rel	xch A. R6	djnz R6, rel	mov A, R6	mov R6, A
F	mov dir, R7	subb A, R7	mov R7, dir	cjne R7 #data, rel	xch A, R7	djnz R7, rel	mov A, R7	mov R7, A

48

MEMORY ORGANIZATION

Thought
Experience is great to have.
I just do not like getting it.
MOD

Harvard vs. Princeton

Computer memory organization is typically divided into two types of architecture. The Princeton architecture, also called Von Neumann, has a common memory for data and program code. This was the most common technology when magnetic cores were used for storage.

Harvard architecture has separate memory devices. This became more popular with solid-state memory. Programmable read-only memory (PROM) was one technology that could be used for code. Random access memory (RAM) was a different technology used for data.

The Motorola scheme uses the Princeton approach, while the Intel philosophy developed around the Harvard technique. This applies both to the personal computer systems as well as the microcontroller devices.

Code memory can exist internally as well as on an external chip. In contrast, all versions have a limited internal data memory. External

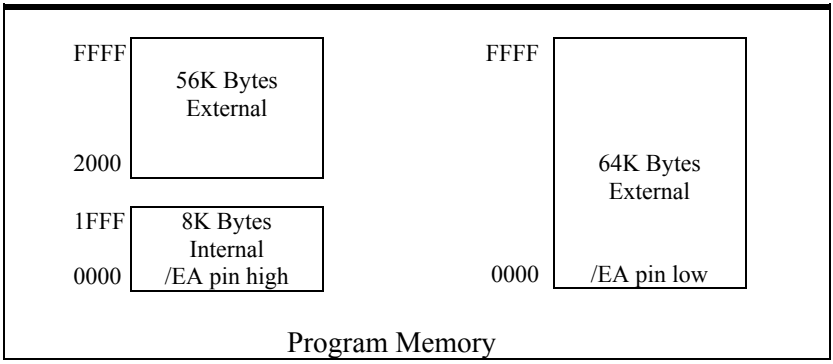
data memory can also be used. In some models, additional internal memory is available in the form of RAM or EEPROM.

Code addresses

The processor memory is arranged in a variety of structures. The lower code memory has reserved space for interrupt vectors.

Symbol	Hex	Meaning
RESET	00	Power on (reset)
EXTI0	03	External interrupt 0
TIMER0	0B	Timer 0 interrupt
EXTI1	13	External interrupt 1
TIMER1	1B	Timer 1 interrupt
SINT	23	Serial port interrupt
	2B	Expansion interrupts

Code memory is accessed based on the chip line /EA. If the line is low, control goes to external memory. If the line is high, initial control goes to internal memory. The next line of code after the top of internal memory is external memory. This transition is regardless of /EA setting. The next address will be in sequence. In other words, the low external memory is inaccessible.



Access to code memory causes the program storage enable (/PSEN) line to be asserted low. This provides the Chip Select for the storage chip.

Instructions that access code memory use the movc mnemonic. This also activates the /PSEN line.

External memory is accessed by sixteen-bit addressing lines. The low lines are located on port 0 and the upper lines are on port 2. This provides a 64k byte page.

Since both data and code are multiplexed on these lines, it is necessary to capture the address. An address latch enable (ALE) line is triggered when addressing is placed on the line by the processor. The ALE is connected to the chip select line of a latch.

External data addresses

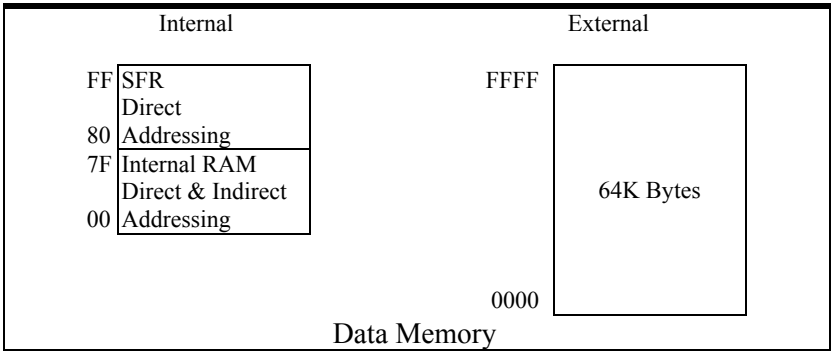
External data memory is typically static ram. It can occupy up to 64K bytes in one page. This is accomplished by sharing the sixteen-bit addressing lines with the code memory. Similarly, the ALE selects the address latch.

The instruction mnemonic for accessing external data memory is movx. The instruction will activate either the read (/RD) or the write (/WR) lines. The write line is connected to the memory-chip write-enable pin. Similarly, the read line is connected to the output enable.

Data memory expansion

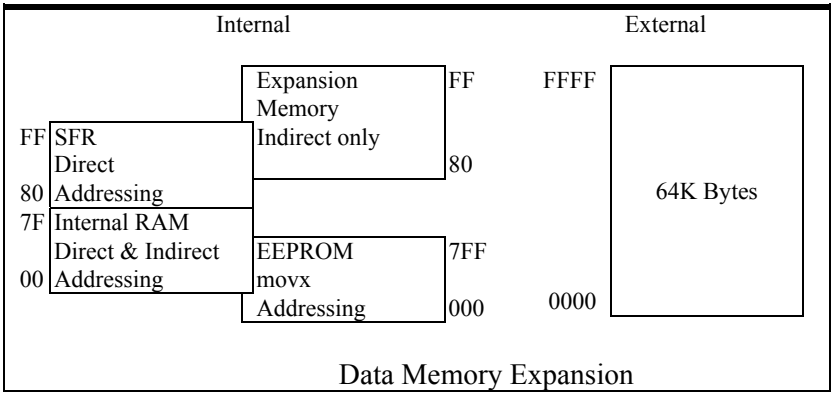
Data memory can be segregated into three types - internal ram, external ram, and internal EEPROM.

Internal and external ram capability is standard across the microcontroller line. The internal ram contains the lower 128 bytes with only 21 special function registers in the upper 128 area.



Enhanced microcontrollers permit access to the remaining locations of the upper 128 bytes of internal ram.

In addition, some models have EEPROM or other memory that is internal on the chip. However, addressing this memory is done exactly as if it were external. Therefore, movx instructions are required.



Internal data addresses

The basic processor has 128 bytes of internal data that is available. The entire area may be used as general-purpose data. In addition,

the lower part can be used as registers. Another group of bytes is bit addressable.

The basic instruction that accesses this area is mov. It does not influence any external pins or lines. Numerous other instructions can also be used, depending on the addressing mode.

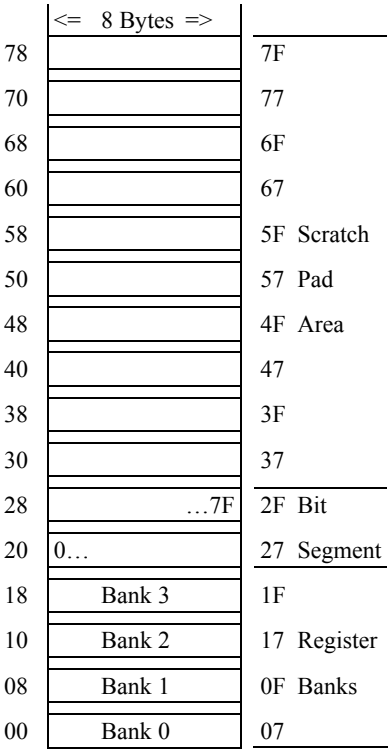
The next 128 bytes of internal memory depend on the manufacturer design. For the basic microcontroller, 21 special function registers (SFR) are implemented from this memory. That leaves a substantial portion that is unused. In terms of memory space, a table identifies which locations are reserved for the registers. Direct addressing accesses the data in the SFR.

The special function registers permit control of many features and capabilities. The next chapter is dedicated to the functions and implementation.

Some versions of the microcontroller allow access to the remaining bytes of the internal memory. The internal memory will be stated as 256 bytes. Indirect addressing accesses data in the upper 128 bytes.

Other internal memory such as EEPROM is available in certain models. Discussion on this topic is specific to the device and will be broached in a different section.

Internal RAM low



Internal RAM high

Locations that are divisible by eight are bit addressable. Therefore, any location that has a hex address that ends with 0 or 8 has bit access. The first bit corresponds to the byte address.

These locations are also called special function registers. Registers that are identified with a plus (+) symbol are added features that may be in various designs.

Symbol	Name	Address
ACC	Accumulator	0E0H
B	B or multiplication register	0F0H
PSW	Program Status Word	0D0H
TH2+	Timer/Counter 2 High Byte	0CDH
TL2+	Timer/Counter 2 Low Byte	0CCH
RCAP2H+	T/C 2 Capture Reg. High Byte	0CBH
RCAP2L+	T/C 2 Capture Reg. Low Byte	0CAH
T2MOD+	Timer/Counter 2 Mode Control	0C9H
T2CON+	Timer/Counter 2 Control	0C8H
IP	Interrupt Priority Control	0B8H
P3	Port 3	0B0H
IE	Interrupt Enable Control	0A8H
P2	Port 2	0A0H
SBUF	Serial Data Buffer	99H
SCON	Serial Control	98H
P1	Port 1	90H
TH1	Timer/Counter 1 High Byte	8DH
TH0	Timer/Counter 0 High Byte	8CH
TL1	Timer/Counter 1 Low Byte	8BH
TL0	Timer/Counter 0 Low Byte	8AH
TMOD	Timer/Counter Mode Control	89H
TCON	Timer/Counter Control	88H
PCON	Power Control	87H
DPTR	Data Pointer 2 Bytes	
DPL	Low Byte	82H
DPH	High Byte	83H
SP	Stack Pointer	81H
P0	Port 0	80H

Predefined bit addresses

Bit addresses are the components of bit definable special function registers (SFR). These can be accessed either by the byte or by the individual bits.

Sym.	Position	Hex	Meaning
CY	PSW.7	D7	Carry flag
AC	PSW.6	D6	Auxiliary carry flag
FO	PSW.5	D5	Flag 0
RS1	PSW.4	D4	Register bank select bit 1
RSO	PSW.3	D3	Register bank select bit 0
OV	PSW.2	D2	Overflow flag
P	PSW.O	D0	Parity flag
TF1	TCON.7	8F	Timer 1 overflow flag
TR1	TCON.6	8E	Timer 1 run control bit
TFO	TCON 5	8D	Timer 0 overflow flag
TRO	TCON.4	8C	Timer 0 run control bit
IE1	TCON 3	8B	Interrupt 1 edge flag
IT1	TCON.2	8A	Interrupt 1 type control bit
IE0	TCON.1	89	Interrupt 0 edge flag
IT0	TCON.0	88	Interrupt 0 type control bit
SMO	SCON.7	9F	Serial mode control bit 0
SM1	SCON.6	9E	Serial mode control bit 1
SM2	SCON.5	9D	Serial mode control bit 2
REN	SCON.4	9C	Receive enable
TB8	SCON.3	9B	Transmit bit 8
RB8	SCON.2	9A	Receive bit 8
TI	SCON.1	99	Transmit interrupt flag
RI	SCON.0	98	Receive interrupt flag
EA	IE.7	AF	Enable all interrupts
ES	IE.4	AC	Enable serial port interrupt
ET1	IE.3	AB	Enable timer 1 interrupt
EX1	IE.2	AA	Enable external interrupt 1
ETO	IE.1	A9	Enable timer 0 interrupt
EXO	IE.O	A8	Enable external interrupt 0
PS	IPA	BC	Priority of serial port interrupt
PT1	IP.3	BB	Priority of timer i interrupt
PX1	IP.2	BA	Priority of external interrupt 1
PTO	IP.1	B9	Priority of timer 0
PXO	IP.O	B8	Priority of external interrupt 0

Predefined bits port 3

Port 3 is an enhanced function location. It can be accessed as an input / output port. In addition, each bit is associated with control of external devices.

