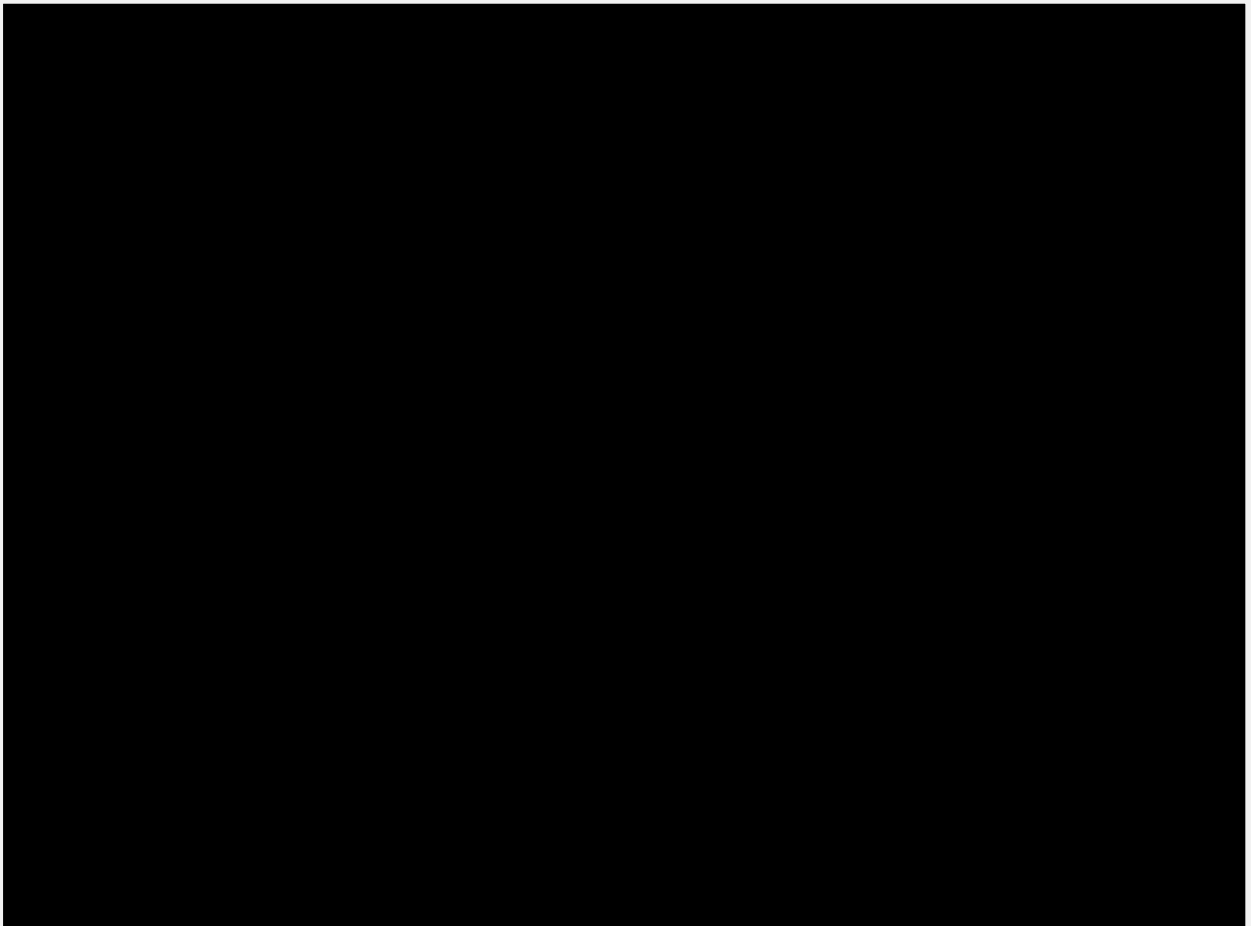# Laboratory Introduction to

Embedded Control

lab manual v15.3

Interrupts on the C8051

Interrupt Source	Interrupt Vector	Priority Order	Interrupt Source	Interrupt Vector	Priority Order
Reset	0x0000	Top	Comparator 0 Falling Edge	0x0053	10
External Interrupt 0 (/INT0)	0x0003	0	Comparator 0 Rising Edge	0x005B	11
Timer 0 Overflow	0x000B	1	Comparator 1 Falling Edge	0x0063	12
External Interrupt 1 (/INT1)	0x0013	2	Comparator 1 Rising Edge	0x006B	13
Timer 1 Overflow	0x001B	3	Timer 3 Overflow	0x0073	14
UART0	0x0023	4	ADC0 End of Conversion	0x007B	15
Timer 2 Overflow (or RXF2)	0x002B	5	Timer 4 Overflow	0x0083	16
Serial Peripheral Interface	0x0033	6	ADC1 End of Conversion	0x008B	17
SMBus Interface	0x003B	7	External Interrupt 6	0x0093	18
ADC0 Window Comparator	0x0043	8	External Interrupt 7	0x009B	19
Programmable Counter Array	0x004B	9	UART1	0x00A3	20
			External Crystal OSC Ready	0x00AB	21

Interrupts and Priority Order

7	6	5	4	3	2	1	0
EA	IEGF0	ET2	ES0	ET1	EX1	**ET0**	EX0

IE: Interrupt Enable Register (Bit Addressable)

7	6	5	4	3	2	1	0
ECP1R	ICP1F	ECP0R	ECP0F	**EPCA0**	EWADC0	ESMB0	ESPIO

EIE1: Extended Interrupt Enable 1 Register

7	6	5	4	3	2	1	0
TF1	TR1	**TF0**	TR0	IE1	IT1	IE0	IT0

TCON: Timer Control Register (Bit Addressable)

Table of Contents

Chapter 1 - Introduction

Microprocessor-controlled systems have become ubiquitous in our day-to-day lives. In addition to the microprocessor's familiar role as the central processing unit (CPU) in a computer, they are well suited to serve as dedicated controllers for various applications. For these dedicated applications it proves cost effective to add hardware features directly into the microprocessor silicon wafer. These typically include timers, analog-to-digital converters, digital-to-analog converters, on chip memory, and serial bus interfaces. With such addition hardware, the chips are called microcontrollers.

From the time you woke up this morning, you've probably interacted with at least a dozen different systems that utilize a microcontroller. As an illustration, most of you have a digital alarm clock that may have a small microcontroller in it. If you switched on your TV or stereo, then you've probably interacted with another microcontroller. If you cooked your breakfast or lunch in a microwave, then you've communicated your cooking instructions to its microcontroller. If you chose instead to eat in the cafeteria and used your *RAD* card, then a microcontroller in the card reader of the cash register processed your transaction. If you drove to class, your car may have fuel control, emission control, anti-lock brakes, air bags...etc. Each of these typically has a dedicated microcontroller.

Uses of Embedded Control

The term *embedded* refers to the fact that the microcontroller is an *integral part* of the unit that it controls. Examples of systems that typically utilize an embedded microcontroller include:

Consumer products

Stopwatches, cell phones, mp3 players, alarm clocks, microwave ovens, dishwashers, washing machines, printers, computer cards, telephones, FAX machines, photocopiers, calculators, audio and video entertainment systems, camcorders, cameras, burglar alarms, home thermostats, and countless electronic games and toys.

Embedded systems in automotive and related products

Anti-lock brakes, engine management, climate control, cruise control, automatic transmission, stereo systems, cellular phones, filling station gas pumps.

Medical and related systems

Cardiac defibrillators, cardiac monitors, sudden infant death syndrome (SIDS) monitors, blood gas analyzers, breathalyzers, digital thermometers.

Commercial products and applications

Barcode scanners, anti-theft systems in retail stores, vending machines, intelligent traffic lights, remote data entry systems.

Military applications

Guided missiles, smart bombs, avionics, communication equipment, global positioning satellite (GPS) receivers, UAVs.

A number of different microcontrollers have been specifically designed for embedded control applications. In this lab, you will be using a popular 8-bit microcontroller *specifically* designed for embedded control, the Silicon Labs C8051F020. Considering its size, the C8051 is both powerful and flexible and has proven ideal for many demanding consumer applications.

The Embedded Control Course

The purpose of this course is to introduce you to the development of an integrated embedded control system. Through this experience, you will gain an understanding of how such systems are designed and integrated into a typical consumer product. Whether your future job requires you to develop cutting-edge sound equipment, design fuel-injection systems, or monitor the ozone layer, the chances are very high that you will be involved with embedded control to some degree. Therefore, with microcontroller-controlled systems playing an increasingly vital role in the world we live in, this course can give you, no matter what your specialty, an enormous advantage over those engineers who have never designed an embedded control system.

The Target Systems

There are three different, but closely related, objectives for the semester. At the end of the semester you will have developed controllers for three types of systems, each using sensors and actuators. Concepts learned to control the *Smart Car* in Lab 4 will be extended in two ways to control the car on an incline and control the *Gondola* on a turntable. Both the *Gondola* and the *Smart Car* have an on-board microcontroller, an electronic compass and an ultrasonic ranger. The compass will control the steering servo on the car or the steering fan on the gondola. The ranger will control various performance objectives on the car or on the gondola/turntable. The car will prove useful in the development of the *control algorithms*.

The students' goal with the car is to build the microcontroller interface circuitry and develop the software, which will enable the car to follow the magnetic field in the Core Studio, along the floor of the lab, as accurately as possible while monitoring the ranger. The distance reading of the ultrasonic ranger is used to react to environmental changes. The code and knowledge gained will

be used to port that code to the gondola and implement control code on the turntable. The development of the system is spread over several lab exercises, each moving to accomplish various tasks.

The steering mechanism of the car consists of a servomotor that is connected to the front wheels through push rods. A closed-loop steering control subsystem is developed to maintain the car on the track by periodically reading the signals from the magnetic compass and determining corrective steering action. On the gondola, it is much the same except the signals will control the speed and direction of a fan mounted in the blimp tail.

A DC drive motor mounted at the back of the car powers the car. A separate control subsystem is developed so that the distance reading of the ultrasonic ranger controls the power to the drive motor and/or the steering servo. The code will then be modified to become closed-loop control of the gondola by controlling the power to thrust fans based on the difference between the desired distance and the actual distance.

Thus the basic system consists of an electronic compass and steering control subsystem to maintain the orientation of the car or gondola modified by the ultrasonic ranger that may also control the car's movement or modify the gondola's orientation. Additionally an accelerometer will be introduced to measure the car's pitch and roll on an incline and take action based on these measurements.

As is standard engineering practice for a system as complex as the smart car and gondola, after a system-level design, you will design, construct, and test *one subsystem at a time.* For the hardware elements of most of these subsystems, we will guide you toward a solution that you are free to use in other control system designs. However, if you think you have a better solution, or you want to try something different, you are encouraged to do so. RPI students are known to be exceptionally creative, and we encourage that creativity. However, keep in mind limitations of time and the availability of materials. Your goal is to produce the best product you can by utilizing the tools and materials available to you, within the time constraints of a one-semester course. And there is an additional constraint - the gondola's angular momentum must be taken into consideration in the design of the control system.

Lab Manual and Assignments

The lab assignments will introduce you to several topic areas of embedded microcontroller control, and will then help you to integrate what you learn at each stage into a working control system. Since the goal of the final lab exercise is to integrate what you have learned and actually built in all of the previous lab exercises, it is essential that you keep up with the work. The specific lab assignments listed in *Appendix E- Lab Assignments* are listed on the LMS web page. These include the necessary reading assignments, both in this manual and in the other sources, as well as

relevant interactive tutorials since much of this information may be new to you. It will be necessary for you to read the assigned material before the start of each laboratory session, and it would be enormously advantageous to work through the tutorials in advance of the lab session. Moreover, the required reading material will be considered *fair game* on exams. Throughout the course, homework will also be assigned along with short in-class exercises to assist you in preparing for the lab exercises.

Future of Embedded Control

The applications of embedded control are limited only by one's imagination and there is a growing opportunity for engineers to utilize embedded control to solve a variety of important problems of today and in the future. In the future, you will continue to see embedded control appear in almost every household item. Some products may actually become feasible because of an inexpensive embedded control system. For this reason, engineers from all areas and with different specialties will at some time in their careers be involved with the development of systems that utilize embedded control.

The microcontroller kit used in the course is available at the Computer Store in the VCC. At the time of this printing, the price is $45. It doesn't include everything you will be using this semester, but it is remarkably self contained and ready to go out of the box. You might consider this microcontroller for projects in other course.

Chapter 2 - Lab Equipment

The *Embedded Control Lab* provides a complete suite of development and diagnostic tools to aid you in the development of your embedded control system. You are expected to load the Silicon Laboratory IDE software and the SDCC compiler on your laptop and to bring the laptop to every class. The Integrated Development Environment, IDE, is a package of programs that allow you to develop the software using the SDCC compiler for writing and compiling C codes for the C8051 on the EVB (evaluation board).

There are also several types of diagnostic tools available for you to use including *logic probes*, *multimeters*, and *oscilloscopes*. These are indispensable aids for diagnosing problems with the circuitry you will build, and the basic uses of these tools are described in the sections that follow.

Diagnostic Tools

Logic probe

The logic probe is a tool that is used to test digital circuitry. Most digital electronic circuitry utilizes binary digital logic. This means that only two possible logic *states*, or *levels*, are recognized by the circuitry. Commonly used designations for these levels are (1 and 0), (*TRUE* and *FALSE*), and (*HIGH* and *LOW*). Depending on the type of electronic circuitry, the logic levels themselves can be represented either by distinct current levels* or by distinct *voltage* levels. The digital circuitry that you will be using in your design work utilizes discrete *voltage levels,* by far the most common type.

Digital circuitry is designed to recognize any voltage above a specified threshold as *logic HIGH*, and any voltage below another specified threshold as *logic LOW* (nominally 0 volts). The threshold voltages may vary among families of logic circuitry. The circuitry you will be using is TTL† or TTL-compatible, which recognizes any voltage below 0.8 volts as logic *LOW* and any voltage above 2.0 volts as logic *HIGH* (nominally 5 volts). Voltages between 0.8 and 2.0 volts are considered ambiguous and hence undesirable.

Although there may be differences between models, logic probes generally have two LEDs to indicate logic state (*HIGH* or *LOW*), and one LED to indicate the occurrence of a transition

---

* Two examples are high-speed current mode logic (CML), and the 20mA current interface utilized in the RS-232-C standard. In the latter, a 20mA current represents logic LOW, and zero current represents logic HIGH.

† Transistor-Transistor Logic. Other logic families include *complementary metal-oxide semiconductor* (CMOS) and *emitter-coupled logic* (ECL).

between logic states. Additionally, they usually have a switch to select between CMOS or TTL circuitry, and another switch labeled *pulse* or *memory* (depending on the manufacturer) to turn a *transition memory mode* on or off. When the tip of the logic probe is placed upon a digital data line, the probe will yield the following types of information:

Logic state

The logic probe can indicate either of *two* possible *logic* states, *LOW* or *HIGH*, or neither of these states, namely, the *floating* state. The first two states are indicated when the appropriately labeled LED is on, and the *floating* state is indicated when neither the *HIGH* nor *LOW* LEDs are on. A common situation where the *floating* state is indicated occurs when the probe touches a wire that is *entirely isolated* from the circuit. Another *floating* state situation occurs when the probe is touching a signal path that is connected *only* to a logic gate *input* terminal.

Logic transitions

By lighting its *transition* or *pulse* LED, the logic probe can also give an indication of whether logic level transitions are occurring. Some probe models also emit a beep to indicate the occurrence of a transition. Because logic level pulses can be extremely brief, perhaps on the order of nano- or picoseconds, it may not be possible to observe the occurrence of such brief *HIGH* or *LOW* pulses on the respective LEDs. However, the logic probe's internal circuitry is designed to detect logic transitions occurring as closely as 60 nanoseconds apart, and report these events by lighting its *pulse* LED, and/or emitting a short beep. Most logic probes also have a transition *memory* or *pulse* mode switch that simply causes the probe's transition LED to remain lit indefinitely after a transition is detected. The probe's *transition* LED can then be cleared by manually turning the transition memory mode switch off and on (i.e., toggling the switch).

Using the logic probe

The logic probe requires an external power source and should be connected to the same power source utilized by the circuitry you are testing. The following list describes the steps necessary to use the logic probe:

1. Connect the probe's red and black power cord leads to +5 volts and ground respectively on your protoboard.
2. Set the CMOS/TTL switch appropriately (in the lab, you will most likely be using TTL components).
3. Set the pulse mode switch in accordance with how you plan to use the probe (see *Logic transitions* on page 6)
4. Touch the probe's tip to any signal path you wish to analyze, and the probe's LEDs will provide you with an indication of the logic activity.

In the Embedded Control lab, the logic probe finds typical application in debugging logic circuitry developed to interface to the C8051 microprocessor's parallel I/O ports. Such circuitry is developed by students to provide the means for the C8051 to control LED displays and/or stepper motors or to read push-button switch settings. More information on the logic probe can be found at *flitec.rpi.edu* in the *Tutorials.*

Multimeter

The multimeter is an instrument that can be used to measure AC and DC voltages and currents or to measure resistance or to determine electrical continuity. The multimeter will be especially useful for analyzing the analog circuitry. The multimeters in the lab are also able to test diodes by using the setting marked by a diode symbol. The diode test feature also allows circuit continuity to be tested. By placing the multimeter's red lead on one point and the black lead on the other point, the multimeter will emit a beep if the resistance between the two points is less than 200Ω. More information on the multimeter can be found at *flitec.rpi.edu* in the *Tutorials.*

Oscilloscope

An oscilloscope is used to analyze high-speed analog or digital signals. The oscilloscope provides a *continuous* representation of a signal in the form of a trace on its display screen. In comparison, the logic probe and multimeter provide only *discrete* data readings. Oscilloscopes are useful for determining the presence of unwanted voltage glitches or other types of noise on a signal path. If you are unable to isolate a circuit problem with either the logic probe or multimeter, the oscilloscope will provide a more complete "picture" of the signal.

The oscilloscope is somewhat more complicated to operate than either the logic probe or multimeter, so if you would like to use it during a lab and are not familiar with its operation, please asks the TA for assistance.

Development Tools

Computers

The software development tools that you will use will be run on your laptop computer. We will help you install the software during the first week of classes. You can reinstall at any point, if it becomes necessary.

The Silicon Labs C8051F020 evaluation boards (EVBs) use two ports (connections to your computer.) One is a USB port that is used to load your code onto the microcontroller. The other is a serial RS-232 port that can be used for data input or output while your program is running on the microcontroller. Most laptop computers don't have a serial port; it is handled by using a USB-to-Serial adapter provided in the classroom. There is a driver that must be downloaded for the USB-

to-Serial adapter. If you are using the Vista operating system, you will allow your computer to search the web to find the driver. We have installation links for other operating systems.

Any user input to the code running on the C8051 microcontroller and any ASCII output will be displayed on your laptop using terminal emulator software. SecureCRT is the recommended package and it is available free from the Help Desk in the VCC for any of the RPI machines. Other options include HyperTerminal and PuTTY. Both are available for free from the web. HyperTerminal is built into Windows XP but not Vista.

The installation instructions on LMS will be the most up-to-date ones available. What follows is for your convenience, but may be slightly dated.

8051 Software Installation and Configuration

Introduction

The SiLabs IDE is a convenient way to edit, compile, and download source code written for the microcontroller. While SiLabs provides a nice interface for making source code changes and easily downloading them to the development boards, it lacks the actual compiler portion, which converts C code to hex files, the common format used by the 8051. To do this, a free and widely used open source tool called Small Device C Compiler (SDCC) is used. SDCC compiles the C code written, and automatically optimizes and converts it to hex. Due to its popularity, support for SDCC in the SiLabs IDE comes standard, making it easy and convenient to use.

The detailed instructions for installing the SiLabs IDE and SDCC compiler are in a file on the main course web page, called **Installing_SiLabs-SDCC-Drivers_Win7.docx**. This file is updated frequently to be compatible with the dynamic web page URLs that are constantly changing. The instructions guide you step-by-step through the procedures to:

1. Download and install the SDCC compiler for code compilation
2. Download and install the IDE for code development and execution in the EVB
3. Add required C language support files for the EVB
4. Create a "Project" for each new lab assignment
5. Compile program files in a "Project"
6. Download and install support tools and Windows drivers
7. Provide a reference list of common problems and solutions

It cannot be emphasized enough that these instructions must be followed extremely carefully in order to be able to complete all the software assignments for homework and lab exercises required by this class.

Using the SiLabs IDE and HyperTerminal Software / SecureCRT / PuTTY

You can use either the HyperTerminal software (or SecureCRT/PuTTY) for serial communication on the computer. This will be used to receive information from the C8051 microcontroller. PuTTY is the preferred software.

If you choose the HyperTerminal, you can use the icon for the c8051.ht HyperTerminal communication link on your desktop (or elsewhere if you chose a different destination during installation described above). Double-click this icon to begin the communication link. Once the program begins, it will automatically attempt to establish a connection. You may disconnect this connection by selecting "Disconnect" in the "Call" menu or clicking the "Disconnect" icon. If you need to change any settings, you should disconnect the call.

The first time you use this software, you should check the properties for this communication link. Under the "File" menu, choose the "Properties" menu item. For use on an RPI laptop with a serial port, the software should connect using COM1. For laptop computers using the USB-to-serial adapter, one must check the COM port number in the device manager; right click my computer and select properties -> hardware -> device manager -> Ports. Find the COM port number there. Windows 8 users just type "device manager" in the search window.

On the Properties window, click on the "Configure" button below the COM port selection. The COM port properties should be set as follows:

- Bits per second: 38400

- Data bits: 8

- Parity: None

- Stop bits: 1

- Flow control: None

Click "OK" on both windows to close them and save any changes made.

To establish a connection, select "Call" under the "Call" menu, or click on the "Call" icon. Once the microcontroller is running a program, any screen outputs will appear in the HyperTerminal window.

If you choose the SecureCRT or PuTTY, the details are quite similar. After opening the SecureCRT software window, under the "File" menu, select "Connect (Alt + C)". Choose a new Session by selecting "Serial" under the "Protocol" drop-down list. Then set the Port, BaudRate, Databits, Parity, and Stopbits similar to the HyperTerminal as described above. Make sure that all the FlowControls are unchecked.

File management on the computer

It is strongly recommended that you create a folder for your code files. A good option is to create a folder within the C:\SiLabs\MCU folder. We will assume you have named that folder "Projects" for this manual, C:\SiLabs\MCU\Projects.

For file management on your laptop computers, remember to keep backup copies of your programs and other documents on a CD, a USB drive, or on your RCS account. **Make sure that all partners have access to the backup files**. Loss of files or inaccessible files is not a valid excuse for late work. Always keep a copy of your current project code on a USB thumb drive!

LITEC Multimedia Tutorials

The on-line tutorials contain much of the information needed for the course along with additional information on many other subjects extending beyond the required course material. While information found in the lab manual is often duplicated in the tutorials, some topics are covered in more detail in the latter.

The *LITEC* tutorials were originally developed using *HyperCard*, a development system used to create custom applications on the *Macintosh*. They have since been converted to the hypertext markup language (HTML) and are available on-line via the RPILMS class pages. A user can navigate through the tutorials to find further information by clicking on the various hypertext links within the website. The tutorials are available under the course materials on RPILMS (htpps://lms.rpi.edu).

The "main menu" of topics can be found on the top of the window and is always available to navigate to a new area of interest. These general topics can be selected by simply clicking on the topic name. A sub-menu of topics will then be listed below the main menu. Clicking on the items in the sub-menu will bring up the desired information in the main lower window. You will find that the tutorials contain helpful information on the lab assignments, hardware (circuit components and tools) and software aspects of the lab, and the basic concepts that you will be learning in this course. Additionally, the tutorial for each lab assignment contains a list of pertinent topics for that particular lab.

To exit the tutorials, either select the **Close** option from the *File* menu located in the title bar of *Internet Explorer* or simply click on the **X** button on the top right corner of the title bar.

Using the tutorials

Upon starting the tutorials, new users find that it takes just a bit of exploring and experimenting before they quickly gain a feeling for how the tutorials work and how to find information within them.

Connecting and downloading to the EVB

After successfully compiling and linking code in a project, you are ready to download it to the EVB on the cars for execution. Make sure the USB debug adapter from the car is plugged into your laptop. The **Connect** button on the toolbar will start the computer communicating with the EVB - you may have to wait a short time while the connection is established. Remember that the EVB must be powered in order for it to communicate with the computer. Once the connection is made, you can download your program using the **Download code** button on the toolbar. Once the code has finished downloading, choosing the "Go" icon or option in the "Debug" menu can start it.

If the code is correct, it will begin running and you can observe the results. The code will continue to run until the program ends, or until the "Stop" icon is selected.

Output from the EVB

If your program is going to produce some sort of text output, you will need to use a program called SecureCRT, HyperTerminal or PuTTY to show the output. SecureCRT is recommended for *Embedded Control* and the one that is best supported.

Protoboards

A protoboard is often used to test a logic circuit before the design is transferred to a printed circuit board. By first building the circuit on a protoboard, a circuit designer is able to test the functionality of the circuit and easily make necessary changes.

The type of protoboard you will be using has been specially designed to be mounted on the chassis of the *Smart Car*. *Figure 2.1* shows how this protoboard should be used.
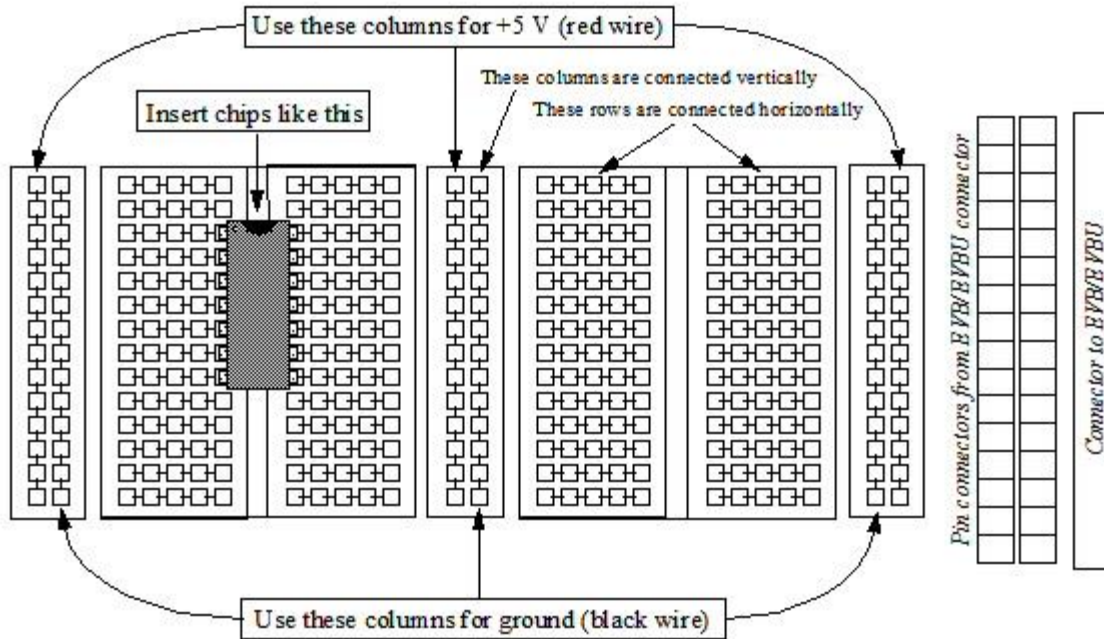
Figure 2.1 - Smart Car Protoboard

This protoboard has a ribbon cable connector for connecting to an EVB (or EVBU), a wire-insertion block strip to connect to the EVB pins, and a *main wiring area* where circuits are developed. In the wiring area shown in *Figure 2.1*, the lines indicate that the "holes" in the protoboard are connected underneath. This figure also illustrates how to properly position a chip along specific columns on the protoboard surface so that none of its pins are shorted together.

As you begin to assemble your circuitry on the protoboard, please take note of following suggestions and tips:

1. Thoroughly familiarize yourself on how the holes on the protoboard are connected.
2. Insert chips only in the special columns on the protoboard as illustrated in *Figure 2.1*.
3. **Use black wire only for ground connections and red wire only for +5 volts. Do not use red or black wire for any other purpose.** This is an industry-wide standard that should be followed.

If you follow the suggestions for neat wiring and the red/black color convention for power and ground, it will be much easier for you and your TA to diagnose any circuit problems. The *key* to successful protoboarding is to adopt *meticulous wiring methods*. Such methods will help you avoid making wiring mistakes in the first place, will make troubleshooting easier, and will result in a *much* more impressive-looking project!

Finding More Information

All of the information you will need for this course should be in this manual, but you may want to know more about some of the equipment or circuit components. The following list summarizes the manuals that are available for you to use in the lab.

C8051F02x Manual

SDCC Manual

Both manuals are available on RPILMS and the SDCC and SiLabs websites.

Chapter 3 - Programming in C

Since the heart of an embedded control system is a microcontroller, we need to be able to develop a program of instructions for the microcontroller to use while it controls the system in which it is embedded. When programs are developed on the same type of computer system on which they will be run, as is most commonly done, it is called *native platform development.* An example of native platform development is the use of *Borland's C/C++* to develop a program on an *Intel* or *AMD*-based computer such as an *Windows* PC, and subsequently running the program on the same computer.

However, the type of program development that you will be doing in this course is known as *cross platform development*, where your laptop computer (one platform) is used to develop programs that are targeted to run on the *SiLabs* C8051F020 EVB (another platform). Thus, even before such a program can be tested, it must be transmitted (downloaded) from the computer to the EVB.

For this embedded microcontroller, we will be using a programming language called 'C'. C is extremely flexible, and allows programmers to perform many low-level functions that are not easily accessible in languages like FORTRAN or Pascal. Unfortunately, the flexibility of C also makes it easier for the programmer to make mistakes and potentially introduce errors into their program. To avoid this, you should be very careful to organize your program so that it is easy to follow, with many comments so that you and others can find mistakes quickly. The example programs were written with this in mind, so that you get an idea of what well-structured programs look like. In order to get you started in C programming, this chapter will explain the basics that you will need to begin. If you would like more examples, they can be found in the *LITEC Tutorials* under *Software: C Programming*.

Brief Overview

As stated above, the C language allows many things that other languages do not, making it very easy to make errors. For this reason, it is suggested that you study this entire unit, including the tutorials if necessary, and become familiar with the specifics before beginning the labs. Although this may seem like a lot of work before the first lab, it will be worth your time and will pay off quickly. If there is a question you have regarding the C language that is not included in the manual, or in the tutorials, you will probably find the answer in a C reference text.

A Simple Program in C

The following program is similar to the first programming example used in most C programming books and illustrates the most basic elements of a C program.

```
#include <stdio.h>     /* include file */

main()                 /* begin program here */
{                      /* begin program block */

      printf("Hello World\r\n");

                       /* send Hello World to the terminal */

}                      /* end the program block */
```

The first line instructs the compiler to include the *header file* for the standard input/output functions. This line indicates that some of the functions used in the file (such as printf) are not defined in the file, but instead are in an external library (stdio.h is a standard library header file). This line also illustrates the use of comments in C. Comments begin with the two character sequence "/*" and end with the sequence "*/". Everything between is ignored and treated as comments by the compiler. Nested comments are not allowed.

The second line in the program is `main()`. Every C program contains a function called `main()` which is the function that executes first. The next line is a curly bracket. Paired curly brackets are used in C to indicate a *block* of code. In the case above, the block belongs to the `main()` statement preceding it.

The `printf` line is the only statement inside the program. In C, programs are broken up into *functions*. The `printf` function sends text to the terminal. In our case, the C8051 will send this text over the serial port to a "computer terminal", where we can view it. (You will use software on your laptop to simulate a terminal.) This line also illustrates that a semicolon follows every statement in C. The compiler interprets the semicolon as the end of one statement, and then allows a new statement to begin.

You may also notice that the comment after the `printf` statement continues over more than one line. It is important to remember that *the compiler ignores everything between the comment markers*. (See *Specifics of the SDCC Compiler*)

The last line is the closing curly bracket that ends the block belonging to the main function. More examples can be found in the tutorials under the C examples section.

Syntax Specifics

There are many syntax rules in C, but there is neither room nor time here to discuss everything in this manual. Instead, we explain the basics, along with the specifics of the *SDCC C Compiler*,

which are not in your textbook. Additional information about the C language can be found in the tutorials, and in any C reference text.

Declarations

One thing that was distinctly missing from the first example program was a variable. The type of variables available with the *SDCC C Compiler* for the C8051 microcontroller and their declaration types are listed below in *Table 3.1*:

Table 3.1 - SDCC C Compiler variable types

Type[*]	Size (bytes)	Smallest Value	Largest Value
integer			
(unsigned) char	1	0	255
signed char	1	-128	127
(signed) short	2	-32768	32767
unsigned short	2	0	65535
(signed) int	2	-32768	32767
unsigned int	2	0	65535
(signed) long	4	-2147483648	2147483647
unsigned long	4	0	4294967295
floating point			
float	4	1.2×10^{-38}	1.2×10^{38}
SDCC specific			
bit	1/8 = 1 bit	0	1
sbit	1/8 = 1 bit	0	1

The format for declaring a variable in a C program is as follows:

```
<type> variablename;
```

For example, the line

```
int i;
```

would declare an integer variable `i`. Although there are a large variety of variable types available, it is important to realize that the larger the size of the data type, the more time will be required by the C8051 to make the calculations. Increased calculation time is also an important consideration

---

[*] The items in parentheses are not required, but are implied by the definition. We recommend that you state these definitions explicitly to avoid errors due to misdefinition.

when using floating-point variables. It is suggested that in the interest of keeping programs small and efficient, you should not use floating point numbers unless absolutely necessary.

Repetitive Structures

Computers are very useful when repeating a specific task and almost every program utilizes this capability. The repetitive structures `for`, `while`, and `do...while` are all offered by C.

for Loops

The most common of looping structures is the `for` loop, which looks like this

```
for (initialize_statement; condition; increment) {
    ...
}
```

In the example above, the `for` loop will perform the "`initialize_statement`" one time before commencing the loop. The "`condition`" will then be checked to make sure that it is true (non-zero). As long as the "`condition`" is true the statements within the loop block will be performed. After each loop iteration, the increment statement is performed. For example:

```
for (i=0; i<10; i++) {
    display(i);
}
```

The statement above will initially set i equal to zero, and then call a user-defined function named `display()` 10 times. Each time through the loop, the value of `i` is incremented by one. After the tenth time through, `i` is set to 10, and the for loop is ended since `i` is not less than ten.

while Loops

Another frequently used loop structure is the `while` loop, which follows this format

```
while (condition){
    ...
}
```

When a `while` loop is encountered, the condition given in parenthesis is evaluated. If the condition is true (evaluates to non-zero), the statements inside the braces are executed. After the last of these statements is executed, the condition is evaluated again, and the process is repeated. When the condition is false (evaluates to zero), the statements inside the braces are skipped over, and execution continues after the closing brace. As an example, consider the following:

```
i = 0;
while (i<10)
```

```
    {
        display(i);
        i++;
    }
```

The above `while` loop will give the same results as the preceding example given with the `for` loop. The variable `i` is first initialized to zero. When the `while` is encountered in the next line, the computer checks to see if `i` is less than 10. Since `i` begins the loop with the value 0 (which is less than ten), the statements inside the braces will be executed. The first line in the loop, `i++`, increments `i` by 1 and is equivalent to `i=i+1`. The second line calls a function named `display` with the current value of `i` passed as a parameter. After `display` is called, the computer returns to the `while` statement and checks the condition `(i < 10)`. Since after the first iteration of the loop the value of `i` is 1, the condition `(i < 10)` evaluates to logical *TRUE* or equivalently "1", the loop will again be executed. The looping will continue until `i` equals 10 when the condition `(i < 10)` will evaluate as being false. The program will then skip over all the statements within the braces of the `while` construct, and proceed to execute the next statement following the closing brace "`}`".

Arrays

It may be necessary to store a list or table of values of the same type that you want to associate in some way. An array can be used to do this. An array is a group of memory locations all of the same name and data type. Each memory location in the array is called an element and is numbered with the first element as number "0". Note: Be aware, though, that arrays can quickly use up the available variable space, and the compiler does not necessarily check this potential problem. The array is declared with the type of data to be stored in the array as follows:

```
    <type> arrayname[maxsize];
```

For example, the lines

```
    int values[10];
    float timer[60]; /* floating point arrays may take up a lot of memory */
```

would declare an array named values that can store up to ten integers (values[0].... values[9]) and an array named timer that can store up to sixty floating point values (timer[0].... timer[59]). The array can be initially filled with values when it is declared, or it can later be filled with data by the program as follows:

```
    int c[5]={23, 10, 35, 2, 17};    /* c[0]...c[4] is filled with listed values */
    int f[5]={0};                    /* f[0]...f[4] is filled with zeros */
    for (i=0;i<=4;i++)
        f[i]=i;                      /* fills the elements of f[0]..f[4] with 0..4 */
```

19

Arrays can also have multiple dimensions. A simple example is an array with multiple rows and columns. These arrays can also be *initialized* and filled as follows. The simplest way to reset an entire array *within* a program is using `for()` loops that clear every element individually.

```
int data[2][10]={{0},{0}};    /* initialized data to have 2 rows and 10 columns filled with
                                 zeros -initialization only works when variable is declared */
data[0][5]=26;                /* puts the value 26 into the element at row 0, column 5 */
data[1][9]=5;                 /* puts the value 5 into the element at row 1, column 9 */
```

Operators

In addition to a full complement of keywords and functions, C also includes a full range of operators. Operators usually have two arguments, and the symbol between them performs an operation on the two arguments, replacing them with the new value. You are probably most familiar with the mathematical operators such as + and -, but you may not be familiar with the bitwise and logical operators which are used in C. *Table 3.2 - Table 3.7* list some of the different types of operators available. The operators are also listed in the order of precedence in *Table 3.8*. Similar to operation precedence in algebra where multiplication precedes addition, all C operators obey a precedence, which is summarized in *Table 3.8*.

Mathematical

The symbols used for many of the C mathematical operators are identical to the symbols for standard mathematical operators, e.g., add "+", subtract "-", and divide "/". *Table 3.2* lists the mathematical operators.

Table 3.2 - Mathematical operators

operator	description
*	multiplication
/	division
%	mod (remainder)
+	addition
-	subtraction

Relational, Equality, and Logical

The C language offers a full range of control structures including `if…else`, `while`, `do…while`, and `switch`. Most of these structures should be familiar from previous computing classes, so the concepts are left to a reference text on C. In C, remember that any non-zero value is true, and a value of zero is false. Relational, equality, and logical operators are used for tests in control structures, and are shown in *Table 3.3*. All operators in this list have two arguments (one on each side of the operator).

Table 3.3 - Relational, equality, and logical operators

operator	description	operator	description
<	less than	==	equal to
>	greater than	!=	not equal to
<=	less than or equal to	||	logical OR
>=	greater than or equal to	&&	logical AND

Bitwise

C can perform some low-level operations such as bit-manipulation that are difficult with other programming languages. In fact, some of these bitwise functions are built into the language. *Table 3.4* summarizes the bitwise operations available in C.

Table 3.4 - Bitwise and shift operators

operator	description	example	result
&	bitwise AND	`0x88 & 0x0F`	`0x08`
^	bitwise XOR	`0x0F ^ 0xFF`	`0xF0`
|	bitwise OR	`0xCC	0x0F`	`0xCF`
<<	left shift	`0x01 << 4`	`0x10`
>>	right shift	`0x80 >> 6`	`0x02`

Table 3.5 - Truth Tables

`X|Y=Q`

X	Y	Q
0	0	0
0	1	1
1	0	1
1	1	1

`X&Y=Q`

X	Y	Q
0	0	0
0	1	0
1	0	0
1	1	1

Unary

Some C operators are meant to operate on one argument, usually the variable immediately following the operator. *Table 3.6* gives a list of those operators, along with some example for reference purposes.

Assignment

Most mathematical and bitwise operators in the C language can be combined with an equal sign to create an assignment operator. For example `a+=3;` is a statement which will add 3 to the

current value of `a`. This is a very useful shorthand notation for `a=a+3;`. Note all 10 variations on this syntax that are shown in *Table 3.8* on page 23. All of the assignment operators have the same precedence as equals, and are listed in the precedence table.

Miscellaneous

Many of the operators in C are specific to the syntax of the C language, and bear other meanings depending on their operands. *Table 3.7* below is a list of some miscellaneous operators that are specific to the C language. This table has been included only as a reference, and you may wish to refer to a C reference manual for complete descriptions of these operators

Table 3.6 - Unary operators

operator	description	example	equivalent
++	post-increment	`j = i++;`	`j = i;` `i = i + 1;`
++	pre-increment	`j = ++i;`	`i = i + 1;` `j = i;`
--	post-decrement	`j = i--;`	`j = i;` `i = i - 1;`
--	pre-decrement	`j = --i;`	`i = i - 1;` `j = i;`
*	pointer dereference	`*ptr`	value at a memory location whose address is in `ptr`
&	reference (pointer) of	`&i`	the address of `i`
+	unary plus	`+i`	`i`
-	unary minus	`-i`	the negative of `i`
~	ones complement	`~0xFF`	`0x00`
!	logical negation	`!(0)`	`(1)`

Table 3.7 - Miscellaneous operators

operator	description
()	function call
[]	array
->	pointer to structure member access
.	structure member access
(*type*)	type cast
sizeof	size of type (in bytes)
?:	if ? then: else
,	combination statement

Precedence and Associativity

All operators have a precedence and an associativity. *Table 3.8* illustrates the precedence of all operators in the language[*1]. Operators on the same row have equal precedence, and precedence decreases as you move down the table.