Sym.	Position	Hex	Meaning
RD	P3.7	87	Read data for external memory
WR	P3.6	B6	Write data for external memory
T1	P3.5	B5	Timer/counter 1 external flag
TO	P3.4	B4	Timer/counter 0 external flag
INT1	P3.3	B3	Interrupt 1 input pin
INTO	P3.2	B2	Interrupt 0 input pin
TXD	P3.1	B1	Serial port transmit pin
RXD	P3.0	BO	Serial port receive pin

⇐⇑⇒

49

SPECIAL FUNCTION REGISTERS

Thought
What is so special
about function registers?
First time user

Reserved memory

The microprocessor has many unique capabilities that give it features of very powerful machines. One of those discussed has been some of the characteristics of a reduced instruction set (RISC) machine. Another was sequential processing as well as powerful stack capability. Another is serial or parallel data transfer. The next is internal memory is register accessible.

There are 21 special function registers (SFR) in the standard microcontroller. These are located in the upper 128 bytes of the internal static RAM. Hence, the bytes are both memory and registers. These registers can be accessed using instructions for memory. In addition, each register has instructions that are special just for that register.

The registers that will be addressed in this section are the ports as well as the control and mode for status flags, power control, timers, and serial.

Ports

Four ports are connected to external pins. As such, they are the input and output connections to the processor. All ports can be used for general-purpose input and output. In addition, ports 0, 2, and 3 have special applications. On some machines, even port 1 has additional functions. The four ports are similar, but the hardware for each is different because of the functions.

In all cases, data stored in the special function register or data memory address will be displayed on the ports. This information may be identified as a byte or individual bits.

To make the port pins perform as an input, first place a 1 on each bit that is to be read. This will pull the line high so that external switches may be connected. Then the data that is read from the port address is the status of the external switches.

Port 0

Port 0 is at internal data address 80h. When external memory is accessed, the port becomes multiplexed. First, the lower eight bits of the address are displayed on the port. Then the data for that address is on the port. Depending on the instruction, the data is either written to the address or read from the address.

AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0

The port has very weak internal pull-up resistors. When used as an output, the port functions like the other ports. However, it is limited in the driving current. When the port is used as an input, it must have external pull-up or pull-down resistors attached. Connect a 2.2 KΩ resistorto 5 volts or to ground as required.

Port 1

Port 1 is at internal data address 90h. It is the general-purpose input output register. Most external connections are made to this port. It has strong internal pull-up resistors; therefore, it does not float.

SCK	MISO	MOSI	P1.4	P1.3	P1.2	P1.1	P1.0

On machines that have serial peripheral interface (SPI) capability, the upper bits of the port are used. Devices that have in-system programming, use the SPI connections.

After the SPI exchange, the port pins can revert to standard input / output. However, the port pins should not be used for several instruction cycles after the in system programming, or the machine may lock up.

Port 2

Port 2 is at internal data address 0A0h. When external memory is accessed, the upper eight bits of the address are displayed on the port. The port is not readily shared as an input / output if external memory is used.

A15	A14	A13	A12	A11	A10	A9	A8

Port 3

Port 3 is at internal data address 0B0h. It is used as a function management location. Each bit is associated with control of external devices. This includes external memory, serial communications, interrupts, and timers.

RD	WR	T1	T0	INT1	INT0	TXD	RXD

Sym.	Position	Hex	Meaning
RD	P3.7	87	Read data for external memory
WR	P3.6	B6	Write data for external memory
T1	P3.5	B5	Timer/counter 1 external flag
T0	P3.4	B4	Timer/counter 0 external flag
INT1	P3.3	B3	Interrupt 1 input pin
INTO	P3.2	B2	Interrupt 0 input pin
TXD	P3.1	B1	Serial port transmit pin
RXD	P3.0	B0	Serial port receive pin

PSW: Program status word

The program status word is a bit addressable location. The register bank selected is determined by the combination of bits RS1-RS0. The result is bank 0 to 3.

CY	AC	F0	RS1	RS0	OV	—	P

CY	PSW.7	Carry flag.
AC	PSW.6	Auxiliary carry flag.
F0	PSW.5	Flag 0 available to the user for general purpose.
RS1	PSW.4	Register Bank selector bit 1.
RS0	PSW.3	Register Bank selector bit 0.
OV	PSW.2	Overflow flag.
—	PSW.1	User definable flag.
P	PSW.0	Parity flag. Set/cleared by hardware each instruction cycle to indicate an odd/even number of 1 bits in the accumulator.

RS1	RS0	Bank	Address
0	0	0	00H-07H
0	1	1	08H-0FH
1	0	2	10H-17H
1	1	3	18H-1FH

PCON: Power control register

The PCON register is not bit addressable. Therefore values must be moved into the location.

SMOD	—	—	—	GF1	GF0	PD	IDL

SMOD	Double baud rate bit. If Timer 1 is used to generate baud rate and SMOD = 1, the baud rate is doubled when the Serial port is used in modes 1, 2, or 3.
	Serial port is used in modes 1, 2, or 3.
—	Not implemented, reserved for future use.
—	Not implemented, reserved for future use.
—	Not implemented, reserved for future use.
GF1	General purpose flag bit.
GF0	General purpose flag bit.
PD	Power Down bit. Setting this bit activates Power Down operation in select processors.
IDL	Idle Mode bit. Setting this bit activates Idle Mode operation in select processors. If 1s are written to PD and IDL at the same time, PD takes precedence.

Interrupts

To use any of the interrupts in the microcontroller, take the following three steps.

1. Set the EA (enable all) bit in the IE register to 1.
2. Set the corresponding individual interrupt enable bit in the IE register to 1.
3. Begin the interrupt service routine at the corresponding vector address of that interrupt. See the following table.

If the bit is 0, the corresponding interrupt is disabled. If the bit is 1, the corresponding interrupt is enabled.

In addition, for external interrupts, pins INT0 and INT1 (P3.2 and P3.3) must be set to 1, and depending on whether the interrupt is to be level or transition activated, bits IT0 or IT1 in the TCON register may need to be set to 1. ITx = 0 is used for level activated, while ITx = 1 makes the interrupt transition activated.

Both the interrupt enable and the interrupt priority registers are bit addressable. These can be changed with setb, clr, anl, orl, and xrl instructions.

Function.	**Interrupt Source**	**Vector Address Hex**
External 0	IE0	0003H
Timer 0	TF0	000BH
External 1	IE1	0013H
Timer 1	TF1	001BH
Serial	R1 & T1	0023H
Timer 2	TF2 & EXF2	002BH

To assign higher priority to an interrupt, the corresponding bit in the IP register must be set to 1. While an interrupt service is in progress, it cannot be interrupted by an interrupt of the same or lower priority.

The only purpose of priority within a level is to resolve simultaneous requests of the same priority level. If the bit is 0, the

corresponding interrupt has a lower priority. If the bit is 1, the corresponding interrupt has a higher priority. The interrupt sources are listed below in order from highest to lowest priority.

IE0	TF0	IE1	TF1	RI / TI	TF2 / EXF2

IE: Interrupt enable register

EA	—	ET2	ES	ET1	EX1	ET0	EX0

EA	IE.7	Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
—	IE.6	Not implemented, reserved for future use.(1)
ET2	IE.5	Enables or disables the Timer 2 overflow or capture interrupt on select models.
ES	IE.4	Enables or disables the serial port interrupt.
ET1	IE.3	Enables or disables the Timer 1 overflow interrupt.
EX1	IE.2	Enables or disables External Interrupt 1.
ET0	IE. 1	Enables or disables the Timer 0 overflow interrupt.
EX0	IE.0	Enables or disables External Interrupt 0.

IP: Interrupt priority register

—	—	PT2	PS	PT1	PX1	PT0	PX0

—	IP. 7	Not implemented, reserved for future use.
—	IP. 6	Not implemented, reserved for future use.
PT2	IP. 5	Defines the Timer 2 interrupt priority level.+
PS	IP. 4	Defines the Serial port interrupt priority level.
PT1	IP. 3	Defines the Timer 1 interrupt priority level.
PX1	IP. 2	Defines External Interrupt 1 priority level.
PT0	IP. 1	Defines the Timer 0 interrupt priority level.
PX0	IP. 0	Defines the External Interrupt 0 priority level.

Timer / counters

The microcontroller has two timers. These can be used as timers or counters. A timer simply counts the number of clock cycles. Two registers operate the timer/counters.

The Control register determines if the timer is running, has completed its count, and what type signal provides the trigger. The Control register is bit addressable. Therefore, discrete values can be changed.

The Mode register operates the timers as 8, 13, or sixteen bit devices. The Mode register is not bit addressable, so only mov instructions are appropriate.

Two tables are given for selecting the mode. The last table has TMOD values that can be used to set up Timer 0. It is assumed that only one timer is used at a time. If Timers 0 and 1 must run simultaneously in any mode, the value in TMOD for Timer 0 must be ORed with the value required for Timer 1.

For example, if Timer 0 is in mode 1 with Gate (external control), and Timer 1 is run in mode 2 as a counter, then the value that must be loaded into TMOD is 69H (09H for Timer 0 ORed with 60H for Timer 1.

Moreover, it is assumed that the timer is not turned on at this point. It can be started at another point in the program by setting bit TRx (in TCON) to 1.

1. Timer x is turned ON/OFF by setting/clearing bit TRx in the software.
2. Timer 0 is turned ON/OFF by the 1 to 0 transition on INT0 (P3.2) when TR0 = 1 (hardware control).
3. Timer 1 is turned ON/OFF by the 1 to 0 transition on INT1 (P3.3) when TR1 = 1 (hardware control).

TCON: Timer/counter control register

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON. 7	Timer 1 overflow flag. Set by hardware when the Timer/Counter 1 overflows. Cleared by hardware as the processor vectors to the interrupt service routine.
TR1	TCON. 6	Timer 1 run control bit. Set/cleared by software to turn Timer/Counter 1 ON/OFF.
TF0	TCON. 5	Timer 0 overflow flag. Set by hardware when the Timer/Counter 0 overflows. Cleared by hardware as the processor vectors to the service routine.
TR0	TCON. 4	Timer 0 run control bit. Set/cleared by software to turn Timer/Counter 0 ON/OFF.
IE1	TCON. 3	External Interrupt 1 edge flag. Set by hardware when the External Interrupt edge is detected.
		Cleared by hardware when the interrupt is processed.
IT1	TCON. 2	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.
IE0	TCON. 1	External Interrupt 0 edge flag. Set by hardware when External Interrupt edge detected. Cleared by hardware when interrupt is processed.
IT0	TCON. 0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low level triggered External Interrupt.

TMOD: Timer/counter mode register

GATE-1	C/T -1	M1 –1	M0 -1	GATE-0	C/T -0	M1 -0	M0 -0

GATE	When TRx (in TCON) is set and GATE = 1, TIMER/COUNTERx runs only while the INTx pin is high (hardware control). When GATE = 0, TIMER/COUNTERx will run only while TRx = 1 (software control).
C/T	Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation(input from Tx input pin).
M1	Mode selector bit.
M0	Mode selector bit.

M1	M0	Mode	Operation
0	0	0	13-bit Timer
0	1	1	sixteen bit Timer/Counter
1	0	2	eight bit Auto-Reload Timer/Counter
1	1	3	Split Timer Mode: (Timer 0) TL0 is an eight bit Timer/Counter controlled by the standard Timer 0 control bits, TH0 is an eight bit Timer and is controlled by Timer 1 control bits.
1	1	3	(Timer 1) Timer/Counter 1 stopped.

Mode	Function	TMOD Count		TMOD Time	
		Internal	External	Internal	External
0	13-bit Timer	00H	08H	04H	0CH
1	sixteen bit Timer	01H	09H	05H	0DH
2	eight bit Auto-Reload	02H	0AH	06H	0EH
3	two eight bit Timers	03H	0BH	07H	0FH

Serial

Serial communications is directed by a universal asynchronous receive transmit (UART) circuit. Timer 1, or Timer 2 if it is available, determines the baud rate. The serial control register is bit addressable.

Operation mode determines the use as a shift, variable baud, or fixed baud register.

SCON: Serial control register

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0	SCON. 7	Serial port mode selection.
SM1	SCON. 6	Serial port mode selection.
SM2	SCON. 5	Enables the multiprocessor communication feature in modes 2 and 3. In mode 2 or 3, if SM2 is set to 1, then RI is not activated if the received 9th data bit (RB8) is 0. In mode 1, if SM2 = 1, then RI is not activated if a valid stop bit was not received. In mode 0, SM2 should be 0.
REN	SCON. 4	Set/Cleared by software to Enable/Disable reception.
TB8	SCON. 3	The 9th bit that is transmitted in modes 2 and 3. Set/Cleared by software.
RB8	SCON. 2	In modes 2 and 3, is the 9th data bit that was received. In mode 1, if SM2 = 0, RB8 is the stop bit that was received. In mode 0, RB8 is not used.
TI	SCON. 1	Transmit interrupt flag. Set by hardware at the end of the 8th bit time in mode 0 or at the beginning of the stop bit in the other modes. Must be cleared by software.
RI	SCON. 0	Receive interrupt flag. Set by hardware at the end of the 8th bit time in mode 0 or halfway through the stop bit time in the other modes (except see SM2). Must be cleared by software.

SM0	SM1	Mode	Description	Baud Rate
0	0	0	Shift register	Fosc./12
0	1	1	eight bit UART	Variable
1	0	2	9-Bit UART	Fosc./64 OR Fosc./32
1	1	3	9-Bit UART	Variable

MODE	SCON	SM2 VARIATION
0	10H	Single Processor
1	50H	Environment
2	90H	(SM2 = 0)
3	D0H	
0	NA	Multiprocessor
1	70H	Environment
2	B0H	(SM2 = 1)
3	F0H	

⇐ ⇑ ⇒

50

SFR EXTENDED

Thought
Be strong, be courageous,
Be not afraid.
General Joshua, ~1500 BC

Enhanced registers

The previous registers are standard to the family of microprocessors. However, there are numerous other registers that various enhanced microprocessor versions offer. Some of the more common registers handle additional timers, on-board memory, additional data pointer, and interfaces such as SPI. Illustrations of some of these extended group or registers are included.

Timer/counter 2

Some versions of the microcontroller have a third timer/counter register. It is operated very similar to the standard Timer 0 and 1.

T2CON is located at address 0C8h and T2MOD is address 0C9h. The control register is bit addressable, but the mode register is not. The reset value for the mode register is = xxxx xx00b.

Four data registers are associated, RCAP2L, RCAP2H, TL2, and TH2.

A table provides typical initialization values. Except for the baud-rate generator mode, the values given for T2CON do not include the setting of the TR2 bit. Therefore, bit TR2 must be set separately to turn the Timer on.

1. Capture/Reload occurs only on Timer/Counter overflow.
2. Capture/Reload occurs on Timer/Counter overflow and a 1 to 0 transition on T2EX (P1.1) pin except when Timer 2 is used in the baud-rate generating mode.

T2CON: Timer/counter 2 control register

Address = 0C8h Reset Value = 0000 0000b

TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2	CP/RL2

TF2	T2CON. 7	Timer 2 overflow flag set by hardware and cleared by software. TF2 cannot be set when either RCLK = 1 or CLK = 1
EXF2	T2CON. 6	Timer 2 external flag set when either a capture or reload is caused by a negative transition on T2EX, and EXEN2 = 1. When Timer 2 interrupt is enabled, EXF2 = 1 causes the CPU to vector to the Timer 2 interrupt routine. EXF2 must be cleared by software.
RCLK	T2CON. 5	Receive clock flag. When set, causes the Serial port to use Timer 2 overflow pulses for its receive clock in modes 1 and 3. RCLK = 0 causes Timer 1 overflow to be used for the receive clock.
TLCK	T2CON. 4	Transmit clock flag. When set, causes the Serial port to use Timer 2 overflow pulses for its transmit clock in modes 1 and 3. TCLK = 0 causes Timer 1 overflows to be used for the transmit clock.
EXEN2	T2CON. 3	Timer 2 external enable flag. When set, allows a capture or reload to occur as a result of negative transition on T2EX if Timer 2 is not being used to clock the Serial port. EXEN2 = 0 causes Timer 2 to ignore events at T2EX.
TR2	T2CON. 2	Software START/STOP control for Timer 2. A logic 1 starts the Timer.
C/T2	T2CON. 1	Timer or Counter select. 0 = Internal Timer. 1 = External Event Counter (triggered by falling edge).
CP/RL2	T2CON. 0	Capture/Reload flag. When set, captures occur on negative transitions at T2EX if EXEN2 = 1. When
		cleared, auto-reloads occur either with Timer 2 overflows or negative transitions at T2EX when
		EXEN2 = 1. When either RCLK = 1 or TCLK = 1, this bit is ignored and the Timer is forced to auto- reload on Timer 2 overflow.

T2MOD: Timer 2 mode register ____

Address = 0C9hh Reset Value = xxxx xx00b

-	-	-	-	-	-	T2OE	DCEN

-	Not implemented, reserved for future use
T2OE	Timer 2 Output Enable bit
DCEN	When set, this bit allows Timer 2 to be configured as an up/down counter.

Function	**TMOD2**	**Time**	**TMOD2**	**Count**
	Internal	**External**	**Internal**	**External**
Sixteen bit Auto-Reload	00H	08H	02H	0AH
Sixteen bit Capture	01H	09H	03H	0BH
Receive only	24H	26H		
Transmit only	14H	16H		
Baud rate generator receive and transmit same baud rate	34H	36H		

Timer 2 data registers _____

RECAP2L: Address = 0CAh Reset Value = 0000 0000b
RECAP2H Address = 0CBh Reset Value = 0000 0000b
TL2 Address = 0CCh Reset Value = 0000 0000b
TH2 Address = 0CDh Reset Value = 0000 0000b

D7	D6	D4	D3	D2	D1	D0

Serial peripheral interface

The serial peripheral interface (SPI) allows high-speed synchronous data transfer between a master and peripheral chips. The characteristics are listed.

1. Full-Duplex, 3-Wire Synchronous Data Transfer
2. Master or Slave Operation
3. 1.5 MHz Bit Frequency (max.)
4. LSB First or MSB First Data Transfer
5. Four Programmable Bit Rates
6. End of Transmission Interrupt Flag
7. Write Collision Flag Protection
8. Wakeup from Idle Mode (Slave Mode Only)

A unique select line is connected to each device. In addition, there are three common lines.

MOSI – master out, slave in.
MISO – master in, slave out.
SCK – clock generated by the master.

The SCK pin is the clock output in the master mode but is the clock input in the slave mode. Writing to the SPI data register of the master CPU starts the SPI clock generator, and the data written shifts out of the MOSI pin and into the MOSI pin of the slave CPU. After shifting one byte, the SPI clock generator stops, setting the end of transmission flag (SPIF). If both the SPI interrupt enable bit (SPIE) and the serial port interrupt enable bit (ES) are set, an interrupt is requested.

The Slave Select input, SS/P1.4, is set low to select an individual SPI device as a slave. When SS/P1.4 is set high, the SPI port is deactivated and the MOSI/P1.5 pin can be used as an input.

There are four combinations of SCK phase and polarity with respect to serial data, which are determined by control bits CPHA and CPOL.

The data register (SPDR) is double buffered, so data can be written both directions simultaneously.

The clock frequency is determined by the control register bits 0 and 1. The frequency is the microprocessors clock divided by the selected divisor. The divisor is generated in the table.

SPR1	SPR0	DIVISOR
0	0	4
0	1	16
1	0	64
1	1	128

A generic technique for SPI is shown in the SPI projects chapter. It will work with any processor, including those without SPI registers. Another procedure for implementing on-board SPI for read and write is also included.

SPCR: SPI control register

Address = 0D5h Reset Value = 0000 01xxb

SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0

SPIE	SPCR. 7	SPI Interrupt Enable. This bit, in conjunction with the ES bit in the IE register, = 1 enable SPI interrupts. SPIE = 0 disables SPI interrupts.
SPE	SPCR. 6	SPI Enable. SPI = 1 enables the SPI channel and connects SS, MOSI, P1.7. SPI = 0 disables the SPI channel.
DORD	SPCR. 5	Data Order. DORD = 1 selects LSB first data transmission. DORD = 0 selects MSB first data transmission
MSTR	SPCR. 4	Master/Slave Select. MSTR = 1 selects Master SPI mode. MSTR = 0 selects slave SPI mode
CPOL	SPCR. 3	Clock Polarity. When CPOL = 1, SCK is high when idle. When CPOL =0, SCK of the master device is low when not transmitting.
CPHA	SPCR. 2	Clock phase. The CPHA bit together with the CPOL bit controls the clock and data relatinship between master and slave.
SPR0	SPCR. 1	SPI clock rate select. These 2 bits control the SCK rate of the device configured as amster. They have no effect on the slave. The SCK frequency is the oscillator frequency divided by the values. SPR1 SPR0 Divisor 0 0 4 0 1 16 1 0 64 1 1 128
SPR1	SPCR. 0	

SPSR: SPI status register _____

Address = 0AAh Reset Value = 00xx xxxxb

SPIF	WCOL						

SPIF	SPSR. 7	SPI Interrupt Flag. When a serial transfer is complete, the SPIF bit is set and an interrupt is generated if SPIE = 1 and ES = 1. The SPIF bit is cleared by reading the SPI status register with SPIF and WCOL bits set, and then accessing the SPI data register
WCOL	SPSR. 6	Write collision flag. The bit is set if the SPI data register is written during a data transfer. During transfer, the result of reading the SPDR register may be incorrect, and writing to it has no effect. The WCOL bit and the SPIF bit are cleared by reading the SPI status register with SPIF and WCOL set, and then accessing the SPI data register.

SPDR: SPI data register

Address = 86h Reset Value = unchanged

SPD7	SPD6	SPD5	SPD4	SPD3	SPD2	SPD1	SPD0

WMCON: Watchdog

The watchdog and memory control register (WMCON) contains control bits for several expansion functions. One is the Watchdog Timer. The EEMEN and EEMWE bits are used to select the 2K bytes on-chip EEPROM, and to enable byte-write. The DPS bit selects which of the two DPTR registers are active.

Address = 96h Reset Value = 0000 0010b

PS2	PS1	PS0	EEMWE	EEMEN	DPS	WDTRST	WDTEN

PS2	WMCON.7	Prescaler Bits for the Watchdog Timer..
PS1	WMCON.6	When all three bits are set to “0”, the watchdog timer has a nominal period of16 ms.
PS0	WMCON.5	When all three bits are set to “1”, the nominal period is 2048 ms
EEMWE	WMCON.4	EEPROM Data Memory Write Enable Bit. Set this bit to “1” before initiating byte write to on-chip EEPROM with the
EEMEN	WMCON.3	movx instruction. User software should set this bit to “0” after EEPROM write is completed.
DPS	WMCON.2	Internal EEPROM Access Enable. When EEMEN = 1, the movx instruction with DPTR will access on-chip EEPROM
WDTRST	WMCON.1	Watchdog Timer Reset and EEPROM Ready/Busy Flag. Each time this bit is set to “1” by user software, a pulse is
RDY/BSY	WMCON.0	generated to reset the watchdog timer. The WDTRST bit is then automatically reset to “0” in the next instruction cycle.