Table 3.8 - Operator precedence and associativity

Operators[†]	associativity
() [] ->.	left to right
! ~ ++ -- * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Programming Structure Hints

The C programs you develop should be written in a style that is easy for yourself and others to read and maintain (modify). Since issues pertaining to programming style are the topic of entire textbooks, the following are a few helpful guidelines and hints that are generally regarded as hallmarks of good programming style.

− Strictly follow the C style convention for the indentation of blocks of code.

− Select identifier names for C variables and functions that implicitly describe their functionality.

---

* Kernighan and Ritchie, *The C Programming Language*
† This table is from Kernighan and Ritchie, *The C Programming Language.*

- Keep the scope of all variables as *local* within functions unless absolutely necessary to globalize their scope.

- As much as possible, encapsulate the functionality of chunks of code into C functions, i.e. adopt a **modular** programming style. It is especially important to avoid having functionally equivalent copies of code dispersed throughout a project comprising one or more files. If the functionality of similar chunks of code can be encapsulated into a single C function, not only will this result in a reduction of the number of lines of code, but also more of the code maintenance can be isolated to *solitary* functions.

- Use comments liberally throughout a program. A good rule of thumb is to include a descriptive comment for about every three lines of code. Comments are extremely helpful when referring to old code or someone else's code.

- For convenience, a decimal/hexadecimal/binary conversion table is included below:

Decimal	Hexadecimal	Binary	Decimal	Hexadecimal	Binary
0	0x0	0000	8	0x8	1000
1	0x1	0001	9	0x9	1001
2	0x2	0010	10	0xA	1010
3	0x3	0011	11	0xB	1011
4	0x4	0100	12	0xC	1100
5	0x5	0101	13	0xD	1101
6	0x6	0110	14	0xE	1110
7	0x7	0111	15	0xF	1111

Specifics of the *SDCC C* Compiler

The Small Device C Compiler is specifically built for programming microcontrollers including the 8051 series microcontroller. SDCC is free, open-source software available for download to anyone.

Library Functions

As seen in the prior example programs, most of the things done in C (with the exception of low-level functions) involve using library functions. All compilers have their own library functions, but they are usually very close to those defined by the ANSI standard. The *SDCC C* compiler is no exception since it includes most of the ANSI functions and contains some

extensions specifically for the C8051. A list of useful functions is included in *Appendix A - Programming Information* for your reference.

In particular, **it must be noted** that SDCC supports two separate print functions: `printf( )` and `printf_fast_f( )`. Both are described in *Appendix A*. The first, `printf( )`, should be used regularly but note that it can't be used to print floating point variables. When floating point variables must be printed the `printf_fast_f( )` function is needed. Also pay attention to using the proper type %d, %i, %u, %l, %x, %lu, and %f with the variable. **An unsigned long variable can be printed as %d without any error or warning but the output will not be what would be expected.**

Comments

In accordance with the ANSI standard, nested comments (comments within comments) are not allowed.

Definition of an Interrupt Handler Function

The SDCC C compiler will generate a function as an interrupt handler if it is defined as such using the `__interrupt` keyword in the function definition. The `__interrupt` keyword is not portable to most other compilers. If you wish to test compile your code on another compiler, such as *Borland C/C++* or *gcc* on RCS, you will need to remove `__interrupt` from your function prototypes and definitions.

Limitations of the demo version

The Silicon Labs IDE that comes with the develop kits natively supports a variety of microcontroller compilers, including SDCC. Since both are available online, all code can be tested for compilation errors just as it would work within the lab.

If you desire to use your knowledge from this course to include an embedded microcontroller in a project in IED or the MDL, you may be able to purchase one at the VCC Computer Store. They may have to order the microcontroller, so be aware that a delivery time will exist.

Chapter 4 - The *Silicon Labs* C8051F020 and the EVB*

Each station in the lab has been provided with a *Silicon Labs C8051F020 EVB* (**EV**aluation **B**oard). The EVB consists of a *C8051F020* microcontroller (64k bytes of internal flash and 4k bytes of internal RAM), and an RS232 transceiver and connector that provide a means of serial communication between the EVB and the host computer. The EVB also has a ribbon cable connector that is used to connect the EVB to a protoboard where students build their circuitry. Since the EVB is itself a computer, after a program is downloaded from the computer to the EVB and begins running, the serial cable between the EVB and the computer may be disconnected without affecting the operation of the program running on the EVB. If you would like more detailed information on the Silicon Labs C8051F020 (sometimes called the SiLabs C8051 for short) or the EVB, the *Silicon Labs C8051F020 Reference Manual* is available on RPILMS and at www.silabs.com.

Powering the EVB

When the car is at the station, the charger and battery should always be connected to the appropriate connections on the *Smart Car* power board (See *Appendix C - Helpful Information*). When the EVB is not in use, as when the student is writing code on the computer, the EVB switch on the side of the protoboard mounting should be in the *off* position, turning the red LED off. When the student leaves the lab, they should make sure the charger is plugged in and the EVB switch is in the *off* position.

Input/Output Ports on the C8051

The C8051 has seven ports, P0 through P7, which can be used for digital input or output. In this class, we will only be using ports *0* through *3*, which are all 1 byte (i.e., 8 bits) wide - these ports are addressable as 1-byte entities, or else you can address the individual pins of the port. When the C8051 reads one of its digital ports, 0 volts on the data line of the port is interpreted as logic *FALSE* and 3+ volts is interpreted as logic *TRUE*. Ports 0 through 3 each has a data register and an output mode register associated with it. Sections to follow will cover reading from and writing to individual bits within these 1-byte entities.

---

* Portions of this chapter are taken from the Silicon Laboratories *C8051F020 Reference Manual*, copyright 2004. All rights reserved, used with permission.

Table 4.1 - Predefined port addresses

Port	Associated Data Register	Associated Output Mode Register	Other Associated Registers
Port 0	P0	P0MDOUT	
Port 1	P1	P1MDOUT	P1MDIN - Port 1 Input Mode Register
Port 2	P2	P2MDOUT	
Port 3	P3	P3MDOUT	P3IF - Port 3 Interrupt Flag Register
Port 4	P4	P74OUT	
Port 5	P5	P74OUT	
Port 6	P6	P74OUT	
Port 7	P7	P74OUT	

7	6	5	4	3	2	1	0
P*x*.7	P*x*.6	P*x*.5	P*x*.4	P*x*.3	P*x*.2	P*x*.1	P*x*.0

P*x*: Port*x* Data Register, *x* = 0…7
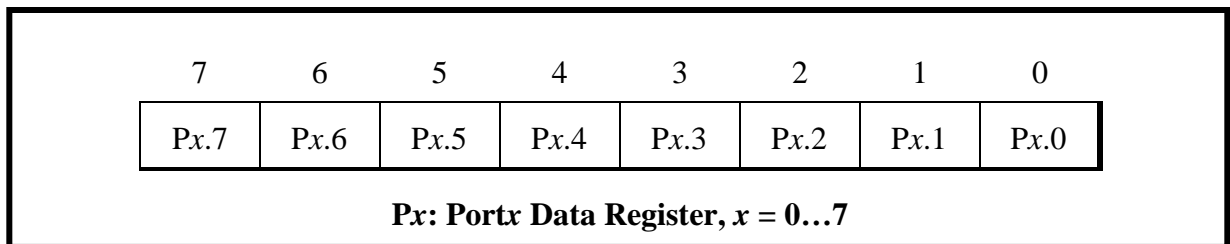
Note that these data register are both bit- and byte-addressable - this means that each bit can be written to or read from individually using its assigned name or the register as a whole can be written to or read from. For example, bit 7 of Port 0 can be set to logic 1 using the code:

```
__sbit __at 0x87 bit7;      /* create sbit for Port 0, pin 7 */
bit7 = 1;                    /* set just pin 7 of Port 0 to high */;
```

that sets the particular bit to 1, or using the line:

```
P0 |= 0x80;
```

that uses a bitwise "Or" operator to "mask" the bits that are not to be changed and set the desired bit.

As the name implies, the output mode registers are used to set the output mode for the associated port.

7	6	5	4	3	2	1	0

Bits 7-0:   PxMDOUT.[7:0]: Port x Output Mode Bits
          0: Port Pin output mode is configured as Open-Drain (input).
          1: Port Pin output mode is configured as Push-Pull (output).

P*x*MDOUT: Port*x* Output Mode Register, *x* = 0...3

27
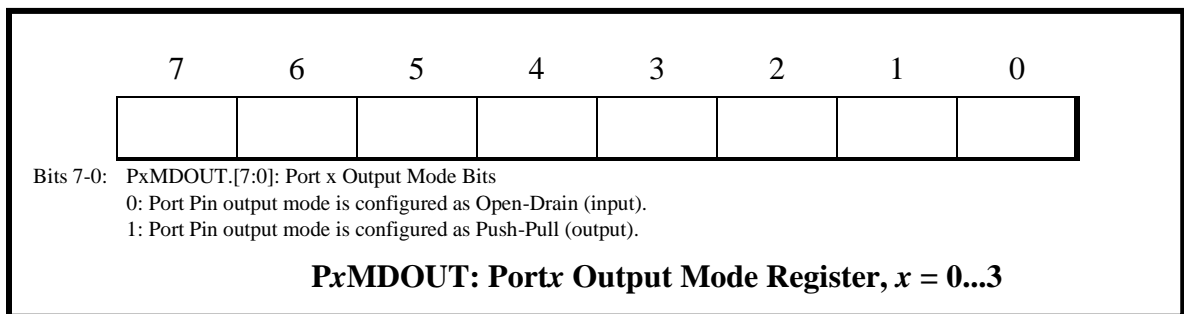
For more information on the I/O ports and their associated registers, see p.164 in the *C8051F020 Reference Manual*.

Setting Bits for Input or Output

A Port pin is configured as a digital input by setting its output mode to "Open-Drain" and writing a logic 1 to the associated bit in the Port data register. For example, P3.7 (Port 3, bit 7) is configured as a digital input by setting P3MDOUT.7 (Port 3 Output Mode register, bit 7) to a logic 0 and P3.7 to a logic 1. The following lines of code illustrate one way to do this in your program:

```
P3MDOUT &= ~0x80;      // Configure Port 3, pin 7 to Open Drain mode
P3 |= 0x80;            // Set Port 3, pin 7 equal to 1 (logic high)
```

A Port pin can be configured as a digital output by setting its output mode to "Push-Pull", as in the following code:

```
P2MDOUT |= 0x04;       // Configure Port 2, pin 2 to Push-Pull mode
```

Digital output can also be performed in Open-Drain mode, but for the purposes of this course, we will be using Push-Pull mode for output. For further information on the output modes, refer to the *C8051F020 Reference Manual*, p. 161.

Reading and Writing Individual Bits

For a program to read the value of a *particular* bit of a digital input port, it is sometimes necessary (or just easier) to first read the *entire* port (all 8 bits), and then, with a second operation called *masking*, to determine the value of any *individual* bit. Similarly, for a program to change a particular bit it may be necessary or desirable to write to the entire byte. This section will cover the techniques that are applied to read and write individual bits within a register.

Example 4.1 below illustrates a code segment from a C program which will set bits 0 - 1 of port 2 and bits 3-6 of port 3 for *output*, while setting bits 4 - 7 port 2 and bits 0 - 2 of port 3 for *input*. The remaining bits of both ports should not be touched and are to be left as they are. The program will then enter a loop to read bit 4 of port 2 through *masking*. The *masking* operation sets those bits that are not of interest to 0, thereby ignoring the state of all bits on port 2 except those of interest (P2.4). In *Example 4.1*, the variable value will equal 0x10 if the state of P2.4 is high, and value will equal 0x00 if the state of P2.4 is low. Upon determining that bit 4 of port 2 is logic *TRUE* (1), the program will then set bits 4 and 6 of port *3* to *TRUE* (1) and bits 3 and 5 of port *3* to logic *FALSE* (0).

```
OD -> Open-Drain, PP -> Push-Pull, XX -> Don't Touch, in -> input, out -> output

Port 2:
MODE (P2MDOUT) OD OD OD OD   XX XX PP PP   (0000 XX11 => in  in  in  in   xxx xxx out out)
IMPEDANCE (P2) HI HI HI HI   -- -- -- --   (1111 XXXX)

Port 3:
MODE (P3MDOUT) XX PP PP PP   PP OD OD OD   (X111 1000 => xxx out out out  out in  in  in )
IMPEDANCE (P3) -- -- -- --   -- HI HI HI   (XXXX X111)


    char value, mask;
    mask = 0x10;                     // "anding" mask to ignore all but bit 4

        P2MDOUT |= 0x03;             // Port 2, pins 0-1 push-pull
        P2MDOUT &= 0x0F;             // Port 2, pins 4-7 open-drain
        P2 |= ~0x0F;                 // Set pins 4-7 of Port 2 to 1
                                     // this will always be the ~ of the MDOUT "&=" value
        P3MDOUT |= 0x78;             // Port 3, pins 3-6 push-pull
        P3MDOUT &= 0xF8;             // Port 3, pins 0-2 open-drain
        P3 |= ~0xF8;                 // Set pins 0-2 of Port 3 to 1
                                     // this will always be the ~ of the MDOUT "&=" value

    do
    {
        value = P2 & mask;
        while (value != mask);       // do while bit 2.4 is LOW
    }
                                     // Want P3 <- X101 0XXX
    P3 |= 0x50;                      // 0x50 in binary is: 0101 0000, use to set bits to 1
    P3 &= 0xD7;                      // 0xD7 in binary is: 1101 0111, use to set bits to 0
```

Example 4.1 - Setting bits of Ports 2 and 3 for input and output, and waiting until bit 4 of Port 2 goes HIGH

Individual bits in a register may be written to by using bitwise operations. *Example 4.2* illustrates clearing certain bits in a port while leaving other bits unchanged, and also setting only certain bits in a port.

```
P2 = P2 & 0x6A;  // Clear bits 0,2,4 and 7 in port 2, don't change bits 1,3,5 and 6
P3 |= 0x95;      // Set bits 0,2,4 and 7 in port 3, don't change bits 1,3,5 and 6
```

Example 4.2 - Setting and clearing bits

The first line in *Example 4.2* performs a bitwise "and" between the current state of Port 2 and 0x6A (binary 0110 1010). This will clear (set them low) bits 0, 2, 4 and 7 while leaving the other bits in the register unchanged. The second line in the example sets bits 0, 2, 4 and 7 (sets them high) by performing a bitwise "or" between the current state of the register and 0x95 (binary 1001 0101).

Crossbar
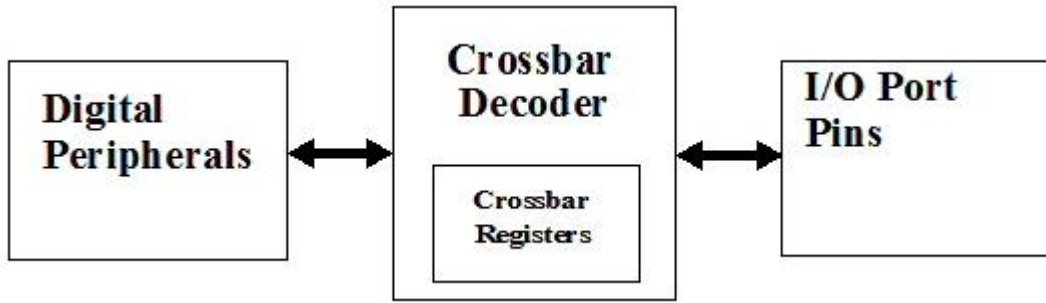
The C8051 EVB comes equipped with a number of "digital peripherals", such as timers, a Programmable Counter Array (see "Programmable Counter Array" on page 44), etc. These peripherals do not have dedicated I/O pins; rather, the Crossbar assigns them pins for input and output.

PIN I/O	P0 0	1	2	3	4	5	6	7	P1 0	1	2	3	4	5	6	7	P2 0	1	2	3	4	5	6	7	P3 0	1	2	3	4	5	6	7	Crossbar Register Bits
TX0	●																																UART0EN: XBR0.2
RX0		●																															
SCK	●	●																															SPI0EN: XBR0.1
MISO		●	●																														
MOSI			●	●																													
NSS				●	●																												
SDA	●		●		●																												SMB0EN: XBR0.0
SCL		●		●		●																											
TX1	●		●		●				●								●																UART1EN: XBR2.2
RX1		●		●		●				●								●															
CEX0	●		●		●				●	●	●						●	●															PCA0ME: XBR0.[5:3]
CEX1		●		●		●			●	●	●						●	●	●														
CEX2			●		●		●		●	●	●						●	●	●	●													
CEX3				●		●		●	●	●	●						●	●	●	●	●												
CEX4					●		●		●	●	●	●					●	●	●	●	●	●											
ECI	●	●	●	●	●	●	●	●	●	●	●	●	●				●	●	●	●	●	●	●										ECI0E: XBR0.6
CP0	●	●	●	●	●	●	●	●	●	●	●	●	●	●			●	●	●	●	●	●	●	●	●								CP0E: XBR0.7
CP1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●							CP1E: XBR1.0
T0	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●						T0E: XBR1.1
/INT0	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●					INT0E: XBR1.2
T1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●				T1E: XBR1.3
/INT1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●			INT1E: XBR1.4
T2	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●		T2E: XBR1.5
T2EX	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	T2EXE: XBR1.6
T4	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	T4E: XBR2.3
T4EX	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	T4EXE: XBR2.4
/SYSCLK	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	SYSCKE: XBR1.7
CNVSTR	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	CNVSTE: XBR2.0

Figure 4.1 - Priority Crossbar Decode Table

The Priority Crossbar Decoder, or "Crossbar", allocates and assigns Port pins on Port 0 through Port 3 to the digital peripherals (UARTs, SMBus, PCA, Timers, etc.) on the device. The Port pins are allocated in order starting with P0.0 and continue through P3.7 if necessary. The digital

peripherals are assigned Port pins in a priority order that is listed in *Figure 4.1* with UART0 having the highest priority and CNVSTR having the lowest priority.

The Crossbar assigns Port pins to a peripheral if the corresponding enable bits of the peripheral are set to a logic 1 in the Crossbar configuration registers XBR0, XBR1, and XBR2. The Port I/O crossbar registers are shown below. Priority assignments for item such as UART0, SPI0, SMB0, UART1, and the CEXn lines were shown previously in the Priority Crossbar Decode Table on page 33.

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
CP0E	ECI0E		PCA0ME		UART0EN	SPI0EN	SMB0EN	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xE1

Bit7:   CP0E: Comparator 0 Output Enable Bit.
           0: CP0 unavailable at Port pin.
           1: CP0 routed to Port pin.
Bit6:   ECI0E: PCA0 External Counter Input Enable Bit.
           0: PCA0 External Counter Input unavailable at Port pin.
           1: PCA0 External Counter Input (ECI0) routed to Port pin.
Bits5-3:   PCA0ME: PCA0 Module I/O Enable Bits.
           000: All PCA0 I/O unavailable at Port pins.
           001: CEX0 routed to Port pin.
           010: CEX0, CEX1 routed to 2 Port pins.
           011: CEX0, CEX1, and CEX2 routed to 3 Port pins.
           100: CEX0, CEX1, CEX2, and CEX3 routed to 4 Port pins.
           101: CEX0, CEX1, CEX2, CEX3, and CEX4 routed to 5 Port pins.
           110: RESERVED
           111: RESERVED
Bit2:   UART0EN: UART0 I/O Enable Bit.
           0: UART0 I/O unavailable at Port pins.
           1: UART0 TX routed to P0.0, and RX routed to P0.1.
Bit1:   SPI0EN: SPI0 Bus I/O Enable Bit.
           0: SPI0 I/O unavailable at Port pins.
           1: SPI0 SCK, MISO, MOSI, and NSS routed to 4 Port pins.
Bit0:   SMB0EN: SMBus0 Bus I/O Enable Bit.
           0: SMBus0 I/O unavailable at Port pins.
           1: SMBus0 SDA and SCL routed to 2 Port pins.   **XBR0: Port I/O Crossbar Register 0**

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
SYSCKE	T2EXE	T2E	INT1E	T1E	INT0E	T0E	CP1E	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xE2

Bit7: SYSCKE: /SYSCLK Output Enable Bit.
0: /SYSCLK unavailable at Port pin.
1: /SYSCLK routed to Port pin.

Bit6: T2EXE: T2EX Input Enable Bit.
0: T2EX unavailable at Port pin.
1: T2EX routed to Port pin.

Bit5: T2E: T2 Input Enable Bit.
0: T2 unavailable at Port pin.
1: T2 routed to Port pin.

Bit4: INT1E: /INT1 Input Enable Bit.
0: /INT1 unavailable at Port pin.
1: /INT1 routed to Port pin.

Bit3: T1E: T1 Input Enable Bit.
0: T1 unavailable at Port pin.
1: T1 routed to Port pin.

Bit2: INT0E: /INT0 Input Enable Bit.
0: /INT0 unavailable at Port pin.
1: /INT1 routed to Port pin.

Bit1: T0E: T0 Input Enable Bit.
0: T0 unavailable at Port pin.
1: T0 routed to Port pin.

Bit0: CP1E: CP1 Output Enable Bit.
0: CP1 unavailable at Port pin.
1: CP1 routed to Port pin.

XBR1: Port I/O Crossbar Register 1

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
WEAKPUD	XBARE	-	T4EXE	T4E	UART1E	EMIFLE	CNVSTE	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address: 0xE3

Bit7: WEAKPUD: Weak Pull-Up Disable Bit.
0: Weak pull-ups globally enabled.
1: Weak pull-ups globally disabled.

Bit6: XBARE: Crossbar Enable Bit.
0: Crossbar disabled. All pins on Ports 0, 1, 2, and 3, are forced to Input mode.
1: Crossbar enabled.

Bit5: UNUSED. Read = 0, Write = don't care.

Bit4: T4EXE: T4EX Input Enable Bit.
0: T4EX unavailable at Port pin.
1: T4EX routed to Port pin.

Bit3: T4E: T4 Input Enable Bit.
0: T4 unavailable at Port pin.
1: T4 routed to Port pin.

Bit2: UART1E: UART1 I/O Enable Bit.
0: UART1 I/O unavailable at Port pins.
1: UART1 TX and RX routed to 2 Port pins.

Bit1: EMIFLE: External Memory Interface Low-Port Enable Bit.
0: P0.7, P0.6, and P0.5 functions are determined by the Crossbar or the Port latches.
1: If EMI0CF.4 = '0' (External Memory Interface is in Multiplexed mode)
  P0.7 (/WR), P0.6 (/RD), and P0.5 (ALE) are 'skipped' by the Crossbar and their output
  states are determined by the Port latches and the External Memory Interface.
1: If EMI0CF.4 = '1' (External Memory Interface is in Non-multiplexed mode)
  P0.7 (/WR) and P0.6 (/RD) are 'skipped' by the Crossbar and their output states are
  determined by the Port latches and the External Memory Interface.

Bit0: CNVSTE: External Convert Start Input Enable Bit.
0: CNVSTR unavailable at Port pin.
1: CNVSTR routed to Port pin.

XBR2: Port I/O Crossbar Register 2

For example, if the UART0EN bit (XBR0.2) is set to a logic 1, the TX0 and RX0 pins will be mapped to P0.0 and P0.1 respectively. Because UART0 has the highest priority, its pins will always be mapped to P0.0 and P0.1 when UART0EN is set to logic 1. SPI has the next highest priority and the port pins mapped will depend on whether UART0 is enabled or not. If UART0 is enabled, P0.2, P0.3, P0.4, and P0.5 will be mapped to SPI. If UART0 is not enabled, then P0.0, P0.1, P0.2 and P0.3 will be mapped to SPI. Continuing to the next highest priority, SMB will reserve two port pins for mapping. The pins selected will depend on whether UART0 and SPI are enabled. If a digital peripheral's enable bits are not set to a logic 1, then its ports are not accessible at the Port pins of the device. Also note that the Crossbar assigns pins to all associated functions when a serial communication peripheral is selected. It would be impossible, for example, to assign TX0 to a Port pin without assigning RX0 as well. Each combination of enabled peripherals results in a unique device pinout. Note that since UART0 (Universal Asynchronous Receiver/Transmitter 0) is used for communication between the PC and EVB, UART0EN (XBR0.2) should always be set to logic high (see the next section, Initializing the system). Further more, pins 0 and 1 of Port 0 are not available for use with any other peripherals.

All Port pins on Ports 0 through 3 that are not allocated by the Crossbar can be accessed as General-Purpose I/O pins by reading and writing the associated Port Data registers.

Initializing the system

All of the necessary initialization and declarations are already made for you, and are included in the header files *c8051.h* and *c8051_SDCC.h*. These files should be saved in the following directory: *C:\Program Files\SDCC\include\mcs51*. On computers running Vista, the directory may be *C:\Program Files (x86)\SDCC\include\mcs51*. The file can be included for your use by adding the following line to your source code:

```
#include <c8051_SDCC.h>.
```

In addition, you will need to call the function *Sys_Init()*, which is defined in c8051_SDCC.h, in the *main()* function of your program.

Table 4.1 summarizes the necessary Port I/O registers that are available to you, and examples 4.1 and 4.2 illustrate the use of these predefined registers. There are additional definitions in this file for features such as timers and serial interfaces. If you wish to make use of any of these features, a complete listing of this header file is listed in *Appendix A - Programming Information* (see *c8051_SDCC.h header file*). See the *C8051F020 Reference Manual* for more information.

One of the processes that the c8051_SDCC.h file initializes is the use of UART0 for serial communication. UART0 is an enhanced serial port with features such as frame error detection and address recognition hardware. You will not need to make any changes to UART0, but it should be

noted that bit 2 of Crossbar Register 0 (XBR0) must be set in order to enable UART0 for input/output.

These predefined port addresses are set up for the programmer to use them as variables. In order to read from a location, simply set another variable equal to the port's register. For example:

```
char a;
a = P3;
```

To write to a memory location (if it is allowed), simply set the port label listed above to a value. For example:

```
P3 = 0x6F;
```

The process of initializing the ports is explained in *Setting Bits for Input or Output* on page 29.

Timer Functions

The clock in a microcontroller is the heart of the microcontroller—it dictates the timing of the execution of each command in a sequence of instructions. Various options are provided through user instructions by which the user can control the timing of certain commands, incorporate delays, and most interestingly, perhaps, generate different functions like square waves and pulses which, in turn, can be used to generate other types of waveforms. Besides the generation of different waveforms, some of the properties of these waveforms can also be studied using the timer functions in the microcontroller. Some of the timer functions capabilities provided by the C8051 are discussed in the following sections.

System Clock (SYSCLK)

The system clock can be used to time actions to occur at fixed periodic intervals, or to measure the time between actions or events. The C8051F020 can use either an internal oscillator or an external oscillator as its system clock source. The internal oscillator operates at a frequency of 2 MHz by default, but can be configured by software to operate at other frequencies. The external oscillator operates at 22.1184 MHz.

Timers[*]

C8051 devices have 5 counter/timers, known as Timer 0, Timer 1... Timer 4. These timers can be configured to be 8, 13, or 16 bits long, and can be used to measure time intervals, count external events, and generate periodic interrupt requests. Detailed information on the counter/timers can be found starting on page 225 of the *C8051F020 Reference Manual*. For LITEC, we will be using Timer 0 only; therefore, this section will focus on describing the use of Timer 0. Timers 0 and 1

---

[*] Chapter 22 of *C8051F020 Reference Manual*

are nearly identical; however, for this course, Timer 1 is dedicated to the use of UART0 for serial communication, and therefore is not available for your use.

Timer 0 can be configured as a 16-bit, 13-bit, or 8-bit counter; it can count the system clock (SYSCLK), the system clock/12, or external events. In addition, the counting can be turned on and off by an external event or by a program. The timer/counter is configured and controlled using three special function registers: CKCON (the Clock Control Register), TMOD (the Timer Mode Register), and TCON (the Timer Control Register).

CKCON can be used to select any of the 5 timers (Timer 0 through Timer 4), and to specify whether a timer will count the system clock, or the system clock/12. Bit 3 of CKCON is used to select whether Timer 0 will count SYSCLK, or SYSCLK/12.

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
-	T4M	T2M	T1M	T0M	Reserved	Reserved	Reserved	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
								0x8E

Bit7:   UNUSED. Read = 0b, Write = don't care.
Bit6:   T4M: Timer 4 Clock Select.
          This bit controls the division of the system clock supplied to Timer 4. This bit is ignored when the
          timer is in baud rate generator mode or counter mode (i.e. C/T4 = 1).
          0: Timer 4 uses the system clock divided by 12.
          1: Timer 4 uses the system clock.
Bit5:   T2M: Timer 2 Clock Select.
          This bit controls the division of the system clock supplied to Timer 2. This bit is ignored when the
          timer is in baud rate generator mode or counter mode (i.e. C/T2 = 1).
          0: Timer 2 uses the system clock divided by 12.
          1: Timer 2 uses the system clock.
Bit4:   T1M: Timer 1 Clock Select.
          This bit controls the division of the system clock supplied to Timer 1.
          0: Timer 1 uses the system clock divided by 12.
          1: Timer 1 uses the system clock.
Bit3:   T0M: Timer 0 Clock Select.
          This bit controls the division of the system clock supplied to Counter/Timer 0.
          0: Counter/Timer uses the system clock divided by 12.
          1: Counter/Timer uses the system clock.
Bits2-0: Reserved. Read = 000b, Must Write = 000.

CKCON: Clock Control Register

The Timer Mode Register (TMOD) is used to configure Timers 0 and 1. Bits 0 and 1 set the number of bits (8, 13, or 16) for Timer 0, bit 2 is used to specify whether the counter should increment on a clock tick or on an input from an external pin, and bit 3 determines whether enabling Timer 0 is dependent on the external input pin /INT0.
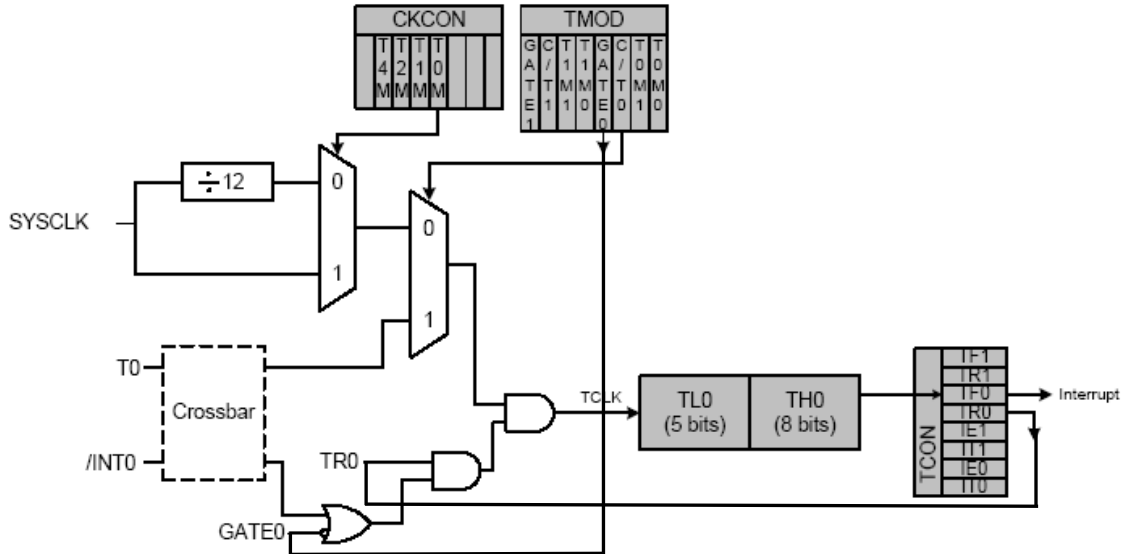
Figure 4.2 - T0 Mode 0 Block Diagram

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
GATE1	C/T1	T1M1	T1M0	GATE0	C/T0	T0M1	T0M0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
								0x89

Bit7:   GATE1: Timer 1 Gate Control.
          0: Timer 1 enabled when TR1 = 1 irrespective of /INT1 logic level.
          1: Timer 1 enabled only when TR1 = 1 AND /INT1 = logic 1.
Bit6:   C/T1: Counter/Timer 1 Select.
          0: Timer Function: Timer 1 incremented by clock defined by T1M bit (CKCON.4).
          1: Counter Function: Timer 1 incremented by high-to-low transitions on external input pin (T1).
Bits5-4: T1M1-T1M0: Timer 1 Mode Select.
          These bits select the Timer 1 operation mode.

T1M1	T1M0	Mode
0	0	Mode 0: 13-bit counter/timer
0	1	Mode 1: 16-bit counter/timer
1	0	Mode 2: 8-bit counter/timer with auto-reload
1	1	Mode 3: Timer 1 inactive

Bit3:   GATE0: Timer 0 Gate Control.
          0: Timer 0 enabled when TR0 = 1 irrespective of /INT0 logic level.
          1: Timer 0 enabled only when TR0 = 1 AND /INT0 = logic 1.
Bit2:   C/T0: Counter/Timer Select.
          0: Timer Function: Timer 0 incremented by clock defined by T0M bit (CKCON.3).
          1: Counter Function: Timer 0 incremented by high-to-low transitions on external input pin (T0).
Bits1-0: T0M1-T0M0: Timer 0 Mode Select.
          These bits select the Timer 0 operation mode.

T0M1	T0M0	Mode
0	0	Mode 0: 13-bit counter/timer
0	1	Mode 1: 16-bit counter/timer
1	0	Mode 2: 8-bit counter/timer with auto-reload
1	1	Mode 3: Two 8-bit counter/timers

TMOD: Timer Mode Register

Bit 4 of the Timer Control Register (TCON) can be used to enable or disable Timer 0. It should be noted that TCON is one of the registers that is "bit-addressable" - this means that the bits of the register have been given specific names, and these names can be used to access the individual bits. Bit 4 of TCON is called TR0 - thus, the following line of code:

```
TR0 = 1;
```

will have the same effect as the line:

```
TCON |= 0x10;
```

As with the other registers, the user can still access TCON as a whole and use bit masking to single out a particular bit.

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
						(bit addressable)		0x88

Bit7:   TF1: Timer 1 Overflow Flag.
        Set by hardware when Timer 1 overflows. This flag can be cleared by software but is automatically
        cleared when the CPU vectors to the Timer 1 interrupt service routine.
        0: No Timer 1 overflow detected.
        1: Timer 1 has overflowed.
Bit6:   TR1: Timer 1 Run Control.
        0: Timer 1 disabled.
        1: Timer 1 enabled.
Bit5:   TF0: Timer 0 Overflow Flag.
        Set by hardware when Timer 0 overflows. This flag can be cleared by software but is automatically
        cleared when the CPU vectors to the Timer 0 interrupt service routine.
        0: No Timer 0 overflow detected.
        1: Timer 0 has overflowed.
Bit4:   TR0: Timer 0 Run Control.
        0: Timer 0 disabled.
        1: Timer 0 enabled.
Bit3:   IE1: External Interrupt 1.
        This flag is set by hardware when an edge/level of type defined by IT1 is detected. It can be cleared
        by software but is automatically cleared when the CPU vectors to the External Interrupt 1 service rou-
        tine if IT1 = 1. This flag is the inverse of the /INT1 input signal's logic level when IT1 = 0.
Bit2:   IT1: Interrupt 1 Type Select.
        This bit selects whether the configured /INT1 signal will detect falling edge or active-low level-sensi-
        tive interrupts.
        0: /INT1 is level triggered.
        1: /INT1 is edge triggered.
Bit1:   IE0: External Interrupt 0.
        This flag is set by hardware when an edge/level of type defined by IT0 is detected. It can be cleared
        by software but is automatically cleared when the CPU vectors to the External Interrupt 0 service rou-
        tine if IT0 = 1. This flag is the inverse of the /INT0 input signal's logic level when IT0 = 0.
Bit0:   IT0: Interrupt 0 Type Select.
        This bit selects whether the configured /INT0 signal will detect falling edge or active-low level-sensi-
        tive interrupts.
        0: /INT0 is level triggered.
        1: /INT0 is edge triggered.

TCON: Timer Control Register (Bit Addressable)

Counter/Timers and Overflow

Whether the counter/timer is counting clock ticks or external event signals, the maximum count can only go as high as can fit into the memory assigned to the counter. Thus, the maximum count for a 16-bit counter is 0xFFFF ($65,535_{10}$), the maximum count for a 13-bit counter is 0x1FFF

($8,191_{10}$), and the maximum count for an 8-bit counter is 0xFF (255_{10}). When a counter has reached its maximum, an event called an "overflow" occurs, and the counter will reset to 0 and begin counting again. An overflow can be used to trigger other events, including *interrupts*.

As mentioned above, setting or clearing bits 0 and 1 of TMOD, the Timer Mode Register, can configure Timer 0 as an 8-bit, 13-bit, or 16-bit timer. Timer 0 stores its count in two 8-bit registers - when it is configured as a 16-bit timer, TL0 holds the eight least significant bits and TH0 holds the eight most significant bits. When Timer 0 is configured as a 13-bit timer, TH0 holds the eight most significant bits, and TL0 holds the five least significant bits. For more information about the modes available for Timer 0, see the *C8051F020 Reference Manual*, p. 227.

The following example code shows a function, *Timer0_Init*, which will initialize and configure Timer 0 for use as a 16-bit counter/timer that counts using SYSCLK/12 as a source. Note that the timer should be disabled (turned off) using TR0 before the program sets the count to 0.

```
void Timer0_Init(void);

main()
{
    Timer0_Init();                      /* Configure Timer 0 */

    TR0 = 1;                            /* Enable Timer 0 */
}

void Timer0_Init(void)
{
    CKCON &= ~0x08;                     /* Timer 0 uses SYSCLK/12 as source */
    TMOD &= 0xF0;                       /* Clear bits 0-3 of the Timer Mode Register */
    TMOD |= 0x01;                       /* Set Timer 0 to Mode 1 (16-bit counter/timer) */
    TR0 = 0;                            /* Disable Timer 0 */
    TMR0 = 0;                           /* Clear all 16 bits of Timer 0's count */
//  TL0 = 0; TH0 = 0;                   /* Or Clear low & high bytes of Timer 0 separately */
}
```

Example 4.3 - Configuring and Initializing Timer 0

Counting External Events

In addition to counting clock ticks based on the system clock, Timer 0 can count external events, such as a logic transition on an input pin. A counter/timer is incremented on each high-to-low transition at the selected input pin. Events with a frequency of up to one-fourth the system clock's frequency can be counted. The input signal need not be periodic, but it should be held at a given level for at least two full system clock cycles to ensure the level is sampled.

The Timer Mode Register (TMOD) is used to set up Timer 0 to count external pulses. Note that when *C/T0* (bit 2 of TMOD) is set to logic 1, high-to-low transitions at the selected input pin (T0) increment the timer register. (See Figures 17.3 and 22.2 in the C8051 Reference Manual.)

In addition, bit 1 of XBR1 (Port I/O Crossbar Register 1), which is the T0 Input Enable Bit, must be set.

Interrupts

In a real-time system, much of the interaction between the microcontroller and the various peripheral devices is through interrupts. When an abnormal condition arises or when a peripheral device requires some service from the microcontroller, an interrupt can be generated to request attention from the CPU. This way, the CPU does *not* have to periodically check the status (poll) of various devices to query whether or not they need servicing, which economizes the CPU's processing time since devices may require only *very infrequent* servicing.

The concept of an interrupt in a microcontroller may be understood by considering the following situation. When the telephone rings, you temporarily suspend whatever activity you are engaged in, answer the phone, have the phone conservation, end the phone call, and then resume your activity at the point where you had suspended it. The *ringing* of the telephone is analogous to an interrupt (*setting* of the *interrupt flag*); the *answering* of the telephone and *conversing* is analogous to invoking the *interrupt service routine (ISR)*, the code that is invoked by the interrupt (also called the *interrupt handler*). Saying good-bye indicates you are done with this call and the caller doesn't need to call you back, this is analogous to *clearing* the *interrupt flag*. Hanging up the phone and returning to your original task is analogous to ending the interrupt and returning to code that was running when the interrupt occurred. The program then proceeds until the next interrupt occurs.

An interrupt can be *enabled* (the interrupt calls are serviced) or *disabled* (the interrupt calls are not answered) by appropriate instructions. This is analogous to the feature where the ringing of the telephone can be switched on or off. Interrupts can also be *nested,* i.e., another interrupt can occur while a *previous* interrupt is being serviced. This is analogous to answering a call waiting on the telephone in the midst of another conversation. Interrupts can also be assigned *priority,* i.e., the order in which they are serviced when two or more interrupts occur at once.