Using onboard EEPROM

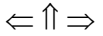
The onboard eeprom requires the WMCON register. This is very similar to an external IIC or SPI function.

```
;-------------------------------------------------
;Program: EEPROM.ASM
;Initial: July 28, 2004
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c) 2004.  All rights reserved
;  Original adapted from Atmel.
;
;Purpose:
;  A set of routines are provided to write and
;  read from the on board EEPROM.
;
;Processor: 8031 family
;PROM:      8k (2000H) onboard
;Crystal:   11.059 MHz
;Assembler: Intel ASM51


;################################################
;                      ASSIGNMENTS
;################################################
;CONSTANTS
;-------------------------------------------------
;                      ;WMCON REGISTER
Spcr      data  0d5h  ;SPI control register

Wmcon     data  96h   ;watchdog & memory register
Eemen     equ   00001000b ;EEPROM access enabl bit
Eemwe     equ   00010000b ;EEPROM write enable bit
Wdtrst    equ   00000010b ;EEPROM RDY/BSY bit


;################################################
;                      PROGRAM
;################################################
          org   00h
```

```
START:    ljmp  INITIAL

          org   0033h ;Address past vectors
          db    'Marcus O. Durham, PhD, PE'

;-------------------------------------------------
          org   0080h        ;Address past reserve
INITIAL:
MAIN:
;-------------------------------------------------
          lcall WMEEPRD      ;read eeprom

MAN9:     sjmp  MAIN


;-------------------------------------------------
WMEEPRD:
;-------------------------------------------------
;  WMCON is used to access internal EEPROM.
;  WMCON is not bit addressable, so Boolean
;  functions are necessary to control bits.
;  EEPROM read

          orl   Wmcon,#Eemen ;enable EEPROM access
          mov   DPTR,#Address;address to read
          movx  A,@DPTR      ;read EEPROM
          xrl   Wmcon,#Eemen ;disable EEP access

          ret                ;home again

;-------------------------------------------------
WMEEPWRD:
;-------------------------------------------------
;  EEPROM write example, utilizing fixed delay
;  for write cycle. Delay is worst case (10 ms).
;  Write is followed by verify (read & compare)
;  Code for delay is not shown.
;  Code to handle verification failure not shown.

          orl   Wmcon,#Eemen ;enable EEPROM access
          orl   Wmcon,#Eemwe ;enable EEPROM writes
          mov   Dptr,#Address;address to write
          mov   A,#Data      ;data to write
          movx  @Dptr,A      ;write EEPROM
```

```
          CALL  DELAY_10_MS  ;wait 10 ms
          movx  A,@Dptr      ;read EEPROM
          cjne  A,#Data,Error;data compare fails
          xrl   Wmcon,#Eemwe ;disable EEPROM write
          xrl   Wmcon,#Eemen ;disabl EEPROM access

          ret                ;home again

;-------------------------------------------------
WMEEPWRB:
;-------------------------------------------------
;  EEPROM write example, utilizing RDY/BSY
;  to determine the end of the write cycle.
;  Write is followed by verify (read and compare)
;  Needs timeout to prevent write error from
;  causing an infinite loop.

          orl   Wmcon,#Eemen ;enable EEPROM access
          orl   Wmcon,#Eemwe ;enable EEPROM writes
          mov   Dptr,#Address;address to write
          mov   A,#Data      ;data to write
          movx  @Dptr,A      ;write EEPROM

Loop:     mov   A,Wmcon      ;EEPROM write status
          anl   A,#Wdtrst    ;check RDY/BSY
          jz    Loop         ;Jump if busy
          movx  A,@Dptr      ;read EEPROM
          cjne  A,#Data,Error;data compare fails
          xrl   Wmcon,#Eemwe ;disable EEPROM write
          xrl   Wmcon,#Eemen ;disabl EEPROM access

          ret                ;home again

;-------------------------------------------------
WMEEPWRP:
;-------------------------------------------------
;  EEPROM write example, utilizing data polling
;  to determine the end of; the write cycle.
;  After data is loaded, the code loops on read
;  until data is returned true.
;  Write verification is implicit in this method.
;  Needs timeout to prevent write error from
;  causing an infinite loop.
```

```
        orl   Wmcon,#Eemen ;enable EEPROM access
        orl   Wmcon,#Eemwe ;enable EEPROM writes
        mov   Dptr,#Address;address to write
        mov   A,#Data      ;data to write
        movx  @Dptr,A      ;write EEPROM

Loop:   movx  A,@Dptr      ;read EEPROM
        cjne  A,#Data,loop ;data compare (busy)
        xrl   Wmcon,#Eemwe ;disable EEPROM write
        xrl   Wmcon,#Eemen ;disabl EEPROM access

        ret                ;home again
```



SECTION VI – COMMUNICATION

⇐⇑⇒

51

ASCII

Thought
Seldom does one problem produce a failure.
It takes at least two problems to cause a catastrophe.
MOD

What is it _____

Traditionally, communications between teletypes was done using the American Standard Code for Information Interchange (ASCII). As computer equipment developed and replaced the early Teletype systems, communications between computer devices used the same ASCII format.

It is interesting that the Teletype technology operated with a current loop. The instrumentation system of 4-20 mA is a direct descendent. Therefore both computer communications and instrumentation signaling can trace their lineage to the Teletype.

Computers internally are considered to operate with a hexadecimal (hex) format. This is simply a selection or combination of four bits.

Therefore, software within the computer must convert between hex and ASCII. This table will provide the normal standards. An additional table provides the next 127 characters. However, the additional special characters are seldom used.

Because of the history, most of the first characters are somewhat arcane. These were used as control messages for the early machines. Only a few of them have application to current transmissions.

The remaining tables provide the characters, numbers, and punctuation used by most messages.

ASCII-hex table

ASCII	Hex	Symbol		ASCII	Hex	Symbol	
0	0	NUL	null, empty	16	10	DLE	data link escape
1	1	SOH	start of heading	17	11	DC1	device control 1
2	2	STX	start of text	18	12	DC2	device control 2
3	3	ETX	end of test	19	13	DC3	device control 3
4	4	EOT	end of xmission	20	14	DC4	device control 4
5	5	ENQ	enquire	21	15	NAK	not acknowledge
6	6	ACK	acknowledge	22	16	SYN	synchronous idle
7	7	BEL	bell ring	23	17	ETB	EOT block
8	8	BS	back space	24	18	CAN	cancel
9	9	TAB	horizontal tab	25	19	EM	end of medium
10	A	LF	line feed	26	1A	SUB	substitute
11	B	VT	vertical tab	27	1B	ESC	escape
12	C	FF	form feed	28	1C	FS	file separator
13	D	CR	carriage return	29	1D	GS	group separator
14	E	SO	shift out	30	1E	RS	record separator
15	F	SI	shift in	31	1F	US	unit separator

ASCII	Hex	Symbol
32	20	(space)
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/

ASCII	Hex	Symbol
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

ASCII	Hex	Symbol
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O

ASCII	Hex	Symbol
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

ASCII	Hex	Symbol
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

ASCII	Hex	Symbol
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	□

52

RS 232

Thought
God does not throw dice.
Dr. Albert Einstein

Connections

The Electrical Industry Apparatus (EIA) has established numerous standards for connections and communications. One of the originals was RS232. This is for serial exchange of data between two machines. Other standards, such as RS 422/485 are used for communications with multiple devices.

These standards were developed for connections in environments that are uncontrolled. Any device can be connected and it can be remote. Therefore, the signal levels are basically 9 - 25 volts. Any device connected using the standard must be able to operate under these voltage constraints.

Numerous connection possibilities exist with serial cables. The most common designations are shown in the tables. The function is compared to the standard db 25 designations. A ‘P’ indicates plug while ‘S’ is a socket.

When connecting to other termination types such as a microcontroller board, treat the cable as a null modem. Make the terminations according to the function. With a null cable, the

functions are crossed. For example TX connects to RX. Only lines 2, 3, & 7 are required for bi-directional communications.

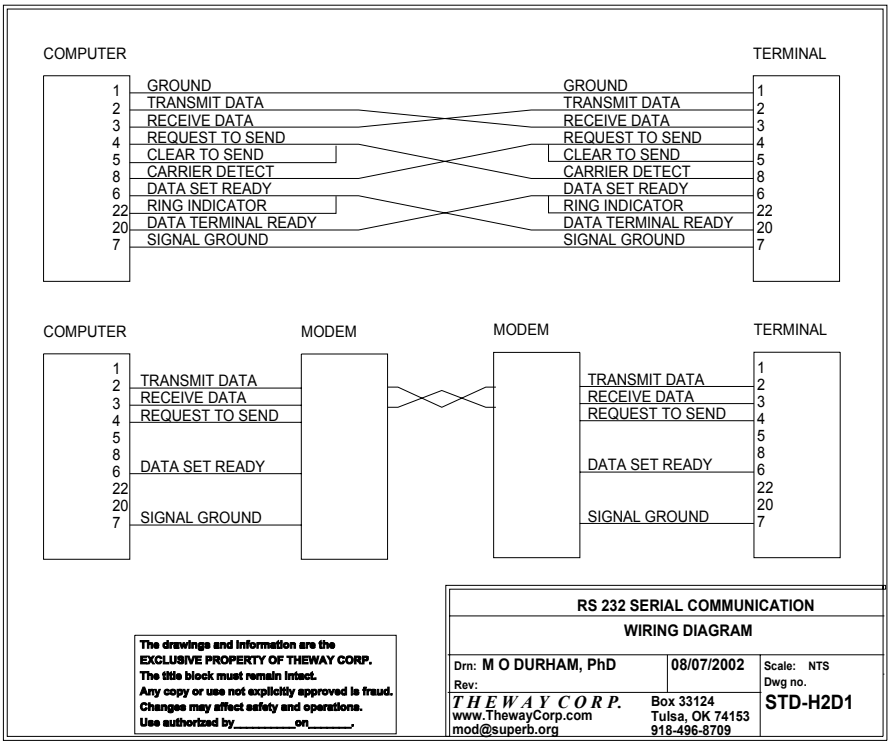
If using an RJ45 Ethernet adapter, consider the pins to correspond with the 9-pin db-9. Since RI is typically jumpered to DSR, leave it as the pin that is unconnected. For a jumper used as a loop back, only pins 2, 3, and 5 would be connected.

RS 232 pin outs

		25 to 9 straight		25 to 25 straight		25 to 25 null	
		25-pin	9-pin	25-pin	25-pin	25-pin	25-pin
Funct	**DB25**	**DB25P**	**DB9S**	**DB25P**	**DB25S**	**DB25P**	**DB25S**
CGround	1			1	1	1	1
TX	2	2	3	2	2	2	3
RX	3	3	2	3	3	3	2
RTS	4	4	7	4	4	4	5
CTS	5	5	8	5	5	5	4
DSR	6	6	6	6	6	6	20
GROUND	7	7	5	7	7	7	7
DCD	8	8	1	8	8		
DTR	20	20	4	20	20	20	6
RI	22	22	9	22	22		

25 to 3 jumper]			
	25-pin	3-pin	3-pin
Funct	**DB25P**	**DB25S**	**DB9S**
TX	2	2	3
RX	3	3	2
	4]		
	5]		
	6]		
	8]		
Gnd	7	7	5

Schematic


COMPUTER
TERMINAL
1 GROUND
2 TRANSMIT DATA
3 RECEIVE DATA
4 REQUEST TO SEND
5 CLEAR TO SEND
8 CARRIER DETECT
6 DATA SET READY
22 RING INDICATOR
20 DATA TERMINAL READY
7 SIGNAL GROUND
GROUND 1
TRANSMIT DATA 2
RECEIVE DATA 3
REQUEST TO SEND 4
CLEAR TO SEND 5
CARRIER DETECT 8
DATA SET READY 6
RING INDICATOR 22
DATA TERMINAL READY 20
SIGNAL GROUND 7
COMPUTER
MODEM
MODEM
TERMINAL
1
2 TRANSMIT DATA
3 RECEIVE DATA
4 REQUEST TO SEND
5
8
6 DATA SET READY
22
20
7 SIGNAL GROUND
TRANSMIT DATA 2
RECEIVE DATA 3
REQUEST TO SEND 4
5
8
DATA SET READY 6
22
20
SIGNAL GROUND 7
The drawings and information are the
EXCLUSIVE PROPERTY OF THEWAY CORP.
The title block must remain intact.
Any copy or use not explicitly approved is fraud.
Changes may affect safety and operations.
Use authorized by_____on_____.
RS 232 SERIAL COMMUNICATION
WIRING DIAGRAM
Drn: M O DURHAM, PhD
08/07/2002
Rev:
Scale: NTS
Dwg no.
STD-H2D1
THEWAY CORP.
www.ThewayCorp.com
mod@superb.org
Box 33124
Tulsa, OK 74153
918-496-8709

Development board pin outs

The RJ45 connector on the board supplies serial RS232 communications. The pins are separated by function.