Since the CPU executes instructions in a sequential fashion, it may be necessary on occasion to execute sets of instructions when requested by certain peripheral devices or on the occurrence of certain conditions. These requests are asynchronous with the execution of the main program. Thus *interrupts* provide a way to temporarily suspend normal program execution so that the CPU can be freed to service these requests. After an interrupt has been serviced, the main program resumes as if there had been no interruption.

Interrupts on the C8051

The C8051F020 has 22 kinds of interrupts available for use, each of which is triggered by a different event. In the following section, we will be discussing the use of the *Timer 0 Overflow Interrupt*.

Timer 0 Overflow Interrupt

As was mentioned above, when a timer overflows (reaches its maximum count), an overflow event occur and the timer reset to 0 and resumes counting. The overflow event can be used to activate an interrupt. In this example, when Timer 0 overflows, it can trigger the Timer 0 Overflow Interrupt. An interrupt is triggered when both its *interrupt enable* and *interrupt flag* are set. The interrupt enable "turns on" the interrupt, that is, it makes that particular interrupt available for use. The interrupt flag is set automatically when the event that triggers that interrupt occurs.

The interrupt enable for Timer 0 is bit 1 in the IE (Interrupt Enable) register. Because the IE register is bit-addressable, each of its bits have names which can be used to access that bit directly. The name for bit 1, the interrupt enable for Timer 0, is ET0. Bit 7 of IE is the global interrupt enable bit - this bit (called EA) must be set in order for any interrupt to be used.

7	6	5	4	3	2	1	0
EA	IEGF0	ET2	ES0	ET1	EX1	**ET0**	EX0

Bit 7: EA: Enable all interrupts
   This bit globally enables/disables all interrupts. When set to '0', individual interrupt mask settings are overridden.
      0: Disable all interrupt sources
      1: Enable each interrupt according to its individual mask setting
Bit 6: IEGF0: General purpose flag 0
      This is a general purpose flag for use under software control
Bit 5: ET2: Enable Timer 2 Interrupt
      This bit sets the masking of the Timer 2 interrupt
      0: Disable Timer 2 interrupt
      1: Enable interrupt requests generated by the TF2 flag (T2CON.7)
Bit 4: ES0: Enable UART0 Interrupt
      This bits sets the masking of the UART0 interrupt
      0: Disable UART0 interrupt
      1: Enable UART0 interrupt
Bit 3: ET1: Enable Timer 1 Interrupt
      This bit sets the masking of the Timer 1 interrupt
      0: Disable all Timer 1 interrupt
      1: Enable interrupt requests generated by the TF1 flag (TCON.7)
Bit 2: EX1: Enable External Interrupt 1
      This bit sets the masking of external interrupt 1
      0: Disable external interrupt 1
      1: Enable interrupt requests generated by the /INT1 pin
Bit 1: ET0: Enable Timer 0 Interrupt
      This bit sets the masking of the Timer 0 interrupt
      0: Disable all Timer 0 interrupt
      1: Enable interrupt requests generated by the TF0 flag (TCON.5)
Bit 0: EX0: Enable External Interrupt 0
      This bit sets the masking of external interrupt 0
      0: Disable external interrupt 0
      1: Enable interrupt requests generated by the /INT0 pin

IE: Interrupt Enable Register (Bit Addressable)

The Timer 0 interrupt flag is bit 5 of the TCON (Timer Control Register) - bit 5 of TCON has the name TF0.

In summary, when Timer 0 overflows, its interrupt flag, TF0 (bit 5 in TCON) is set. If the Timer 0 interrupt has been enabled by setting the interrupt enable, ET0 (bit 1 in IE), then the interrupt will pause execution of the main program and launch execution of the associated interrupt service routine (ISR). Once the ISR has completed, the interrupt flag will be cleared automatically, and execution of the main program will resume.

Interrupt Service Routines

An interrupt service routine is a function that will execute when its associated interrupt occurs. The specification of which interrupt will launch the ISR is given in the ISR's function declaration, which is of the form:

```
void ISR_name(void) __interrupt X
  {
      // Put code here
  }
```

where ISR_name is the name of the function, and X is a number from 0 to 21, which uniquely identifies the interrupt. The interrupt numbers have been pre-assigned and are the same as the priority order numbers - these can be found in *Table 4.2*.

Table 4.2 - Interrupts and Priority Order

Interrupt Source	Interrupt Vector	Priority Order	Interrupt Source	Interrupt Vector	Priority Order
Reset	0x0000	Top	Comparator 0 Falling Edge	0x0053	10
External Interrupt 0 (/INT0)	0x0003	0	Comparator 0 Rising Edge	0x005B	11
Timer 0 Overflow	0x000B	1	Comparator 1 Falling Edge	0x0063	12
External Interrupt 1 (/INT1)	0x0013	2	Comparator 1 Rising Edge	0x006B	13
Timer 1 Overflow	0x001B	3	Timer 3 Overflow	0x0073	14
UART0	0x0023	4	ADC0 End of Conversion	0x007B	15
Timer 2 Overflow (or RXF2)	0x002B	5	Timer 4 Overflow	0x0083	16
Serial Peripheral Interface	0x0033	6	ADC1 End of Conversion	0x008B	17
SMBus Interface	0x003B	7	External Interrupt 6	0x0093	18
ADC0 Window Comparator	0x0043	8	External Interrupt 7	0x009B	19
Programmable Counter Array	0x004B	9	UART1	0x00A3	20
			External Crystal OSC Ready	0x00AB	21

Thus, if you want to call the function *Timer0_ISR* when a Timer 0 overflow triggers an interrupt, your code would contain the following lines:

```
void Timer0_ISR(void) __interrupt 1
{
    // Put code here
}
```

The following example code builds on a previous example to show how to initialize and configure Timer 0 for use as a 16-bit counter/timer, and also to enable the Timer 0 Overflow Interrupt.

```
void Timer0_Init(void);
void Timer0_ISR(void) __interrupt 1;

main()
{
    Timer0_Init();                       /* Configure Timer 0 */

    IE |= 0x02;                          /* Enable Timer 0 interrupts */
    EA = 1;                              /* Enable global interrupts */

    TR0 = 1;                             /* Enable Timer 0 */
}

void Timer0_Init(void)
{
    CKCON &= ~0x08;                      /* Timer 0 uses SYSCLK/12 as source */
    TMOD &= 0xF0;                        /* Clear bits 0-3 of the Timer Mode Register */
    TMOD |= 0x01;                        /* Set Timer 0 to Mode 1 (16-bit counter/timer) */
    TR0 = 0;                             /* Disable Timer 0 */
    TMR0 = 0;                            /* Clear Timer 0's count */
//  TL0 = 0; TH0 = 0;                    /* Or Clear the bytes of Timer 0's count separately */
}

void Timer0_ISR(void) __interrupt 1
{
                                         // Put code here
}
```

Example 4.4 - Configuring and enabling the Timer 0 Overflow Interrupt

Programmable Counter Array

The Programmable Counter Array (PCA0) provides enhanced timer functionality while requiring less CPU intervention than the standard 8051 counter/timers. PCA0 consist of:

1. A dedicated 16-bit counter/timer whose count is stored in two 8-bit registers, PCA0H (high byte) and PCA0L(low byte)
2. A mode register, PCA0MD
3. A control register, PCA0CN
4. Five capture/compare modules, each consisting of:
   - A 16-bit capture/compare data register consisting of two 8-bit registers, PCA0CPHn (for high byte) and PCA0CPLn (for low byte), n=0:4
   - A capture/compare module mode register, PCA0CPMn, n=0:4
   - An associated I/O line (CEXn) routed through the crossbar to Port I/O when enabled, n=0:4
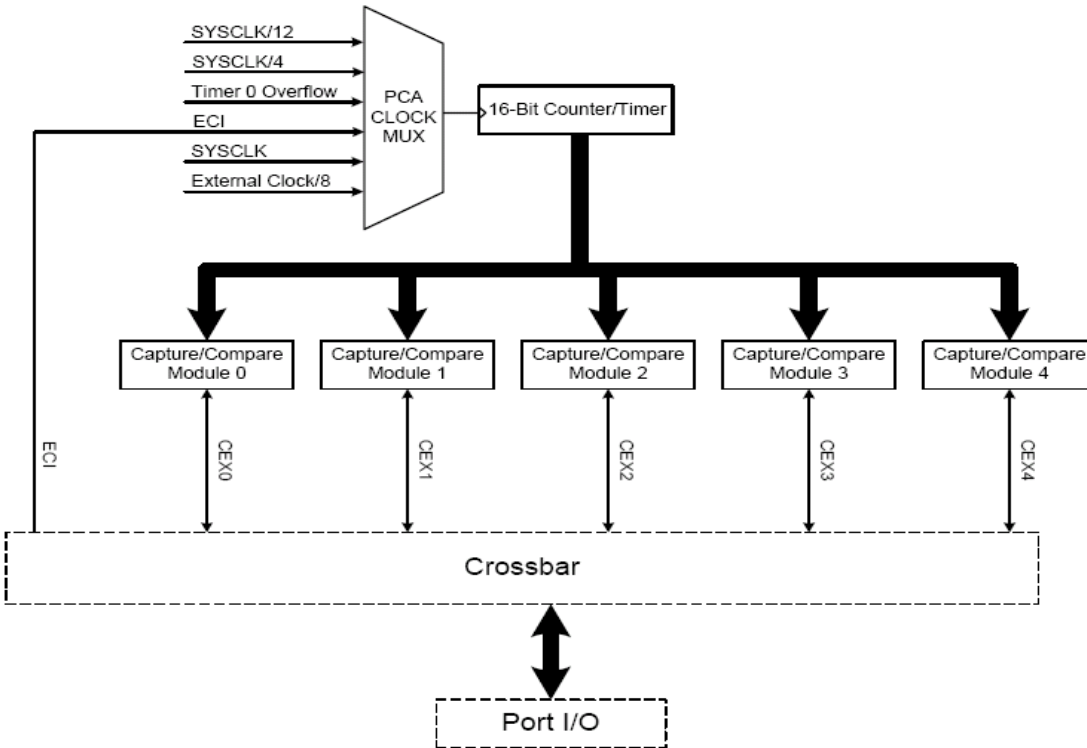
Figure 4.3 - PCA Block Diagram

The heart of the PCA timer system is the 16-bit free-running counter - depending on how it is configured, this timer can count one of six possible inputs, including the system clock (SYSCLK), SYSCLK/4, or SYSCLK/12. The count takes place using two 8-bit registers, PCA0H, which contains the high byte (eight most significant bits) of the count, and PCA0L, which contains the low byte (eight least significant bits). As with the other C8051 counter/timers, when the count has reached its maximum value, an overflow occurs, and the counter reset to 0 and resumes counting. Because the PCA timer is a 16-bit counter, it can count up to 0xFFFF, or $65,535_{10}$.

The five capture/compare modules can be used to contain values which will be compared to the PCA0 count - when the count matches the value in a module, then an event may be triggered, such as setting an output pin to logic 1. Like the PCA0, each module stores its contents in two associated registers, one of which holds the high byte (PCA0CPH*n*), and the other holds the low byte (PCA0CPL*n)*, where *n* = 0-4.
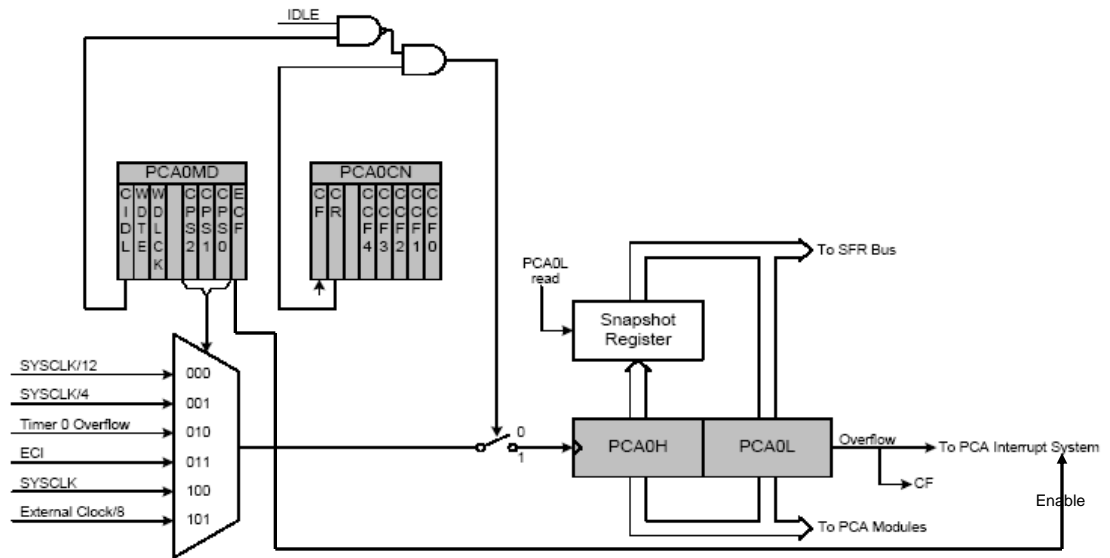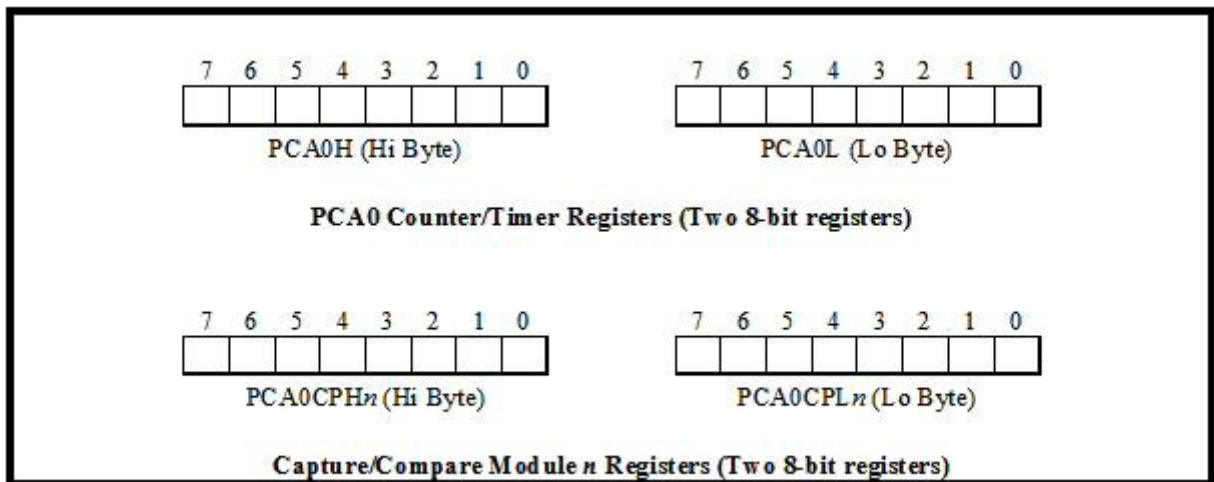
Figure 4.4 - PCA Counter/Timer Block Diagram

Each capture/compare module may be configured to operate independently in one of six modes - we will focus on the 16-bit Pulse Width Modulator Mode. (For more information on the PCA and the modes available to the capture/compare modules, refer to p. 249 in the *C8051F020 Reference Manual.*)

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
ECP1R	ECP1F	ECP0R	ECP0F	EPCA0	EWADC0	ESMB0	ESPI0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
								0xE6

Bit7:   ECP1R: Enable Comparator1 (CP1) Rising Edge Interrupt.
          This bit sets the masking of the CP1 interrupt.
          0: Disable CP1 Rising Edge interrupt.
          1: Enable interrupt requests generated by the CP1RIF flag (CPT1CN.5).
Bit6:   ECP1F: Enable Comparator (CP1) Falling Edge Interrupt.
          This bit sets the masking of the CP1 interrupt.
          0: Disable CP1 Falling Edge interrupt.
          1: Enable interrupt requests generated by the CP1FIF flag (CPT1CN.4).
Bit5:   ECP0R: Enable Comparator0 (CP0) Rising Edge Interrupt.
          This bit sets the masking of the CP0 interrupt.
          0: Disable CP0 Rising Edge interrupt.
          1: Enable interrupt requests generated by the CP0RIF flag (CPT0CN.5).
Bit4:   ECP0F: Enable Comparator0 (CP0) Falling Edge Interrupt.
          This bit sets the masking of the CP0 interrupt.
          0: Disable CP0 Falling Edge interrupt.
          1: Enable interrupt requests generated by the CP0FIF flag (CPT0CN.4).
Bit3:   EPCA0: Enable Programmable Counter Array (PCA0) Interrupt.
          This bit sets the masking of the PCA0 interrupts.
          0: Disable all PCA0 interrupts.
          1: Enable interrupt requests generated by PCA0.
Bit2:   EWADC0: Enable Window Comparison ADC0 Interrupt.
          This bit sets the masking of ADC0 Window Comparison interrupt.
          0: Disable ADC0 Window Comparison Interrupt.
          1: Enable Interrupt requests generated by ADC0 Window Comparisons.
Bit1:   ESMB0: Enable System Management Bus (SMBus0) Interrupt.
          This bit sets the masking of the SMBus interrupt.
          0: Disable all SMBus interrupts.
          1: Enable interrupt requests generated by the SI flag (SMB0CN.3).
Bit0:   ESPI0: Enable Serial Peripheral Interface (SPI0) Interrupt.
          This bit sets the masking of SPI0 interrupt.
          0: Disable all SPI0 interrupts.
          1: Enable Interrupt requests generated by the SPIF flag (SPI0CN.7).

EIE1: Extended Interrupt Enable 1

Pulsewidth modulation

Pulsewidth modulation (PWM) refers to the procedure in which the width of a pulse in a rectangular waveform is varied in correspondence to some information. Each PCA0 module may be operated in 16-bit PWM mode, which will allow it to produce a pulsed signal on an output pin. In this mode, the 16-bit capture/compare module defines the number of PCA0 clock ticks for the low time of the PWM signal. When the PCA0 counter matches the contents of module n, the output on CEXn is asserted high; when the counter overflows, CEXn is asserted low.

Bit 6 of the PCA Control Register, PCA0CN, is used to enable the PCA0 counter/timer:

7	6	5	4	3	2	1	0
CF	**CR**	-	CCF4	CCF3	CCF2	CCF1	CCF0

Bit 7: CF: PCA Counter/Timer Overflow Flag.
> Set by hardware when the PCA0 Counter/Timer overflows from 0xFFFF to 0x0000. When the Counter/Timer Overflow (CF) interrupt is enabled, setting this bit causes the CPU to vector to the CF interrupt service routine. This bit is not automatically cleared by hardware and must be cleared by software.

Bit 6: CR: PCA0 Counter/Timer Run Control.
> This bit enables/disables the PCA0 Counter/Timer.
> 0: PCA0 Counter/Timer disabled.
> 1: PCA0 Counter/Timer enabled.

Bit 5: UNUSED. Read = 0b, Write = don't care.

Bit 4: CCF4: PCA0 Module 4 Capture/Compare Flag.
> This bit is set by hardware when a match or capture occurs. When the CCF interrupt is enabled, setting this bit causes the CPU to vector to the CCF interrupt service routine. This bit is not automatically cleared by hardware and must be cleared by software.

Bit 3: CCF3: PCA0 Module 3 Capture/Compare Flag.
> This bit is set by hardware when a match or capture occurs. When the CCF interrupt is enabled, setting this bit causes the CPU to vector to the CCF interrupt service routine. This bit is not automatically cleared by hardware and must be cleared by software.

Bit 2: CCF2: PCA0 Module 2 Capture/Compare Flag.
> This bit is set by hardware when a match or capture occurs. When the CCF interrupt is enabled, setting this bit causes the CPU to vector to the CCF interrupt service routine. This bit is not automatically cleared by hardware and must be cleared by software.

Bit 1: CCF1: PCA0 Module 1 Capture/Compare Flag.
> This bit is set by hardware when a match or capture occurs. When the CCF interrupt is enabled, setting this bit causes the CPU to vector to the CCF interrupt service routine. This bit is not automatically cleared by hardware and must be cleared by software.

Bit 0: CCF0: PCA0 Module 0 Capture/Compare Flag.
> This bit is set by hardware when a match or capture occurs. When the CCF interrupt is enabled, setting this bit causes the CPU to vector to the CCF interrupt service routine. This bit is not automatically cleared by hardware and must be cleared by software.

PCA0CN: PCA Control Register (Bit Addressable)

The PCA0 Mode Register, PCA0MD, is used to set the mode for PCA0:

7	6	5	4	3	2	1	0
CIDL	-	-	-	**CPS2**	**CPS1**	**CPS0**	**ECF**

Bit 7: CIDL: PCA0 Counter/Timer Idle Control
    Specifies PCA0 behavior when CPU is in idle mode
    0: PCA0 continues to function normally while the system controller is in idle mode
    1: PCA0 operation is suspended while the system controller is in idle mode
Bits 6-4: UNUSED
Bits 3-1: CPS2-CPS0: PCA0 Counter/Timer Pulse Select
    These bits select the timebase source for the PCA0 counter:

CPS2	CPS1	CPS0	Timebase
0	0	0	System clock/12
0	0	1	System clock/4
0	1	0	Timer 0 overflow
0	1	1	High-to-low transition on EC1
1	**0**	**0**	**System clock (SYSCLK)**
1	0	1	External clock/8
1	1	0	Reserved
1	1	1	Reserved

Bit 0: ECF: PCA Counter/Timer Overflow Interrupt Enable
    This bit sets the masking of the PCA0 Counter/Timer Overflow (CF) interrupt
    0: Disable the CF interrupt
    1: Enable a PCA0 Counter/Timer Overflow interrupt request when CF (PCA0CN.7) is set

PCA0MD: PCA0 Mode Register

Each capture/compare module has an associated Capture/Compare Mode Register, called PCA0CPM*n*, where *n* = 0-4, one for each module. Each of the Capture/Compare Mode Registers can be used to configure the corresponding module for 8-bit or 16-bit PWM, to enable or disable the module's comparator function, and to enable Pulse Width Modulation mode. 16-bit PWM mode is enabled by setting the ECOM*n*, PWM*n*, and PWM16*n* bits in the PCA0CPM*n* register.
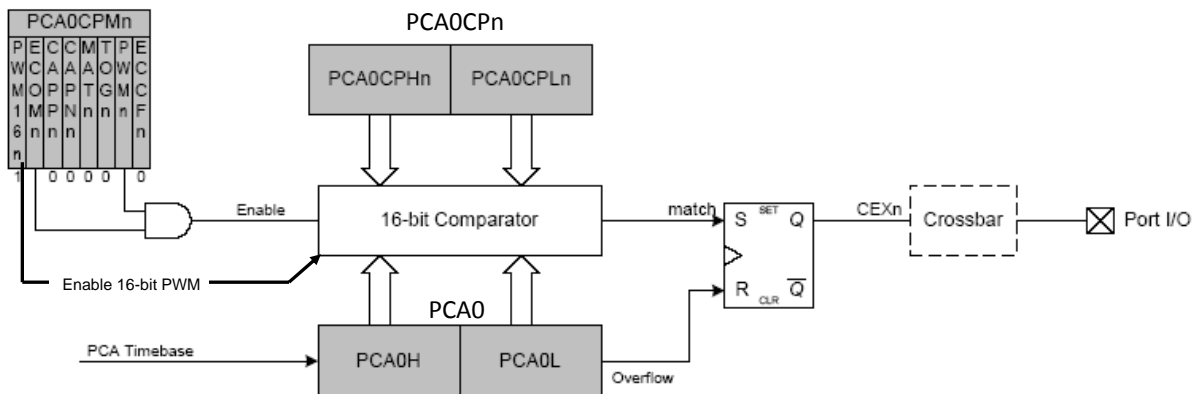

Figure 4.5 - PCA 16-Bit PWM Mode

7	6	5	4	3	2	1	0
PWM16*n*	**ECOM***n*	CAPP*n*	CAPN*n*	MAT*n*	TOG*n*	**PWM***n*	ECCF*n*

Bit 7: PWM16n: 16-bit Pulse Width Modulation Enable
    This bit selects 16-bit mode when Pulse Width Modulation mode is enabled (PWMn = 1)
    0: 8-bit PWM selected
    1: 16-bit PWM selected

Bit 6: ECOMn: Comparator Function Enable
    This bit enables/disables the comparator function for PCA0 module n
    0: Disabled
    1: Enabled

Bit 5: CAPPn: Capture Positive Function Enable
    This bit enables/disables the positive edge capture for PCA0 module n
    0: Disabled
    1: Enabled

Bit 4: CAPNn: Capture Negative Function Enable
    This bit enables/disables the negative edge capture for PCA0 module n
    0: Disabled
    1: Enabled

Bit 3: MATn: Match Function Enable
    This bit enables/disables the match function for PCA0 module n. When enabled, matches of the PCA0 counter with a
    module's capture/compare register cause the CCFn bit in PCA0MD register to be set to logic 1
    0: Disabled
    1: Enabled

Bit 2: TOGn: Toggle Function Enable
    This bit enables/disables the toggle function for PCA0 module n. When enabled, matches of the PCA0 counter with a
    module's capture/compare register cause the logic level on the CEXn pin to toggle. If the PWMn bit is also set to logic 1,
    the module operates in Frequency Output Mode
    0: Disabled
    1: Enabled

Bit 1: PWMn: Pulse Width Modulation Mode Enable
    This bit enables/disables the PWM function for PCA0 module n. When enabled, a pulse width modulated signal is
    output on the CEXn pin. 8-bit PWM is used if PWM16n is logic 0; 16-bit mode is used if PWM16n is logic 1. If the
    TOGn bit is also set, the module operates in Frequency Output Mode.
    0: Disabled
    1: Enabled

Bit 0: ECCFn: Capture/Compare Flag Interrupt Enable
    This bit sets the masking of the Capture/Compare Flag (CCFn) interrupt
    0: Disable the CCFn interrupts
    1: Enable a Capture/Compare Flag interrupt request when CCFn is set

PCA0CPM*n***: PCA0 Capture/Compare Mode Registers**

As was noted above, each capture/compare module has its own associated I/O line (CEX*n*), which is routed through the Crossbar to Port I/O when enabled. The Port I/O Register XBR0 is used to route the CEX*n* I/O lines to port pins.

A segment of C code is shown in *Example 4.5* illustrating the use of PCA0 and capture/compare module 0 (CCM0) to implement Pulse Width Modulation.

```
#include <c8051_SDCC.h>
#include <stdlib.h>
#include <stdio.h>

int CEX0_PW = 20000;                        /* designed PW at cex0 20,000=0x4E20*/

void Port_Init(void);
void PCA_Init(void);

void main(void)
{
   int temp0_lo_to_hi;

   Sys_Init();                              /* Initialize the C8051 board */
   Port_Init();                             /* Set port(s) for input/output */
   PCA_Init();                              /* Configure the PCA */

   while (1)
   {
      temp0_lo_to_hi = 0xFFFF - CEX0_PW;    /* = 0xB1DF = 45,535 */
      PCA0CP0 = temp0_lo_to_hi;             /* = 0xB1DF */
   }
}

void Port_Init(void)
{
   P0MDOUT |= 0x04;                         /* Set pin 2 of Port 0 to push-pull */
   XBR0 = 0x0C;                             /* Configure crossbar to use CEX0 */
}

void PCA_Init(void)
{
   PCA0CPM0 = 0xC2;                         /* CCM0 in 16-bit compare mode */
   PCA0CN = 0x40;                           /* Enable PCA counter */
}
```

Example 4.5 - Pulse Width Modulation implemented using the PCA (Programmable Counter Array)

Remember that the output pin is set high when the PCA0 count matches the contents of capture/compare module 0, so in order to output the correct pulsewidth, the contents of module 0 should be set to 0xFFFF minus the desired pulsewidth.

Within Port_Init(), pin 2 of Port 0, which will be the output pin used by CEX0, is set to push-pull mode, and the Crossbar register XBR0 is set to route CEX0 to a Port pin. When *Capture/Compare Module 0 (CCM0)* makes a successful compare, it changes the state of pin *P0.2* to logic *high*. When *PCA0* overflows, it clears pin *P0.2* (sets it *LOW*).

The 16-bit counter rolls over to zero every 2^{16} (65536) clock cycles meaning that *PCA0* will set P0.2 low every 65536 clock cycles. In the main loop, the contents of CCM0 is set to 45535 = 65535 – 20000 or 0xFFFF - 20000. Therefore, P0.2 is set *High* whenever the free-running counter equals 45,535. This process will repeat itself as long as the program is running.

The waveform at pin *P0.2* for *Example 4.5* can be viewed using an oscilloscope and will be as shown in *Figure 4.6*.
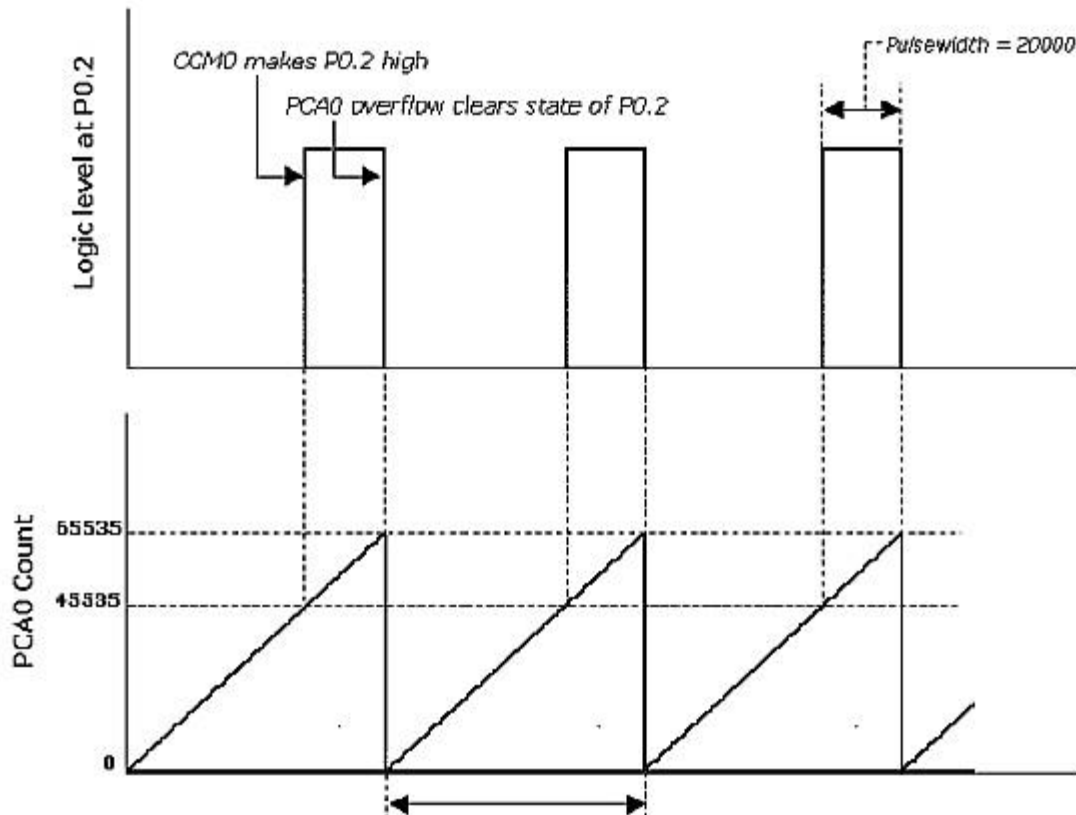
Figure 4.6 - Pulsewidth Modulation Waveform

PCA Overflow Interrupt Using 'Preset' Technique

As with Timer 0 (see "Timer 0 Overflow Interrupt" on page 41), when the PCA overflows, it can trigger an interrupt. Using an interrupt service routine, we can control how often to update the compare value, which represents the magnitude of the desired pulsewidth. We may simply call interrupt service routine every 2^x clock counts using x-bit counter. Or we may call interrupt service routine every y clock counts, where $y \leq 2^x$, using 'preset' values.

The C code shown in *Example 4.6* uses PCA0 and capture/compare module 0 (CCM0) to implement Pulse Width Modulation, but uses the PCA overflow interrupt (interrupt 9) to regulate the frequency with which the compare value is updated. Note that in the code, every time interrupt service routine is called, PCA0 counter is pre-set to 21,300 so that interrupt service routine is called every 2 ms using System Clock. This 'preset' technique can also be used in Timer 0 Overflow Interrupt to regulate the time interval in which interrupt service routine is called. **See Java Applet for PCA on LMS.**
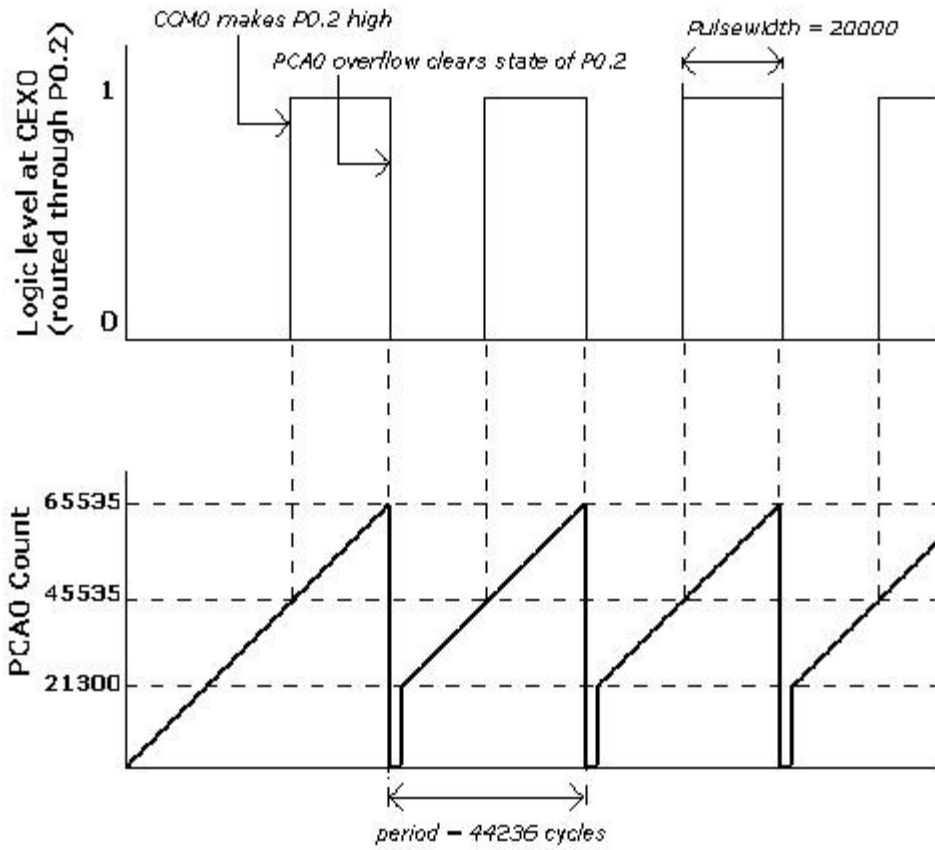
Figure 4.7 - Pulsewidth Modulation Waveform

```
/*This is sample code. It uses a 2.5 ms period; NOT 20 ms that the lab exercise requires*/

#include <c8051_SDCC.h>
#include <stdlib.h>
#include <stdio.h>

int CEX0_PW = 10000;
void Port_Init(void);
void PCA_Init(void);
void PCA_ISR (void) __interrupt 9;

void main(void)
{
   char input;
   unsigned int tmp0_lo_to_hi;

   Sys_Init();                        // Initialize the C8051 board
   Port_Init();                       // Set port(s) for input/output
   PCA_Init();                        // Configure the PCA

   while (1)
   {
      input = (char)getchar();        // Wait for a key to be pressed
      if (input == 'r')               // If 'r' is pressed by the user
      {
         if (SERVO_PW < PW_RIGHT)
         CEX0_PW = CEX0_PW + 100;     // Increase the pulsewidth by 100
      }
      else if (input == 'l')          // If 'l' is pressed by the user
      {
         if (SERVO_PW > PW_LEFT)
         CEX0_PW = CEX0_PW - 100;     // Decrease the pulsewidth by 100
      }
      tmp0_lo_to_hi = 0xFFFF - CEX0_PW;
      // Set next compare value to pulsewidth
      PCA0CP0 = tmp0_lo_to_hi;        // Load 16 bits for CEX0 on P0.2
      PCA0CP2 = CEX0_PW + 10240;      // Load 16 bits for CEX2 on P0.4 (inverse Duty Cycle)
      printf("Pulsewidth = %d\r",CEX0_PW);
   }
}
void Port_Init(void)
{
   P0MDOUT |= 0x1C;                   // Set pins 2 - 4 of Port 0 to push-pull
   XBR0 = 0x1C;                       // Configure crossbar to use CEX0 - CEX2, and UART0
}
void PCA_Init(void)
{
   PCA0MD = 0x89;                     // Enable CF interrupt & SYSCLK (not SYSCLK/12)
   PCA0CPM0 = PCA0CPM2 = 0xC2;        // CCM0 & CCM2 in 16-bit compare mode
   PCA0CN = 0x40;                     // Enable PCA counter
   EIE1 |= 0x08;                      // Enable PCA interrupt
   EA = 1;                            // Enable global interrupts
}
void PCA_ISR(void) __interrupt 9
{
   if (CF)
   {
      CF = 0;                         // Clear overflow flag
      PCA0 = 10240;                   // (or PCA0 = 0x2800) Start count for 2.5 ms
   }                                  // NOTE: motors on Smartcar need a 20 ms period
                                      // DO NOT USE THIS VALUE!
   PCA0CN &= 0x40;                    // Handle other PCA interrupt sources
}
```

Example 4.6 - Pulse Width Modulation implemented using the PCA (Programmable Counter Array) and Interrupts

The C8051 A/D Converter[*]

Analog-to-digital (A/D) conversion is a process whereby a continuous-valued voltage is converted into an integer that is proportional to the voltage. For example, the C8051F020 has an 8-bit and a 12-bit A/D converter. The remainder of this section will apply specifically to the 8-bit converter, although the concepts can also be applied to the 12-bit converter. The 8-bit A/D converter (ADC1) can be programmed to acquire analog input from Port 1 through any of its 8 external pins (*P1.0 - P1.7*). Since the converter has 8 bits of resolution, the result of an A/D conversion will be an integer in the range 0 to 255, and this integer result has a linear relationship to the original analog input voltage as will be described.
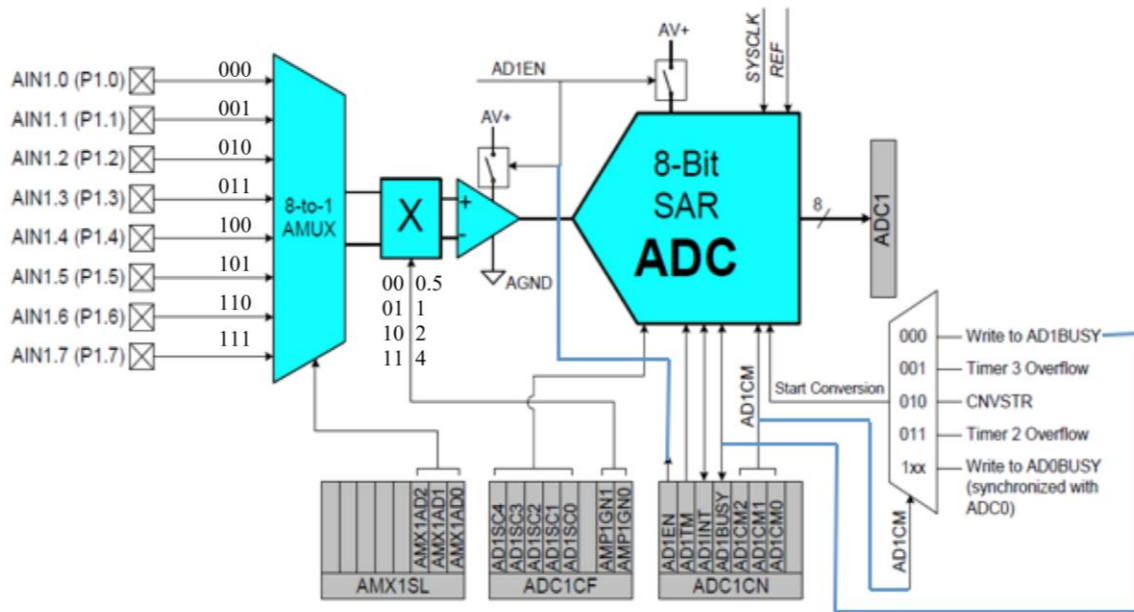

Figure 4.8 – A/D Converter Function Block Diagram

Conceptually, the A/D converter examines the voltage on an analog input pin and tries to determine approximately how far this voltage lies between the allowable extreme voltages V_{RL} *(Voltage Reference Low)* and V_{RH} *(Voltage Reference High)*. The A/D conversion result will then be in proportion to where the input voltage falls between the V_{RL} and V_{RH} extremes. For example, if $V_{RL} = 0$V and $V_{RH} = 2.4$V, and if the voltage at the input pin $= 1.2$V, then 1.2V is half or 50% of the way between 0 and 2.4V. Thus the A/D conversion result, (which is always a discrete number between 0 and 255), will be 50% of its maximum value, or 128.

For the C8051, the V_{RL} will always be 0 Volts, and V_{RH} (which will generally be referred to as *Vref*) will equal 2.4 Volts. Input voltages in the range from 0 to Vref will map to a digital number from 0x00 to 0xFF (0_{10} to 255_{10}). In general, if the voltage at *P1.x* is *less than* or equal to 0 V,

---

[*] Chapter 7 of C8051 User Manual

then the conversion result will saturate at 0. Similarly, if the voltage at *P1.x* is *greater than* or equal to the voltage at *Vref*, then the conversion result will saturate at 255. Otherwise, the conversion result will be proportional to where the voltage at P1.x lies between *0 V* and *Vref*. You should try to avoid inputting voltages of greater than Vref or less than 0 V. The equation below describes the relationship between the reference voltage, a voltage on one of the 8 analog inputs *P1.x (x = 0, 1... 7)* and the A/D result. Depending upon how the A/D converter has been configured by program instructions, the user can select which of the 8 pins will be used for input. Note that the *Gain* can be set to determine how much the output signal should be amplified (see the information on the ADC1 Configuration Register, ADC1CF, on page 57); for this course, the *Gain* will normally be set to one. Configuration of the A/D converter will be discussed in the next section.

$$ADresult = \text{ floor} \left[256 \cdot \frac{V_{P1.x} \cdot Gain}{V_{Ref}} \right] \quad \text{for} \quad 0 \leq (V_{P1.x} \cdot Gain) < V_{Ref}$$

Configuring the A/D converter

In order to perform an A/D conversion using the C8051, you will need to set and access a number of SFRs (Special Function Registers). The pins on Port 1 can serve as analog inputs to the ADC1 analog MUX. A Port pin is configured as an Analog Input by writing a logic 0 to the associated bit in the P1MDIN register. All Port pins default to a Digital Input mode.

7	6	5	4	3	2	1	0
P1MDIN.7	P1MDIN.6	P1MDIN.5	P1MDIN.4	P1MDIN.3	P1MDIN.2	P1MDIN.1	P1MDIN.0

Bits 7-0: P1MDIN.[7:0]: Port 1 Input Mode Bits
      0: Port pin is configured in Analog Input mode
      1: Port pin is configured in Digital Input mode

P1MDIN: Port 1 Input Mode Register

You may recall from Table 4.1 that Port 1 has an associated data register, P1, and an associated output mode register, P1MDOUT. In order to use a pin of Port 1 for analog input, a logic 1 must be sent to its corresponding pin in P1, and that pin must be set to "open drain" by writing a logic 0 to the corresponding pin in P1MDOUT. Thus, in order to configure pin 0 of Port 1 for analog input, the following lines should appear in your code:

```
P1MDIN &= ~0x01;        // Set pin 0 of Port 1 for analog input
P1MDOUT &= ~0x01;       // Set pin 0 of Port 1 as "open drain"
P1 |= 0x01;             // Send logic 1 to pin 0 of Port 1
```

The next step is to be sure that Vref will be set to the 2.4 V internal reference voltage. This is accomplished using the Reference Control Register (REF0CN). Bits 0, 1, and 3 of REF0CN will be used to ensure that Vref is set correctly.

Table 4.3 - Voltage Reference Electrical Characteristics

VDD = 3.0 V, AV+ = 3.0 V, -40°C to +85°C unless otherwise specified

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS
INTERNAL REFERENCE (REFBE = 1)					
Output Voltage	25°C ambient	2.36	2.43	2.48	V
VREF Short-Circuit Current				30	mA
VREF Temperature Coefficient			15		ppm/°C
Load Regulation	Load = 0 to 200 μA to AGND		0.5		ppm/μA
VREF Turn-on Time 1	4.7μF tantalum, 0.1μF ceramic bypass		2		ms
VREF Turn-on Time 2	0.1μF ceramic bypass		20		μs
VREF Turn-on Time 3	no bypass cap		10		μs
EXTERNAL REFERENCE (REFBE = 0)					
Input Voltage Range		1.00		(AV+)-3	V
Input Current			0	1	μA

Table 4.3 shows 2ms delay in Voltage Reference Start-up. **This is a long delay**; no ADC conversions can be done before this. **If attempted, the conversion value will return as 255** since the input voltage will be compared to a reference that is close to 0V and being larger than the reference results in a saturated voltage and the maximum value possible.

Figure 4.9 shows the configuration requirements for the analog conversion reference voltage. Before any analog-to-digital or digital-to-analog conversions can take place, the 1.2V band-gap bias voltage must be enabled (BIASE = 1). In addition to this voltage, a 2nd reference voltage must be provided. This 2nd reference can be created 2 ways: by attaching an external source, or by generating it internally using the 1.2V band-gap and an amplifier with a gain of 2 to supply a fixed 2.4V reference voltage. In all Embedded Control labs using the A/D converter, the internal ×2 amp in combination with the band-gap source will provide the required reference voltage, but students should understand how other reference voltage values may be used.

Note that the value of the reference voltage determines the range of the converter. The largest positive voltage applied to the converter must be less than the reference voltage. That means that with the internally generated 2.4V reference the maximum applied voltages must be less than 2.4V. The minimum voltage is always 0V, or ground. Also note that if the ×2 amp is not enabled (REFBE = 0), the 1.2V band-gap potential is NOT connected to the VREF line (effectively an open switch on the amp output) and an external voltage supply must be used to provide the reference.
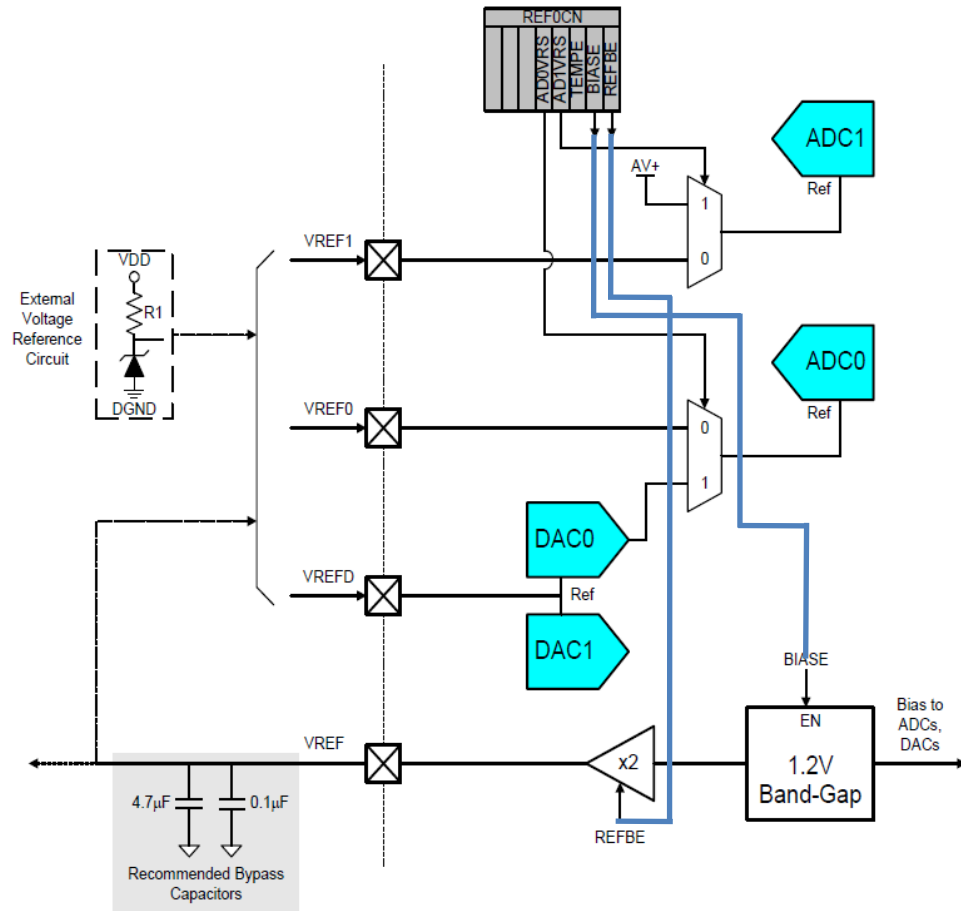
Figure 4.9 – Voltage Reference Sources Functional Block Diagram

7	6	5	4	3	2	1	0
-	-	-	AD0VRS	**AD1VRS**	TEMPE	**BIASE**	**REFBE**

Bits 7-5: UNUSED

Bit 4: AD0VRS: ADC0 Voltage Reference Select
     0: ADC0 voltage reference from VREF0 pin
     1: ADC0 voltage reference from DAC0 output

Bit 3: AD1VRS: ADC1 Voltage Reference Select
     0: ADC1 voltage reference from VREF1 pin
     1: ADC1 voltage reference from AV+

Bit 2: TEMPE: Temperature Sensor Enable Bit
     0: Internal temperature sensor OFF
     1: Internal temperature sensor ON

Bit 1: BIASE: ADC/DAC Bias Generator Enable Bit (Must be '1' if using ADC or DAC)
     0: Internal bias generator OFF
     1: Internal bias generator ON

Bit 0: REFBE: Internal Reference Buffer Enable Bit
     0: Internal reference buffer OFF
     1: Internal reference buffer ON. Internal voltage reference is driven on the VREF pin

REF0CN: Reference Control Register

The ADC1 Configuration Register (ADC1CF) and the ADC1 Control Register (ADC1CN) will allow you to set the gain for internal amplification, enable ADC1, set the "Start of Conversion" mode, and start the conversion. ADC1CN also contains the flag that will indicate when the A/D conversion is completed. The gain for the internal amplification should be set to 1 using ADC1CF.

7	6	5	4	3	2	1	0
AD1SC4	AD1SC3	AD1SC2	AD1SC1	AD1SC0	-	**AMP1GN1**	**AMP1GN0**

Bits 7-3: ADC1 SAR Conversion Clock Period Bits
    SAR Conversion clock is derived from system clock by the following equation, where AD1SC refers to the 5-bit value held in AD1SC4-0.

$$AD1SC = \frac{SYSCLK}{CLK_{SAR1}} - 1$$

Bit 2: UNUSED
Bits 1-0: AMP1GN1-0: ADC1 Internal Amplifier Gain (PGA)
    00: Gain = 0.5
    01: Gain = 1
    10: Gain = 2
    11: Gain = 4

ADC1CF: ADC1 Configuration Register

7	6	5	4	3	2	1	0
AD1EN	AD1TM	**AD1INT**	**AD1BUSY**	**AD1CM2**	**AD1CM1**	**AD1CM0**	-

Bit 7: AD1EN: ADC1 Enable Bit
    0: ADC1 Disabled. ADC1 is in low-power shutdown
    1: ADC1 Enabled. ADC1 is active and ready for data conversions
Bit 6: AD1TM: ADC1 Track Mode Bit
    0: Normal Track Mode: When ADC1 is enabled, tracking is continuous unless a conversion is in process
    1: Low-power Track Mode: Tracking Defined by AD1STM2-0 bits
Bit 5: AD1INT: ADC1 Conversion Complete Interrupt Flag
    This flag must be cleared by software
    0: ADC1 has not completed a data conversion since the last time this flag was cleared
    1: ADC1 has completed a data conversion
Bit 4: AD1BUSY: ADC1 Busy Bit
    0: ADC1 conversion is complete or a conversion is not currently in progress. AD1INT is set to logic 1 on the falling edge of AD1BUSY
    1: ADC1 conversion is in progress
Bits 3-1: AD1CM2-0: ADC1 Start of Conversion Mode Select
    AD1TM = 0:
    000: ADC1 conversion initiated on every write of '1' to AD1BUSY
    001: ADC1 conversion initiated on overflow of Timer 3
    010: ADC1 conversion initiated on rising edge of external CNVSTR
    011: ADC1 conversion initiated on overflow of Timer 2
    1xx: ADC1 conversion initiated on write of '1' to AD0BUSY (synchronized with ADC0 software-commanded conversions)
    AD1TM = 1:
    000: Tracking initiated on write of '1' to AD1BUSY and lasts 3 SAR1 clocks, followed by conversion
    001: Tracking initiated on overflow of Timer 3 and lasts 3 SAR1 clocks, followed by conversion
    010: ADC1 tracks only when CNVSTR input is logic low; conversion starts on rising CNVSTR edge
    011: Tracking initiated on overflow of Timer 2 and lasts 3 SAR1 clocks, followed by conversion
    1xx: Tracking initiated on write of '1' to AD0BUSY and lasts 3 SAR1 clocks, followed by conversion
Bit 0: UNUSED

ADC1CN: ADC1 Control Register

Note that we have not yet specified which specific pins of Port 1 should be used for analog input - this will be done using the AMUX1 Channel Select Register (AMX1SL). AMUX1 refers to the configurable analog multiplexer which is one component of the ADC1 (8-bit A/D converter) subsystem of the C8051. More information about the ADC1 subsystem and the role of the multiplexer can be found on page 75 of the *C8051F020 Reference Manual*. The pins of AIN1 also function as Port 1 I/O pins - specifying the desired AIN1 pins for analog input will correctly set the corresponding Port 1 pins.

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
-	-	-	-	-	AMX1AD2	AMX1AD1	AMX1AD0	00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
								0xAC

Bits7-3:   UNUSED. Read = 00000b; Write = don't care
Bits2-0:   AMX1AD2-0: AMX1 Address Bits
            000-111b: ADC1 Inputs selected as follows:
            000: AIN1.0 selected
            001: AIN1.1 selected
            010: AIN1.2 selected
            011: AIN1.3 selected
            100: AIN1.4 selected
            101: AIN1.5 selected
            110: AIN1.6 selected
            111: AIN1.7 selected

AMX1SL: AMUX1 Channel Select Register

The code in *Example 4.7* demonstrates how to configure the 8-bit A/D converter, and also how to start the conversion, wait for it to complete, and then read the resulting digital value. Note that upon completion of the A/D conversion, the digital value will be placed in the 8-bit ADC1 Data Word Register, ADC1.

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
								00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address:
								0x9C

Bits7-0:   ADC1 Data Word.

ADC1: ADC1 Data Word Register

```c
#include<c8051_SDCC.h>
#include<stdio.h>

void Port_Init(void);
void ADC_Init(void);
unsigned char read_AD_input(unsigned char n);

main()
{
    unsigned char result;

    Sys_Init();                         /* Initialize the C8051 board */
    putchar(0);                         /* Need to do this to prevent error */
    Port_Init();                        /* Configure P1.0 for analog input */
    ADC_Init();                         /* Initialize A/D conversion */

    while (1)
    {
        result = read_AD_input(0);      /* Read the A/D value on P1.0 */
    }
}

void Port_Init(void)
{
    P1MDIN &= ~0x01;                    /* Set P1.0 for analog input */
    P1MDOUT &= ~0x01;                   /* Set P1.0 to open drain */
    P1 |= 0x01;                         /* Send logic 1 to input pin P1.0 */
}

void ADC_Init(void)
{
    REF0CN = 0x03;                      /* Set Vref to use internal ref. voltage (2.4 V) */
    ADC1CN = 0x80;                      /* Enable A/D converter (ADC1) */
    ADC1CF &= ~0x03;                    /* Clear gain bits to 0 (gain = 0.5) */
    ADC1CF |= 0x01;                     /* Set A/D converter gain to 1 */
}

unsigned char read_AD_input(unsigned char n)
{
    AMX1SL = n;                         /* Set P1.n as the analog input for ADC1 */
    ADC1CN = ADC1CN & ~0x20;            /* Clear the "Conversion Completed" flag */
    ADC1CN = ADC1CN | 0x10;             /* Initiate A/D conversion */

    while ((ADC1CN & 0x20) == 0x00);    /* Wait for conversion to complete */

    return ADC1;                        /* Return digital value in ADC1 register */
}
```

Example 4.7 - Code for A/D Conversion on pin 0 of Port 1

Serial Communication

There are many reasons that one needs to send data to or receive data from other devices. For example, the 'printf' and 'scanf' command use an RS-232 serial protocol to communicate between the micro controller and a terminal program on your laptop. Serial connections send data one bit at a time, which require fewer wires than parallel connections. The electronic compass and the ultrasonic ranger used in the *Gondola* use a different serial communication protocol - I^2C bus. A brief introduction to I^2C bus is given next. For more details refer to Chapter 18 of C8051 user manual.

SMBus and I^{2}C bus

The I^{2}C bus, Inter IC bus, was developed by Philips in the 1980s to allow for communication between devices in a TV. SMBus, System Management Bus, is an Intel standard developed in 1995 to avoid patent issues. Since SMBus includes the I^{2}C standards, we use either name interchangeable. There are versions of the I^{2}C bus that work up to 3.4MHz. In your lab, you will use 100kHz, which is the original specification.

Serial ports transmits one bit of data at a time; sending high and low signals on a wire. Then, how does the receiver know when one bit ends and the next starts? The answer is providing a clock, which is used to determine the bit rate. Synchronous communication sends a clock signal with the data. I^{2}C is one such system, as is SPI[*]. Asynchronous communication uses separate clocks; one on each end (e.g., USB, Ethernet) and asynchronous communication requires both ends to agree on the clock rate, the baud rate. The clocks won't exactly match, so additional information must be sent to match up the data.

For I^{2}C, two wires carry signals, so it is called a 2-wire system. It needs power and ground, so four wires will be connected. It operates synchronous communication with only one master at a time and one or more slaves (see *Figure 4.10*). The Master device sends the clock on one wire, the SCL or serial clock. Then, every device is synchronized to the Master. The C8051 can be a slave, however, for our application the C8051 will always be the Master and sensors are Slaves.

The other wire transmits serial data (SDA). Data line is high to signify that a bit is a 1 or true, low for a 0 or false. Unit sending data must set the data line high or low while clock is low. When clock goes high data is read by the other device(s). Data is sent in bytes; 8 bits sent in serially one after another. After one byte is sent, receiver must acknowledge data received before next byte can be sent. Data is sent either to the slave devices (write) or data is received from the slave devices (read) on the SDA line. Because data has to flow either direction, SDA line can't be push pull; both SDA and SCL are automatically configured as open drain on the C8051. Pull-up resistors are used to pull both the SDA and SCL lines high to 3.3V. Any device can pull the lines low, so any device can send data.

Slaves have addresses; all devices are on the same bus, so each needs a unique identifier. Slaves and Master must keep track of who is talking, so there is a Read/Write bit, a Start signal and a Stop signal.

---

[*] Serial Peripheral Interface (SPI) is a synchronous protocol that uses 3 wires and is also available on the C8051F020.
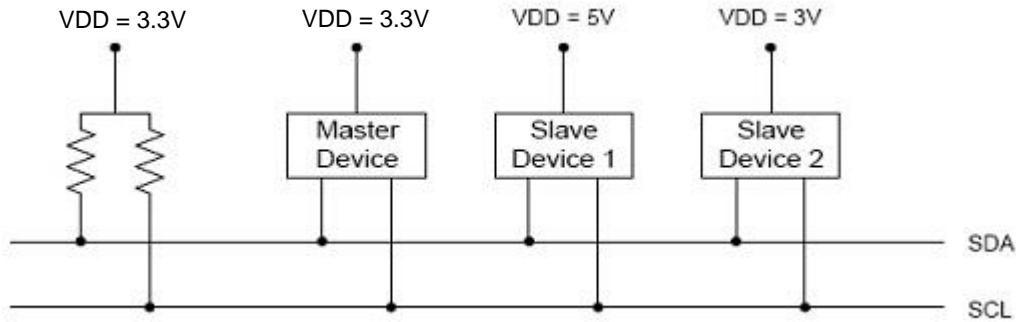
Figure 4.10 - Typical SMBus Configuration

There are two modes; (1) Master Transmitter Mode is used to talk to a slave device and (2) Master Receiver Mode is used to listen to a slave device. The Master Transmitter Sequence (see *Figure 4.11*) starts with the Master sending a START signal followed by a byte containing the 7-bit slave address (bits 7-1) with the R/W bit (bit 0) *low*[*], indicating a write sequence. The Slave should then send an acknowledge (ACK) signal. The Master will then send one byte of data, check for ACK from the Slave, and repeat until all data is sent. The transmission is ended by the Master sending the STOP command. The Master must know how many bytes to write.
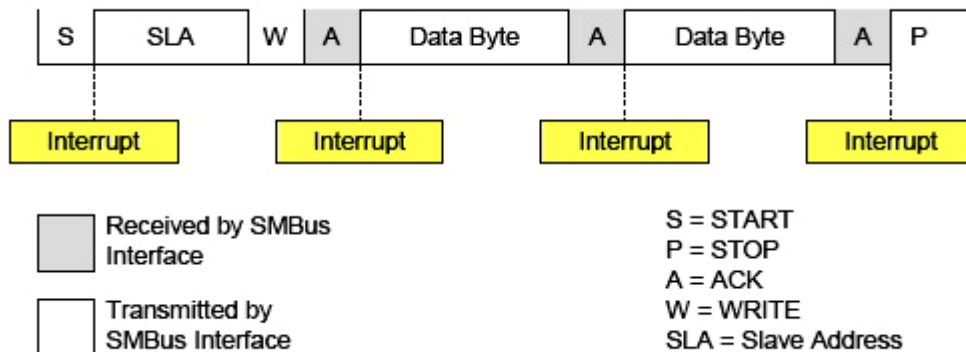

Figure 4.11 - Typical Master Transmitter Sequence

In the Master Receive mode (see *Figure 4.12*) the Master again sends a START signal and sends a single byte with the 7-bit slave address (bits 7-1) and the R/W bit (bit 0) *high*, indicating a read sequence. The Slave should then send an acknowledge (ACK) signal. Then, the Slave will send one byte of data, and the Master should acknowledge. Transmission of data bytes is repeated until the Master responds with NACK (Not Acknowledge) and STOP instead of ACK. Note that the Master must know how many bytes to read.

---

[*] See Chapter 18 of the C8051 User Manual for additional details.
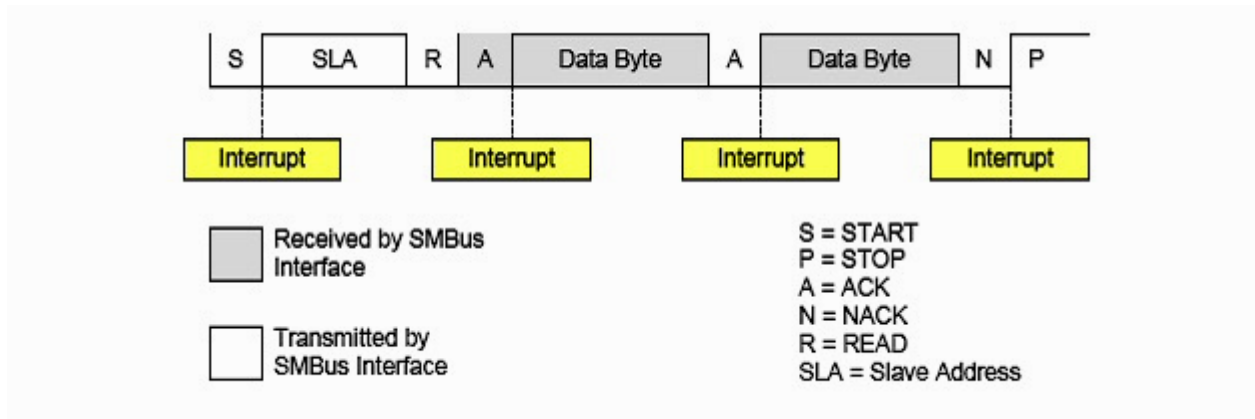
Figure 4.12 - Typical Master Receiver Sequence

Using Interrupts might be the most eloquent way to implement the I^{2}C bus. However, there are 28 possible status states of the I^{2}C bus hardware on the C8051 and a complete interrupt service routine would have to handle all these states. Since the C8051 is always the Master in our application and not all states will occur, our code will not use interrupts. Instead the code will control each step of the communication, and will monitor the SI bit (of the SMB0CN register) to determine when a step is complete.

There are five SFRs, special function registers associated with the I^{2}C bus:

- SMB0CN – SMBus 0 Control: Used to control the bus (functions use sbits)

- SMB0STA – SMBus 0 Status: Read to know present status (not used in our implementation)

- SMB0CR – SMBus 0 Clock Register: Set for 100kHz clock (set once)

- SMB0ADR – SMBus 0 Address: slave address (not used in our implementation)

- SMB0DAT – SMBus 0 data: data for reads and writes (used by functions)

7	6	5	4	3	2	1	0
BUSY	ENSMB	STA	STO	SI	AA	FTE	TOE

Bit 7: BUSY: Busy Status Flag
    0: SMBus0 is free
    1: SMBus0 is busy
Bit 6: ENSMB: SMBus Enable
    This bit enables/disables the SMBus serial interface
    0: SMBus0 is disabled
    1: SMBus0 is enabled
Bit 5: STA: SMBus Start Flag
    0: No START condition is transmitted
    1: When operating as a master, a START condition is transmitted if the bus is free. (If the bus is not free, the START is
      transmitted after a STOP is received) If STA is set after one or more bytes have been transmitted or received and
      before a STOP is received, a repeated START condition is transmitted. To ensure proper operation, the STO bit
      should be explicitly cleared to '0' before setting the STA bit to '1'.
Bit 4: STO: SMBus Stop Flag
    0: No STOP condition is transmitted
    1: Setting STO to logic 1 causes a STOP condition to be transmitted. When a STOP condition is received, hardware clears
      STO to logic 0. If both STA and STO are set, a STOP condition is transmitted followed by a START condition. In
      slave mode, setting the STO flag causes SMBus to behave as if a STOP condition was received.
Bit 3: SI: SMBus Serial Interrupt Flag
    This bit is set by hardware when one of 27 possible SMBus0 states is entered. (Status code 0xF8 does not cause SI to be
    set.) When the SI interrupt is enabled, setting this bit causes the CPU to vector to the SMBus interrupt service routine.
    This bit is not automatically cleared by hardware and must be cleared by software.
Bit 2: AA: SMBus Assert Acknowledge Flag
    This bit defines the type of acknowledge returned during the acknowledge cycle on the SCL line.
    0: A "not acknowledge" (high level on SDA) is returned during the acknowledge cycle
    1: An "acknowledge" (low level on SDA) is returned during the acknowledge cycle
Bit 1: FTE: SMBus Free Timer Enable Bit
    0: No timeout when SCL is high
    1: Timeout when SCL high time exceeds limit specified by the SMB0CR value
Bit 0: TOE: SMBus Timeout Enable Bit
    0: No timeout when SCL is low

SMB0CN: SMBus0 Control Register (Bit Addressable)

7	6	5	4	3	2	1	0

Bits 7-0: SMB0DAT: SMBus0 Data
    The SMB0DAT register contains a byte of data to be transmitted on the SMBus0 serial interface or a byte that
    has just been received on the SMBus0 serial interface. The CPU can read from or write to this register whenever
    the SI serial interrupt flag (SMB0CN.3) is set to logic 1. When the SI flag is not set, the system may be in the
    process of shifting data in/out and the CPU should not attempt to access this register.

SMB0DAT: SMBus0 Data Register

R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	Reset Value
								00000000
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	SFR Address 0xCF

Bits 7-0: SMB0CR.[7:0]: SMBus0 Clock Rate Preset

The SMB0CR Clock Rate register controls the frequency of the serial clock SCL in master mode. The 8-bit word stored in the SMB0CR Register preloads a dedicated 8-bit timer. The timer counts up, and when it rolls over to 0x00, the SCL logic state toggles.

The SMB0CR setting should be bounded by the following equation , where *SMB0CR* is the unsigned 8-bit value in register SMB0CR, and *SYSCLK* is the system clock frequency in Hz:

$$SMB0CR < (\,(288 - 0.85 \cdot SYSCLK/10^6)/1.125)$$

The resulting SCL signal high and low times are given by the following equations:

$$T_{LOW} = (256 - SMB0CR)/SYSCLK$$

$$T_{HIGH} \cong (258 - SMB0CR)/SYSCLK + 625\text{ns}$$

Using the same value of SMB0CR from above, the Bus Free Timeout period is given in the following equation:

$$T_{BFT} \cong \frac{10 \cdot (256 - SMB0CR) + 1}{SYSCLK}$$

SMB0CR: SMBus0 Clock Rate Register

For implementation, first, you need to initialize SMBus registers; setting SCL to 100kHz and enabling SMBus0 (note that you also need to initialize the crossbar (XBR0) appropriately in order to use SMBus):

```
void SMB_Init(void)
{
     SMB0CR=0x93;        /* set SCL to 100KHz (actual freq ~ 94,594Hz)*/
     ENSMB=1;            /* bit 6 of SMB0CN, enable the SMBus */
}
```
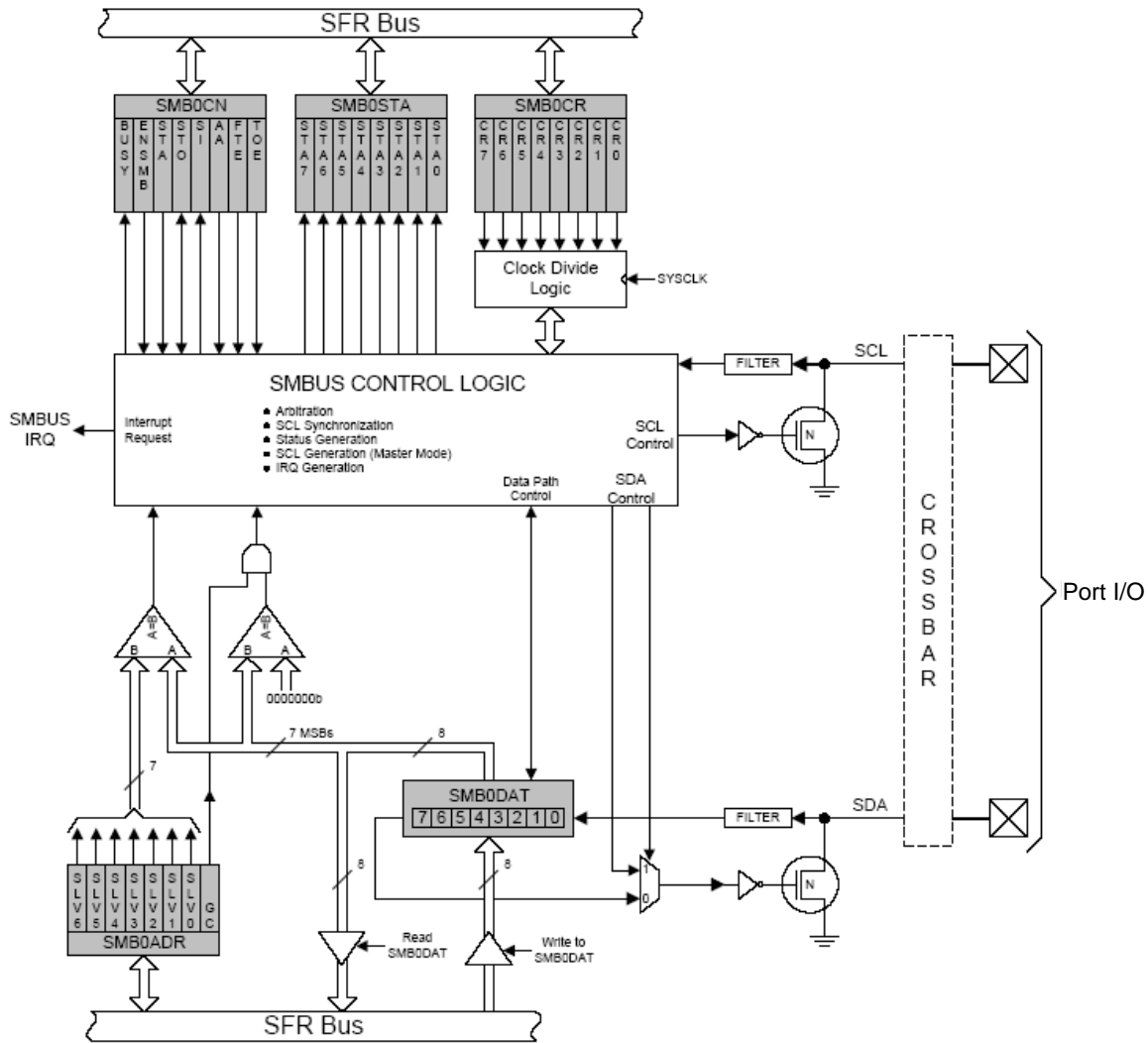
Figure 4.13 - SMBus0 Block Diagram

After initialization, you will need a series of simple functions:

- `void i2c_start (void)`: to sends a start signal, which must be followed by a write to be useful

- `void i2c_write (unsigned char output_data)`: to send a byte of data or a slave address with the R/W bit, which must be preceded by a start or another write

- `void i2c_write_and_stop (unsigned char output_data)`: to send last byte of a write to slave, which is followed by a stop signal

- `unsigned char i2c_read (void)`: to read a byte of data from a slave, which must be preceded by a start and a write. The write contains the address of the slave of interest.

Most often there are three writes before the read. The first addresses the slave, the second picks which data is desired from the slave and the third says to start sending.

– `unsigned char i2c_read_and_stop (void)`: to read the last signal in a sequence and send a stop signal.

You will also need two higher-level functions:

– `void i2c_write_data (unsigned char addr, unsigned char slave_reg, unsigned char *buffer, unsigned char num_bytes)`: (1) to start the I2C, (2) to write to the slave address, addr, with R/W bit low, (3) to send a byte of data which is the register inside the slave to be written, slave_reg, (4) to send the data that is to be written in the slave_reg, buffer, (5) to repeat if more than one byte is to be written, and (6) to end with a stop signal on last write.

– `void i2c_read_data (unsigned char addr, unsigned char slave_reg, unsigned char *buffer, unsigned char num_bytes)`: (1) to start the I^2C, (2) to write to the slave address, addr, with R/W bit low, (3) to write a byte of data which is the register inside the slave to be read, slave_reg, and send stop, (4) to start the I^2C, (5) to write to the slave address, addr, again but with R/W bit high, (6) to read data, and to repeat read to get all the data, num_bytes, (7) to set ACK high during each read except on the last read, and (8) to stop the I^2C.

Chapter 5 - Circuitry Basics and Components

Building Circuits

It is essential that you are familiar with the information in this section before you begin building your first circuit. When you begin assembling your circuitry, please keep in mind that it is *absolutely essential* to wire your circuits as neatly as possible! The more time invested into building a neat, cleanly wired circuit, the less time will be spent in trying to isolate problems with it later.

Grounding

One of the most common sources of problems causing circuits to work unpredictably and sporadically is incorrect grounding practices. Since voltage is defined as potential difference, all interacting circuitry must have a common reference. All voltages are measured from this point, which is defined as 0V, or ground. Make sure that the power supply; the EVB, and your circuitry all are using a common ground by explicitly connecting the grounds together at a *single* point.

Noise

Noise is defined as an unwanted signal that interferes with a transmitted signal. Another extremely important reason why it is essential to wire the protoboards neatly is to eliminate as much noise as possible. Recall from physics that wires carrying electric current also have magnetic fields that can *induce* unwanted voltages and currents in adjacent wires. The degree to which the unwanted induction takes places is largely a function of how efficiently a wire serves as an *antenna*. A long loop of wire sticking up serves as a much better antenna than a short wire close to the surface of the protoboard. Therefore, **try to keep your wires short and close to the protoboard to reduce antenna effects.** Also, avoid running power and ground for motors to the vertical rails on the protoboard that would induce a significant amount of noise into your circuitry as the relatively large motor currents fluctuate.

Preventing errors

The best way to avoid facing most hardware problems is to prevent them from ever occurring. All wires should be flat against the protoboard to make component removal simple, and no bare wires should protrude above the protoboard holes in order to avoid unintended connections and shorts. Select wire colors according to industrial standards—use red wire *only* for 5V power and black wire *only* for ground. This makes it much easier to visually inspect the board to see if power and ground are connected. Select colors for other wires so that they will be easier to trace. A single

wire within a bundle of other identically colored wires is very difficult to differentiate and trace. Finally, place all chips on the board *in the same orientation and with their tops facing the top of the protoboard* so that it is easy to quickly see which pins are connected.

Schematics

Throughout this course, you will be given schematics for the logic circuits you will be expected to build. A few common schematic symbols are shown in *Figure 5.1*. A schematic shows the interconnections between integrated circuit (IC) chips and other electronic components that are necessary to realize the circuit. Because a schematic is made primarily to show *logical* connections, it often **does not** closely resemble the physical layout of the circuit. For example, the IC chips are often shown on a schematic with the pins *rearranged*, usually so that the input pins are on one side and the output pins are on the other. This is done for organizational purposes, to make the schematic more readable.
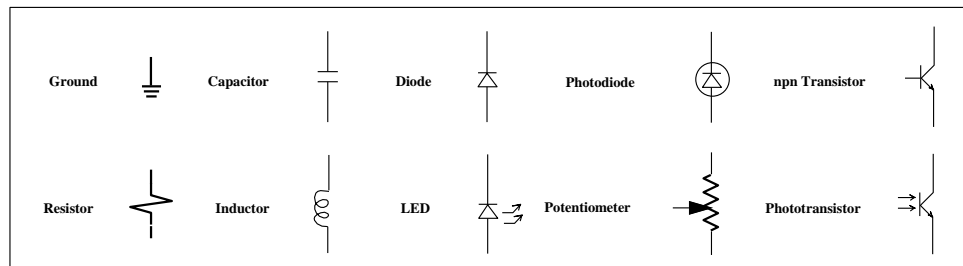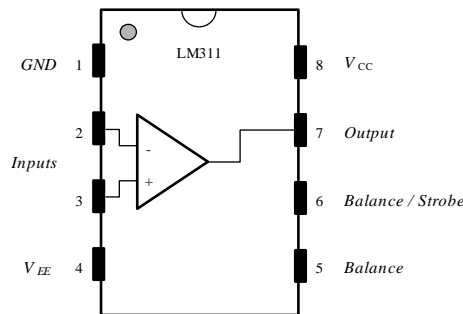

Figure 5.1 - Common schematic symbols

Figure .2 - IC illustrating counter-clockwise pin number scheme

5

On the physical chip, the pin numbers are arranged from the top-left around the chip in a ***counter-clockwise (CCW)*** direction as shown in *Figure 5.2*. The top-left is found by locating the small dot or notch on the chip.

Additionally, it is important to remember that schematics often **do not** show pins labeled for *power* and *ground* on ICs. This is because power and ground connections are *always* required, and showing these connections would simply add clutter to the drawing. So when building your circuit,

don't forget to attach power and ground to each chip even if these connections are not shown on the schematic!

Chip handling precautions

Integrated circuit chips are very susceptible to damage by static electricity. You can permanently damage an IC if you don't discharge any accumulated static charge from yourself before handling it. To avoid this, you must make sure that you discharge any static electricity your body may have accumulated before you handle a chip. One way to do this is to touch a well-grounded object just before handling any chips.

The Buffer

In integrated circuits, a buffer gate receives a logic input and sends the same logic output. Buffer chips typically contain multiple buffer gates with pairs of pins for input and output for each gate. The chip will to have power (V_{CC}) at pin 16 and ground (GND) at pin 8 connections to operate properly. Applications of buffers include isolating specific parts of a circuit and acting as a current supply or current sink. Although the output logic state (low or high) always matches the input state, buffers also interface different logic families with supply voltages ranging from 5V down to 2.5V or lower. Output voltages will match logic thresholds of connected family.

The 74F365 Hex Buffer/Driver chip contains six pairs of input/output pins (see *Figure 5.3*). In addition, it has two Output Enable pins, 1 and 15, which must be set low in order for the chip to function as a buffer. When both Output Enable pins are set low, then a low input on one of the input pins will produce a low output on the corresponding output pin, and a high input will produce a high output on the corresponding pin.
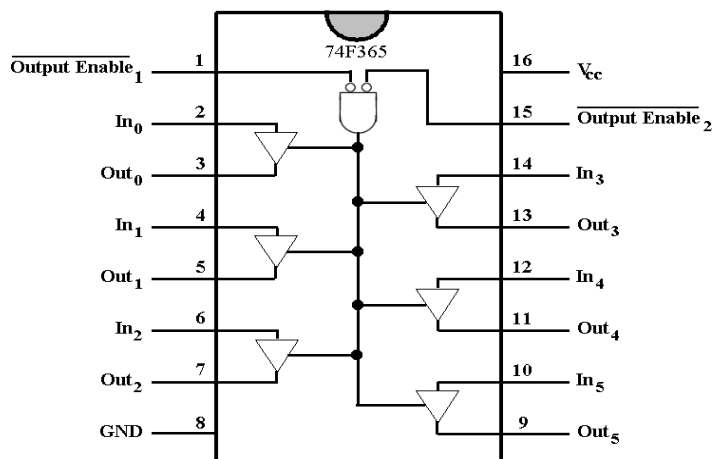

Figure 5.3 - 74F365 Hex Buffer/Driver Chip

Note that setting only one of the Enable pins high will still produce a high impedance state, which will effectively isolate the input from the output (see *Table 5.1*). Thus, the chip can actually output three states: high, low, and high impedance. In class, make sure both of these pins are grounded. Note that the 3.3V logic levels of the C8051F020 are boosted to the 5V logic levels of the protoboard circuits by the 74F365 buffer.