Cable	**Cable**	**Board**	**Funct**	**To**	**Serial**	**Funct**
TIA568B	*Telco*	*RJ45*		*RJ11*	*db 9*	
Color	*Alt*	*Pin*		*Pin*	*Pin*	
W/Orange	Black	1			1	Jump
Orange	Yellow	2			6	Jump
W/Green	White	3	T1X	1	3	TXD
Blue	Red	4	R1X	2	2	RXD
W/Blue	Green	5	Ground	3	5	Ground
Green	Blue	6	R2x	4	4	nc
W/Brown	Brown	7			7	Jump
Brown	Orange	8			8	Jump

⇐ ⇑ ⇒

53

NETWORK CONNECTION

Thought
To err is human.
To forgive is not a computer function.
Popular quip

Network

Two fundamental networks are used with computer systems. These are the analog telephone or telecom system and digital Ethernet networking connections. Since termination to these networks are very similar, but different, it is prudent to have the pin-out diagrams.

The Telecommunications Industry Association (TIA) develops the standards for interconnection. The standards for this type wiring is TIA568. The telecom system uses the Universal System Ordering Codes (USOC). USOC was developed by the Bell system to provide standard access. It has been adopted by most current users of telecommunications equipment.

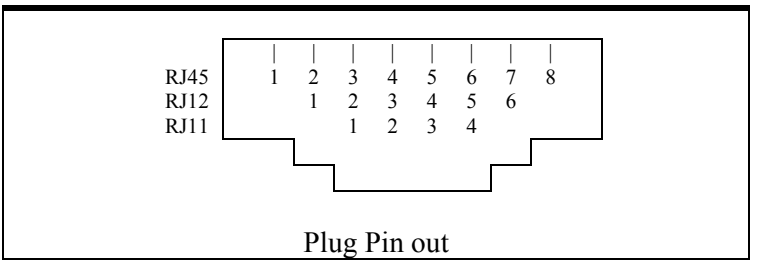
Wire for these networks should be enhanced category 5, commonly called Cat 5E or better. This cable contains four twisted pairs. To prevent cross talk, these have different twist rates. When properly installed and terminated, it permits communications up to 1000 megabits per second.

The diagrams show the color-coding and pin arrangements for these designs. A design designation can be used with any connector that has that number of pins or more. The common connector designation is based on the maximum number of pins.

Designation	Standard	Pairs	Pins
RJ48	USOC	4	
RJ45			8
RJ25	USOC	3	
RJ14	USOC	2	
RJ12			6
RJ11	USOC	1	4
568	TIA		

Plug designation

The pins are numbered when looking at a plug with the contacts facing up. Pin number 1 is on the left. The corresponding pin 1 is observed in the jack, when the contacts are on the top. The strip designation is the standard termination strip for networks.



Plug Pin out

The USOC standards start with Pair 1 on the middle two pins. Pair 2 is on the next two pins outward. Pair 3 is on the next two pins outward. Finally, Pair 4 is on the outermost two pins.

The TIA standards also start with Pair 1 on the middle two pins, 4&5. For TIA568A, Pair 2 is on the next two pins outward, 3 & 6, and Pair 3 is on the first two pins, 1 & 2. For TIA568B, Pair 3 is on the next two pins outward and Pair 2 is on the first two pins. Finally, Pair 4 is on the last two pins, 7 & 8.

The expansion connection from the microcontroller design board also uses an RJ45 connector. This is the same as the connection used for computer networking. As a result, the color-coding and pin-outs will be the same.

Diagram – digital network

Networks use an 8-pin connector called an RJ45. T568B is the most common connection pattern and is used by AT&T. T568A was developed to more closely match telephone pin-outs.

Only two pairs are used for computer networking. Therefore, the other two pairs are available for other digital service, such as digital telephone.

PAIR	NETWORK	T568B	T568A
	Standard⇒	*ATT*	*Tele*
	Connector⇒	*RJ45*	*RJ45*
	Function	*8-Pin*	*8 Pin*
White/blue		5	5
Blue/white		4	4
White/orange	TX+	1	3
Orange/white	TX-	2	6
White/green	RX+	3	1
Green/white	RX-	6	2
White/brown		7	7
Brown/white		8	8

Diagram – analog telephone

Standard telephones are analog with a ringing voltage exceeding 90 volts. Since computer networking is digital, these signals should not be mixed in a cable. Because of these considerations, it is standard practice to run one cable for digital networking and one cable for analog service.

PAIR	PHONE	USOC				
	Standard⇒	*ATT*				
	Connector⇒	*RJ45*	*RJ12*	*RJ11*	*Term*	
	Function	*8-Pin*	*6-Pin*	*4-Pin*	*Strip*	*Telco*
White/blue	Line 1 tip +	5	4	3	1	Green
Blue/white	Line 1 ring -	4	3	2	2	Red
White/orange	Line 2 tip +	3	2	1	3	Black
Orange/white	Line 2 ring -	6	5	4	4	Yellow
White/green	Line 3 tip +	2	1		5	White
Green/white	Line 3 ring -	7	6		6	Blue
White/brown	Line 4 tip +	8			7	Brown
Brown/white	Line 4 ring -	1			8	Orange

Diagram – analog audio

Since networking cable is often used for audio systems, the following format is one that has been used. At the time of writing, there was not broad standard for this use. Notice that the selection of the stereo pairs permits them to be placed on the same cable as the analog telephone.

PAIR	AUDIO				
	Function	*Strip*	*Jack*	*Speaker*	*Plug*
White/blue	Surround R +	1	Red low	Red	Green
Blue/white	Surround R -	2	Red low	Black	
White/orange	Surround L +	5	White low	Red	White
Orange/white	Surround L -	6	White low	Black	
White/green	Stereo R +	3	Red top	Red	Red
Green/white	Stereo R -	4	Red top	Black	
White/brown	Stereo L +	7	White top	Red	Black
Brown/white	Stereo L -	8	White top	Black	

54

PROGRAMMABLE LOGIC DEVICE

Thought
What you believe
is what happens.
MOD

It is just logic

A programmable logic device (PLD) is used to provide combinational logic and register based sequential circuits. The simple PLDs are rather inexpensive and powerful devices with significant flexibility. They can be programmed as firmware.

The earlier versions are referred to as programmable array logic (PAL). An enhanced version by one vendor is called generic array logic (GAL). The more powerful version is a programmable electrically erasable logic (PEEL) device which is a superset of the basic devices.

A common superset peel variation is a 22cv10a. The 22 indicates there are 22 I/O pins on the device. The 10 indicates that 10 of those can be used as input/output while the remaining 12 are input only. The package is a 24 pin skinny DIP.

These devices are programmed using software for the particular chip. This is often manufacturer specific. Many variations of the

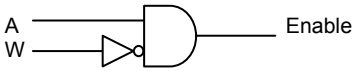
devices are reprogrammable either using ultraviolet or electrically erasable technology.

The speed is quiet fast. Typical propagation delays are less that 15 ns and as low as 5 ns - faster than standard TTL 7400 series logic.

Combinational logic

A peel is structured with input pins and input/output cell pins. Boolean logic is used to describe the relationships with the pins.

The process of developing a PLD program is very simple. It is a direct substitution for combinational logic. A very unpretentious circuit will effectively illustrate the process.



The Boolean equation is not complicated.

Enable = A and not W

When written in the software syntax, the equation is equally obvious.

Enable = A & !W

This is actually the equation for a latch enable when A is the address and W is the write not line.

From observing the programmable chip pin-out, inputs can be on pins 2 through 11 and on 13. Similarly, the output can be placed on pin 14 through 23. Pin one is a clock for register logic and an input for combinational logic.

First time user

Create a text file using a word processor or editor. Save it as unformatted text. Save with the extension '*.PSF' for the programming assembly software used in these examples.

The file begins with header information. The only required field is the filename. The device type must be specified. Then the file lists assignments for the input pins. Next comes the specification for the input/output cell pins. Finally, the equations are listed. Comments or descriptions can be placed anywhere within the program.

```
Title: 'File: ModPeel.psf'
PEEL22CV10A

A pin 2
W pin 3

IOC (14 'Enable' POS COM FEED_PIN)

EQUATIONS
Enable.com = A & !W
```

That is it. That is all and there is no more. An explanation of the IOC statements is given in the example file.

These programs are processed using ICT WinPlace software. It can be downloaded from their website.

The PLD software prefers a *.PSF extension. The comments are removed in a *.RED file. The output of the PLD software is assembled to a *.MAP file. This is translated to a format that is used to burn the PEEL. The industry standard is a JEDEC file with the extension *.JED.

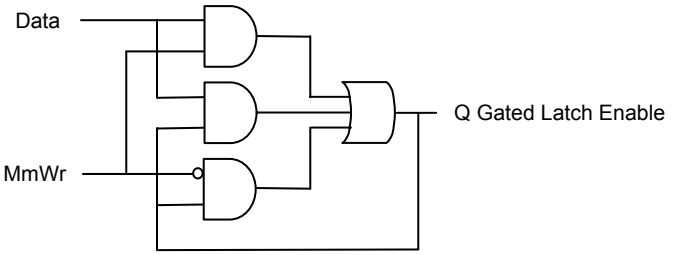
The JED file is loaded into a programmer / burner to change the links in the PLD.

Gated latch

The combinational logic circuit implementation is very straightforward. Much more complex circuits can be implemented with the PLD. These include internal flip/flops and registers. In addition the feedback type can be described as combinational, register, or latch. Nevertheless, the definitions will be only marginally more complex than the simple example.

Numerous projects use latches for external data expansion. One of the problems with latches is hazards. The inputs will not change at exactly the same time. Therefore, there will be a glitch in the output. This may not be critical in some circuits, but it can be catastrophic in timing systems.

Therefore, it is appropriate to address the design of a gated latch circuit.



The center AND gate is a redundant term to prevent a hazard or race. Therefore, if the software reduces the logic to minimum, the minimization feature will need to be off for this section of code.

```
@R-
GatedLatch = (MmWr & D)#(!MmWr & GL)#(D & GL)
@R+
```

OLMC and pin assigments

Each of the output pins is controlled by an output logic macro cell (OLMC). The OLMC allows the pin to act as input, combinational

output, register output, and input/output functions. There are 12 combinations of these variations. The decision is simply controlled by the equations used to program the pin.

Pin 1 is the clock for registered logic or an input for combinational. Pins 2 through 11 can be used for any input. For earlier devices, pin 13 is a common output enable for registers or an input for combinational. For 22cv10 and similar peel devices, each output has its own enable, so this pin is simply another input. Pins 14 through 23 are controlled by the OLMC.

With this much flexibility, it is necessary to define the input output cell (IOC) logic with a specification statement for each pin. The format follows.

IOC (pin-number 'name' polarity logic feedback)

Name is the label for that pin. Polarity can be pos(itive) or neg(ative). Logic can be com(binational) or reg(ister) based. Feedback can be from pin feed_pin (output pin) or feed_reg (register out).

The peel devices have a common synchronous preset (SP) and an asynchronous clear (AC) for the registers. These terms can be defined by a software equation.

SP := logic equation
AC := logic equation

The superset peel devices also have one buried combinational feedback, one buried combinational feedback with register, and one buried register feedback.

Registers

If the OLMC is used as a register, it can be implemented as a D flip/flop. By use of the registers, sequential logic such as counters and shift registers can be constructed.

If any of the outputs are configured as a register, then pin 1 is the clock input to the register.

Combinational logic uses a .COM extension on the label to set up the OLMC. All the previous examples have been this type sum of products.

```
Enable.COM = A & !W
```

Registered logic uses a .D extension, as a D flipflop, to set up the OLMC.

```
Q.D = D
```

Q will take the value of D following the rising edge of the clock on pin 1. Multiple registers can be configured to form counters. Each intermediate state will be an output pin.

For earlier versions, the registered outputs have a common output enable on pin 13. When the output enable is low, all the registered outputs are enabled. When the pin is high, the registered outputs are tri-stated (high impedance). Nevertheless, the registers can still be used internally as feedback to another equation.