Table 5.1 - Function Table for the 74F365

L = Low Voltage Level, H = High Voltage Level, X = Immaterial, Z = High Impedance

Inputs			Output
OE_1	OE_2	In	Out
L	L	L	L
L	L	H	H
X	H	X	Z
H	X	X	Z

The Inverter

The function of an inverter is to transform a logic high at the input, A, to a logic low at the output, Y, and visa versa. Similar to a buffer chip, and inverter chip also typically contains multiple gates and requires power and ground connections.

The *Motorola SN74LS05N* and *SN74LS04N* hex inverter ICs, shown in *Figure 5.4*, contains six independent inverters. The *Motorola SN74LS05N* has been designed with open-collector outputs that require the use of pull-up resistors in order to function properly.
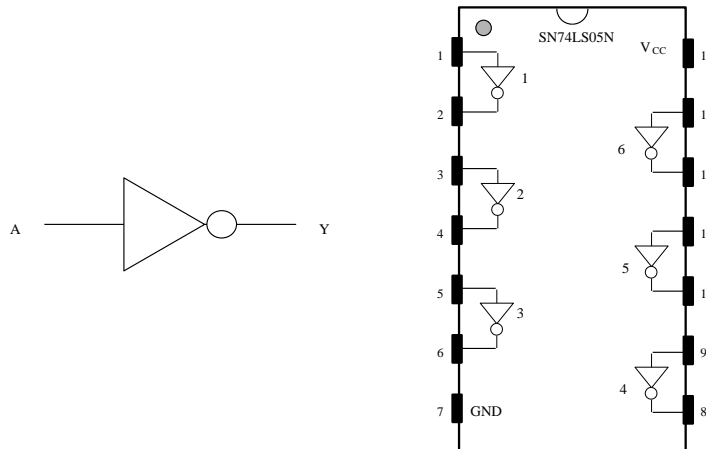

Figure 5.4 - Symbol for an inverter (left) and a Motorola SN74LS05N or SN74LS04N hex inverter IC

The *Motorola* **SN74LS04N** hex inverter IC serves the same function and has the same pin configuration as the *SN74LS05N*. The main difference between the two ICs is that the *SN74LS04N* does not require pull-up resistors at its outputs in order for the inverter to function correctly.

Common Digital Gates

Some of the common digital gates frequently used in digital electronics are given below.

LEDs

An LED, or light emitting diode, is a very simple means of displaying *information* to a user. An LED is simply a diode that emits light when current flows across it. An important characteristic to remember about diodes is that a diode only allows current to flow in one direction. Therefore, an LED can be lit by placing a potential difference across the diode that will cause current to flow from the anode to the cathode of the LED. It should also be noted that the intensity of the light emitted by the LED is directly proportional to the amount of current passing through the LED. *Figure 5.5* shows a physical and schematic representation of a standard LED.
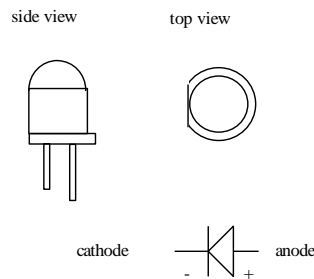

Figure 5.5 - An LED, note flat side is on cathode (-) lead

The question that should be on your mind is *"How can we control the LED using the microprocessor?"* Remember, the microprocessor can send a 1 or 0 to each digital output pin resulting in a logic high or low level, where logic low corresponds to ground and logic high to +5V.

When implementing a circuit as shown in *Figure 5.6*, a 0 [V] (or logic low) must be output from the EVB pins connected to the LEDs in order to create a potential difference across the LEDs and cause them to light. Note, a buffer is used between the LED and the EVB pin in order to protect the EVB in case of a short across the LED or other potentially damaging occurrence. The buffer will be used for isolation extensively in this course.
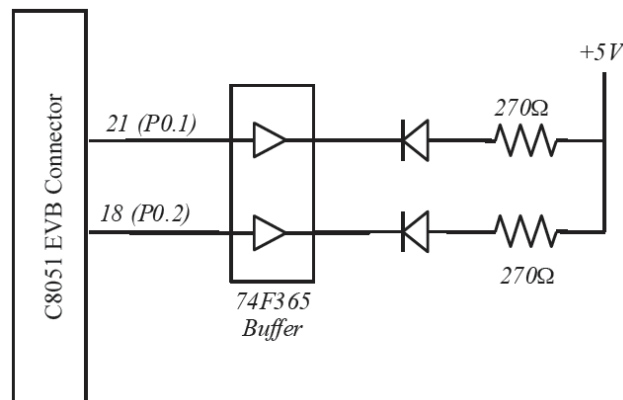

Figure 5.6 - Simple LED/buffer schematic

Switches

The next developmental stage is the integration of switches. The lab for Embedded Control is equipped with momentary push button switches and non-momentary slide switches, which can be configured for a variety of uses. One particular use for the switches is to provide a means to input digital data to the C8051. The C8051 can read data through the data lines of its digital input ports from an external device such as a switch. Since values provided to the C8051's digital input ports must either be 0 volts (logic *LOW*) or 5 volts (logic *HIGH*), switches used for digital input should be configured to provide these two voltage levels.

Toggle and Push-button Switches

The push-button switches in the lab, and some types of toggle switches, are momentary switches. The slide switches in the lab are non-momentary toggle switches. As soon as you release a push-button switch, contact is broken. When you release a momentary toggle switch, its spring returns it to the '*A*' position. *Figure 5.7* illustrates the relation between switch position and contact configuration of a toggle switch, whether it is momentary or non-momentary.
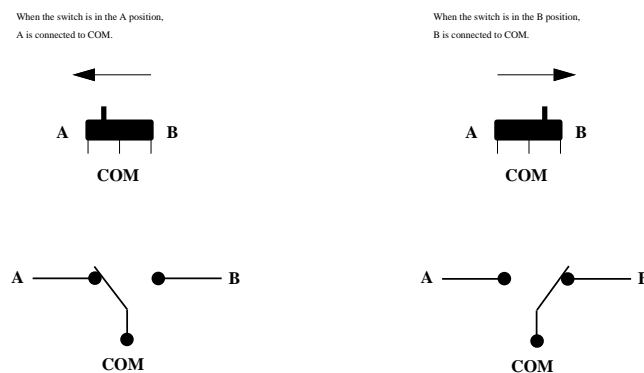
When the switch is in the A position, A is connected to COM.

When the switch is in the B position, B is connected to COM.

A   B   COM

A   B   COM

A   B   COM

A   B   COM

Figure 5.7 - Slide (toggle) switch positions

The distinguishing feature of the momentary switch is its *normal* position. The term *normal* as applied here means "without outside intervention, as by manually holding the switch in one position or the other"[*] . Switches that have a *normal* position also have contacts referenced as *normally open* (N.O.) and *normally closed* (N.C.). The *normal* position for the momentary toggle switches is for COM to be connected to '*A*'. Because the toggle switches can connect one line (COM) to either of two alternate lines (*A* or *B*), they are further classified as *single-pole double-throw* (SPDT) switches. The slide switches available in the lab for Embedded Control do not have a normal position.

---

* The terminology "normally open/closed contacts" also applies to electro-mechanical switches (relays) where the relay is in its normal position when its coil is not energized.

Figure 5.8 illustrates the internal configuration of the push-button switches. Since these switches are configured to connect a single line to another line, they are classified as *single-pole single-throw* (SPST) switches. When the push-button is not pressed, the contacts are normally open.
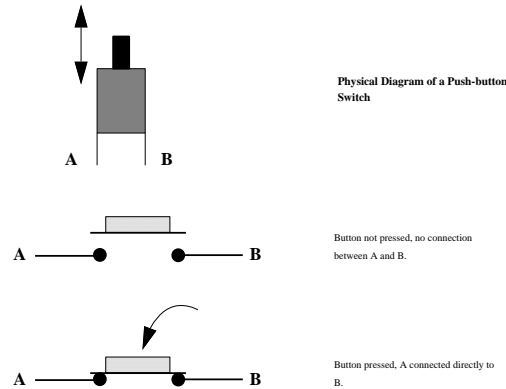

Physical Diagram of a Push-button Switch

A   B

Button not pressed, no connection between A and B.

A ———————————— B

Button pressed, A connected directly to B.

A ———————————— B

Figure 5.8 - Push-button switch positions

Table 5.2 below summarizes the types of switches available in the lab for Embedded Control.

Table 5.2 - Switches available in the lab for Embedded Control

Physical Type	Momentary/ Non-Momentary	Normal Position	Electrical Type
Pushbutton	Momentary	Open	SPST
Slide Switch (Toggle)	Non-Momentary	N/A	SPDT

Configuring switches for 0 or 5 volt digital output

Figure 5.9 illustrates a useful push-button switch configuration for generating a digital (0 or 5 volt) signal, while *Figure 5.10* illustrates a toggle switch configuration for producing a 0 or 5 volt signal. For illustrative purposes only, the outputs from these switch configurations are shown as being routed to various data lines of the C8051's digital input port *2*. In practice, these switch outputs could be routed to any logic gate input terminal which accepts 0 and 5 volt logic inputs, e.g., the input terminals of logic gates such as *AND*, *OR*, *NOT*, etc.
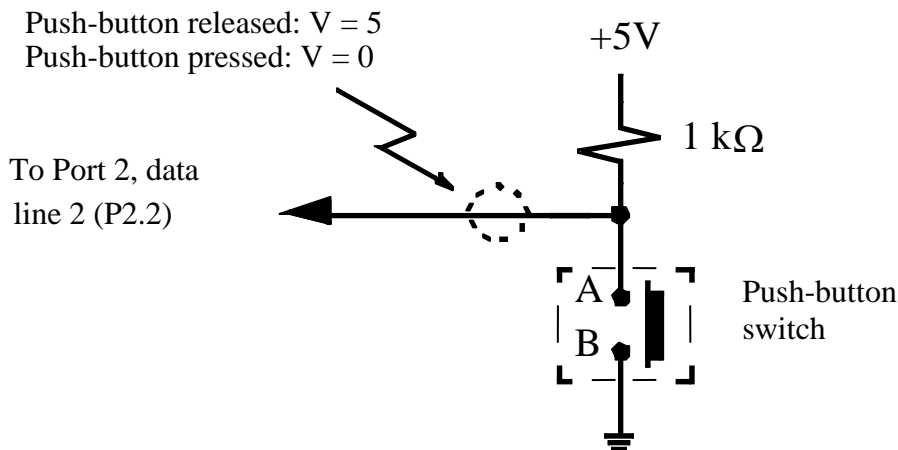
Push-button released: V = 5
Push-button pressed: V = 0

+5V

1 kΩ

To Port 2, data
line 2 (P2.2)

A

B

Push-button
switch

Figure 5.9 - Push-button configuration for producing 0 and 5 volt outputs

+5V

1 k Ω

To Port 2, data
line 4 (P2.4)

COM

A

B

Toggle
switch

Switch in position A: V = 5
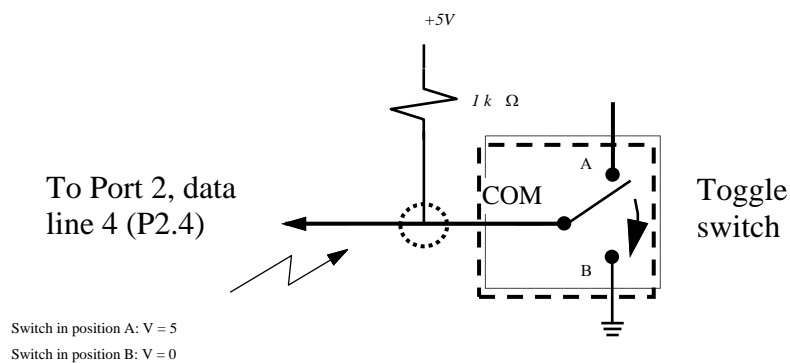Switch in position B: V = 0

Figure 5.10 - Toggle switch configuration for producing 0 and 5 volt outputs

Electric Compass

The electric compass yields the current direction with respect to magnetic north. A magnetic field sensor in the compass is sensitive enough to detect the Earth's magnetic field. As shown in *Figure 5.11*, the compass requires a 5V power line at Pin 1 and a ground line at Pin 9. Also, you need to connect Pin 2 (SCL - Serial Clock) and Pin 3 (SDA - Serial Data) to I^2C bus to get readings from the compass. The SCL and SDA Port pin connections on the EVB will depend on the XBR0 setting. As with all I^2C devices the SDA and SCL lines must be pulled up to a 3.3V supply through a 1.8k resistor. Pin 7 on the 60-pin connector is the only pin assigned to 3.3V and must be wired carefully in all circuits. **Never short 3.3V to 5V anywhere in a circuit.** Doing so will damage both the EVB and the 5V supply on the car.
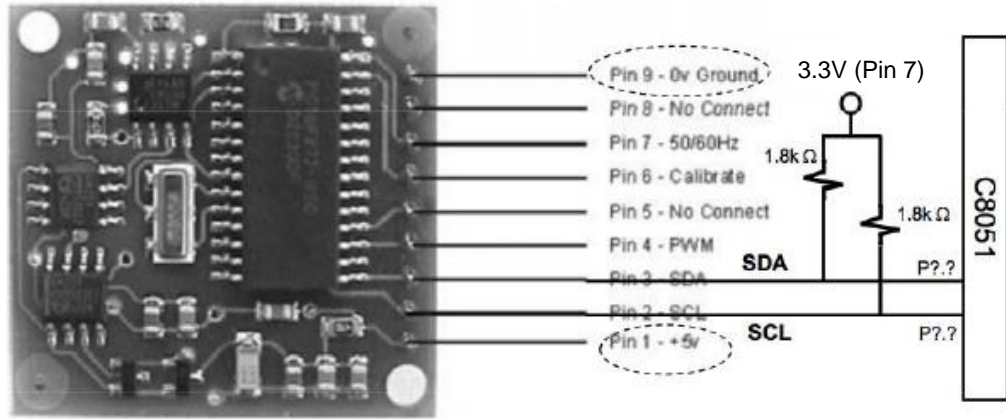
Figure 5.11 - Wiring for Compass

Ultrasonic Ranger

The ultrasonic ranger detects the distance to objects or surfaces. It does so by creating a short burst of high frequency sound waves, which travel a the speed of roughly 0.9 feet/msec and reflect back to the ranger from any object encountered in the path. After transmitting the signal, the ranger waits for the reflected signal (echo). If the echo is received, the ranger computes the distance to the object based on the elapsed time. The necessary wiring is shown in *Figure 5.12*.

The ranger requires 5V power and ground lines. It also requires I^2C bus connections (SDA and SCL) to read the distance to an object. Again, the SCL and SDA Port pin connections on the EVB depend on the XBR0 settings but still must be pulled up to 3.3V on Pin 7.
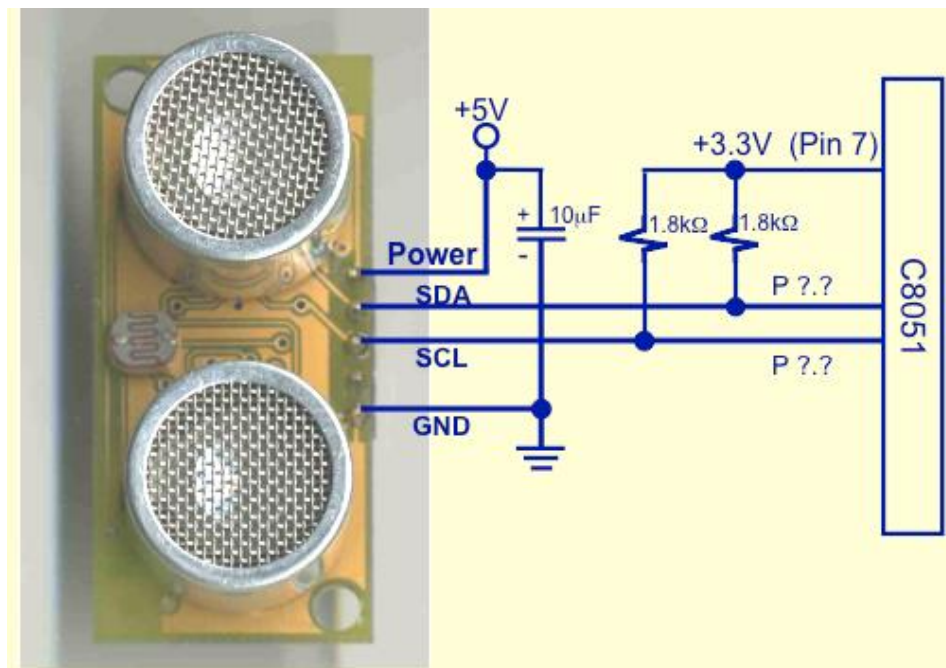
Figure 5.12 - Wiring for Ranger

LCD and Keypad

The LCD and keypad drivers provide an easy way to display characters on a LCD screen and read key presses using the I^2C bus. The LCD screen is capable of displaying 20 characters per line, with a total of 4 lines for the entire display. The keypad is capable of reading 12 characters total (numbers 0-9, *, and #).

The LCD requires 5V power and ground lines. It also requires I^2C bus connections (SDA and SCL) to send data to the LCD, and read data from the keypad. The SCL and SDA Port pin connections on the EVB will depend on the XBR0 settings.
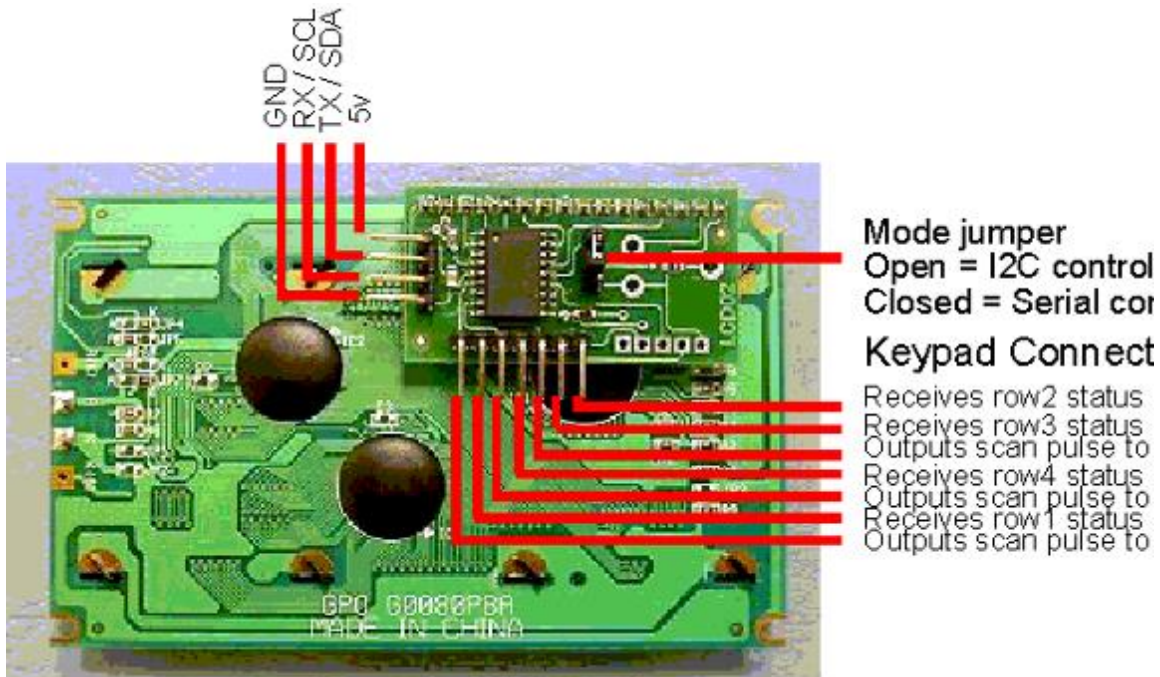
Figure 5.13 - Wiring for LCD: GND is black, SCL is white, SDA is gray, and 5V is violet. If the white & gray wires are crossed at the cable connector the signals are GND, SDA, SCL, 5V.

Figure 5.14 - Wiring for keypad. Pins 1 and 9 are unconnected. Only pins 2-8 are used.

Accelerometer

A 3-axis accelerometer with an SMB interface allows the car to detect accelerations along all 3 axes. The data can also be used to determine pitch and roll angles of the car on a flat surface. The chip used on the module is an STMicroelectronics LSM303C, which also includes a 3-axis magnetometer. The magnetometer will not be used in this class, but could be used to calculate compass heading the same way the SMB compass module does.

Different from the others, **this module requires 3.3V power** and ground lines. As typical it requires I^2C bus connections (SDA and SCL) to send data to the 8051 and receive configuration information. The SCL and SDA Port pin connections on the EVB will depend on the XBR0 settings. The accelerometer's I^2C bus address is 0x3A.

Note the directions for the x-, y-, and z-axis on the board: +x is to the right; +y is to the bottom; and +z is up, out of the board. If the module is held flat, neither the x- nor y-axis will detect any acceleration (tilt). If the board is tilted around the x- or y-axis, a signed acceleration value will be returned and the sign will indicate the direction of the pitch or roll while the magnitude will indicate the size of the angle. Lab 5 explains the details of how to set up and use the module. Your main program must **call the accelerometer initialization function** `Accel_Init_C(void)` (**not** `Accel_Init(void)`) in `i2c.h` before using the device. The data from the chip is inherently noisy and some extra processing is required to allow the car to be well controlled. The simplest method to clean up the signal (although unsophisticated) is to average a number of individual readings together. It has been determined that 8 or 16 will do a reasonable job of reducing the noise. Also, due to variations in how the module mounts on protoboards, there will be constant offsets in both the x- and y-axis that must be measured when the car is on a flat surface (off the foam block and on its wheels). Average about 32 data points to determine an offset value to be subtracted from every reading when in normal operation.
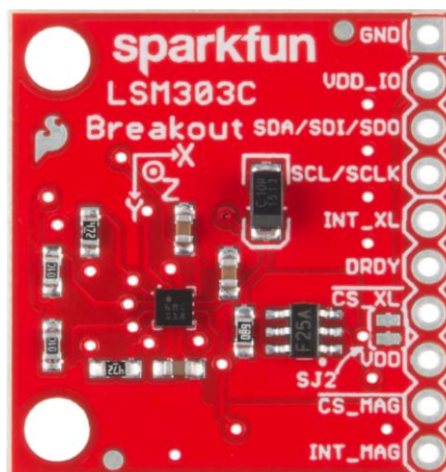

 Figure 5.15 - The 3-axis Acceleration Module (or compatible version). GND and VDD_IO are the ground and +3.3V connections, SDA and SCL are self-explanatory. No other pins are used.

Wireless RF Serial Link Modules

A wireless RF transceiver pair sets up a serial link between the car and the laptop without using the wired USB/RS-232 adapter. The car module interfaces directly with the transmit and receive UART0 signals (P0.0 & P0.1) on the EVB Port Connector. The laptop module plugs into a USB port and is assigned a COMn: port number by Windows similar to when the USB/RS-232 adapter is plugged in, after the drivers are loaded. The same procedure should be followed to connect your SecureCRT terminal to the correct serial port. With everything in place, this allows the car to send and receive serial data using `printf()` and `getchar()` commands. Laptops can more conveniently be used to select expanded menu options with easier to understand instructions than those imposed by the limits of the LCD/Keypad interface. Additionally, real-time data may be collected while the car drives up the ramp for later plotting without any outside physical restraints pulling against the car. Similar hardware will be used on the gondolas in Lab 6 for wireless connections, but will already be connected internally on the gondolas.

For additional information on setting up SecureCRT to collect data from the car that will be passed to Excel or MATLAB for plotting, see the sections **Terminal Emulator Program** and **Drivers for USB RF link serial adapter** in the course website file **Installing_SiLabs-SDCC-Drivers**.

It is important to realize that transmitter/receiver pairs are set and students must be sure to use the correct matching laptop module with the car module. Unmatched modules are configured to different radio channels and can't communicate with each other. The channel number (written on the white tag on the car module, ranging from 1 to 10) must match the number written on the laptop USB module. Also, the wired RS-232 connection **must not be used and must be unplugged** when the RF car module is connected to the EVB bus.

As usual, the car module requires +5V power and ground lines. It also needs connections to the UART0 TX0 (Transmit, P0.0) and RX0 (Receive, P0.1) lines, as show in Figure 5.16. A 5th pin needs to be grounded during normal operation.
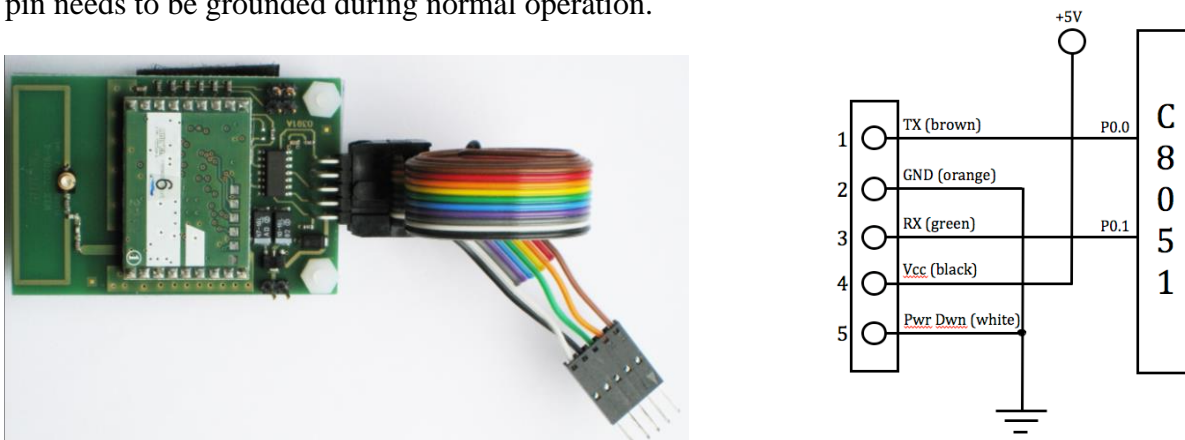
Figure 5.16 – The wireless serial RF connection module and diagram.

A common problem with the RF links used on the cars and the similar units used on the gondolas is signal distortion due to over-modulation at the receiver. If the receiver unit plugged into the laptop is too close to the car or gondola (less than 3 feet away), the characters received may be corrupted. This usually shows up as garbled characters or graphical symbols on the terminal. If this happens move the receiver further away from the transmitter and restart SecureCRT. In most cases SecureCRT must be completely closed down and restarted to reset the defaults or it will not go back to receiving and displaying ASCII characters correctly.

Chapter 6 - Motor Control

Three types of motors commonly used in embedded control applications are DC (direct current), servo, and stepper motors. For the *Smart Car*, the steering system is controlled by a pulsewidth-modulated signal applied to a servo motor while the speed is controlled by pulsewidth modulation of the signal sent to the drive motor. The process by which the motors interpret these signals is vastly different however. An overview of DC, servo, and stepper motor operation and the process by which the DC and servo motors process the pulsewidth-modulated signals will be discussed below as well as the associated driver circuitry.

Each type of motor requires a driver to act as an interface between the motor and the C8051. A driver is a circuit that uses a small control signal to control a large load, such as a motor or a relay. It is also used to isolate the control system from this large load. The outputs from the C8051 are 5V logic outputs, which means that they are either 0 or 5V, and cannot supply enough current to run the motors. Therefore, these signals must be amplified if they are to be used to control anything other than logic circuits.

Servo Motors

Servo motors are typically used in applications which require controlled rotary movement. They are cheap ($1 to $20), run very fast (1000 RPM and up), and require very little driving circuitry.

Actuation

A common method for controlling the position of a servo motor is through pulsewidth modulation of the driving signal. The internal circuitry of the servo motor includes a potentiometer, a comparator circuit, and an internal clock - these allow the motor to vary its functioning depending on the pulsewidth of the driving signal. The pulsewidth refers to the amount of time the driving signal is high during the period of the signal. The comparator circuit in the servo motor determines whether the pulsewidth of the driving signal is equal to the pulsewidth of the internal clock. When the driving signal pulsewidth is less than the internal clock pulsewidth, the comparator circuit supplies a control signal that allows the servo motor to rotate and turn the wheels to the left. A potentiometer is coupled to the shaft of the servo motor and adjusts the pulsewidth of the internal clock. As the servo rotates, the potentiometer rotates, and the pulsewidth generated by the internal clock changes. The motor shaft continues to rotate until the pulsewidth of the internal clock is equal to the pulsewidth of the driving signal (as determined by the comparator circuit).

The same procedure is followed when the driving pulsewidth is greater than the internal clock pulsewidth, except that the comparator circuit now supplies a control signal that allows the servo motor to rotate and turn the wheels to the right.

Driver

The modest power and voltage requirements allow the servo to run using a single buffer. The servo does not need any voltage amplification, but a buffer or dual inverter gates are needed as a current source, thus amplifying the power from the logic signal output by the C8051 to a signal having sufficient current to drive the servo motor. The 74F365 buffer chip can supply 48mA and the 74LS04 inverter chip can supply 40mA; the servo motor typically draws approximately 10mA (small in comparison to the DC drive motor which draws close to 1A).

DC Motors

A DC drive motor mounted on the car chassis gets its power from the 12 volt power source. Regulating the power applied to the drive motor controls the speed of the car: the rotational speed of the motor will increase with an increasing analog voltage. A pulsewidth-modulated signal can be generated by the control program running on the EVB that is used to switch a current-driver chip (for power amplification), the output of which is connected to a DC drive motor. The average power applied to the drive motor depends on the duty cycle of the pulsewidth-modulated signal that is under computer control.

Actuation

Pulsewidth modulation is a simple way to actuate a DC motor with a variable voltage, even if a variable voltage source is not available. The inductance and series resistance of the motor coils acts as a low pass filter, filtering out high frequency spikes and averaging an input signal to the motor. By varying the time that a constantly varying pulse is on (T_{on}) within a constant time period (T_{total}), the voltage on the motor can be varied (see *Figure 6.1*). The output voltage for this scheme is thus controlled by varying the duty cycle, $\dfrac{T_{on}}{T_{total}}$. A duty cycle of 100% corresponds to maximum power applied to the drive motor.
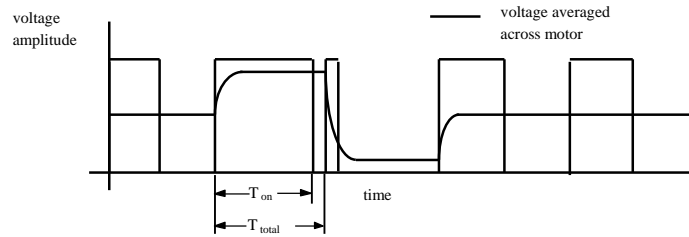
Figure 6.1 - Pulsewidth modulation averaging

Speed Controller

The speed controller can receive pulse width modulation input signals and map it to pulse width modulation output signals. The *Table 6.1* provides mapping between input and output signals. Note that the numbers are approximate because there is deadband near neutral pulse width. The speed controller has been calibrated for you based on *Table 6.1*. When PCA is configured as System Clock/12 as our lab application, the corresponding pulse width will be calculated as:

pulse width (counts) = input pulse width (ms) * System Clock/12 (MHz)

The input pulse width must be refreshed every 20 ms. If the program refreshed pulsewidth much less often, the program shuts down the motor. If it refreshes much more often than 20 ms, the program starts to confuse and does not work properly.

To make the motor work for the maximum and minimum pulse width, the warm-up period is necessary; the micro controller must send the neutral pulse width for a second after power is on. Also note that the Bandwidth of minimum and maximum pulse width must be maintained because the motor shuts down if input pulse width is sufficiently larger than maximum pulse width or sufficiently shorter than minimum pulse width.

The model number of speed controller used in the *Smart Car* and *Gondola* is EZX-R and HFX-R by HITEC, respectively.

Table 6.1 - Pulse Width Table for Speed Controller

Input Pulse Width	Output Signal
1.9 ms (maximum)	100% Forward Duty Cycle
1.7 ms	50% Forward Duty Cycle
1.5 ms (neutral)	No Output
1.3 ms	50% Reverse Duty Cycle
1.1 ms (minimum)	100% Reverse Duty Cycle

Chapter 7 - Control Algorithms

Control is a common concept, which you have probably encountered before. When you drive an automobile, you are controlling it in order to safely arrive at a planned destination. Systems such as an automobile are in the category of manual control. When no human interaction is involved, we refer to this as automatic control. A good example is room-temperature control, where a furnace is turned on and off depending on the desired temperature and a thermostat reading of the current temperature.

This section will serve as a brief introduction to basic control algorithms. Automatic control is a mature discipline, with an extensive body of knowledge, most of which is beyond the scope of this course. For more detailed information you should refer to a control textbook such as *Modern Control Systems*, by Richard Dorf, which is available in the library. The book *Modelling and Analysis of Dynamic Systems,* by Charles Close and Dean Fredrick, is another reference with which you may be more familiar and contains some material in more detail.

To solve a specific control problem, the designer must make decisions about the type of control to use, including whether to add more hardware or make the software more complex. There is a constant trade-off depending on the requirements for speed, expendability, and cost. One of the advantages of embedded control is that the bulk of the solution can be handled in software, which can be changed without much difficulty. The microcontroller remains the center of the control system by coordinating all the components that interface with the environment.

The basic objective of a control system is to force the output *y* (*Figure 7.1*) to equal the input or reference *r* by designing the controller so that the input to the system from the controller *u* (the *manipulated variable*) drives the output *y* (*controlled variable*) to equal the reference. This is normally done by controlling an actuator, such as a motor, valve, etc. When the manipulated variable is adjusted according to measurements of the controlled variable, we call this *closed-loop*, or *feedback*, control.

Closed-Loop Control

To illustrate the concept of control, let us look at a system you are familiar with. First, we will take a look at the manual control of such a system, and then at how it is controlled automatically.

To regulate the speed of an automobile the driver uses the gas pedal, adjusting the flow of gasoline into the engine. Say the driver wants to maintain a speed of 55 m.p.h.; if the current speed is below this reference, she will have to press on the gas; conversely, if it is above she will have to release it a little. Eventually, she will be able to maintain the pedal at a given position and maintain the desired speed, so long as the grade of the road does not change.

In a car that has cruise control, the driver has the option of setting the desired speed (reference), and allows the car to take over the adjustment of the gas. If we have the same situation where the grade of the road does not change, we could easily map the different positions of the gas pedal to different speeds, and just select the one that corresponds to the desired reference. But in practice the grade of the road will change. Another factor that makes such a solution impractical is the degradation of the system over time, as well as changes in the quality of the gasoline, etc. In control terms, we call these effects *disturbances*.

The approach taken in control to eliminate or minimize the effect of such disturbances is to use feedback. This is, we take a measurement of the variable we want to regulate, and we feed it back to the controller to compare with our desired reference value. Then, the controller can adjust the manipulated variable as a function of the error - the difference between the reference and the actual measurement (*Figure 7.1*).
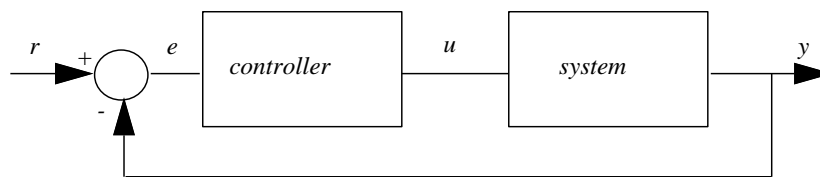
Figure 7.1 - Simple closed-loop control system

In the case of the cruise control, if the error starts to increase (actual speed of the car is lower than the desired reference), the system has to increase gas flow. If the error becomes negative, then we are going faster than we want to, and the flow of gas has to be decreased. In essence, the controller is mimicking the human driver. The process by which the controller calculates the manipulated variable as a function of the error is the control algorithm.

Once a control algorithm is picked, it can then be tuned to obtain the desired performance of the system. In the case of cruise control, by tuning we could set how aggressively the controller will react to changes in the speed. We could have it be very fast in responding (resulting in constant and abrupt acceleration and deceleration), or very slow (taking a long time to correct for changes in the grade of the road). Ideally, we want to tune it so the car will respond smoothly and in a reasonable amount of time.

In order to specify the desired performance of a closed-loop system (this is, a system with feedback control), there is a whole set of terminology that is used. This allows us to give actual performance values and criteria that can be used in the design of such a system, rather than verbal descriptions, which are difficult to quantify.

Control Terms
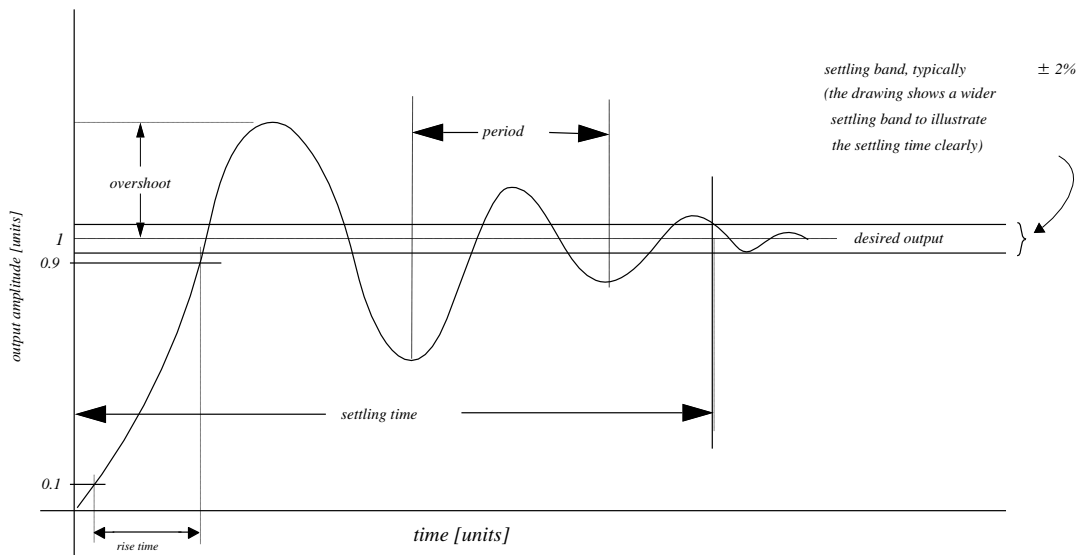
Figure 7.2 - System response for control term definitions

There are several important terms that make it easier to describe the performance of control systems. Most of these terms are indicated in *Figure 7.2*. The *settling band* is the region around the reference (setpoint, desired output) in which it does not matter whether the system oscillates, though the controller may still have an effect on the system output. The settling band can be specified as a percentage as well; a typical value is ±2%, but could also be ±1%, ±5% or some other value, depending on the application and the performance required. The *steady state* response is the value the system output attains as time approaches infinity, provided that the system is stable. Ideally, the steady state output is equal to the reference, but in practice it is acceptable if these values are close (how close will depend on the application). If the final steady-state value of the system differs from the desired setpoint, we call this the *steady-state error*. If a settling band is defined, it should enclose this difference. The *settling time* is the time it takes the output amplitude to reach and stay within the settling band. The *rise time* is the time it takes for the output amplitude to go from 10% to 90% of the steady state value of the system (the concept of rise time is independent of the settling band). Typical rise time values will vary for different types of systems; for an aircraft rudder control, the rise time could be of the order of milliseconds, while in a chemical process control system, the rise time could be of the order of hours! Like settling time, this specification depends on the speed of the system response. The *overshoot* is the amount the output amplitude goes above the reference (or under when the reference is negative). Some systems can tolerate no overshoot while others can handle a large amount. Typically systems that allow overshoot may show an oscillatory response, as is the case in *Figure 7.2*. In these cases, a *period of oscillation* can be determined. The *frequency of oscillation* is defined as the inverse of the period

of oscillation. Additionally, there are other control concepts that will be introduced briefly in the next section: *overdamped*, *critically-damped* and *underdamped* systems.

Before going on to the algorithms, keep in mind that for most systems control engineers will work with mathematical models of the system they want to control. This allows them to perform simulations on a computer instead of experimenting with the real system. More importantly, a good mathematical model of a dynamic process captures the behavior of the system (sometimes called the "plant" in control systems textbooks), which then allows the engineer to design the controller to perform according to the specifications without the need to tune it experimentally. Because of the limitations in time and scope of this course, we will resort to the more ad-hoc approach of experimentation to tune our controllers.