Combination output enable _____

Combinational logic can define an output enable. It is limited to a single product term.

```
Result.OE = B & C
```

In actuality, it is simply an additional term to the descriptive equation.

Limitations

One of the limitations of the simple PLD such as the 22V10 is the limit on internal logic. Every term that is used must be either an input or an output. There are no internal states. As a result, intermediate states use an output pin and preclude its use for other connections. This limitation simply results from the device being a simple array that has connections 'burned'.

The enhanced versions, such as the peel, have limited internal feedback states that overcome some of the problems.

Nevertheless, it is excellent for direct implementation of combinational logic and simple gates.

Program: combinational logic (*.psf)

```
TITLE 'FILE: PeelII.psf'
DESIGNER 'Dr. Marcus O. Durham'
DATE '28 July 2004'

DESCRIPTION
  The description is optional. The title block
  information must be enclosed within single
  quotation marks.
  The device must be specified.
end_DESC;

PEEL22CV10
```

```
"
"        ___  ____
"       |   \/    |
"  CLK  {  1   24 }  Vcc
"  I1   {  2   23 }  F9
"  I2   {  3   22 }  F8
"  I3   {  4   21 }  F7
"  I4   {  5   20 }  F6
"  I5   {  6   19 }  F5
"  I6   {  7   18 }  F4
"  I7   {  8   17 }  F3
"  I8   {  9   16 }  F2
"  I9   { 10   15 }  F1
"  I10  { 11   14 }  F0
"  Gnd  { 12   13 }  I11
"       |_________|

"Pins
"Registered clock or input: 1
"Input: 2,3,4,5,6,7,8,9,10,11, 13
"Input or output logic macro cell (OLMC):
"  14,15,16,17,18,19,20,21,22,23
"
"-------
"Symbols are used for programming functions
" Quote = comments not printed
" ' = comments for header to be printed'
" ! = not
" & = AND
" # = OR
" $ = exclusive-or
" = is equation for combination logic
" ; = last character in an equation


"-------
"Application
"Memory can be changed from EPROM to SRAM
"by changing the chip & /OE jumper.
"PLD / PEEL program selects MemP1 and MemP27.

"EPROM: MemP1=Vpp, MemP27=A14 requires /OE=/PSEN
jumper
"SRAM:  MemP1=A14, MemP27=/WE, requires /OE=/RD or
Ground jumper
```

```
"MMIO:  uses A15
"SRAM:  !A15

"CHECK MemP27: IT CAN GO LOW IF !PSENn, IE RDn
ACTIVATES A14

"-------
"Address assignments.
"Because of the Don't Cares, the effective
"memory-mapped address can be 8xxn.

"Memory-mapped read or write is address A15 only.
"8000h
"Keypad latches are read row & write column
"8001h
"DisLE latch is selected for 7Segment & LCD.
"8002h
"
"Isp OE is select low on invert reset input.

"-------
"Memory switching latch was for old BIOS
"Kept for record purposes, original file is lost.
"CE = WRFF & ucA7 & ucD0 # !WRFF&CE
"OE = PSEN&RD # RD&!CE

"---------------
"PIN ASSIGNMENTS
"INPUTS

A00   pin 1
A01   pin 2
A02   pin 3
A14   pin 4
A15   pin 5
AD00  pin 6
Reset pin 7
"     pin 8
PSEN  pin 9
RDn   pin 10
WRn   pin 11
"           pin 13
```

```
"INPUT / OUTPUT CELLS
"The input/output cells require specification.
"The format is
"IOC (pin-number 'name' polarity logic feedback)
"Polarity can be pos(itive) or neg(ative).
"Logic can be com(binational) or reg(ister)
"Feedback can be from the feed_pin or feed_reg.

IOC (14 'MmRd' POS COM FEED_PIN)
IOC (15 'MmWr' POS COM FEED_PIN)
"IOC (16 'Spare' POS COM FEED_PIN)
IOC (17 'DisLE' POS COM FEED_PIN)
IOC (18 'KeyLE' POS COM FEED_PIN)
IOC (19 'KeyOEn' POS COM FEED_PIN)
IOC (20 'ISPOEn' POS COM FEED_PIN)
IOC (21 'MemP27' POS COM FEED_PIN)
IOC (22 'Oen' POS COM FEED_PIN)
IOC (23 'MemP1' POS COM FEED_PIN)


DEFINE
"Define variables do not have feedback.
MmAd = A15


EQUATIONS
"Equations define logic functions
"Register variables have .D extension for D FF.
"Combinational variables have .COM extension.

IspOEn.com   = !Reset;

MemP1.com  = A14 & (!A15 & (!WRn #!RDn));
MemP27.com = (!PSEN & A14) # (WRn & !A15);
Oen.com = PSEN # !(!RDn & !A15);

MmWr.com = A15 & !WRn;
MmRd.com = A15 & !RDn;

KeyOEn.com = !(MmRd& !A02& !A01& A00);
KeyLE.com = (MmWr& !A02& !A01& A00);

DisLE.com = (MMWr & !A02 & A01 & A00);
```

```
TEST_VECTORS
"Test vectors are used to verify the logic works.
"This is a partial truth table.
"The first line specifies input output variables.
"Next line are input output values.
"C is clock transition.
"0 is an input low, 1 is an input high
"L is output low, H is output high.
"X is don't care

(A00 A15 A00 Wrn MmWr DisLE -> KeyOEn)
C     0     X     0     L     x     X
C     0     X     1     L     x     X
C     1     X     0     H     x     X
C     1     X     1     L     x     X
```

Created Files

The redefine file (*.red) is an output of the program. It eliminates all the comments and extraneous information. It only contains the necessary information to execute the program. This is the title, device, pin assignments for input, input output cell description, and equations.

The assembled file (*.map) contains the pin node connections and all the product terms.

The JEDEC file (*.jed) is the information that is sent to burn the programmable logic device. This is a common language among many vendors.

55

CIRCUIT TIME & PHASE SHIFT

Thought
Educating process:
Encourage – instruction – example.
Dr. Jerry Falwell

Background

Every electrical device and circuit has a difference in time between the input and the output. This difference is called by a variety of names, depending on the specialty within the electrical field. Various names are time delay, propagation delay, time constant, phase shift, and wait state. A trigonometric interpretation is referred to as power factor or lead/lag angle.

The property characteristics of every material causes some resistance. In addition, bending the circuit path creates an inductance. In contrast, placing two conductors in proximity causes a capacitance.

The combination of resistance with inductance or with capacitance defines the delay in terms of a time constant. The time delay is measured in seconds, while the resistance is Ohms, inductance is Henries, and capacitance is Farads.

$$t_D = L / R$$

$$t_D = R * C$$

The combination of inductance and capacitance causes a frequency that may be called oscillation, ringing, vibration, revolutions, or cycling. The frequency is in cycles per second called Hertz.

$$f = 1 / 2\pi\sqrt{(L * C)}$$

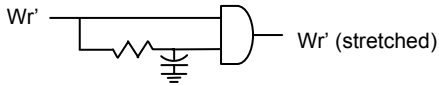
Integrated circuit devices have very complex internal connections. Therefore, it is virtually impossible for the user to determine the delay. The manufacturer will describe the lag between the input and the output as propagation delay.

Digital circuits have either an on or off state. The state will change based on the input. If the input branches have different time delays, then the digital state will not change at precisely the time expected.

This different response due to a delay can result in a hazard condition called a race. That is a situation when the response is not absolutely defined due to differences in time for the state changes on the input.

Delay

Occasionally it is desired to create a delay. For example if an enable pulse is just a little too short, it can be stretched with a AND gate. The input to one side is delayed.



Suppose a total delay of 37 ns is required. The 74AC08 has an eight ns propagation delay. The values for the resistor and capacitor are chosen to make the difference of 37 – 8 = 29 ns.

The resistor value must be large enough to keep Wr' from sinking too much current. A value in excess of 2 k Ohms is generally

acceptable, as was found in the fundamentals chapter. A capacitor of 10 pF will yield a time value in the nanosecond region.

The resistor value should correspond to the exact timing needed. For 29 ns, the value would be 2.9 k Ohms.

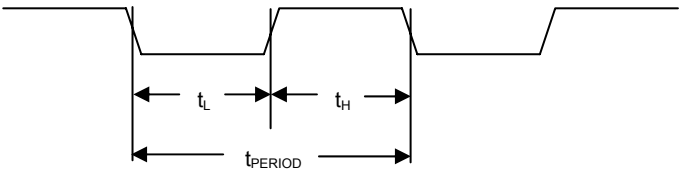
$$t_D = R * C = 2900 * 10\ e^{-12} = 29\ ns$$

The capacitor should have a matched negative temperature coefficient so the RC time constant will not drift with temperature. Voltage and temperature variations and noise can cause the above circuit to malfunction. A safety pad of 10 ns should be subtracted from the maximum stretched pulse width allowed to assure adequate delay.

Clock signals

Clock signals are critical to sequential circuits. Likewise, they establish the timing relationship between all the components associated with a computer.

Clocks are simply signals that cycle between low and high values. State variables change state only at the clock edge. State changes are level independent. The edge may be a rising edge on the transition from low to high. Alternately it can be a trailing or falling edge on the transition from high to low.



Interaction

Signals must be asserted in the correct sequence and for a long enough duration that the next circuit can respond. In computer

circuits, the clock timing is determined by an oscillator connected to the chip. The chip in turn, must perform multiple chores within its purview.

Similarly, the processor must interface to other devices such as memory and latches. All these perform at a certain rate. Since the computer is the system controller, it will generate the timing signals for the other devices.

A timing diagram is developed by the manufacturer of each integrated circuit chip to illustrate the duration of each signal. The computer primarily interfaces to external chips in one of three ways. Read code is used to access program memory. Read data is used to access input devices. Write data is used to access output devices.

The timing diagrams of the microprocessor for these three operations are shown on the next page.

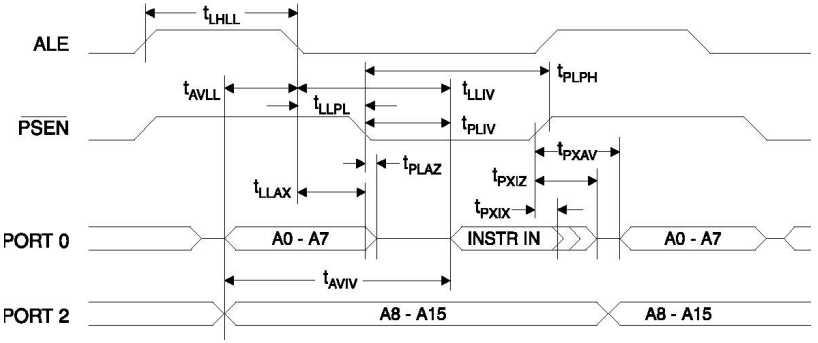
Consider the timing for addressing. The processor asserts the ALE (address latch enable) line high. This is usually connected to a 74573 latch enable. Then the processors asserts the address on port 0. After adequate set-up time, the processor asserts the ALE low. This causes the latch to trap the information that was on port 0.

Next the input control line is asserted low. This is the PSEN' or RD'. Then the data is brought in on port 0. For writing, the data is first placed on port 0, then the WR' is asserted low.

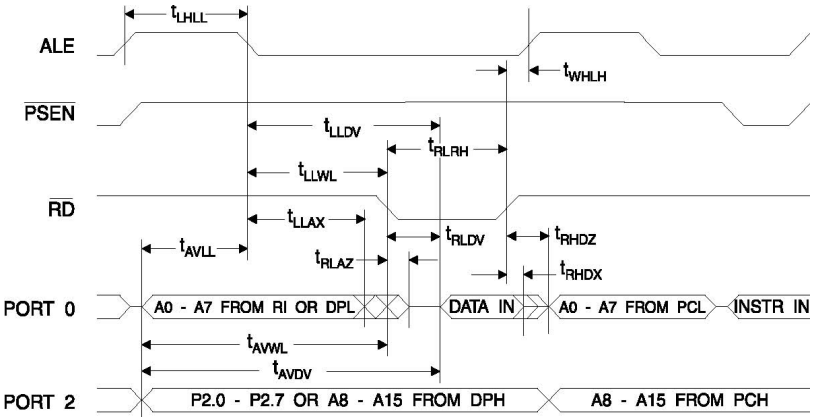
Any chips controlled by these lines, must respond within the time frame that the line is asserted. Otherwise, the data will not transfer between the chip and the processor.