Proportional Control

Proportional control is a simple type of closed-loop feedback control. The input to the system adjusts linearly to the difference between the system output and the reference. This may allow the closed-loop system to quickly approach its goal if the closed-loop system is stable (if the closed-loop system is unstable the output will not go to its goal, it will diverge). Depending on the system properties, THIS TYPE OF CONTROL MAY OR MAY NOT WORK! There are several techniques used to analyze stability properties. You may learn these in other courses such as Modeling and Control of Dynamic Systems, Signals and Systems, Discrete Time Systems, Control Systems Engineering, Chemical Process Dynamics and Control, Dynamic Systems for Biomedical Engineering, or Mechatronics.
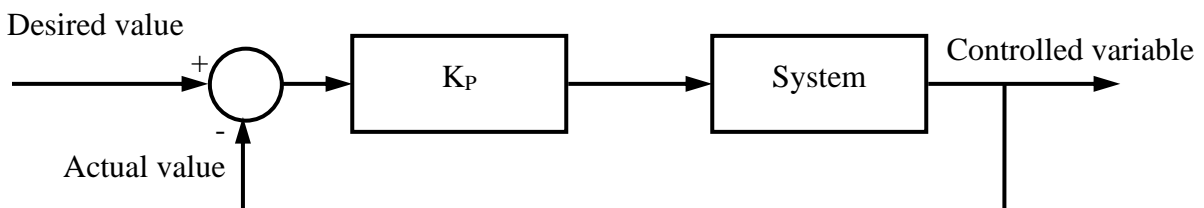
Figure 7.3 - Proportional control block diagram

In a proportional controller, the gain, K_P, is defined as the value of the multiplier that is applied to the difference between the desired value and the measured value of the output, which may or may not be the same as the actual output because of sensor errors (*Figure 7.3*). By varying the gain, the system response can be varied. For some gains, the system may approach the desired value with no overshoot but have long rise and settling times (*Figure 7.4*, curve A). This is referred to as an *overdamped* system. For some other gains, the system may overshoot the desired value but still decay toward the desired value (*Figure 7.4*, curve B). This is referred to as an

underdamped system. For other gains or all gains, depending on the system properties, the system may go unstable (*Figure 7.4*, curve C). There is an intermediate case defined between the overdamped and underdamped system, the *critically damped* system (*Figure 7.4*, curve D); suffice to say that it overshoots slightly, but with no oscillations and settles into the steady-state value. We are not going to get into further explanations here, because it would require a lengthy analytical procedure that is beyond the scope of this chapter. You may refer to any basic control system theory book for further information. These concepts are usually introduced in the context of second order systems.
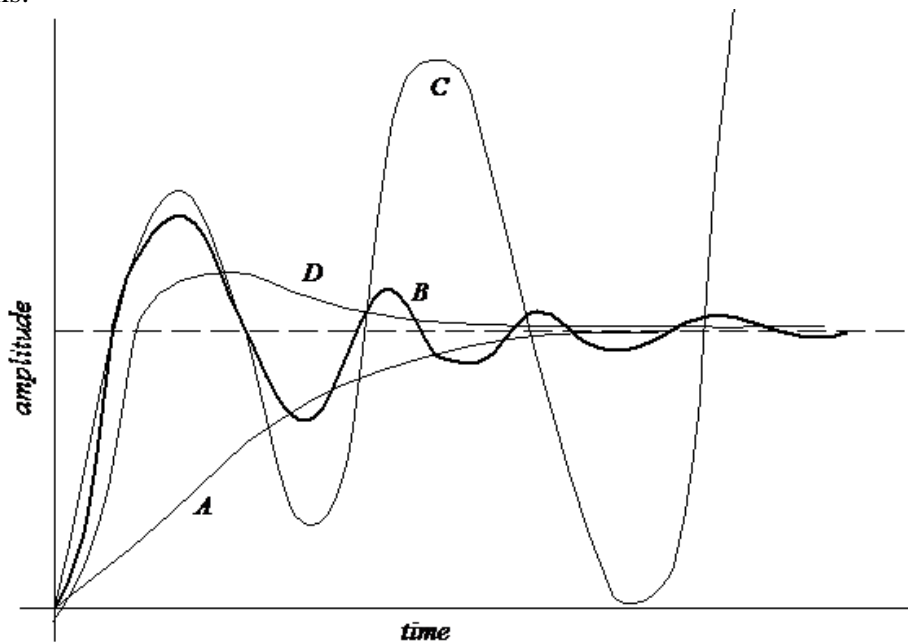
Figure 7.4 - System responses with different gains

The three parameters (rise time, settling time, and overshoot) may be used to determine the gain, depending on the requirements of the system. There are various techniques used in control system design to find the adequate gain analytically. Since real life systems are inherently non-linear, this can be non-trivial. Another method for control system design is "trial and error". This method is NOT GUARANTEED TO WORK! While less efficient and reliable, this technique requires a less accurate system model and less understanding of control design techniques. The gain is chosen following some "rules of thumb": Typically, decreasing the gain increases the system rise time (i.e., it decreases the response speed). On the other hand, increasing the gain decreases the system rise time and increases the system overshoot, and in some cases could make the system unstable.

Analytically, the control law for a proportional controller is given as:

$$u(t) = K_P e(t)$$

where $u(t)$ is the output control signal, K_P is the proportional gain constant, and $e(t)$ is the error.

Since we are dealing with a discrete-time system, we can replace t by k, which is the time-step at which the calculation is done. In proportional control this is not a critical factor, as the new value of the manipulated variable u does not depend on what has happened in the past.

Proportional plus Integral Control (PI Control)

To improve the system response without making the gain large, an integral term can be added to improve steady-state control (this is, to minimize the steady-state error, in many cases to zero).
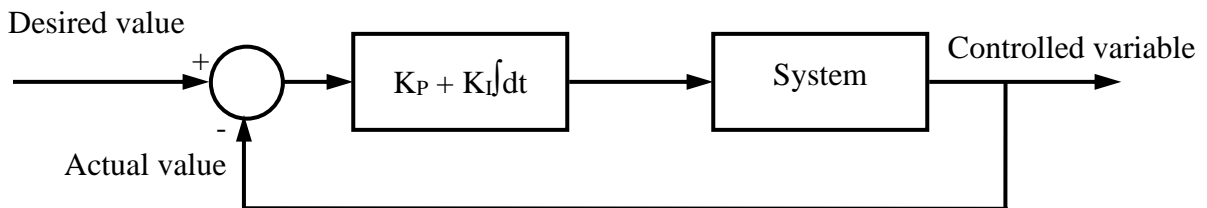
Note that PI control works if the system has natural damping. Since the damping for *Gondola* is very small, we use PD control instead.

By integrating the error over time, an additional element will be utilized that will force the output to the setpoint. As long as the error is non-zero, the integral will keep growing; when the error finally reaches zero, then the integral's value will stay constant, providing the needed extra forcing of the system. If the instantaneous error is zero, then the value of the manipulated variable will only be the integral term.

Analytically, the control law for a PI controller is given as:

$$u(t) = K_P e(t) + K_I \int e(t)dt$$

where $u(t)$ is the control signal being sent, K_P is the proportional gain constant, K_I is the integral gain constant, and $e(t)$ is the error.

When implementing the PI control law in discrete time, we have to approximate the integral by a summation. If we again use k to denote the current time instance, we can write:

$$u(k + 1) = K_P e(k) + K_I \sum_{j=0}^{k} e(j)$$

where $u(k+1)$ is the next control signal to be implemented and $e(k)$ is the current error. The summation considers all the previous error values found at each discrete time interval.

An alternative form of this equation can also be used: this considers the current control signal being implemented, $u(k)$, the current error, $e(k)$, and the previous error, $e(k-1)$.

$$u(k + 1) = u(k) + K_P[e(k) - e(k - 1)] + K_I e(k)$$

You should be able to derive this last equation from the previous one through simple algebra. When implementing an integral control it is important that the time interval at which the next control move $u(k+1)$ is calculated remains constant.

Another implementation of a PI controller calculates the integrated error $e_{int}(t)$ separately and combines the terms as:

$$e_{int}(k) = e_{int}(k-1) + e(k)$$
$$u(k+1) = K_P e(k) + K_I e_{int}(k)$$

Proportional plus Derivative Control (PD Control)

PI control works if the system has natural damping. Since the *Gondola* has small damping effects, you will use PD control in your lab. The derivative term provides damping, thereby allowing larger proportional gains that results in quicker response. Note that PD control will not remove steady state errors.
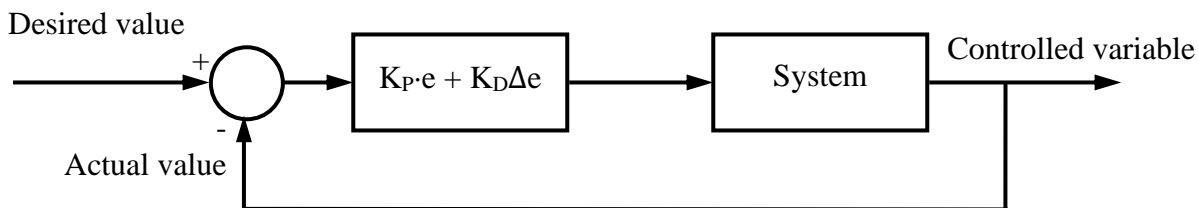
Analytically, the control law for a PD controller is given as:

$$u(t) = K_P e(t) + K_D \frac{d}{dt} e(t)$$

where $u(t)$ is the control signal being sent, K_P is the proportional gain constant, K_D is the derivative gain constant, and $e(t)$ is the error.

When implementing the PD control law in discrete time, we have to approximate the derivative by a subtraction. If we again use k to denote the current time instance, we can write:

$$u(k+1) = K_P e(k) + K_D[e(k) - e(k-1)]$$

where $u(k+1)$ is the next control signal to be implemented, $e(k)$ is the current error and $e(k-1)$ is the previous error.

Other Considerations

Even though the principles of analog control using the trial and error approach can be applied to this control problem, they may not always work. With more difficult problems, it is necessary to develop and analyze the system model with your controller before you build it so that catastrophic mistakes can be avoided.

There are also many other strategies for controlling a system. You may have heard of fuzzy-logic control for instance, or neural network control. Beyond these, there are more advanced strategies such a model-predictive control (MPC), adaptive control (with many variations), optimal control, and many more. These other topics are usually covered in graduate level courses.

Chapter 8 - Troubleshooting

Hardware

In this course you will be assembling a substantial amount of circuitry. Occasionally you may make an error in wiring, or you may encounter a malfunctioning circuit component. Since hardware problems are common during circuit development, it is crucial to know how to diagnose circuit problems independently rather than depending on the TA. Being able to troubleshoot a problem with a system is much more than just a skill—it demonstrates true understanding of underlying principles. It is difficult to diagnose a system that isn't understood.

As an engineer, you are *expected* to possess troubleshooting skills. Engineers, like medical professionals, are counted upon to be able to identify a problem, analyze its symptoms, and determine its cause. Developing a critical mind is the key to debugging, and this chapter gives some hints on what to look for when debugging electronics.

Short Circuits

A short circuit or "short" occurs when two wires in the circuit are unintentionally connected. This often occurs in haphazardly wired circuits when adjacent bare wires touch or when a wire is connected to the wrong location. A short between power and ground can be dangerous if the power supply is not short protected. Such a short might cause the power supply to rapidly heat up, quickly destroying itself and possibly other nearby equipment. If a short circuit occurs on your protoboard, one of the fuses on the power board will blow to protect the battery. This fuse will need to be replaced. Your TA can supply you with another fuse, but first, you must check your board to correct the problem to prevent another fuse from blowing.

Determining the existence of a short between power and ground is very easy—just use the multimeter to read the voltage across power and ground to see if there is a potential difference. If there isn't, then a power-to-ground short is present somewhere in the circuit. You should turn off the power supply and visually inspect for the short.

While the power supply is *off*, another way to determine the existence of a power-to-ground short is by checking the continuity between the power and ground terminals. Disconnect your circuit from the power supply. Set the multimeter to its *diode-check* mode and connect it across the power and ground terminals on your protoboard. If the multimeter emits a continuous beep, then a short exists between power and ground somewhere in the circuit. Visually inspect your circuit for red and black wires connected to the wrong vertical rails. If a visual inspection doesn't reveal the short, then try using the multimeter to test the continuity between power and ground

while removing power or ground wires one at a time from the circuit. When the short has been removed, the multimeter will stop beeping.

Shorts, other than those between power and ground, are somewhat more difficult to diagnose. Bare wires, unintentionally touching, fall under the general heading of *shorts*. Remember, that sometimes problems are misdiagnosed as being hardware-related when in fact the problem is rooted in faulty software.

Crossed Wiring

The term *crossed wiring* is often loosely applied to many different types of wiring errors, but it generally refers to a *set* of wires that have been misconnected to a *set* of points, usually in an orderly fashion. For example, consider the situation where two wires, W_1 and W_2 should be connected to two points on a circuit, P_1 and P_2 respectively. If the wires are mistakenly connected in the reverse order, W_1 to P_2 and W_2 to P_1, then the term *crossed* wiring would apply—W_1 and W_2 were *crossed*.

The easiest way to check for this type of error is to check the order of connections on sets of associated wires. For example, the input and output ports on the EVB each have eight data lines. If a known, *distinguishable* pattern is written to one of the output ports via software, then by using the logic probe to test the data lines, it will become immediately apparent if connections to these lines have been crossed.

Logical Errors

Logical errors are the most difficult to find because they are caused by incorrect circuit design (i.e., your schematic is incorrect), not careless wiring. An error of this type can only be found by having a thorough understanding of what the circuit is *intended* to do, and a thorough understanding of how each circuit component operates. One example of an error of this type is that an incorrect integrated circuit (IC) chip was mistakenly used, e.g., you took the wrong IC from the parts bin. If you understand how the "correct" IC is supposed to function, then by checking the input-output relation of the IC in question, you will quickly determine that the IC is not functioning as specified. A closer examination of the IC may then reveal that you've mistakenly used the *wrong* IC. Use the multimeter or logic probe to test the functionality of the IC. With the voltages on the IC's input pins at a certain level, check if the voltages on the IC's output pins are correct.

EVB Not Responding

One common problem is an EVB not responding when the user attempts to download a program to it. This will usually produce the message, "Target could not be reset. Confirm cable and power connections and retry." The first thing that should be checked is the power and ground

connections. If the connections are correct, check that the source of power is on (make sure the red EVB power LED is on). If the power source does not light the LED, check the fuses located on the car's power connection board. Another possible cause is the serial cable. Check to ensure that a proper connection is made at the EVB connector, as well as the computer's connector port.

During the car sequence, the battery should be checked with a voltmeter to ensure that the battery has enough voltage. If the batteries have a sufficient charge, check the output of the voltage regulator located on the chassis of the car. The output voltage should be approximately 5 volts. If the voltage falls below 4.8 volts, the EVB may not respond.

Software

During the lab sequences, complex software codes will be developed with multiple functions and routines. These complex codes leave a great deal of room for error. Syntax errors are easily remedied with the aid of the C compiler. Logical errors, on the other hand, are quite difficult to find. Logical errors can place your code in an endless loop or even prevent your program from entering a significant loop. During debugging, these logical errors can leave the programmer stumped for several hours. The only sure fire way of remedying the logical errors is with a logical, systematic approach. First, determine the intended response of the program. Next, determine the actual response obtained when the program is running, and compare the responses. If the cause of the problem is still undeterminable, placing print statements at strategic locations, such as before and after loops and conditional statements, can help locate the errors. Once the location of the faulty statement is discovered, attempts to rectify the error can take place.

To reduce the chance for error and simplify the debugging effort, programs should be well constructed. The program should be made modular in structure; well commented, and variable names should be well thought out to minimize any possible confusion.

Output problems

The EVB displays output by using a serial connection to send data to a program called HyperTerminal, which may be running on the same computer as the development IDE, or on a different computer. Either way, a serial cable must be connected from the EVB's output serial port to a serial port on the computer being used for display, and Secure CRT or HyperTerminal must be running on that computer. If terminal program is not displaying the output, it may be necessary to quit the program and restart it.

If restarting HyperTerminal does not solve the problem, you may need to create a new connection to be sure the settings are correct. On starting HyperTerminal, you may be asked to designate a new connection, or else you can choose the "New Connection" option under the "File" menu. The first screen will request a name for the connection - choose a simple name such as

"litec", and click on the "OK" button. The next screen will be titled "Connect To", and the last setting on the screen will be "Connect using:" and offer some options - choose one of the COM port options (usually COM1 or COM2), and click on "OK". The next screen will allow you to set the COM port settings; set "Bits per second:" to **38400**, and "Flow control:" to **None**; click on "OK". From there, HyperTerminal should be set to correctly communicate with the EVB. The baud rate is 39400 and all flow control must be turned off.

Glossary

!

   This is the logical negation operator and if the operand following ! is TRUE, the result is
   FALSE. If the operand following the ! is FALSE, the result is true.

   Usage: *! (5 > 4)* is FALSE, because (5 > 4) is TRUE, and ! TRUE is FALSE
   Context: logical operator
   See also: !=, &&, ||, ~

!=

   The not-equal-to operator. It tests for inequality between two operands. It returns TRUE only
   if the left operand is not equal to the right operand.

   Usage: The code *(5 != 5)* returns FALSE, the code *(5 != 3)* returns TRUE
   Context: C relational operator
   See also: =, !

~, |, &

   This is the bitwise negation operator, bitwise OR operator and bitwise AND operator.

   Usage: ~0xF3 is equivalent to 0x0C.
   Context: bitwise operator
   See also: !, &

|=, &=

   This is a C abbreviation. The code variable1 |= variable2; is equivalent to the code variable1 = variable1 |
   variable2;

   Usage: The code *a = 0x01; a |= 0x02;* results in *a* containing the value 0x03
   Context: C language syntax
   See also: |, ||

||, &&

   This is logical OR operator and logical AND operator. The OR operator takes two operands
   and evaluates them, and it returns TRUE only if either of the operands are TRUE.

   Usage: || is often used in conditional statements such as "if" or "while"
   Context: logical operator
   See also: &, |

^(1)

   In other compilers, this is used in variable declarations to specify the position of the desired bit
   for bit-addressable registers such as P0, P1, etc.

   Usage: The code *push_button = P3^4;* will associate the variable *push_button* with bit 4 of port 3 of the C8051 EVB.
   Context: C compiler syntax

^(2)

   When used in a calculation in a C program, this operator performs a bitwise exclusive OR
   operation.

   Usage: The code *a = 0x01; b = a^0x03;* will set variable *b* equal to 0x02;
   Context: ANSI C bitwise operator
   See also: ^(1), bitwise operator

#define

   This is an instruction to the C compiler that replaces all occurrences of a single word with the
   text that follows it in the define command. This is done before the program is compiled. The
   word to be replaced is usually in capital letters to distinguish it from program variables and
   other strings.

Usage: The C program line *#define GAIN 300* replaces all occurrences of the text GAIN with the text 300 before the program
               is compiled.
Context: C preprocessor directive
See also: #include

#include

Tells the C compiler to include code and function declarations contained in the file indicated. Many of these files, such as stdio.h and math.h, are included with the compiler.

Usage: The C program line *#include <stdio.h>* includes the definitions and code contained in the file stdio.h.
Context: C preprocessor directive
See also: stdio.h, c8051.h, math.h, #define

%

This is the modulus operator. It returns the remainder of a division operation.

Usage: The expression *6%4* will return a result of 2
Context: C arithmetic operator
See also: mod

&

This character has two purposes depending on how it is used. If it placed between two operands, it is the bitwise (or binary) AND operator. The bitwise AND operator returns the result of an AND operation performed on each pair of bits in the two operands. If & is placed in front of a variable it is the address operator. It takes the address of the variable it is placed in front of. i.e. it returns the location in memory where that variable is located.

Usage: The bitwise AND operator is used quite often in Embedded Control, especially for I/O port operations. Don't use the
               address operator unless you are familiar with the concept of a pointer.
Context: bitwise operator, C unary operator
See also: bit mask, &&
See also:

0x

This is used to designate a number expressed in hexadecimal notation (hex). Hexadecimal numbers use the digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F.

Usage: 0x3F is the same as 63 in base 10 or 0011 1111 in binary
Context: C language syntax
See also: binary, decimal, number conversion

74F04, 74LS04

This is a hex inverter chip used in Embedded Control. An inverter is a logic device with a single input and a single output. When the input is 0 volts (low), the inverter makes the output 5 volts (high). When the input is 5 volts (high), the inverter makes the output 0 volts (low). Hence, it "inverts" the logical state of the input. The "hex" part of the name means that there are six independent inverters in one chip package.

Usage: The 7404 has TTL level outputs, meaning it produces enough current to drive other logic inputs, or microprocessor
               inputs. It can drive an LED, but it will not be incredibly bright (max output is 8 mA). If you wish to have slightly
               brighter LEDs, or to drive higher current loads consider using the 7405.
Context: Electronic component
See also: 74F05, 74LS05

74F05, 74LS05

This is a hex inverter chip with open collector outputs. Open collector outputs provide a higher current capacity for driving LEDs, buzzers, and other higher current load devices. However, to use the 7405 to drive other logic inputs or microprocessor inputs, you must use what is called a "pull-up" resistor. This is simply a high value resistor (1k - 10k), which is connected between the output of the 7405 and 5 volts. This resistor supplies the current necessary to "pull" the

output up to a high value. The open collector output cannot do this without that resistor. See the description of the 7404 for information on general-purpose inverters.

Usage: Use the 7405 to drive any high current devices you might want to use.
Context: Electronic component
See also: 74F04, 74LS04

<, >, ==, <=, >=

The less-than, greater-than, equal to, less-than-equal-to and greater-than-equal-to operators. For example, the less than returns TRUE only if the left operand is of lesser value than the right operand.

Usage: The code *(4 < 5)* returns TRUE, the code *(5 < 4)* returns FALSE.
Context: C relational operator
See also: <<, >>

<<, >>

The left shift and right shift operator. Takes the left operand and shifts it to the left the number of times indicated by the right operand. For example, 0x01 << 1 shifts the value 0x01 to the left 1 bit, giving a result of 0x02.

Usage: A left shift of one bit is equivalent to a multiplication by 2.
Context: C shift operator
See also: <, >, <<=

<<=, >>=

A C abbreviation. The code a <<= 1; is equivalent to the code a = a << 1.

Usage: This is rarely used in Embedded Control.
Context: C shift operator
See also: <<

=

This is the assignment operator. It assigns a variable (left operand) to a value (right operand). This must not be confused with the == operator, which tests for equality.

Usage: The code *a = 5;* assigns the value 5 to the variable a.
Context: C assignment operator
See also: ==

\n

This is the notation used in the C language to designate a new line character. If used in a print statement in a program, it causes the text to scroll up one line and the cursor to return to the beginning of the line.

Usage: This is sometimes called an escape sequence (a series of characters that represent one single character).
Context: C syntax
See also: \r, \t, printf()

\r

This is the notation used to designate a carriage return. If used in a print statement in a program, it causes the cursor to go back to the beginning of the current line without jumping to a new line. It is similar to the \n (new line) character, but doesn't start a new line.

Usage: This is sometimes called an escape sequence (a series of characters that represent one single character).
Context: C syntax
See also: \n, \t, printf()

\t

This is the notation used to represent a tab character. This can be used when outputting data that will be imported into a spreadsheet. The spreadsheet will expect the individual data items to be separated by the tab character.

Usage: This is sometimes called an escape sequence (a series of characters that represent one single character).
Context: C syntax
See also: \n, \r, printf()

abs()

Math library function that returns the absolute value of an integer.

Usage: You must *#include <math.h>* to use this function.
Context: Math library function
See also: fabs()

address

The address of a variable is the location in memory where its value is stored. Addresses are usually expressed in hexadecimal.

Usage: In Embedded Control we are not usually concerned with specific addresses.
Context: Microprocessor concept
See also: variable, RAM, ROM

analog

This refers to a value or signal that is continuous in nature. It can take on an infinite number of values. Analog signals must be converted to digital values before they can be understood by a microprocessor.

Usage: The analog signals used in Embedded Control are usually voltages between 0 and 5 volts.
Context: Electronic term
See also: digital

analog circuits

Circuits composed of non-discrete components such as resistors, capacitors, transistors, and op-amps. The signals generated by these circuits are continuous in nature. The opposite of an analog circuit is a digital circuit.

Usage: The OTU circuit is an analog circuit.
Context: Electronics Concept
See also: analog, digital, digital circuit, resistor, capacitor, op-amp

analog-to-digital conversion (A/D Conversion)

This is the process of quantizing an analog signal into a digital signal. The precision of the quantized value is determined by number of bits assigned to the digital value. For example, an 8-bit conversion is more precise than a 4-bit conversion.

Usage: The Evaluation Board (EVB) can perform A/D conversion on any of 8 different pins.
Context: Microprocessor Concept
See also: analog, digital

anode

Positive terminal of a two terminal semiconductor component, such as an LED. On an LED it is designated by the longer lead, or the side of the LED opposite the flat spot.

Usage: The anode of an LED is usually connected through a series resistor to power.
Context: Electronic term
See also: LED, cathode

ANSI C

This refers to the C standard published by the American National Standards Institute (1989).

Usage: The SDCC C compiler is based on the ANSI C standard.
Context: C programming term
See also: SDCC C compiler

array

A group of associated variables that can be accessed with a numerical index.

Usage: *myarray[4]* refers to the variable in the array *myarray* with index 4. In C, array indices start with 0, so it is actually
            the 5th element in that array.
Context: Programming concept
See also: index

ASCII

ASCII is a set of characters usable on a computer and numbers that represent them. For instance, the lower case g has an ASCII value of 103.

Usage: In C, you can get the ASCII value of a character by enclosing it in single quotes (e.g. 'g' is equal to 103).
Context: Programming concept
See also: getchar()

assembly language

A very low level computing language in which you are dealing with the basic commands that the microprocessor is executing. The C programs you write in the lab are translated to assembly language so they can be executed by the microprocessor.

Usage: You will not have to write programs in assembly language in Embedded Control.
Context: Programming language
See also: compile, C programming language

associativity

Refers to the order in which a mathematical expression is evaluated. For instance, the expression 3 + 4 * 5 could be evaluated as (3 + 4) * 5 or as 3 + (4 * 5), depending on the rules for associativity.

Usage: The rules of associativity for C can be found in most C programming texts
Context: C Programming syntax
See also: Kernighan & Ritchie, C Programming language

baud

A measure of serial transmission speed. Stands for bits per second.

Usage: The serial transmission speed to the Evaluation Board is 57,600 Baud
Context: Computer unit
See also: serial

binary

Refers to a number expressed in base 2. Binary numbers use the digits 0 and 1. Binary number are usually only used in the context of microprocessors.

Usage: 11011011 is a binary number equivalent to decimal 219
Context: Mathematical concept
See also: hexadecimal, decimal

bit

This is the smallest unit of computer memory. It consists of one location that can store one of two values, usually 0 or 1.

Usage: A byte is made up of eight bits
Context: Microprocessor concept
See also: byte, nibble

bit mask

102

In order to isolate a particular bit or bits of a result, bitwise AND the result with a value called a bit mask. The purpose is to clear all of the other bits so that all that is left is the information in the bit(s) we are concerned with. For instance, to examine the rightmost bit of the Port 3 data register, bitwise AND it with 0x01 (P3 & 0x01).

Usage: You will use a bit mask for much of the digital I/O you will do in Embedded Control.
Context: Programming concept
See also: bit, bitwise operator

bitwise operator

Refers to an operator that works on the individual bits that make up a value. For instance, the bitwise AND operator (&) performs an AND operation on each individual bit of its operands.

Usage: You should be careful not to confuse the bitwise operators with logical and assignment operators.
Context: Programming term
See also: logical operator, assignment operator

brace (curly bracket) {}

Used in C to indicate the beginning and end of a program block.

Usage: The body of a function is set out in braces (curly brackets)
Context: C language syntax
See also: bracket, parenthesis

bracket (square bracket) []

Used in C to designate the index of an array.

Usage: *my_array[i]* indicates the ith element in the array my_array[]
Context: C language syntax
See also: brace, parenthesis

breadboard

This is an area where electronic components can be temporarily connected. Components can be easily inserted and removed without soldering or special tools.

Usage: Breadboards are used for all of the circuits constructed in Embedded Control.
Context: Lab equipment
See also: protoboard, Evaluation Board

buffer

A region of memory (or a file) where incoming and outgoing data is temporarily stored until it is required.

Usage: When printing a constant flow of data, if the monitor cannot respond quickly enough, the data is stored in the buffer.
Context: Microprocessor Concept
See also: memory, Random Access Memory (RAM)

bus

This is a term for several points that are connected together electrically. It can refer to a point in a circuit where many signals are connected together.

Usage: Breadboards utilize several buses where many components are connected to a single point.
Context: Electronic term
See also: breadboard, protoboard

byte

A byte is a group of eight bits. It is a common unit of computer memory. One byte is usually used to represent one character. It is often used with prefixes as in Mb which indicates a Megabyte.

Usage: Many of the special function registers (SFRs) on the C8051 are one byte in size
Context: Microprocessor concept
See also: bit, nibble

SDCC C compiler

This is a program that is used to convert your C program (a text file) into a binary format that the C8051F020 microprocessor can understand. This process is called compiling. This program also informs you of syntax errors in your C program. In the documentation, it may be referred to as the SDCC C compiler.

Usage: This compiler is used in the Embedded Control lab. It is licensed and distributed by a company called SDCC.
Context: Software Tool
See also: compile, program, C8051F020, gcc compiler

C8051F020

This type of microprocessor manufactured by Silicon Labs is used in the Embedded Control lab. You will find it on the Evaluation Board (EVB). It is a 16(??)-bit processor equipped with timers, I/O ports, and analog/digital converters.

Usage: The C8051F020 is a general-purpose microprocessor used in many applications
Context: Microprocessor concepts
See also: Evaluation Board (EVB), microprocessor

c8051.h

A C header file specifically written for the C8051F020 Evaluation Board as it will be used in LITEC. It initializes and configures some components of the microprocessor.

Usage: The C code *#include <c8051.h>* includes the c8051.h header file as part of the program.
Context: C language concept, C8051 microprocessor concept
See also: #include, registers

c8051f020.h

A C header file included with the SDCC C compiler. It contains the names and definitions of the registers used by the microprocessor.

Usage: The header file c8051f020.h enables you to access the registers and ports of the C8051F020.
Context: C language concept, C8051 microprocessor concept
See also: #include, registers

capacitor

An electronic component that stores charge. Capacitance is measured in farads or microfarads (µf).

Usage: Capacitors are used to construct filters.
Context: Electronic component
See also: filter

cathode

Negative terminal or electrode of a load component (not a source), such as an LED. On an LED it is designated by the shorter lead or the side of the LED that has a flat spot.

Usage: The cathode of an LED is usually connected to ground.
Context: Electronic term
See also: Light Emitting Diode, anode

ceil()

This function returns the smallest integer number which is greater than the number passed to it. The name comes from the word ceiling (an upper limit).

Usage: *ceil(4.6)* returns a value of 5
Context: C math library function
See also: floor()

char

The char data type is typically an 8-bit value. It is named this because it is the size used to represent the ASCII character set.

Usage: An unsigned character can represent the values 0 - 255.
Context: C data type
See also: int, float, double, signed, unsigned

clear

Refers to the act of setting a particular register bit to 0. We say we are "clearing" a bit when we set it to 0, and we say we are "setting" a bit when we set it to 1.

Usage: The code *P3 = P3 & 0xFE;* clears the rightmost bit of the Port 3 data register.
Context: Microprocessor term
See also: set, &

clock

Microprocessors use a common signal which oscillates at a fixed frequency to coordinate the various activities that it performs. This signal, or the circuit that produces it, is often called the clock. The frequency at which it oscillates is kept under very tight control. Consequently, time can be measured by counting the oscillations of the clock signal.

Usage: The clock of the C8051F020 used in the lab oscillates at 22.1184 MHz.
Context: Microprocessor concept
See also: Microprocessor, Hz

color band

Resistor values are often indicated by a series of colored bands on the body of the resistor. These colors represent numbers that can be used to determine the value and tolerance of the resistor.

Usage: There is a resistor chart in *Appendix C* which shows how these colors represent resistor values.
Context: Electronic component identification
See also: resistor

comparator

A comparator is an integrated circuit that takes in two analog signals and produces a digital value based on which of the two analog signals is greater.

Usage: A comparator can be used to determine if a signal under investigation is greater than a known reference voltage.
Context: Electronic component
See also: analog, digital, integrated circuit, op-amp

const

The data type const is used to declare a variable as constant. This means that the value it is given initially cannot be changed.

Usage: const can be used instead of #define to declare the gain variable in your control algorithm.
Context: C data type
See also: char, int, double, float, constant, #define

constant

A value or variable that does not change.

Usage: You can use the #define directive to set up names for any constants you may need in your programming

Context: Programming concept

See also: const, #define

counter

In programming, a counter is a variable that is incremented every time a particular event occurs. The value of this variable is the number of times that event occurred.

Usage: Counters are used frequently in Embedded Control to keep track of various events.

Context: Programming term

See also: increment, decrement

CMOS, HCMOS

CMOS is an abbreviation for Complimentary Metal Oxide Semiconductor (HCMOS is a higher speed version of CMOS). It is a type of logic circuit. CMOS chips are resistant to noise and operate from a wide range of supply voltages. They are susceptible to damage due to static discharge.

Usage: HCMOS chips are sometimes used in the lab. They are indicated by chip numbers containing HC such as the 74HC237

Context: Electronic term

See also: TTL logic

curly bracket (brace) {}

Used in C to indicate the beginning and end of a program block.

Usage: The body of a function is set out in braces (curly brackets).

Context: C language syntax

See also: bracket, parenthesis, brace

data type

The name for the several different kinds of variable that can be used in programming language.

Usage: The major C data types you will use are char, int, float, and double.

Context: Programming term

See also: char, int, float, double, declaration

debugging

The process of analyzing the behavior of your program to determine why it is not operating as intended.

Usage: Good debugging practices will save you a lot of time in the Embedded Control course

Context: Programming concept

decimal

Refers to a number expressed in base 10. This is the kind of numbers most commonly used.

Usage: Decimal numbers use the digits 0,1,2,3,4,5,6,7,8 and 9.

Context: Mathematical concept

See also: hexadecimal, binary

declaration

Most programming languages require that you specify some details about a variable or function before you attempt to use it. This is called a declaration.

Usage: The code *int i;* is a declaration for the variable i of type integer.

Context: Programming concept

See also: initialization, data type

decrement

To decrement a variable is to decrease its value by 1.

Usage: The code *i--;* causes the value of i to be decreased by one (same as *i = i - 1;*).
Context: Programming term
See also: increment

digital

This refers to a value or signal that is discrete in nature. That is it can take on one of a limited number of values.

Usage: The digital signals used in Embedded Control are voltages of either 0 or 5 volts.
Context: Electronic term
See also: analog

diode

A diode is a single semiconductor PN junction. It allows current to flow in only one direction. Certain diodes emit light when current is passed through them (Light Emitting Diodes, or LEDs)

Usage: Diodes are used on the smart car chassis to reduce inductive noise from the drive motor.
Context: Electronic component
See also: Light Emitting Diode (LED)

double

A C data type used to store real numbers. The SDCC C Compiler does not recognize the double data type.

Usage: In some compilers, double and float are essentially the same.
Context: C data type
See also: float, long, data type

embedded control

A general term for any time when a microprocessor is "embedded" into a machine to monitor and control its behavior.

Usage: In Embedded Control a microprocessor is "embedded" in the *Smart Car* you construct.
Context: Computer term
See also: microcontroller

Evaluation Board (EVB)

The microprocessor board used in the lab to run Embedded Control programs. It contains a C8051F020 microprocessor, RAM, ROM, and associated circuitry.

Usage: The EVB is what you will use to run your programs and control your lab projects.
Context: Embedded Control term
See also: C8051F020

FALSE

In C, any value that is not equal to 0 is considered to be TRUE. 0 is the only value that is considered FALSE.

Usage: The code *if (0) myvariable = 10;* will not change myvariable, since 0 is considered FALSE.
Context: C language detail
See also: TRUE, if statement, while statement

filter

An electronic circuit which processes a changing signal. It allows certain components of the signal to pass easily, while providing resistance to other components of the same signal.

Usage: We sometimes use low-pass filters to process signals in Embedded Control.
Context: Electronic term
See also: low-pass filter, high-pass filter

flag

A variable that is used to indicate whether a particular event has taken place. For instance, you may have a flag to indicate whether the start button has been pressed yet.

Usage:　　　A flag usually has a value of 1 or 0, but that is not necessarily true.
Context:　　Programming concept
See also:　　variable

float

A C data type used to store real numbers. For the SDCC C compiler, a float is a 32 bit representation.

Usage:　　　In programming, using the float data type can provide greater precision in calculations and data.
Context:　　C data type
See also:　　double, data type

floor()

This function returns the largest integer number which is smaller than the number passed to it. The name comes from the word floor (lower limit).

Usage:　　　The code *floor(4.6)* evaluates to a value of 4
Context:　　Math library function
See also:　　ceil()

for statement

A C statement used to create loops and other repeating structures.

Usage:　　　The code *for (i=0;i<50;i++);* is an example of a "for" statement that creates a short delay.
Context:　　C language
See also:　　while statement

fuse

An electrical device that will interrupt a circuit if the current through it gets too high. When this has happened, we say the fuse is "blown". The type used on teh Smart Car and the Gondola are self resetting.

Usage:　　　There are fuses on the Embedded Control *Smart Car* for protection - these fuses will blow if you have a short circuit.
Context:　　Electrical component
See also:　　short circuit

gate

The general term for a logic circuit, such as an AND or OR circuit. It may also loosely refer to an inverter.

Usage:　　　Gates you will probably be using is the NAND gate and an inverter.
Context:　　Electronic term
See also:　　74F04, 74LS04

getchar()

This function is used to receive an input character from the input buffer of the Evaluation Board. Typically this will be a character that you have typed on the computer keyboard while connected to the EVB. It is included in the stdio.h header file.

Usage:　　　This function can be used to wait for "any key" to be pressed.
Context:　　C library function
See also:　　scanf()

header file

A file that contains function prototypes for library functions. These files end with .h, and are used with the #include directive.

Usage:　　　The header files you will use are included with the compiler.
Context:　　Programming term
See also:　　#include, c8051.h, math.h, stdio.h, string.h

hexadecimal (hex)

This is used to designate a number expressed in hexadecimal notation (hex). Hexadecimal numbers use the digits 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F. In C, the prefix 0x is used to indicate hexadecimal notation.

Usage: 0x3F is the same as 63 in base 10.
Context: C language syntax
See also: binary, decimal, number conversion

high-pass filter

A filter that presents little resistance to high frequency signals but much resistance to low frequency signals. It is used to process a signal to extract out the part of the signal that is changing more quickly.

Usage: You would use an high-pass filter to look only at the alternating part of a signal and ignore the DC level.
Context: Electronics concept
See also: low-pass filter, filter

Hz (Hertz)

The unit of frequency. One Hz is one cycle per second or 1/second.

Usage: The C8051F020 microcontroller on the EVB operates at a frequency of 22.1184 MHz.
Context: Electronic units
See also: ohm, volt

if statement

A C statement used for conditional execution. It evaluates its argument and executes certain code if the argument is TRUE.

Usage: The code *if (5 > 4) myvariable = 1;* will result in myvariable being set to 1 because (5 > 4) evaluates to TRUE.
Context: C language
See also: TRUE, FALSE, for statement, while statement

indentation

It is good programming practice to indent your C code according the level of nesting in the code. For each nested loop or code block, indent the code inside 3-4 spaces more than the loop or code block that contains it.

Usage: It is much easier to read and debug code that is properly indented.
Context: C programming
See also: C programming language

index

Refers to the numerical selector of an array; can refer to a numerical value or a variable used for that purpose.

Usage: In the code *myarray[i];* i would be called the array index.
Context: Programming term
See also: array

input

A signal connection that looks at the voltage or logic level. It does not attempt to change it; it merely determines what it is. Or a value that a function uses to perform actions or calculations.

Usage: You can think of an input as where data flows "into" something.
Context: Computer term
See also: output

int

The C data type for an integer. It can be preceded with several modifiers such as long, unsigned, and signed. An integer is assumed to be signed unless the unsigned modifier is used. A signed integer has a minimum value of -32767 and a maximum value of +32767.

Usage: With the SDCC C Compiler, an int is a 16-bit value and a long int is a 32-bit value.
Context: C data type
See also: integer, double, unsigned, signed, data type

integer

A number that has no fractional part.

Usage: Integers can be stored in variables of type int.
Context: Mathematical term
See also: int

integrated circuit (IC)

A component made up of many transistors (and other components) in a single package. The ICs we use in the lab consist of a rectangular plastic body with eight to sixteen leads coming out. It looks like a large black bug.

Usage: A common IC used in the lab is the 74LS04.
Context: Electronic components
See also: chip

interface

Refers to where a device or program interacts with the outside world.

Usage: The interface of the Evaluation Board includes a 60 pin I/O connector that you connect to the protoboard.
Context: Computer/Electronic term
See also: Evaluation Board

interrupt

Something that "interrupts" the normal linear execution of a computer program. In response to some event, the computer is "interrupted" and goes off to do some other task. When that task is completed, the computer returns to the point it was when the interrupt occurred.

Usage: Interrupts are often used to handle events which occur very often and need only a small amount of processing.
Context: Microprocessor term
See also: interrupt service routine

interrupt service routine (ISR)

This is the name for a function that "handles" an interrupt. When an interrupt occurs, the execution of the program jumps to the interrupt service routine. When the ISR is finished executing, the microprocessor returns to executing the regular program at the point where it left off.

Usage: You will need to write an interrupt service routine for the Real-Time Interrupts used to measure the drive speed.
Context: Programming term
See also: interrupt, Real-Time Interrupt

inverter

An inverter is a logic device with a single input and a single output. When the input is 0 volts (low), the inverter makes the output 5 volts (high). When the input is 5 volts (high), the inverter makes the output 0 volts (low). Hence, it "inverts" the logical state of the input.