⇐⇑⇒

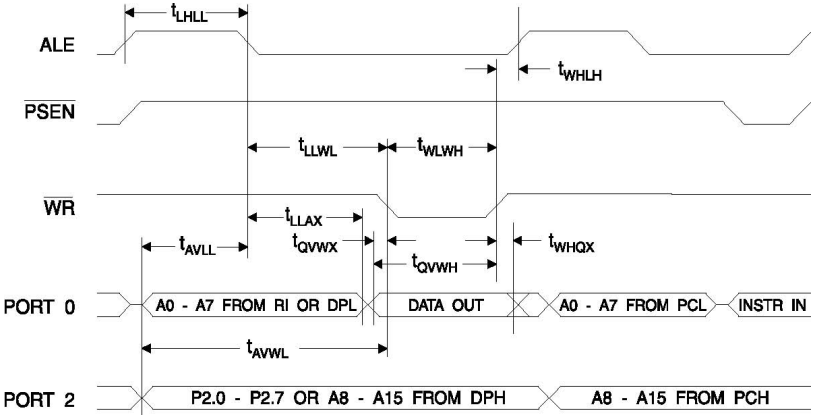
Ext program memory read cycle____



Ext data memory read cycle_____



Ext data memory write cycle



SECTION VII – DOCUMENTATION

⇐ ⇑ ⇒

56

EXTREME PROGRAMMING (XP) HARMONIZATION

Thought
I change my life,
when I change my mind.
Jim Stovall

General guidelines

Extreme programming is a team approach to project development. An early chapter addressed the requirements of the practice.

Each team creates code that looks exactly like all the other. In addition, variables are commonly defined so each team can access them.

Since a large number of items are used to interface with a microprocessor, many variables are commonly used. This segment of program code illustrates the common values that are often required. Therefore, it is prudent to avoid overlaying or changing these values.

User variables can be placed in the upper locations of the internal RAM between the defined assignments and where the stack has been moved.

Program specifics

```
;-----------------------------
;Program: MOD-SYSTEM.asm
;Update:  15 August 2002
;Initial: 17 October 1991
;
;By:      Dr. Marcus O. Durham, PhD, PE
;         Tulsa, OK, USA
;         mod@superb.org
;         www.ThewayCorp.com
;Copyright (c)1991 - 2002.  All rights reserved
;
;Purpose:
;  A set of routines are provided to perform the
;  basic functions of a computer.
;
;  These include ports, timers, serial RS232,
;  mmio, keypad, seven-segment display,
;  liquid crystal display, analog/digital convert,
;  and math.
;
;Processor: 8031 family
;PROM:      4k (1000H) onboard
;Crystal:   11.059 MHz
;Baud:      9600
;Handshake: not used at this speed
;
;  There are three types of processing techniques
;  that are used.
;    MAIN      - is the normal loop processing
;    INTERRUPTS- occur based on events, such as
;                regular time intervals.
;    BUSY      - is called for random tasks that
;                can be processed during the delay
;                in other tasks. Keypad scan
;                is a prime example.
;
;-----------------------------
;ASSEMBLERS
;-----------------------------
;
;  Several assemblers are available for the 8051.
;  The major differences are in the syntax of
```

```
;  the directives.  The three that I have used are
;  compared.
;
;
;  TASM.EXE is a table lookup.
;  tasm -51 -p -l Bios11v2.asm
;         .org  0050h          ;Next inst @ 0050H
;Value    .equ  12h            ;predefined value
;Table    .byte 34h            ;define code byte
;         .text "123"          ;define ASCII
;         .end                 ;last thing
;
;
;  A51.EXE is student version of PseudoCodes.
;  A51 -s Bios11v2.asm
;         .org  0050h          ;Next inst @ 0050H
;         .equ  label,12h
;Table:   .db   34h            ;define code byte
;         .db   "123"          ;define ASCII
;
;
;  ASM51.EXE is Intel assembler with library.
;  ASM51 Bios11v2.asm
;         org   0050h          ;Next inst @ 0050H
;Value    equ   12h            ;predefined value
;Table:   db    34h, '123'     ;define code storage
;         end                  ;last thing
;
;
;-----------------------------
;FORMAT
;-----------------------------
;
;  The format of the code lines is shown. This
;  gives space for opcodes in the *.lst files
;  to fit on one line of display & print.
;
;LABEL:   MNEMO REGISTERS      ;COMMENTS
;Columns  1     1      2      3                 4|
;         1     7      4      1                 9
;
;-----------------------------
;EXTERNAL -VS- INTERNAL PROM
;-----------------------------
```

```
;
;  8051 has external memory on prom
;  8951 has 4K (07ffH) internal flash eprom
;  898252 has 8K flash & 2K eeprom
;
;  To program 8051, burn an external prom/eeprom.
;  To program low memory of 8951, burn flash
;  To program 898252, program using SPI to port 1.
;    P15 = MOSI
;    P16 = MISO
;    P17 = SCLK
;    Ground = common
;    Use program AEC_ISP.exe to download from PC
;
;  When the uP is reset, it looks at EA'.
;  If EA'=0, then external eprom is read.
;
;  IF EA'=1, then internal eprom is read.
;  Internal eprom is executed until it hits end.
;  Then external eprom is used, even if EA'=1.
;
;  EA' is pin 31.
;  To begin program from prom, strap EA' to 0.
;  To use the internal flash, strap EA' to 1.
;
;-----------------------------
;
;  The 8951 has 4K (0FFFh) internal flash prom.
;  Make the last internal executable line be 0FFFh
;  The next instruction will be 1000h on external
;  prom/sram.
;  So, org the external program to 1000H.
;  Initially make program organize to hi memory.
;         org    1000H        ;external
;
;  If it is moved to low memory (onboard flash),
;  the address must be known so the hi memory can
;  call the routine.
;  For example in low memory
;         org   0250H         ;Next inst @ 0250H
;SCRNDATA:mov                 ;low mem routine
;
;  The high memory PROM/SRAM overlays the same
;  space.  It is just not available to the uC.
```

```
;  However, this little feature lets us place
;  an org directive to the assembler at the same
;  location. It will never be executed. Program
;  Control will go to the low memory
;         org   0250H         ;pseudo location
;SCRNDATA:nop                 ;just for assembler


;################################################
;
;                      ASSIGNMENTS
;
;################################################
;
;  Assignments include
;  1. constants for symbols and sizes
;  2. constants for ext mmio addresses
;  3. variables assigned to internal RAM locations


;++++++++++++++++++++++++++++++++++++++++++++++++
;CONSTANTS
;------------------------------

                        ;SYMBOLS
Scrl      equ   42d     ;scroll key,"*",2AH
Entr      equ   35d     ;enter key,"#",23H
None      equ   0FFH    ;blank key

Flash     equ   254     ;black cursor for flash
Colon     equ   3AH     ;colon
Zero      equ   30H     ;zero character
Slash     equ   2FH     ;slash
DecPt     equ   2EH     ;decimal point
Minus     equ   2DH     ;minus sign
CRet      equ   13d     ;carriage return

DigX      equ   8       ;number of decimal digits
MathX     equ   4       ;number of hexadecimal byte

;------------------------------

                        ;EXTERNAL ADDRESS
FilAdH    equ   00      ;address file base for DPH
```

```
FilAdL    equ   50H    ;for DPL, base addr= 0050H

AdKey     equ   8001H  ;MMIO addr for Keypad latch
AdSeven   equ   8002H  ;MMIO addr for 7-segment
AdInst    equ   8003H  ;MMIO addr for LCD instruct
AdData    equ   8004H  ;MMIO addr for LCD data
AdRead    equ   8005H  ;MMIO addr for LCD read


;++++++++++++++++++++++++++++++++++++++++++++++++
;STACK
;------------------------------
;
;  The stack is moved up high to prevent it from
;  overwriting data. Stack counts up from the next
;  location past the stack pointer base.
;
;  The default stack location is 07H. It will over
;  write the registers.
;
;  Two bytes are used by each interrupt and each
;  subroutine call. So enough stack space must be
;  reserved for nested loops.
;
;                       ;STACK RESERVED
;         equ   5FH     ;Move stack to 60H & above
;
;++++++++++++++++++++++++++++++++++++++++++++++++
;USER VARIABLES
;------------------------------
;
;  All the RAM locations I used are shown below.
;  This is so the user will not overwrite them.
;
;  Place any user variables within this range.
;
;------------------------------
                        ;USER DEFINED VARIABLES
                        ;place user variable here
;         equ   5Fh     ;
                        ;and everything in between


SbcTeam2  equ   40h     ;count team 1
```

```
SbcTeam1  equ   3Fh    ;count team 1
TimMin    equ   3EH    ;time minutes counter
TimSec    equ   3DH    ;time seconds counter
TimPer    equ   3CH    ;time periods counter


;++++++++++++++++++++++++++++++++++++++++++++++++
;DEFINED VARIABLES
;-----------------------------
;
;  Defined variables are used by many routines.
;  These are fixed locations that should not be
;  changed since the order is critical to SqROOT.
;
;  Note that the math variables ArgX, RemX, FraX
;  locations are shared with TmpX and GapX
;  Math procedures are seldom done. When they are,
;  the process is completed before other functions
;  are attempted. Therefore, there is no conflict.

                      ;Gap is scratch, display
                      ;Tmp is temporary register
                      ;Hex =32 bit working reg
                      ;ORDER CRITICAL TO SQ ROOT

FraD      equ   3BH    ;4 bytes used in divide
FraC      equ   3AH    ;same space as Gap upper
FraB      equ   39H
FraA      equ   38H

RemD      equ   33H    ;4 digit remain
RemC      equ   32H    ;same space as Tmp upper by
RemB      equ   31H
RemA      equ   30H

SizeT     equ   33H    ;same space as Tmp upper by
ArgG      equ   32H    ;used in multiply
ArgT      equ   31H
ArgH      equ   30H

GapH      equ   3BH    ;8 digit
GapG      equ   3AH    ;BCD digits high byte
GapF      equ   39H    ;& intermediate value hold
GapE      equ   38H
```

```
GapD      equ   37H
GapC      equ   36H
GapB      equ   35H
GapA      equ   34H

TmpH      equ   33H    ;8 byte
TmpG      equ   32H
TmpF      equ   31H
TmpE      equ   30H    ;Delay routine
TmpD      equ   2FH
TmpC      equ   2EH    ;RAM Init Valu, Init ea pas
TmpB      equ   2DH    ;RAM Test Valu, cpl ea addr
TmpA      equ   2CH    ;math low byte

HexD      equ   2BH    ;double word
HexC      equ   2AH
HexB      equ   29H    ;sixteen bit register
HexA      equ   28H    ;low byte of variable

;------------------------------
;                      ;BITS LOCATION
;         equ   27H    ;last byte reserved for bit
;         equ   20H    ;first byte for bit usage
;
;------------------------------
                       ;SCREENS
ScnFil    equ    1FH   ;File where data is stored
ScnMem    equ    1EH   ;bytes of memory required
ScnGet    equ    1DH   ;cursor loc present
ScnCur    equ    1CH   ;cursor loc desired & temp
ScnDis    equ    1BH   ;bytes of display space req
ScnTyp    equ    1AH   ;type data, if #, it is dp

;------------------------------
                       ;BANK3
QikB      equ   19H    ;Interrupt HEX value, msb
QikA      equ   18H    ;Interrupt HEX value, lsb

;------------------------------
                       ;KEYS
KeyCol    equ   17H    ;present column number
KeyBit    equ   16H    ;byte moves a bit w/ column
KeyRow    equ   15H    ;row number pushed
KeyMul    equ   14H    ;multiple key count
```

```
CharK     equ   13H    ;key input character
CharD     equ   12H    ;debounced

;-----------------------------
                       ;BANK2
;R1       equ   11H    ;
;R0       equ   10H    ;

;-----------------------------
                       ;CHARACTERS, HOLD, COUNT
CharP     equ   0FH    ;undebounced previous input
CharL     equ   0EH    ;character to LCD & Serial
MenuL     equ   0DH    ;loop @MENUS
LatCr     equ   0CH    ;control latch bits status
ScnDph    equ   0BH    ;hold DPH
ScnDpl    equ   0AH    ;hold DPL

;-----------------------------
                       ;BANK1
;R1       equ   09H    ;SRAM address, interrupts
;R0       equ   08H    ;SRAM address, background

;-----------------------------
                       ;BANK 0, USE W/ MATH & LOOP
LoopC     equ   07H    ;loop counter

Size      equ   06H    ;multiply, divide, sqrt, GP
;R5       equ   05H    ;carry in multiply, GP
;R4       equ   04H    ;carry in mul,GP

;R3       equ   03H    ;loop 2, math
;R2       equ   02H    ;loop, destination size,mat
;R1       equ   01H    ;source indirect addr,math
;R0       equ   00H    ;destination indirect addr


;++++++++++++++++++++++++++++++++++++++++++++++++
;BITS ASSIGNMENTS
;-----------------------------
;
                       ;AT RAM byte 20H
FgC       bit   01H    ;temporary for C, etx
FgKeyH    bit   00H    ;flag key held down
```

```
;++++++++++++++++++++++++++++++++++++++++++++++++
;SPECIAL FUNCTION REGISTERS
;------------------------------
;
;  SFR are listed for information. With limited
;  assemblers, remove the ; since the address must
;  be defined.
;
;                        ;SFR DEFINED
;B        equ   0F0H    ;second math register
;Acc      equ   0E0H    ;accumulator
;PSW      equ   0D0H    ;processor status word
;IP       equ   0B8H    ;interrupt priority
;P3       equ   0B0H    ;port 3
;IE       equ   0A8H    ;interrupt enable
;P2       equ   0A0H    ;port 2
;SBUF     equ   99H     ;serial buffer
;SCON     equ   98H     ;serial control
;P1       equ   90H     ;port 1
;TH1      equ   8DH     ;timer hi byte 1
;TH0      equ   8CH     ;timer hi byte 0
;TL1      equ   8BH     ;timer lo byte 1
;TL0      equ   8AH     ;timer lo byte 0
;TMOD     equ   89H     ;timer mode
;TCON     equ   88H     ;timer control
 PCON     equ   87H     ;power control reg
;DPH      equ   83H     ;data pointer hi
;DPL      equ   82H     ;data pointer low
;SP       equ   81H     ;stack pointer
;P0       equ   80H     ;port 0

;------------------------------
                        ;bit IN SFR
Acc7     equ   0E7H    ;Accumulator bit 7
Acc6     equ   0E6H    ;accum bit 6
BankH    equ   0D4H    ;PSW register bank high bit
BankL    equ   0D3H    ;PSW register bank low bit
Over     equ   0D2H    ;PSW bit 2, overflow on add
;P        equ   0D0H    ;parity, PSW
;TI       equ   99H     ;transmit is complete
;RI       equ   98H     ;receive is complete
;TR1      equ   8EH     ;timer 1 start
;TF0      equ   8DH     ;timer 0 overflow
```

```
;TR0       equ   8CH     ;timer 0 start

;-----------------------------
                         ;PORT USE
P33        equ   0B3H    ;Port33 Hi=PC Busy,uP no xm
                         ;Not always used in prog
                         ;to use, setb P33 for input
P35        equ   0B5H    ;Port35 Hi=uP BUSY,stop ser
                         ;xmit to uP in download mod

                         ;SPI ADC LINES
                         ;move when know real locate
SpMosi     equ   0B2H    ;SPI data out, INT0, P32
SpMiso     equ   0B4H    ;SPI data in, T0, P34
SpCLk      equ   0B5H    ;SPI serial clock, T1, P35
SpCS       equ   4       ;SPI ADC CS Con latch D4
SpConv     equ   5       ;SPI ADC Cnv Con latch D5


;################################################
;
;                      PROGRAM
;
;################################################
           org   00H
START:
;-----------------------------
;  When processor is reset, program control comes
;  here. Jump to the first executable address
;  after all interrupts reserved locations.

           ljmp  INITIAL


;************************************************
;
;                      INTERRUPTS
;
;************************************************
;
;  Interrupt set-up is discussed in a later
;  section. It is associated with timer/counters.
;
;-----------------------------
```

```
;INTERRUPT-External 0
;-----------------------------

          org   03H
          reti

;-----------------------------
;INTERRUPT-Timer 0
;-----------------------------
;  The procedure provides direction when timer
;  completes count.

          org   0BH
          ljmp  TIMECAL       ;interrupt processor

;-----------------------------
;INTERRUPT-External 1
;-----------------------------

          org   13H
          reti

;-----------------------------
;INTERRUPT-Timer 1
;-----------------------------
;  Timer 1 is used for serial. So this procedure
;  will never be executed.

          org   1BH

;-----------------------------
;INTERRUPT-Serial
;-----------------------------
;  This is not used. The bits are polled in
;  various other routines.

          org   23H


;-----------------------------
;COPYRIGHT
;-----------------------------
;  Ownership of program is stored in memory.
          org   0033H ;Address past vectors
```

```
          db    'Copyright (C) 2004'
          db    'Marcus O. Durham, PhD, PE'
          db    'Tulsa, OK  USA'

;************************************************
;
;                     INITIAL & MAIN
;
;************************************************
          org   0080h         ;Addres past reserve
;         org   2000h         ;Address if external
;-----------------------------
INITIAL:
;-----------------------------
;  Setup the initial conditions.

;-----------------------------
          end
```

57

DOCUMENTATION

Thought
The job is not finished,
until the paperwork is done.
Popular quip

Report

A report should be prepared that is a complete documentation of the project. The report will be a technical manuscript that you provide to your boss or customer. The intent is to provide all the information so the project can be duplicated or advanced. Your next contract depends on the quality of your presentation.

All the information should be typed and professionally printed on a laser-quality printer.

1. A *Cover Sheet* should include project name, course number, your name, date the project is due, and date the project is submitted. When the project is completed, have the page initialed and dated by a supervisor.

2. A *Table of Contents* should list all the sections. This can be on the same page as the cover sheet.

3. An *Abstract* should give the project definition. It should be adequate that an unfamiliar person can read it and determine

what the project is about. The length should be less than 250 words.

4. An *Executive Summary* should give the conclusions and solutions statement. There should be a brief statement of problems. One sentence should discuss future applications and improvements on what you did. The length should be less than 400 words. The abstract and executive summary should tell the complete story. Keep It Simple, Sam!

5. A *Hardware Block Diagram* will illustrate the major building blocks of the equipment.

6. A *Software Block Diagram* will be a flowchart of the major components of the program.

7. A *Schematic* will show the interconnection of all the hardware components. The file should be in a format that is importable for display by a web browser.

8. A *Software Listing* of all programs with comments is a necessary component. This should be an editable executable version of the software.

9. A *Spreadsheet* contains the cost items. This is a list of parts, which contains the following information. A total should be given as a summation of all the costs.

Part	Device	Package	Where Used	Vendor	Number	Qty	Cost

10. A *Time* sheet should breakout the time invested in each phase of the project. As a minimum this will be planning, software, hardware, trouble-shooting, and report preparation.

11. A *Support Equipment* list will include all extraneous equipment. The list will include a personal computer, applications software packages, EPROM burner, power supply, cables, and any other items required.

It is advisable to keep a copy of the report for your records. Much of the information will be used on succeeding projects. This is called file-drawer engineering.

Computer aided design

In today's marketplace, all circuit diagrams should be made with a computer aided design tool. This results in professional schematics and lends itself to changes. If a printed circuit board design program is used, all the following comments will be included.

Occasionally, it is desirable to make a sketch in a word processing or other program. These may be beneficial for enhancing a particular part of the circuit or for illustration purposes. Then the following suggestions are very appropriate.

There are many ways to draw a circuit diagram. While there is no universal standard for drawing schematics, a standard approach for diagrams is a requirement for consistent results.

The first and most obvious practice is to draw a block for each chip. Label the chip number and its function (e.g. uP, ROM, RAM, etc.) inside the block. Labeling the chip's function is especially important when using unusual chips. However, there is one exception to the block labeling; discrete gate diagrams are discussed below.

Every pin of each chip should be labeled with its function on the inside of the block and its pin number on the outside of the block. Some groups of pins have a similar function such as address-bus pins. These can be labeled as a group instead of drawing a pin-for-pin representation. Make sure that the pin numbers listed on the outside of the block correspond appropriately to the pin descriptions on the inside of the block. For example, A0 corresponds to pin 39.

Discrete gates (drivers, flip/flops, etc.) should not be drawn in block form. The number of interconnections to various parts of the circuit would be too cumbersome. Instead, draw the gate in the circuit diagram where it is needed. Draw the part number inside the gate.

Include the pin function if appropriate. Draw pin numbers outside the gate.

Power supply connections for the gate need not be specified unless they are different from the standard. In most cases, power is applied to the upper right corner pin and ground is applied to the lower left corner pin. Consider a 14-pin chip. The upper right pin is number 14 and has +5 V while the lower left pin is number 7 and has ground.

Interconnections between pins on different chips can be drawn in whatever way works best. Avoid making the lines too convoluted. Lines that cross are not considered to connect unless there is a dot at the intersection. This procedure does not require humps every time one line crosses another.

Lines of similar function, such as address lines and data lines, need not be drawn separately. It is common practice to use a single bus line as long as the line is wider than normal. Draw a slash through the bus line with the number of lines on the bus. If several, but not all, lines branch off from the bus, label the branches and make a new bus line if necessary.

Unused pins on any chips, except discrete gates, must still be included in the circuit diagram. Draw a short 3-4 mm line from the pin, but do not connect the wire to anything.

If the pin is tied to ground or to the power supply, draw in a short connection and label it ground or +5 V. Do not use bus lines for ground and power supply connections. This would result in too many lines on the circuit diagram.

⇐ ⇑ ⇒

END

Thought
The end or top of one phase
is simply the beginning or bottom of the next.
Valedictorian speech by K. D. Durham

⇐ ⇑ ⇒

AUTHOR

Dr. Marcus O. Durham brings very diverse experience to his writing and lectures. He is an engineer, who owns THEWAY Corp., an international consulting practice. He is an entrepreneur on the internet with Advanced Business Technology, Inc. He is a Professor at The University of Tulsa. He is formerly Dean of Graduate Studies and Professor at Southwest Biblical Seminary.

He is a commercial pilot who flies his own plane, is a ham radio Extra Class operator, and has a commercial radiotelephone license. He is a registered Professional Engineer and a state licensed electrical contractor.

Professional recognition includes Fellow of Institute of Electrical and Electronic Engineers, Diplomate of American College of Forensic Examiners, Certified Homeland Security Level III (highest), and Kaufmann Medal by IEEE.

Dr. Durham is acclaimed in *Who's Who of American Teachers* (multiple editions), *National Registry of Who's Who*, *Who's Who of the Petroleum and Chemical Industry*, *Who's Who in Executives and Professionals*, *Who's Who Registry of Business Leaders*, Congressional Businessman of the Year, and Presidential Committee Medal of Honor. Honorary recognition includes Phi Kappa Phi, Tau Beta Pi, and Eta Kappa Nu.

He has published over 100 papers and articles and has authored six books. He has developed a broad spectrum of projects for both U.S. and international companies. He has traveled in over 22 countries and has mentoring relationships with students in 15 additional nations.

Dr. Durham received the B.S. from Louisiana Tech University, the M.E. from The University of Tulsa, and the Ph.D. from Oklahoma State University. He has other studies with numerous educational and scholarly organizations.

The author can be contacted at the publisher.

⇐⇑⇒