Usage: You will use inverters in most of your Embedded Control circuits
Context: Electronic component
See also: 74LS04, 74LS05

Kernighan & Ritchie

This refers to a rather well known book called "The C programming language" written by Brian Kernighan and Dennis Ritchie. This book is considered an authoritative reference on the C language, but contains some conventions that differ from the ANSI C standard which was published later.

Usage: Published by Prentice Hall, RPI library call number QA76.73.C15 K47

Context: C language reference book

See also: "A Book on C" by Al Kelley and Ira Pohl

keyword

A reserved word in C. You cannot have functions or variables with the same name as a C keyword.

Usage: For instance, you cannot name a variable "if".

Context: C programming language

See also: variable, function, C programming language

Light Emitting Diode (LED)

This is a semiconductor junction that emits light when a current is passed through it. An LED has two components, an anode and a cathode. The cathode is designated by a shorter lead, or by a flat spot on the side of the LED. Normally the cathode would be connected to the output of an inverter, and the anode would be connected to one side of a resistor. The other side of the resistor would be connected to power. This resistor is required to limit the current through the LED to a safe value.

Usage: Lab component

Context: Electronic component

See also: diode, resistor, semiconductor

logic probe

A device used to indicate logic levels. It indicates either high or low.

Usage: There is a logic probe in the Embedded Control tool kits

Context: Embedded Control equipment

See also: logic levels

long

A C data type modifier. In most cases it increases the number of bits used to store a data type. With additional bits come additional variable ranges.

Usage: For example, a long int has a range of -2147483648 to +2147483647

Context: C data type

See also: signed, unsigned, int, double

loop

This refers to a program structure where a portion of code is repeated several times.

Usage: Loops can be made with the "for" or "while" statements

Context: Programming term

See also: for statement, while statement

mask

In order to isolate a particular bit of a result, we perform a bitwise AND between the result and a value called a bit mask. The purpose is to clear all of the other bits so that all that is left is the bit we are concerned with. To examine the rightmost bit of the data register of Port 3, we would bitwise AND it with 0x01 (P3 & 0x01)

Usage: You will use a bit mask for much of the digital I/O you will do in Embedded Control

Context: Programming concept

See also: bit, bitwise operator

math.h

An include file for C that contains prototypes for math functions such as $\sin()$ and $\cos()$.

Usage: The C code *#include <math.h>* enables you to use the math library functions
Context: C language concept
See also: #include, sin(), cos()

microcomputer

A computer device that utilizes microelectronics. It has become a very general-purpose term which refers to any computing device which is intended to interface directly with a user. Typically, with some type of output screen and keyboard that allows general purpose computing.

Usage: The laboratory computers can be called microcomputers
Context: Computer term
See also: microcontroller

microcontroller

A computer device that has a very specific purpose: to control something. It may or may not have a user interface. It is generally only useful for a small class of tasks.

Usage: The C8051 can be used as a microcontroller.
Context: Computer term
See also: microcomputer

mod

mod is short for modulus. It refers to taking the remainder of an integer division. The % is the mod operator in the C language

Usage: 6 mod 4 is 2
Context: Programming concept
See also: %

multimeter

A meter that can be used to measure voltage, current, or resistance.

Usage: The Embedded Control tool kits each contain a digital multimeter
Context: Embedded Control equipment
See also: voltage, current, resistance

nibble

Half of a byte, 4 bits

Usage: This term is not often used in Embedded Control
Context: Microprocessor term
See also: byte

noise

Unwanted distortions to a signal caused by external interference.

Usage: Noise can sometimes be a problem in sensitive circuits like the Optical Tracking Units
Context: Electrical term
See also: filter

ohm (Ω)

The unit of resistance. Equal to 1 volt / 1 amp.

Usage: All the resistors in the lab are measured in ohms
Context: Electronic unit

operand

The values that an operation are performed on. The subjects of the process performed by an operator.

Usage: The operands of the + operator in (3 + 4) are 3 and 4
Context: Programming term
See also: operator

operator

An entity in a computing language that indicates a process or operation to be performed on some values, called operands.

Usage: The C operators are grouped into mathematical, logical, relational, and bitwise operators
Context: Programming term
See also: operand, logical operator, mathematical operator, relational operator, bitwise operator

output

A signal connection that changes a voltage or logic level. The component attempts to drive the level at that connection to the desired value. Or a value that is produced by a function.

Usage: You can think of an output as where data flows "out of" something.
Context: Computer term
See also: input

overflow

When a variable is changed to a value greater than its range, we say an "overflow" occurs. In some cases, the value will "wrap around" and start again from 0. The variable would then be set to the amount by which the value was outside of the variables range.

Usage: For example, let i be an unsigned char (range 0-255) that is currently set to 200. If you were to attempt to add 100 to i and overflow would occur. The intended value was 200 + 100 = 300, but the actual value will be 300 - 256 = 44.
Context: Microprocessor term
See also: underflow, data type, unsigned, char

parameter

This is a programming term for the value that is passed to a function. It can be thought of as an "input" to the function.

Usage: Often interchangeable with the term argument
Context: Programming term
See also: function, argument

pass by reference

This is when parameters passed to a function can be changed by that function. That is, the value of the original variable that was used as a parameter can be changed.

Usage: To pass by reference in C, you must use pointers
Context: Programming concept
See also: pass by value, function, pointer

pass by value

This is when parameters passed to a function cannot be changed by that function. That is, the value of the original variable that was used as a parameter cannot be changed. The function uses a copy of the parameter rather than the original.

Usage: Pass by value is the normal parameter passing method in C
Context: Programming concept
See also: pass by reference, function

period

When describing a periodic (repeating) signal, the period is the length of time for one cycle of the repeating pattern. Measured in seconds.

Usage: period = 1/frequency
Context: Electronic term
See also: Hz

pointer

A variable whose value is the memory location of another variable or function.

Usage: You are not required to deal with pointers in Embedded Control
Context: Programming concept
See also: &, pass by reference

port

A set of connections through which a microprocessor communicates with external circuitry or devices.

Usage: Microprocessor Concept
Context: The C8051 has 8 ports labeled 0 through 7
See also: input, output

potentiometer (pot)

A variable resistor.

Usage: The volume control on your stereo is probably a potentiometer.
Context: Electronic component
See also: resistor, power supply, ohm

power

A loose term for the connection in a circuit which supplies operating current to the circuit.

Usage: Power is usually 5 volts in Embedded Control
Context: Electrical term
See also: high

printf(), printf_fast_f()

This function allows you to produce formatted output to the screen or terminal device. In Embedded Control, this allows you to print information to the screen of the computer (when it is connected to the Evaluation Board).

Usage: The code *printf("hello world\n");* prints the text "hello world" and a new line to the screen
Context: C programming
See also: Evaluation Board, I/O functions

protoboard

A protoboard is used to test a circuit before it is permanently constructed. It allows the designer to construct circuits quickly and easily. The name comes from the term prototype, which designates an implementation of an untested design.

Usage: All of the circuits for your Embedded Control labs will be constructed on protoboards
Context: Embedded Control equipment

pulsewidth modulation (PWM)

When the width of a pulse in a rectangular waveform is varied. This pulsewidth (width of the pulse) can be used to express a value or can represent a duty cycle.

Usage: The steering system uses a PWM signal where the pulsewidth indicates the desired position of the wheels
Context: Electronic term
See also: steering motor, drive motor, servo motor, Timer Output Compare

pushbutton switch

A switch which uses a button to make and break the connection.

Usage: Pushbutton switches are commonly used as start switches for games in Embedded Control

Context: Electrical component

See also: momentary switch, toggle switch, switch

putchar()

A function used to output a single character.

Usage: *putchar('a')* will send an "a" from the EVB to the computer

Context: C library function

See also: printf(), getchar()

rand()

A C library function used to generate a pseudo-random number. rand() generates a number between 0 and 32767

Usage: You should use the srand() function to set the seed of the pseudo-random number generator.

Context: C library function

See also: srand()

random number generation

The process of picking a random number (or almost random number). The random number generator used in Embedded Control is pseudo-random, meaning it does not produce truly random numbers in the strictest sense.

Usage: You should consult a book on microprocessor programming or engineering probability to understand random
             numbers and their generation.

Context: Computer concept

See also: seed, rand(), srand()

range

Refers to the values that a variable or function can take on. Usually expressed by the minimum and maximum values included.

Usage: The range of an unsigned char is the integers from 0 through 255 (0,1,2,3, ... ,253,254,255 or 0-255)

Context: Mathematical concept

See also: unsigned char, int

real time

Refers to events or programs that are time critical, that is, time measured with respect to real world time (i.e. seconds, minutes, hours). Computers programs are not always concerned with time in the outside world. A program such as a word processor is not concerned with the exact time that a particular event takes place.

Usage: You will need to be concerned about some real time programming aspects in Embedded Control, i.e., how long your
             Smart Car code takes to execute.

Context: Microprocessor concept

See also: Real-Time Interrupt

Real-Time Interrupt

An interrupt that occurs at regular time intervals. This can be used to keep track of the passing of time in your program or to measure how long events take to occur.

Usage: You will use a Real-Time Interrupt to measure your Smart Car speed

Context: Microprocessor concept

See also: real time

register

A specific memory location of a microprocessor for a particular purpose. Usually for special functions or configuration.

Usage: The C8051 ports have data registers and output mode registers associated with them.
Context: Microprocessor concept

relational operators

Operators which determine the relationship between numbers, such as equal, greater than, less than, etc.

Usage: The relational operators in C are ==, !=, >, <, >=, and <=.
Context: Programming term
See also: ==, !=, <, >, >=, <=

resistor

An electronic device used to introduce resistance into a circuit.

Usage: A resistor is often used to limit current flow.
Context: Electronic component
See also: resistor color codes

resistor color codes

Resistor values are shown by colored bands on the body of the resistor. There is a chart in the lab, in this manual, and in the LITEC tutorials which describes how these colors indicate resistor values.

Usage: You need to familiarize yourself with the resistor color codes so you do not use incorrect values.
Context: Electronic component
See also: resistor, resistance

return

This is a C keyword. It is used to pass a value back from a function and to exit the function. Execution of the function stops after the return command so any code following it will be ignored. If the function has no return value, it will simply exit the function.

Usage: The code *return 0;* will make 0 the return value of the function and then exit the function
Context: C language keyword
See also: function

sampling rate

The rate at which a microprocessor (or other device) checks the value of a signal. It is expressed in Hz (times it is checked per second).

Usage: A typical sampling rate for the Optical Tachometer is 30 Hz
Context: Microprocessor term
See also: Hz

scanf()

A library function which receives formatted data input by the user.

Usage: You may use scanf() to input values to your program via the computer and the *IDE*. In <stdio.h>
Context: C Library function
See also: getchar()

seed

The starting point of a pseudo-random number generator. If a program uses the same seed each time it is run, it will get the same sequence of "random" numbers each time.

Usage: You can set the seed using the srand() function. Random numbers are obtained using the rand() function
Context: Microprocessor concept
See also: rand(), srand()

semicolon (;)

A semicolon is used in C to end a line of code.

Usage: If you find errors with a line of code, check the lines before to make sure they have properly included semicolons.

Context: C programming language

servo motor

A motor that is controlled by a pulsewidth modulated signal. It is sensitive to the amount of time during a period that the signal is high. A certain rotary position on the servo motor corresponds to a specific pulsewidth.

Usage: A servo motor is used on the *Smart Car* to control the steering.

Context: Electrical component

See also: steering motor, stepper motor

set

Refers to the act of setting a particular register bit to 1. We say we are "clearing" a bit when we set it to 0, and we say we are "setting" a bit when we set it to 1.

Usage: The code P3 = P3 | 0x01; sets the rightmost bit of the data register for port 3

Context: Microprocessor term

See also: clear, &, register

setpoint

The "goal" of a control system. It is the desired value that we wish the system we are controlling to reach.

Usage: In Embedded Control, you will use a setpoint to represent the desired speed of the Smart Car

Context: Control term

See also: deadband

shift

When each bit of a value is moved to the left (or right), we say that we have "shifted" that value to the left (or right). Specified with the << operator for left shift and >> operator for right shift.

Usage: If we shift the binary value 0010110 one bit to the left, we get 0101100

Context: Microprocessor concept

See also: <<, >>

short circuit

This is when wires are accidentally connected together, causing the current flow in the circuit to be much higher than intended. Alternatively, a mis-wiring that causes the load resistance to be much lower than intended.

Usage: You should check for short circuits (shorts) in your circuits before you apply power. You can do this by using the multimeter to measure the resistance between the power and ground connections of your circuit. Generally it should not be less than 10 ohms.

Context: Electrical term

See also: multimeter, resistance

software

Refers to programs that are written for a computer to run. As opposed to hardware which is the physical components which make up the computer.

Usage: We generally refer to the programs that run on a computer (word processor, spreadsheet, text editor) as software.

Context: Computer term

See also: hardware

sqrt()

A math library function, returns the square root of a floating point number. Contained in the math.h header file.

Usage: *sqrt(4.84);* is equal to 2.2
Context: C math library function
See also: math functions, math.h

srand()

The random seed function. This function allows you to set the seed of the random number generator.

Usage: Including the line srand(TL1); after your program has done some I/O (especially after a getchar();) will automatically
           set the seed to a different value (almost) every time.
Context: C library function
See also: seed, rand(), stdlib.h

standard include files

These files contain function prototypes and code which are helpful for many programs. They are used with the #include statement. Include files are typically named with a .h extension.

Usage: stdio.h, stdlib.h and math.h are some common include files
Context: C programming concept
See also: #include, stdio.h, stdlib.h, math.h, c8051.h

static

A data type modifier in the C language. If a variable in a function is of type static, it does not lose its value when the function quits. Static variables are initialized to zero automatically.

Usage: You may wish to use a static variable if you want a function to "remember" something, but you will not need to use
           that variable in the rest of the program
Context: C programming language
See also: data type

stdio.h

An include file for C that contains prototypes for I/O functions such as printf() and scanf()

Usage: The C code *#include <stdio.h>* enables you to use the I/O library functions
Context: C language concept
See also: #include, standard include files

stdlib.h

An include file for C that contains prototypes for common functions such as abs() and rand()

Usage: The C code *#include <stdio.h>* enables you to use common library functions in your program
Context: C language concept
See also: #include, standard include files

steering motor

This is the servo motor used on the Smart car to turn the front wheels. We control the position of those wheels by a pulsewidth output generated from the Evaluation Board.

Usage: You will write programs which move the steering motor in response to sensor inputs from the Optical Tracking Unit
Context: Embedded Control equipment
See also: servo motor

string

A string is a series of characters. In the C language, a string is represented by a char array. The end of the string is indicated by a character whose value is 0. The values in the array are the ASCII values of the characters in the string.

Usage: You won't find much need for strings in your Embedded Control projects
Context: C programming language
See also: string functions, string.h

string functions

These are library functions for the manipulation of strings. Some of these are strlen(), strcat() and strcpy(); which give the length of a string, concatenate two strings, and copy a string respectively

Usage: A good C book should describe the basic string functions
Context: C programming language
See also: string, string.h

string.h

An include file for C that contains prototypes for string manipulation functions such as strcpy() and strcmp().

Usage: The C code *#include <string.h>* enables you to use the string manipulation library functions
Context: C language concept
See also: #include, string, string functions

switch

A switch is an electrical device that allows a user to easily make or break a connection. We say a switch is closed when the terminals are connected and open when they are not.

Usage: You will use several kinds of switches in Embedded Control
Context: Electrical component
See also: pushbutton switch, toggle switch, momentary switch, switch keyword

switch keyword

The word "switch" is a reserved word in the C language. This means you cannot use it as a variable or function name. You should consult a C book if you are interested in how to use the switch keyword in a program.

Usage: The switch keyword is rarely used in Embedded Control programs
Context: C language
See also: switch

syntax

Refers to the specific command format required by a programming language. Punctuation, command names, etc. are all part of the syntax of a language.

Usage: You should have a good C language book to serve as a syntax reference
Context: Programming term

timer

A subsystem of a microprocessor that allows a program to keep track of time. Timers are used to generate time sensitive output signals such as the pulse width outputs used for the Steering and Drive motors

Usage: The C8051 has several timers that you will use
Context: Microprocessor concept
See also: Timer Output Compare

toggle switch

A switch with a lever that flips back and forth to make and break the connection.

Usage: A typical light switch is probably a kind of toggle switch
Context: Electrical component
See also: momentary switch, push button switch, switch

TRUE

In C, any value that is not equal to 0 is considered to be TRUE. 0 is the only value that is considered FALSE.

Usage: The code *if (1) myvariable = 10;* will result in myvariable being set to 10, since 1 is TRUE

Context: C language detail

See also: FALSE, if statement

TTL logic

Transistor-Transistor logic. This is a family of logic chips that employ this type of logic. It also implies a certain general current driving capacities, speed, and input switching characteristics.

Usage: We use mostly TTL chips in the lab

Context: Electronic concept

See also: CMOS

typecast

In the C language, this refers to a conversion between data types.

Usage: An example would be the code *int_var = ((int) float_var);* which takes the variable float_var (assumed to be a floating point number) and converts it to an integer, then assigns it to the variable int_var (assumed to be an integer).

Context: C language concept

See also: data types, int, float, double

underflow

When a variable is changed to a value less than its range, we say an "underflow" occurs. In some cases, the value will "wrap around" and start again from the maximum value. The variable would then be set to the amount by which the value was outside of the variables range.

Usage: For example, let i be an unsigned char (range 0-255) that is currently set to 50. If you were to attempt to subtract 100 from i an underflow would occur. The intended value was 50 - 100 = -50, but the actual value will be 256 - 50 = 206.

Context: Microprocessor term

See also: overflow, data type, unsigned, char

unsigned

This a modifier for data types, such as unsigned int. A variable of this type is always positive or zero. This increases the range of variable since the computer does not have to store the sign of the number.

Usage: We sometimes use unsigned int to get increased integer range. The char data type is usually assumed to be unsigned

Context: C data type

See also: char, int, signed

variable

A location in memory where a value can be stored.

Usage: The code *int i;* declares i to be a variable of type integer

Context: C programming

See also: data types

V$_{CC}$

A designation for the "power" connection in a circuit.

Usage: In Embedded Control, V$_{CC}$ usually refers to 5 volts

Context: Electronic term

See also: power supply

void keyword

This is used in C to indicate that a function does not have a return value or has no parameters.

Usage: a function prototype of *void myfunction(int);* indicates a function which takes an integer as a parameter and has no return value.

Context: C syntax

See also: int, prototype

volt, voltage

The unit of potential difference. For purposes of Embedded Control, it is generally assumed that voltage is measured relative to "ground". That is ground is assumed to be 0 volts.

Usage: 1 volt = 1 amp * 1 ohm

Context: Electrical unit

See also: ohm, amp

voltage reference (Vref)

This signal defines the input signal range of the A/D converter. The A/D converter will only convert analog values that fall between 0 and Vref.

Usage: Vref=2.4 V for the Smart car and the Gondola

Context: EVB inputs

See also: Analog-to-digital conversion

voltage regulator

This is a component used on the Smart Car that regulates power supply voltage. The voltage from the battery is regulated to 5 volts so it can be used to power the EVB and your logic circuitry.

Usage: You may notice that the voltage regulator on your Smart Car gets slightly warm. This is because it must dissipate power in order to bring the 10-13 volts of the battery down to 5 volts.

Context: Electronic component

See also: voltage, power supply

waveform

Another word referring to a changing signal. Usually one that repeats in some regular fashion.

Usage: The terms waveform and signal are often interchangeable

Context: Electronic term

See also: signal

wavelength

The period of a periodic wave multiplied by its speed in space. Comes from the physical distance that cycle traverses in space. Usually measured in meters.

Usage: Sometimes used loosely to describe the period of a wave

Context: Physics concept

See also: period

while statement

A C statement used to create loops and other repeating structures

Usage: An example of a while statement is *while (myvariable < 20) {/* repeated code */};* which repeats the code block as long as myvariable is less than 20

Context: C language

See also: for

wiring diagram

A physical representation of the connections in a circuit. It generally shows the components in some graphical (not symbolic) way, and shows lines indicating physical connections. A wiring diagram includes some connections that are assumed on a schematic, such as power and ground.

Usage: You should understand the difference between a wiring diagram and a schematic

Context: Electronics concept

See also: schematic

Appendix A - Programming Information

C functions

As mentioned many times, the C language does not really support many functions intrinsic to the language, but allows for extensions in the C libraries. The following section outlines the most useful functions that are available with the *SDCC C* compiler. Some of the functions are ANSI C standard functions, and some are specific to the implementation on the C8051. Below is an outline of the structure of this section, along with an example of each section.

Name
> lists the name of the function

Prototype
> shows the prototype of the function which is necessary to know what types are used in the function call and what include file contains the prototype

Description
> describes the use and operation of the function, including options

Portability
> contains the information about the function as applied to other C compilers

Example program
> a short example of how the function is called or used

Related functions
> a list of other functions that are used for the same purpose or that are used with the function being described

abs

Prototype

       #include <stdlib.h>
       int abs(number);
       int number;

Description

       abs returns the absolute value of an integer.

Portability

       Available on most systems.

Example program

```
/* Absolute value test program
   This program demonstrates the use of the abs function.
*/
/* Include files */
#include <stdlib.h>
main()
{
    int i;
    i = -19;
    printf("\n The absolute value of %d is %d \n\n\n", i, abs(i));
}
```

Related functions

       fabs

ceil

Prototype

       #include <math.h>
       double ceil(Number);
       double Number;

Description

       ceil() is called the ceiling function. It returns the smallest integer greater than Number as a floating point number. For example, ceil(9.3) returns 10.0, and ceil(-6.7) returns -6.0.

Portability

       ANSI C compatible

Example program

```
/* Ceiling function test program
   This program demonstrates the use of the ceil function
*/
/* Include files */
#include <math.h>
main()
{
```

```
    double i;
    i = -17.569;
    printf("\n The smallest integer greater than %f is %f \n\n", i, ceil(i));
}
```

Related functions
        floor

floor

Prototype
        #include <math.h>
        double floor(Number);
        double Number;

Description
        floor returns the greatest integer less than Number as a floating point number. For example
floor(9.3) returns 9.0, and floor(-6.7) returns -7.0.

Portability
        ANSI C compatible

Example program
```
/* Floor function test program
    This program demonstrates the use of the floor function
*/
/* Include files */
#include <math.h>
main()
{
    double i;
    i = 14.378;
    printf("\n The smallest integer less than %f is %f \n\n",i, floor(i));
}
```

Related functions
        ceil

getchar

Prototype
>   #include <c8051_SDCC.h>
>   char getchar();
>   int getc(FILE *stream*);

Description
>   getchar reads the next character from the terminal port and returns its value as a char.

Portability
>   getchar is available in ANSI C, but its exact behavior is compiler dependent.

Example program
```
/* getchar test program
    This program demonstrates the use of the getchar function. The function waits for a character to be
sent from the terminal, then transmits a message with the character back.
*/

main()
{
    int c;

    while (1)
    {
        while(!(c = getchar()));
        printf("The character was %c and its ASCII code is %d \n",c,c);
    }
}
```

Related functions
>   gets, putchar

getchar_nw

Prototype
>   #include <c8051_SDCC.h>
>   char getchar_nw();
>   int getc(FILE *stream*);

Description
>   Similar to getch, getchar_nw reads the next character from the terminal port and returns its value as a char, however, if no character is waiting to be read the function returns immediately with a value of 0xFF (255).

Related functions
>   getchar, putchar

gets

(NOT PRESENTLY AVAILABLE FOR THE SDCC COMPILER)

Prototype
>   #include <string.h>
>   char *gets(String);
>   char *String;

Description
>   gets will read a line of input from the terminal port and will place it in the string pointed to by
String.

Portability
>   Available everywhere.

Example program
```
/* gets test program
   This program illustrates the use of the gets function. It will receive a line of input from the
terminal, and return it. It will also print out the first 40 characters in columns to illustrate simple
string handling functions. The printf function can only output a string of up to 80 characters, so we
check for a string that is too long. If you need to output longer strings use the puts function instead.
*/

#include <string.h>

main()
{
    char String[80];
    int i, length;

    while (1)
    {
        printf("\nPlease enter a string\n");
        gets(String);

        length = strlen(String);

        if (length <= 60)
        {
            printf("\nThe String is \"%s\" \n",String);

            for (i=0;( (i<length) && (i<40) );i++)
                printf("\t%c",String[i]);

            printf("\n");
        }
        else
            printf("\nThat string is too long, try again.\n");
    }
}
```

Related functions
>   getchar, puts

kpd_input

Prototype
> #include <i2c.h>
> unsigned int kpd_input(mode);
> char mode;

Description
> The kpd_input function accepts an unsigned int (16-bit) from the keypad for a decimal integer up to 5 digits. Number with less than 5 digits are terminated (entered) by pressing the '#' key. Five-digit numbers are automatically entered when the 5th digit is pressed. As digit keys are pressed their values are displayed on the LCD screen at the beginning of the line of the cursor position. If mode has the value of 0 a message is displayed on the LCD prompting the user with instuctions. If mode is nonzero the prompt is not displayed and only the digits are echoed to the LCD as the keys are pressed. The '*' key is ignored. Any value entered outside the range of 0 – 65535 results in the value obtained from a normal binary arithmetic overflow operation.

Example program
```
/* kpd_input
    This program illustrates how to use the kpd_input function
*/

main()
{
    unsigned int value;
    value = kpd_input(0);

}
```

lcd_clear

Prototype
>   #include <i2c.h>
>   void lcd_clear();

Description
>   The lcd_clear function clears the LCD of any text that is currently being displayed on the screen.

Example program
```
/* lcd_clear
    This program illustrates how to clear the LCD
*/

main()
{
    lcd_clear();
}
```

lcd_print

Prototype
>   #include <i2c.h>
>   void lcd_print(controlString [, arg1] . . .);
>   char *controlString;

Description
>   The lcd_print function has functionality similar to printf, with the exception of special characters
that the LCD cannot print. By calling this function, the string is sent through I2C to the LCD screen
rather than the terminal (as with printf).

Example program
```
/* lcd_print
    This program illustrates how to print to the LCD
*/

main()
{
    char c = 'a';
    lcd_print("The value of c is: %c", c);
}
```

Related functions
>   printf

printf   *NOTE: Use printf_fast_f for foating point values.*

Prototype

      #include <stdio.h>
      int printf(controlString [, arg1] . . .);
      char *controlString;

Description

      The formatted input and output functions are reminiscent of the days of hardcopy terminals, which prompted for input from the user in a specific form, and then output information, one line at a time. This is a very simple method of communication in contrast to today's interfaces that usually have full-screen feedback of keyboard and mouse input. This simple interface can be used with the EVB because the HyperTerminal program emulates a terminal to interface with other computers. Printf is used to send information, such as variables, that must be converted to a character string form before sending each character to the terminal. To accomplish this conversion and positioning on the line, printf uses a control string, which instructs the function what types of input are being used and what format the output should be in. The control string flags are listed here along with several examples of the usage of control strings.

      When the printf function searches through the control string argument, it looks for flags in the string that indicate that a command follows. There are two characters which are used: a backslash (\) to indicate a control command follows, and a percent sign (%) to indicate the type of the next variable to send to the terminal.

The control commands follow:
      \n   start a new line
      \r   send a carriage return without a line feed
      \t   send a tab
      \b   send a back space
      \   control string continues on the following line

Some data types, what they print, and the applicable variable types are given in the table below:

Type	Purpose	Variable Types
%c	Single ASCII Character	unsigned char signed char
%u	Unsigned Decimal Number	unsigned char unsigned int
%d	Signed Decimal Number	signed char signed int
%lu	Unsigned Decimal Number	unsigned long
%ld	Signed Decimal Number	signed long
%x	Lowercase Hexadecimal Number Note: Does not prepend printed value with '0x'	singed char unsigned char
%X	Uppercase Hexadecimal Number Note: Does not prepend printed value with '0x'	signed int unsigned int

Additionally, the format code %[width].[precision]f prints floating point numbers {float,double}
THIS TYPE IS ONLY AVAILABLE WHEN USING *printf_fast_f* when using SDCC Compiler

Portability
        printf is a standard library function available on all ANSI and original C compilers.

Example program

```
#include <stdio.h>

void test_printf (void)
{
    char a;
    int b;
    long c;
    unsigned char x;
    unsigned int y;
    unsigned long z;
    float f,g;

    a = 1;
    b = 12365;
    c = 0x7FFFFFFF;
    x = 'A';
    y = 54321;
    z = 0x4A6FE00;
    f = 10.0;
    g = 22.95;

    printf("Char %d  int %d  long %ld\n", a,b,c);
    printf("Uchar %u  Uint %u  Ulong %lu\n", x,y,z);
    printf("Xchar %x  xint %x  xlong %lx\n", x,y,z);
        // printf("%f != %f\n", f,g);            NOT AVAILABLE WITH SDCC COMPILER
        // printf("%4.2f != %4.2f\n", f,g);   NOT AVAILABLE WITH SDCC COMPILER
}
```

Related functions
        printf_fast_f, scanf { scanf is not presently available in the SDCC compiler }

printf_fast_f

Prototype
>   #include <stdio.h>
>   int printf_fast_f(controlString [, arg1] . . .);
>   char *controlString;

Description
>   The printf_fast_f function has the same identical functionality as printf, with the exception that printf_fast_f will allow the output of floating point data types.

The data types and how to print them are all those included in printf plus:
float, double   %[width].[precision]f

Example program
```
/* printf_fast_f
    This program illustrates how to print a floating point number with SDCC
*/

main()
{
    float g = 22.95;

    printf_fast_f("The value of g is: %4.2f", g);
}
```

Related functions
>   printf

putchar, puts

(**puts** NOT PRESENTLY AVAILABLE FOR THE SDCC COMPILER)

Prototype
        #include <c8051_SDCC.h>
        void putchar(Character);
        int Character;
        void puts(String);
        char *String;

Description
        Putchar sends Character to the terminal port. Character is the ASCII code (integer) for the character to be sent. Puts sends a string to the terminal port.

Portability
        ANSI C compatible.

Example program
```
/* putchar & puts test program
    This program illustrates the use of the putchar and puts functions. The program will use putchar to
output the ASCII characters in the range 33-125 (the printable characters), then print a message with
the puts function.
*/

main()
{
    int i;
    char String[30];

/* use strcpy() to copy Done message into String */
    strcpy(String,"\nDone\n\n\n\n");
    puts("\n");

    for (i=33;i<=126;i++)
        putchar(i);

    puts(String);
}
```

Related functions
        gets, getchar

rand

Prototype
>   #include <stdlib.h>
>   int rand(void);

Description
>   Returns a pseudo-random integer between 0 and 32767. The **srand()** function may be used to seed the pseudo-random number generator before calling **rand()**.

Portability
>   Available on most systems.

Example program
```
/* rand test program
    This program illustrates the generation of 5 random numbers*/

#include <stdlib.h>

main()
{
    int num,i;
    for (i=0;i<5;i++)
    {
        num = rand();
        printf("the random number = %d\n", num);
    }
}
```

Related functions
>   srand

read_keypad

Prototype
>   #include <i2c.h>
>   char read_keypad();

Description
>   Reads the key currently being pressed on the keypad and returns the character value of the key. If no key is pressed, the function returns a value 0xFF (255).

Example program

```
#include <i2c.h>
void main(void)
{
    char a;
    lcd_print("Enter a character:");
    a = read_keypad();
    lcd_clear();
    lcd_print("The value you entered is %c\n", a);
}
```

scanf

(NOT PRESENTLY AVAILABLE FOR THE SDCC COMPILER)

Prototype
>   #include <stdio.h>
>   int scanf(controlString [, pointer1] . . .);
>   char *controlString;

Description
>   Reads formatted data from stdin and writes the results to memory at the addresses given by the variable arguments. Each variable argument must be a pointer to a datum of type that corresponds to the format of the data.

Portability
>   scanf is a standard library function available on all ANSI and original C compilers

Example program

```
#include <stdlib.h>
#include <stdio.h>
void main(void)
{
    int a;
    printf("Enter an integer:");
    scanf("%d", &a);
    printf("The value you entered is %d\n", a);
}
```

Related functions
>   printf

srand

Prototype
> #include <stdlib.h>
> int srand(int val);

Description
> Seed the **rand()** function with **val**.

Portability
> Available on most systems.

Example program
```
/* srand test program
    This program demonstrates how to seed the random number generator*/

#include <stdlib.h>

main()
{
    int num;

    srand(50);
    num = rand();

}
```

Related functions
> rand

c8051f020.h header file

```
/*---------------------------------------------------------------------------
   Register Declarations for the Cygnal/SiLabs C8051F02x Processor Range

   Copyright (C) 2004 - Maarten Brock, sourceforge.brock@dse.nl

   This library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with this library; if not, write to the Free Software
   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
---------------------------------------------------------------------------*/

#ifndef C8051F020_H
#define C8051F020_H


/*  BYTE Registers  */
__sfr __at (0x80) P0        ; /* PORT 0                                    */
__sfr __at (0x81) SP        ; /* STACK POINTER                             */
__sfr __at (0x82) DPL       ; /* DATA POINTER - LOW BYTE                   */
__sfr __at (0x83) DPH       ; /* DATA POINTER - HIGH BYTE                  */
__sfr __at (0x84) P4        ; /* PORT 4                                    */
__sfr __at (0x85) P5        ; /* PORT 5                                    */
__sfr __at (0x86) P6        ; /* PORT 6                                    */
__sfr __at (0x87) PCON      ; /* POWER CONTROL                             */
__sfr __at (0x88) TCON      ; /* TIMER CONTROL                             */
__sfr __at (0x89) TMOD      ; /* TIMER MODE                                */
__sfr __at (0x8A) TL0       ; /* TIMER 0 - LOW BYTE                        */
__sfr __at (0x8B) TL1       ; /* TIMER 1 - LOW BYTE                        */
__sfr __at (0x8C) TH0       ; /* TIMER 0 - HIGH BYTE                       */
__sfr __at (0x8D) TH1       ; /* TIMER 1 - HIGH BYTE                       */
__sfr __at (0x8E) CKCON     ; /* CLOCK CONTROL                             */
__sfr __at (0x8F) PSCTL     ; /* PROGRAM STORE R/W CONTROL                 */
__sfr __at (0x90) P1        ; /* PORT 1                                    */
__sfr __at (0x91) TMR3CN    ; /* TIMER 3 CONTROL                           */
__sfr __at (0x92) TMR3RLL   ; /* TIMER 3 RELOAD REGISTER - LOW BYTE        */
__sfr __at (0x93) TMR3RLH   ; /* TIMER 3 RELOAD REGISTER - HIGH BYTE       */
__sfr __at (0x94) TMR3L     ; /* TIMER 3 - LOW BYTE                        */
__sfr __at (0x95) TMR3H     ; /* TIMER 3 - HIGH BYTE                       */
__sfr __at (0x96) P7        ; /* PORT 7                                    */
__sfr __at (0x98) SCON      ; /* UART0 CONTROL                             */
__sfr __at (0x98) SCON0     ; /* UART0 CONTROL                             */
__sfr __at (0x99) SBUF      ; /* UART0 BUFFER                              */
__sfr __at (0x99) SBUF0     ; /* UART0 BUFFER                              */
__sfr __at (0x9A) SPI0CFG;  /* SERIAL PERIPHERAL INTERFACE 0 CONFIGURATION */
__sfr __at (0x9B) SPI0DAT;      /* SERIAL PERIPHERAL INTERFACE 0 DATA      */
__sfr __at (0x9C) ADC1;         /* ADC 1 DATA                              */
__sfr __at (0x9D) SPI0CKR;/* SERIAL PERIPHERAL INTERFACE 0 CLOCK RATE CONTROL*/
```

```
__sfr __at (0x9E) CPT0CN      ;  /* COMPARATOR 0 CONTROL                    */
__sfr __at (0x9F) CPT1CN      ;  /* COMPARATOR 1 CONTROL                    */
__sfr __at (0xA0) P2          ;  /* PORT 2                                  */
__sfr __at (0xA1) EMI0TC      ;  /* External Memory Timing Control          */
__sfr __at (0xA3) EMI0CF      ;  /* EMIF CONFIGURATION                      */
__sfr __at (0xA4) PRT0CF      ;  /* PORT 0 CONFIGURATION                    */
__sfr __at (0xA4) P0MDOUT     ;  /* PORT 0 OUTPUT MODE CONFIGURATION        */
__sfr __at (0xA5) PRT1CF      ;  /* PORT 1 CONFIGURATION                    */
__sfr __at (0xA5) P1MDOUT     ;  /* PORT 1 OUTPUT MODE CONFIGURATION        */
__sfr __at (0xA6) PRT2CF      ;  /* PORT 2 CONFIGURATION                    */
__sfr __at (0xA6) P2MDOUT     ;  /* PORT 2 OUTPUT MODE CONFIGURATION        */
__sfr __at (0xA7) PRT3CF      ;  /* PORT 3 CONFIGURATION                    */
__sfr __at (0xA7) P3MDOUT     ;  /* PORT 3 OUTPUT MODE CONFIGURATION        */
__sfr __at (0xA8) IE          ;  /* INTERRUPT ENABLE                        */
__sfr __at (0xA9) SADDR0      ;  /* UART0 Slave Address                     */
__sfr __at (0xAA) ADC1CN      ;  /* ADC 1 CONTROL                           */
__sfr __at (0xAB) ADC1CF      ;  /* ADC 1 CONFIGURATION                     */
__sfr __at (0xAC) AMX1SL      ;  /* ADC 1 MUX CHANNEL SELECTION             */
__sfr __at (0xAD) P3IF        ;  /* PORT 3 EXTERNAL INTERRUPT FLAGS         */
__sfr __at (0xAE) SADEN1      ;  /* UART1 Slave Address Enable              */
__sfr __at (0xAF) EMI0CN      ;  /* EXTERNAL MEMORY INTERFACE CONTROL       */
__sfr __at (0xAF) _XPAGE      ;  /* XDATA/PDATA PAGE                        */
__sfr __at (0xB0) P3          ;  /* PORT 3                                  */
__sfr __at (0xB1) OSCXCN      ;  /* EXTERNAL OSCILLATOR CONTROL             */
__sfr __at (0xB2) OSCICN      ;  /* INTERNAL OSCILLATOR CONTROL             */
__sfr __at (0xB5) P74OUT; /* PORT 4 THROUGH 7 OUTPUT MODE CONFIGURATION    */
__sfr __at (0xB6) FLSCL       ;  /* FLASH MEMORY TIMING PRESCALER           */
__sfr __at (0xB7) FLACL       ;  /* FLASH ACESS LIMIT                       */
__sfr __at (0xB8) IP          ;  /* INTERRUPT PRIORITY                      */
__sfr __at (0xB9) SADEN0      ;  /* UART0 Slave Address Enable              */
__sfr __at (0xBA) AMX0CF      ;  /* ADC 0 MUX CONFIGURATION                 */
__sfr __at (0xBB) AMX0SL      ;  /* ADC 0 MUX CHANNEL SELECTION             */
__sfr __at (0xBC) ADC0CF      ;  /* ADC 0 CONFIGURATION                     */
__sfr __at (0xBD) P1MDIN      ;  /* PORT 1 Input Mode                       */
__sfr __at (0xBE) ADC0L       ;  /* ADC 0 DATA - LOW BYTE                   */
__sfr __at (0xBF) ADC0H       ;  /* ADC 0 DATA - HIGH BYTE                  */
__sfr __at (0xC0) SMB0CN      ;  /* SMBUS 0 CONTROL                         */
__sfr __at (0xC1) SMB0STA     ;  /* SMBUS 0 STATUS                          */
__sfr __at (0xC2) SMB0DAT     ;  /* SMBUS 0 DATA                            */
__sfr __at (0xC3) SMB0ADR     ;  /* SMBUS 0 SLAVE ADDRESS                   */
__sfr __at (0xC4) ADC0GTL     ;  /* ADC 0 GREATER-THAN REGISTER - LOW BYTE*/
__sfr __at (0xC5) ADC0GTH     ;  /* ADC 0 GREATER-THAN REGISTER - HIGH BYT*/
__sfr __at (0xC6) ADC0LTL     ;  /* ADC 0 LESS-THAN REGISTER - LOW BYTE    */
__sfr __at (0xC7) ADC0LTH     ;  /* ADC 0 LESS-THAN REGISTER - HIGH BYTE   */
__sfr __at (0xC8) T2CON       ;  /* TIMER 2 CONTROL                         */
__sfr __at (0xC9) T4CON       ;  /* TIMER 4 CONTROL                         */
__sfr __at (0xCA) RCAP2L      ;  /* TIMER 2 CAPTURE REGISTER - LOW BYTE    */
__sfr __at (0xCB) RCAP2H      ;  /* TIMER 2 CAPTURE REGISTER - HIGH BYTE   */
__sfr __at (0xCC) TL2         ;  /* TIMER 2 - LOW BYTE                      */
__sfr __at (0xCD) TH2         ;  /* TIMER 2 - HIGH BYTE                     */
__sfr __at (0xCF) SMB0CR      ;  /* SMBUS 0 CLOCK RATE                      */
__sfr __at (0xD0) PSW         ;  /* PROGRAM STATUS WORD                     */
__sfr __at (0xD1) REF0CN      ;  /* VOLTAGE REFERENCE 0 CONTROL             */
__sfr __at (0xD2) DAC0L       ;  /* DAC 0 REGISTER - LOW BYTE               */
__sfr __at (0xD3) DAC0H       ;  /* DAC 0 REGISTER - HIGH BYTE              */
__sfr __at (0xD4) DAC0CN      ;  /* DAC 0 CONTROL                           */
__sfr __at (0xD5) DAC1L       ;  /* DAC 1 REGISTER - LOW BYTE               */
__sfr __at (0xD6) DAC1H       ;  /* DAC 1 REGISTER - HIGH BYTE              */
```

139

```
__sfr __at (0xD7) DAC1CN       ;  /* DAC 1 CONTROL                        */
__sfr __at (0xD8) PCA0CN       ;  /* PCA 0 COUNTER CONTROL                */
__sfr __at (0xD9) PCA0MD       ;  /* PCA 0 COUNTER MODE                   */
__sfr __at (0xDA) PCA0CPM0     ;  /* CONTROL REGISTER FOR PCA 0 MODULE 0  */
__sfr __at (0xDB) PCA0CPM1     ;  /* CONTROL REGISTER FOR PCA 0 MODULE 1  */
__sfr __at (0xDC) PCA0CPM2     ;  /* CONTROL REGISTER FOR PCA 0 MODULE 2  */
__sfr __at (0xDD) PCA0CPM3     ;  /* CONTROL REGISTER FOR PCA 0 MODULE 3  */
__sfr __at (0xDE) PCA0CPM4     ;  /* CONTROL REGISTER FOR PCA 0 MODULE 4  */
__sfr __at (0xE0) ACC          ;  /* ACCUMULATOR                          */
__sfr __at (0xE1) XBR0     ;  /* DIGITAL CROSSBAR CONFIGURATION REGISTER 0*/
__sfr __at (0xE2) XBR1     ;  /* DIGITAL CROSSBAR CONFIGURATION REGISTER 1*/
__sfr __at (0xE3) XBR2     ;  /* DIGITAL CROSSBAR CONFIGURATION REGISTER 2*/
__sfr __at (0xE4) RCAP4L       ;  /* TIMER 4 CAPTURE REGISTER - LOW BYTE   */
__sfr __at (0xE5) RCAP4H       ;  /* TIMER 4 CAPTURE REGISTER - HIGH BYTE  */
__sfr __at (0xE6) EIE1         ;  /* EXTERNAL INTERRUPT ENABLE 1          */
__sfr __at (0xE7) EIE2         ;  /* EXTERNAL INTERRUPT ENABLE 2          */
__sfr __at (0xE8) ADC0CN       ;  /* ADC 0 CONTROL                        */
__sfr __at (0xE9) PCA0L        ;  /* PCA 0 TIMER - LOW BYTE               */
__sfr __at (0xEA) PCA0CPL0;/* CAPTURE/COMPARE REGISTER FOR PCA 0 MODULE 0 – LOW BYTE*/
__sfr __at (0xEB) PCA0CPL1;/* CAPTURE/COMPARE REGISTER FOR PCA 0 MODULE 1 - LOW BYTE*/
__sfr __at (0xEC) PCA0CPL2;/* CAPTURE/COMPARE REGISTER FOR PCA 0 MODULE 2 - LOW BYTE*/
__sfr __at (0xED) PCA0CPL3;/* CAPTURE/COMPARE REGISTER FOR PCA 0 MODULE 3 - LOW BYTE*/
__sfr __at (0xEE) PCA0CPL4;/* CAPTURE/COMPARE REGISTER FOR PCA 0 MODULE 4 - LOW BYTE*/
__sfr __at (0xEF) RSTSRC       ;  /* RESET SOURCE                         */
__sfr __at (0xF0) B            ;  /* B REGISTER                           */
__sfr __at (0xF1) SCON1        ;  /* UART1 CONTROL                        */
__sfr __at (0xF2) SBUF1        ;  /* UART1 DATA                           */
__sfr __at (0xF3) SADDR1       ;  /* UART1 Slave Address                  */
__sfr __at (0xF4) TL4          ;  /* TIMER 4 DATA - LOW BYTE              */
__sfr __at (0xF5) TH4          ;  /* TIMER 4 DATA - HIGH BYTE             */
__sfr __at (0xF6) EIP1         ;  /* EXTERNAL INTERRUPT PRIORITY REGISTER 1*/
__sfr __at (0xF7) EIP2         ;  /* EXTERNAL INTERRUPT PRIORITY REGISTER 2*/
__sfr __at (0xF8) SPI0CN       ;  /* SERIAL PERIPHERAL INTERFACE 0 CONTROL */
__sfr __at (0xF9) PCA0H        ;  /* PCA 0 TIMER - HIGH BYTE              */
__sfr __at (0xFA) PCA0CPH0;/* CAPTURE/COMPARE REGISTER FOR PCA 0 MODULE 0 – HIGH BYTE*/
__sfr __at (0xFB) PCA0CPH1;/* CAPTURE/COMPARE REGISTER FOR PCA 0 MODULE 1 - HIGH BYTE*/
__sfr __at (0xFC) PCA0CPH2;/* CAPTURE/COMPARE REGISTER FOR PCA 0 MODULE 2 - HIGH BYTE*/
__sfr __at (0xFD) PCA0CPH3;/* CAPTURE/COMPARE REGISTER FOR PCA 0 MODULE 3 - HIGH BYTE*/
__sfr __at (0xFE) PCA0CPH4;/* CAPTURE/COMPARE REGISTER FOR PCA 0 MODULE 4 - HIGH BYTE*/
__sfr __at (0xFF) WDTCN        ;  /* WATCHDOG TIMER CONTROL               */


/*  WORD/DWORD Registers  */

__sfr16 __at (0x8C8A) TMR0     ;  /* TIMER 0 COUNTER                      */
__sfr16 __at (0x8D8B) TMR1     ;  /* TIMER 1 COUNTER                      */
__sfr16 __at (0xCDCC) TMR2     ;  /* TIMER 2 COUNTER                      */
__sfr16 __at (0xCBCA) RCAP2    ;  /* TIMER 2 CAPTURE REGISTER WORD        */
__sfr16 __at (0x9594) TMR3     ;  /* TIMER 3 COUNTER                      */
__sfr16 __at (0x9392) TMR3RL   ;  /* TIMER 3 CAPTURE REGISTER WORD        */
__sfr16 __at (0xF5F4) TMR4     ;  /* TIMER 4 COUNTER                      */
__sfr16 __at (0xE5E4) RCAP4    ;  /* TIMER 4 CAPTURE REGISTER WORD        */
__sfr16 __at (0xBFBE) ADC0     ;  /* ADC 0 DATA WORD                      */
__sfr16 __at (0xC5C4) ADC0GT   ;  /* ADC 0 GREATER-THAN REGISTER WORD     */
__sfr16 __at (0xC7C6) ADC0LT   ;  /* ADC 0 LESS-THAN REGISTER WORD        */
__sfr16 __at (0xD3D2) DAC0     ;  /* DAC 0 REGISTER WORD                  */
__sfr16 __at (0xD6D5) DAC1     ;  /* DAC 1 REGISTER WORD                  */
__sfr16 __at (0xF9E9) PCA0     ;  /* PCA COUNTER                          */
```

```
__sfr16 __at (0xFAEA) PCA0CP0 ;  /* PCA CAPTURE 0 WORD                   */
__sfr16 __at (0xFBEB) PCA0CP1 ;  /* PCA CAPTURE 1 WORD                   */
__sfr16 __at (0xFCEC) PCA0CP2 ;  /* PCA CAPTURE 2 WORD                   */
__sfr16 __at (0xFDED) PCA0CP3 ;  /* PCA CAPTURE 3 WORD                   */
__sfr16 __at (0xFEEE) PCA0CP4 ;  /* PCA CAPTURE 4 WORD                   */


/*  BIT Registers  */

/*  P0  0x80 */
__sbit __at (0x80) P0_0       ;
__sbit __at (0x81) P0_1       ;
__sbit __at (0x82) P0_2       ;
__sbit __at (0x83) P0_3       ;
__sbit __at (0x84) P0_4       ;
__sbit __at (0x85) P0_5       ;
__sbit __at (0x86) P0_6       ;
__sbit __at (0x87) P0_7       ;

/*  TCON  0x88 */
__sbit __at (0x88) IT0        ;  /* EXT. INTERRUPT 0 TYPE            */
__sbit __at (0x89) IE0        ;  /* EXT. INTERRUPT 0 EDGE FLAG       */
__sbit __at (0x8A) IT1        ;  /* EXT. INTERRUPT 1 TYPE            */
__sbit __at (0x8B) IE1        ;  /* EXT. INTERRUPT 1 EDGE FLAG       */
__sbit __at (0x8C) TR0        ;  /* TIMER 0 ON/OFF CONTROL           */
__sbit __at (0x8D) TF0        ;  /* TIMER 0 OVERFLOW FLAG            */
__sbit __at (0x8E) TR1        ;  /* TIMER 1 ON/OFF CONTROL           */
__sbit __at (0x8F) TF1        ;  /* TIMER 1 OVERFLOW FLAG            */

/*  P1  0x90 */
__sbit __at (0x90) P1_0       ;
__sbit __at (0x91) P1_1       ;
__sbit __at (0x92) P1_2       ;
__sbit __at (0x93) P1_3       ;
__sbit __at (0x94) P1_4       ;
__sbit __at (0x95) P1_5       ;
__sbit __at (0x96) P1_6       ;
__sbit __at (0x97) P1_7       ;

/*  SCON  0x98 */
__sbit __at (0x98) RI         ;  /* SCON.0 - RECEIVE INTERRUPT FLAG      */
__sbit __at (0x98) RI0        ;  /* SCON.0 - RECEIVE INTERRUPT FLAG      */
__sbit __at (0x99) TI         ;  /* SCON.1 - TRANSMIT INTERRUPT FLAG     */
__sbit __at (0x99) TI0        ;  /* SCON.1 - TRANSMIT INTERRUPT FLAG     */
__sbit __at (0x9A) RB8        ;  /* SCON.2 - RECEIVE BIT 8               */
__sbit __at (0x9A) RB80       ;  /* SCON.2 - RECEIVE BIT 8               */
__sbit __at (0x9B) TB8        ;  /* SCON.3 - TRANSMIT BIT 8              */
__sbit __at (0x9B) TB80       ;  /* SCON.3 - TRANSMIT BIT 8              */
__sbit __at (0x9C) REN        ;  /* SCON.4 - RECEIVE ENABLE              */
__sbit __at (0x9C) REN0       ;  /* SCON.4 - RECEIVE ENABLE              */
__sbit __at (0x9D) SM2 ;  /* SCON.5 - MULTIPROCESSOR COMMUNICATION ENABLE  */
__sbit __at (0x9D) SM20;  /* SCON.5 - MULTIPROCESSOR COMMUNICATION ENABLE  */
__sbit __at (0x9D) MCE0;  /* SCON.5 - MULTIPROCESSOR COMMUNICATION ENABLE  */
__sbit __at (0x9E) SM1        ;  /* SCON.6 - SERIAL MODE CONTROL BIT 1   */
__sbit __at (0x9E) SM10       ;  /* SCON.6 - SERIAL MODE CONTROL BIT 1   */
__sbit __at (0x9F) SM0        ;  /* SCON.7 - SERIAL MODE CONTROL BIT 0   */
__sbit __at (0x9F) SM00       ;  /* SCON.7 - SERIAL MODE CONTROL BIT 0   */
__sbit __at (0x9F) S0MODE     ;  /* SCON.7 - SERIAL MODE CONTROL BIT 0   */
```

141

```
/*  P2  0xA0 */
__sbit __at (0xA0) P2_0        ;
__sbit __at (0xA1) P2_1        ;
__sbit __at (0xA2) P2_2        ;
__sbit __at (0xA3) P2_3        ;
__sbit __at (0xA4) P2_4        ;
__sbit __at (0xA5) P2_5        ;
__sbit __at (0xA6) P2_6        ;
__sbit __at (0xA7) P2_7        ;


/*  IE  0xA8 */
__sbit __at (0xA8) EX0         ; /* EXTERNAL INTERRUPT 0 ENABLE        */
__sbit __at (0xA9) ET0         ; /* TIMER 0 INTERRUPT ENABLE           */
__sbit __at (0xAA) EX1         ; /* EXTERNAL INTERRUPT 1 ENABLE        */
__sbit __at (0xAB) ET1         ; /* TIMER 1 INTERRUPT ENABLE           */
__sbit __at (0xAC) ES0         ; /* SERIAL PORT 0 INTERRUPT ENABLE     */
__sbit __at (0xAC) ES          ; /* SERIAL PORT 0 INTERRUPT ENABLE     */
__sbit __at (0xAD) ET2         ; /* TIMER 2 INTERRUPT ENABLE           */
__sbit __at (0xAF) EA          ; /* GLOBAL INTERRUPT ENABLE            */


/*  P3  0xB0 */
__sbit __at (0xB0) P3_0        ;
__sbit __at (0xB1) P3_1        ;
__sbit __at (0xB2) P3_2        ;
__sbit __at (0xB3) P3_3        ;
__sbit __at (0xB4) P3_4        ;
__sbit __at (0xB5) P3_5        ;
__sbit __at (0xB6) P3_6        ;
__sbit __at (0xB7) P3_7        ;


/*  IP  0xB8 */
__sbit __at (0xB8) PX0         ; /* EXTERNAL INTERRUPT 0 PRIORITY      */
__sbit __at (0xB9) PT0         ; /* TIMER 0 PRIORITY                   */
__sbit __at (0xBA) PX1         ; /* EXTERNAL INTERRUPT 1 PRIORITY      */
__sbit __at (0xBB) PT1         ; /* TIMER 1 PRIORITY                   */
__sbit __at (0xBC) PS0         ; /* SERIAL PORT PRIORITY               */
__sbit __at (0xBC) PS          ; /* SERIAL PORT PRIORITY               */
__sbit __at (0xBD) PT2         ; /* TIMER 2 PRIORITY                   */


/*  SMB0CN  0xC0 */
__sbit __at (0xC0) SMBTOE      ; /* SMBUS 0 TIMEOUT ENABLE             */
__sbit __at (0xC1) SMBFTE      ; /* SMBUS 0 FREE TIMER ENABLE          */
__sbit __at (0xC2) AA          ; /* SMBUS 0 ASSERT/ACKNOWLEDGE FLAG    */
__sbit __at (0xC3) SI          ; /* SMBUS 0 INTERRUPT PENDING FLAG     */
__sbit __at (0xC4) STO         ; /* SMBUS 0 STOP FLAG                  */
__sbit __at (0xC5) STA         ; /* SMBUS 0 START FLAG                 */
__sbit __at (0xC6) ENSMB       ; /* SMBUS 0 ENABLE                     */
__sbit __at (0xC7) BUSY        ; /* SMBUS 0 BUSY                       */


/*  T2CON  0xC8 */
__sbit __at (0xC8) CPRL2       ; /* CAPTURE OR RELOAD SELECT           */
__sbit __at (0xC9) CT2         ; /* TIMER OR COUNTER SELECT            */
__sbit __at (0xCA) TR2         ; /* TIMER 2 ON/OFF CONTROL             */
__sbit __at (0xCB) EXEN2       ; /* TIMER 2 EXTERNAL ENABLE FLAG       */
__sbit __at (0xCC) TCLK        ; /* TRANSMIT CLOCK FLAG                */
__sbit __at (0xCD) RCLK        ; /* RECEIVE CLOCK FLAG                 */
__sbit __at (0xCE) EXF2        ; /* EXTERNAL FLAG                      */
```

```
__sbit __at (0xCF) TF2          ;   /* TIMER 2 OVERFLOW FLAG                 */


/*   PSW   0xD0 */
__sbit __at (0xD0) P            ;   /* ACCUMULATOR PARITY FLAG               */
__sbit __at (0xD1) F1           ;   /* USER FLAG 1                           */
__sbit __at (0xD2) OV           ;   /* OVERFLOW FLAG                         */
__sbit __at (0xD3) RS0          ;   /* REGISTER BANK SELECT 0                */
__sbit __at (0xD4) RS1          ;   /* REGISTER BANK SELECT 1                */
__sbit __at (0xD5) F0           ;   /* USER FLAG 0                           */
__sbit __at (0xD6) AC           ;   /* AUXILIARY CARRY FLAG                  */
__sbit __at (0xD7) CY           ;   /* CARRY FLAG                            */


/*   PCA0CN   0xD8H */
__sbit __at (0xD8) CCF0         ;   /* PCA 0 MODULE 0 INTERRUPT FLAG         */
__sbit __at (0xD9) CCF1         ;   /* PCA 0 MODULE 1 INTERRUPT FLAG         */
__sbit __at (0xDA) CCF2         ;   /* PCA 0 MODULE 2 INTERRUPT FLAG         */
__sbit __at (0xDB) CCF3         ;   /* PCA 0 MODULE 3 INTERRUPT FLAG         */
__sbit __at (0xDC) CCF4         ;   /* PCA 0 MODULE 4 INTERRUPT FLAG         */
__sbit __at (0xDE) CR           ;   /* PCA 0 COUNTER RUN CONTROL BIT         */
__sbit __at (0xDF) CF           ;   /* PCA 0 COUNTER OVERFLOW FLAG           */


/*   ADC0CN   0xE8H */
__sbit __at (0xE8) ADLJST       ;   /* ADC 0 RIGHT JUSTIFY DATA BIT          */
__sbit __at (0xE8) AD0LJST      ;   /* ADC 0 RIGHT JUSTIFY DATA BIT          */
__sbit __at (0xE9) ADWINT       ;   /* ADC 0 WINDOW COMPARE INTERRUPT FLAG   */
__sbit __at (0xE9) AD0WINT      ;   /* ADC 0 WINDOW COMPARE INTERRUPT FLAG   */
__sbit __at (0xEA) ADSTM0       ;   /* ADC 0 START OF CONVERSION MODE BIT 0  */
__sbit __at (0xEA) AD0CM0       ;   /* ADC 0 START OF CONVERSION MODE BIT 0  */
__sbit __at (0xEB) ADSTM1       ;   /* ADC 0 START OF CONVERSION MODE BIT 1  */
__sbit __at (0xEB) AD0CM1       ;   /* ADC 0 START OF CONVERSION MODE BIT 1  */
__sbit __at (0xEC) ADBUSY       ;   /* ADC 0 BUSY FLAG                       */
__sbit __at (0xEC) AD0BUSY      ;   /* ADC 0 BUSY FLAG                       */
__sbit __at (0xED) ADCINT   ;   /* ADC 0 CONVERISION COMPLETE INTERRUPT FLAG*/
__sbit __at (0xED) AD0INT   ;   /* ADC 0 CONVERISION COMPLETE INTERRUPT FLAG*/
__sbit __at (0xEE) ADCTM        ;   /* ADC 0 TRACK MODE                      */
__sbit __at (0xEE) AD0TM        ;   /* ADC 0 TRACK MODE                      */
__sbit __at (0xEF) ADCEN        ;   /* ADC 0 ENABLE                          */
__sbit __at (0xEF) AD0EN        ;   /* ADC 0 ENABLE                          */


/*   SPI0CN   0xF8H */
__sbit __at (0xF8) SPIEN        ;   /* SPI 0 SPI ENABLE                      */
__sbit __at (0xF9) MSTEN        ;   /* SPI 0 MASTER ENABLE                   */
__sbit __at (0xFA) SLVSEL       ;   /* SPI 0 SLAVE SELECT                    */
__sbit __at (0xFB) TXBSY        ;   /* SPI 0 TX BUSY FLAG                    */
__sbit __at (0xFC) RXOVRN       ;   /* SPI 0 RX OVERRUN FLAG                 */
__sbit __at (0xFD) MODF         ;   /* SPI 0 MODE FAULT FLAG                 */
__sbit __at (0xFE) WCOL         ;   /* SPI 0 WRITE COLLISION FLAG            */
__sbit __at (0xFF) SPIF         ;   /* SPI 0 INTERRUPT FLAG                  */


/* Predefined SFR Bit Masks */

#define PCON_IDLE       0x01    /* PCON                                     */
#define PCON_STOP       0x02    /* PCON                                     */
#define PCON_SMOD0      0x80    /* PCON                                     */
#define TF3             0x80    /* TMR3CN                                   */
#define CPFIF           0x10    /* CPTnCN                                   */
#define CPRIF           0x20    /* CPTnCN                                   */
```

```
#define CPOUT            0x40    /* CPTnCN                               */
#define TR4              0x04    /* T4CON                                */
#define TF4              0x80    /* T4CON                                */
#define ECCF             0x01    /* PCA0CPMn                             */
#define PWM              0x02    /* PCA0CPMn                             */
#define TOG              0x04    /* PCA0CPMn                             */
#define MAT              0x08    /* PCA0CPMn                             */
#define CAPN             0x10    /* PCA0CPMn                             */
#define CAPP             0x20    /* PCA0CPMn                             */
#define ECOM             0x40    /* PCA0CPMn                             */
#define PWM16            0x80    /* PCA0CPMn                             */
#define PORSF            0x02    /* RSTSRC                               */
#define SWRSF            0x10    /* RSTSRC                               */
#define RI1              0x01    /* SCON1                                */
#define TI1              0x02    /* SCON1                                */
#define RB81             0x04    /* SCON1                                */
#define TB81             0x08    /* SCON1                                */
#define REN1             0x10    /* SCON1                                */
#define SM21             0x20    /* SCON1                                */
#define SM11             0x40    /* SCON1                                */
#define SM01             0x80    /* SCON1                                */

#endif
```

c8051_SDCC.h header file

```
//------------------------------------------------------------------------------
// This file is for use in Embedded Control when using the SDCC compiler
//
// Directions:
//
// This file should be saved to the following directory on your laptop:
// C:\Program Files\SDCC\include\mcs51
// Save as c8051_SDCC.h
//
// In your program, you need to include this header file as #include <c8051_SDCC.h>
// and in the main() program, call the function Sys_Init();
//
// Another approach is to put this file in the working directory and call it as
// #include "c8051_SDCC.h" and in the main() program, call the function Sys_Init();
//------------------------------------------------------------------------------


//------------------------------------------------------------------------------
// Includes
//------------------------------------------------------------------------------

#include <c8051f020.h>                     // Special Function Register (SFR) declarations


//------------------------------------------------------------------------------
// Global CONSTANTS
//------------------------------------------------------------------------------

#define SYSCLK      22118400              // SYSCLK frequency in Hz
#define BAUDRATE    38400                // Baud rate of UART in bps


//------------------------------------------------------------------------------
// Initialization Subroutines
//------------------------------------------------------------------------------


//------------------------------------------------------------------------------
// SYSCLK_Init
//------------------------------------------------------------------------------
//
// This routine initializes the system clock to use an 22.1184MHz crystal
// as its clock source.
//
void SYSCLK_Init (void)
{
    int i;                               // delay counter

    OSCXCN = 0x67;                       // start external oscillator with
                                         // 22.1184MHz crystal
    for (i=0; i < 256; i++);             // wait for oscillator to start

    while (!(OSCXCN & 0x80));            // Wait for crystal osc. to settle

    OSCICN = 0x88;                       // select external oscillator as SYSCLK
                                         // source and enable missing clock
                                         // detector
}

//------------------------------------------------------------------------------
```

```
// UART0_Init
//-----------------------------------------------------------------------------
//
// Configure the UART0 using Timer1, for <baudrate> and 8-N-1.
//
void UART0_Init (void)
{
    SCON0  = 0x50;                      // SCON0: mode 1, 8-bit UART,
                                        // enable RX
    TMOD   = 0x20;                      // TMOD: timer 1, mode 2, 8-bit reload
    TH1    = -(SYSCLK/BAUDRATE/16);     // set Timer1 reload value for
                                        // baudrate
    TR1    = 1;                         // start Timer1
    CKCON |= 0x10;                      // Timer1 uses SYSCLK as time base
    PCON  |= 0x80;                      // SMOD00 = 1 (disable baud rate
                                        // divide-by-two)
    TI0    = 1;                         // Indicate TX0 ready
    P0MDOUT |= 0x01;                    // Set TX0 to push/pull
}


//-----------------------------------------------------------------------------
// Sys_Init
//-----------------------------------------------------------------------------
//
// Disable watchdog timer and call other Init functions.
//
void Sys_Init (void)
{
    WDTCN = 0xde;                       // disable watchdog timer
    WDTCN = 0xad;

    SYSCLK_Init ();                     // initialize oscillator
    UART0_Init ();                      // initialize UART0

    XBR0 |= 0x04;
    XBR2 |= 0x40;                       // Enable crossbar and weak pull-ups
}

void putchar(char c)
{
    while(!TI0);
    TI0=0;
    SBUF0 = c;
}

//-----------------------------------------------------------------------------
// getchar()
//-----------------------------------------------------------------------------
char getchar(void)
{
    char c;
    while(!RI0);
    RI0 =0;
    c = SBUF0;
    putchar(c);                        // echo to terminal
    return SBUF0;
}
```

146

i2c.h header file

```
/*  Copy this header file to: C:\Program Files\SDCC\include\mcs51
    This has the functions to use the SMBus and functions for the LCD
    display and number pad.  */


//#include <stdio.h>  //this line should already be in your .c code
#include <stdarg.h>
#include <stdlib.h>   //this may already be in your .c code
#include <string.h>

//-----------------------------------------------------------------------------
// Value Definitions  - sometimes these are useful
//-----------------------------------------------------------------------------

#define TRUE    0x01      //Value representing TRUE
#define FALSE   0x00      //Value representing FALSE
#define ON      0x01      //Value representing ON
#define OFF     0x00      //Value representing OFF
#define HIGH    0x01      //Value representing ON
#define LOW     0x00      //Value representing OFF

#define addr_accel 0x30
#define addr_accelC 0x3A

//-----------------------------------------------------------------------------
// I2C Bus (SMBus) register bit definitions
//-----------------------------------------------------------------------------
/* The following lines are an example of setting up sbit names for the SMBus
   the c8051F020.h file creates the sbit names that are used in this code */

__sbit __at 0xC7 BUS_BUSY; //SM Bus Busy (bit 7)
__sbit __at 0xC6 BUS_EN;   //SM Bus Enable (bit 6)
__sbit __at 0xC5 BUS_START;//SM Bus Start (bit 5)
__sbit __at 0xC4 BUS_STOP; //SM Bus Stop (bit 4)
__sbit __at 0xC3 BUS_INT;  //SM Bus Interrupt (bit 3)
__sbit __at 0xC2 BUS_AA;   //SM Bus ACK (bit 2)
__sbit __at 0xC1 BUS_FTE;  //SM Bus Clock timeout - high (bit 1)
__sbit __at 0xC0 BUS_TOE;  //SM Bus Clock timeout - low (bit 0)
__sbit __at 0x83 BUS_SCL;

//-----------------------------------------------------------------------------
// Global CONSTANTS
//-----------------------------------------------------------------------------

#define DELAY_WRITE 5000   //~5 ms delay write time (about 1000 cycles/ms)
#define DELAY_BLINK 50000  //Value for delay time - blink

//-----------------------------------------------------------------------------
// Global VARIABLES
//-----------------------------------------------------------------------------

unsigned char Data2[3];

//-----------------------------------------------------------------------------
// Function PROTOTYPES
//-----------------------------------------------------------------------------

#define high_byte(x) ((x & 0xFF00) >> 8) //define a high byte as a shift right

extern void delay_time (unsigned long time_end); //Delay function
```

147

```
extern void lcd_print(const char *fmt, ...);//Print to LCD screen
extern void lcd_clear();                     //Clear LCD screen
extern unsigned int kpd_input(char mode);    //Input a multidigit value on keypad

//extern void lcd_cursor(bit show);          //Unused


extern void key_test();     //Test the functionality of the Keypad, unused

unsigned char i2c_read(void);
unsigned char i2c_stop_and_read(void);
void i2c_write(unsigned char output_data);
void i2c_write_and_stop(unsigned char output_data);
void i2c_write_data(unsigned char addr, unsigned char start_reg, unsigned char *buffer,
unsigned char num_bytes);
void i2c_read_data(unsigned char addr, unsigned char start_reg, unsigned char *buffer,
unsigned char num_bytes);
unsigned char i2c_read_and_stop(void);
void i2c_start(void);
void Accel_Init(void);
void Accel_Init_C(void);


void lcd_print(const char *fmt, ...)
{
    unsigned char len, i;                   //assign counter variables
    unsigned static char __xdata text[80];  //character array
    va_list ap;                             //initialize a pointer

    if ( strlen(fmt) <= 0 ) return;         //If there is no data to print, return

    va_start(ap, fmt);
    vsprintf(text, fmt, ap);
    va_end(ap);

    len = strlen(text);
    for(i=0; i<len; i++)
    {
        if(text[i] == (unsigned char)'\n') text[i] = 13;
    }

    i2c_write_data(0xC6, 0x00, text, len);
}


void lcd_clear()
{
    unsigned char NumBytes=0, Cmd[2];

    while(NumBytes < 64) i2c_read_data(0xC6, 0x00, &NumBytes, 1);

    Cmd[0] = 12;
    i2c_write_data(0xC6, 0x00, Cmd, 1);
}


unsigned char read_keypad()
{
    unsigned char i=0, Data[2];     //Initialize variables

    i2c_read_data(0xC6, 0x01, Data, 2); //Read I2C data on address 192, register 1, 2 bytes
of data.
```

148

```
    if(Data[0] == 0xFF) return 0;    //No response on bus, no display

    for(i=0; i<8; i++)                //loop 8 times
    {
        if(Data[0] & (0x01 << i))    //find the ASCII value of the keypad read, if it is the
current loop value
            return i+49;
    }

    if(Data[1] & 0x01) return '9';  //if the value is equal to 9 return 9.

    if(Data[1] & 0x02) return '*';  //if the value is equal to the star.

    if(Data[1] & 0x04) return '0';  //if the value is equal to the 0 key

    if(Data[1] & 0x08) return '#';  //if the value is equal to the pound key

    return 0xFF;                      //else return a numerical -1 (0xFF)
}


////////////////////////////////////////////////////////////////////////////////////////////////
// Function to enter a multidigit positive digit on the keypad, entered when '#' is pressed.
// When passed an argument of 0 it displays instructions on the LCD
// Otherwise it just displays the key characters on the next line as they are pressed.
// The '*' key is ignored and the '#' key is the <Enter> key.
// Values are automatically entered after 5 key presses. Overflows larger than the max
// allowed in an unsigned 16-bit int follow normal 2's-complement bit overflow operations.
////////////////////////////////////////////////////////////////////////////////////////////////

unsigned int kpd_input(char mode)
{
    unsigned sum;
    unsigned char key=0xFF, i;

    sum = 0;
//  lcd_clear();                      //clear the screen
// If mode is 0, output prompt on LCD
    if(mode==0)lcd_print("\nType digits; end w/#");
// Clear 5 spaces on LCD for 5-digit maximum number to be input
    lcd_print("     %c%c%c%c%c",0x08,0x08,0x08,0x08,0x08);

    delay_time(500000);               //Add 20ms delay before reading i2c in loop
                                      //Helps reduce i2c bus lockup crashes
// For each digit, display character on LCD
// Automatically return value after 5 digits or after '#' is entered
    for(i=0; i<5; i++)
    {
        while(((key=read_keypad()) == 0xFF) || (key == '*'))delay_time(10000);
        if(key == '#')
        {
            while(read_keypad() == '#')delay_time(10000);
            return sum;
        }
        else
        {
            lcd_print("%c", key);
            sum = sum*10 + key - '0';
            while(read_keypad() == key)delay_time(10000); //wait for key to be released
        }
    }
    return sum;
}
```

149

```
///////////////////////////////////////////////////////////////////////////////
//  Routine:    delay_time
//  Inputs:     counter value to stop delaying
//  Outputs:    none
//  Purpose:    to pause execution for pre-determined time
///////////////////////////////////////////////////////////////////////////////
void delay_time (unsigned long time_end)
{
    unsigned long index;         //counter variable

    for (index = 0; index < time_end; index++); //for loop delay
}


void i2c_start(void)
{
    while(BUSY);                 //Wait until SMBus0 is free
    STA = 1;                     //Set Start Bit
    while(!SI);                  //Wait until start sent
    STA = 0;                     //Clear start bit
    SI = 0;                      //Clear SI
}


void i2c_write(unsigned char output_data)
{
    SMB0DAT = output_data;       //Data to be written put into register
    while(!SI);                  //Wait until send is complete
    SI = 0;                      //Clear SI
}


void i2c_write_and_stop(unsigned char output_data)
{
    SMB0DAT = output_data;       //Data to be written put into register
    STO = 1;                     //Set stop bit
    while(!SI);                  //Wait until send is complete
    SI = 0;                      //Clear SI
}


unsigned char i2c_read(void)
{
    unsigned char input_data;

    while(!SI);                  //Wait until we have data to read
    input_data = SMB0DAT;        //Read the data
    SI = 0;                      //Clear SI
    return input_data;           //Return the read data
}


unsigned char i2c_read_and_stop(void)
{
    unsigned char input_data;

    while(!SI);                  //Wait until we have data to read
    input_data = SMB0DAT;        //Read the data
    SI = 0;                      //Clear SI
    STO = 1;                     //Set stop bit
    while(!SI);                  //Wait for stop
```

150

```
    SI = 0;
    return input_data;          //Return the read data
}


void i2c_write_data(unsigned char addr, unsigned char start_reg, unsigned char *buffer,
unsigned char num_bytes)
{
    unsigned char i;            //counter variable

    EA = 0;                     //Don't allow interrupts during I2C work
    i2c_start();                //Initiate I2C transfer
    i2c_write(addr & ~0x01);    //Write the desired address to the bus
    i2c_write(start_reg);       //Write the start register to the bus
    for(i=0; i<num_bytes-1; i++) //Write the data to the register(s)
        i2c_write(buffer[i]);
    i2c_write_and_stop(buffer[num_bytes-1]); //Stop transfer
    EA = 1;                     //Re-Enable interrupts
}


void i2c_read_data(unsigned char addr, unsigned char start_reg, unsigned char *buffer,
unsigned char num_bytes)
{
    unsigned char j;

    EA = 0;                     //Don't allow interrupts during I2C work
    i2c_start();                //Start I2C transfer
    i2c_write(addr & ~0x01);    //Write address of device that will be written to, send 0
    i2c_write_and_stop(start_reg); //Write & stop the 1st register to be read
    i2c_start();                //Start I2C transfer
    i2c_write(addr | 0x01);     //Write address again, this time indicating a read operation
    for(j = 0; j < num_bytes - 1; j++)
    {
        AA = 1;                 //Set acknowledge bit
        buffer[j] = i2c_read(); //Read data, save it in buffer
    }
    AA = 0;
    buffer[num_bytes - 1] = i2c_read_and_stop(); //Read the last byte and stop, save it in
the buffer
    EA = 1;                     //Re-Enable interrupts
}                               //End function


//-----------------------------------------------------------------------------
// Accelerometer Initialization
//-----------------------------------------------------------------------------
// NOTE: Writing multiple registers in one call to i2c_write_data()
// doesn't work correctly, multiple calls are required for LSM303DLM!

void Accel_Init(void)
{
//  unsigned char Data2[1];

    Data2[0]=0x23;  //normal power mode, 50Hz ODR, y & x axes enabled
//  Data2[0]=0x3B;  //normal power mode, 1kHz ODR, y & x axes enabled
    Data2[1]=0x00;  //default - no filtering
    Data2[1]=0x10;  //filtered data selected, HPF = 1.0->0.125Hz
    Data2[2]=0x00;  //default - no interrupts enabled
//  Data2[3]=0x80;  //setting Block Data Update bit locks up I2C bus
    i2c_write_data(addr_accel, 0x20, Data2, 1);
//  i2c_write_data(addr_accel, 0xA0, Data2, 3);  // This only works after calling the
previous line
```

151

```
//  i2c_write_data(addr_accel, 0x21, Data2, 1);
//  i2c_write_data(addr_accel, 0x22, Data2, 1);
//  i2c_write_data(addr_accel, 0x23, Data2, 1);


}


void Accel_Init_C(void)
{
//  unsigned char Data2[3];

// Setting Block Data Update bit locks up I2C bus
    Data2[0]=0x04;  //set register address auto increment bit
    i2c_write_data(addr_accelC, 0x23, Data2, 1);

    Data2[0]=0x6B;  //R20 normal power mode, 800Hz ODR, y & x axes enabled
//  Data2[0]=0x3B;  //R20 normal power mode, 100Hz ODR, y & x axes enabled
    Data2[1]=0x00;  //R21 Default - no HP filtering
//  Data2[1]=0x13;  //R21 filtered data selected, HPF = 1.0->0.125Hz
    Data2[2]=0x00;  //R22 Default - no interrupts enabled
    i2c_write_data(addr_accelC, 0x20, Data2, 1);

}
```

152

Appendix B - Helpful Information

Resistor Color Code

Table B.1 - Resistor color code table

Color	Nominal (1st & 2nd band)	Multiplier (3rd band)	Tolerance (4th band)	Reliability (5th band)
Black	0	1	N/A	N/A
Brown	1	10	N/A	1.0%
Red	2	100	N/A	0.1%
Orange	3	1,000	N/A	0.01%
Yellow	4	10,000	N/A	0.001%
Green	5	100,000	N/A	N/A
Blue	6	1,000,000	N/A	N/A
Violet	7	10,000,000	N/A	N/A
Gray	8	100,000,000	N/A	N/A
White	9	1,000,000,000	N/A	N/A
Gold	N/A	0.1	5%	N/A
Silver	N/A	0.01	10%	N/A
No Band	N/A	N/A	20%	N/A

The first and second bands represent digits, while the third band indicates the power of 10 by which to multiply. The fourth band indicates the tolerance of the indicated value. A silver band represents a tolerance of 10%, and a gold band indicates 5%.

Example:

   Red, Black, Orange, Gold
    2   0   3   → 20 x 10^3 = 20kOhm, 5% tolerance

   Yellow, Violet, Black, Silver
    4   7   0   → 47 x 10^0 = 47Ohm, 10% tolerance

Connections on the Smart Car
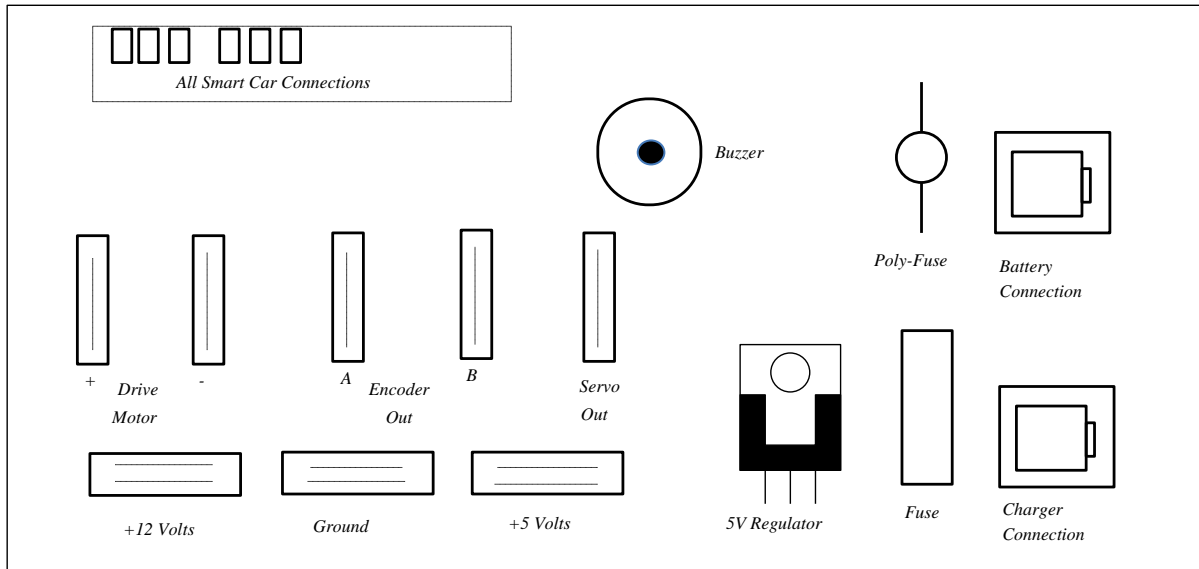
Figure B.1 - Connections on the Smart Car

Figure B.1 shows a diagram of the board for all the power, input, and output connections to the *Smart Car's* components (i.e. servo motor, tachometer, DC drive motor, battery, power supply). It is located on the base of the *Smart Car* chassis. All connections to the smart car, other than the EVB, will be made through this powerboard.

Before you make connections to your protoboard, make sure that the battery connector is securely attached to the power board. Then, check the power and ground connections on the protoboard for shorts using the ohmmeter. If there are no shorts, you can connect the wires to the power blocks. If all is ok, the LED should remain lit once the car has been turned on. If not, you probably have a wiring problem somewhere and already blown the fuse. Correct your wiring first since the fuse will not be replaced until the wiring is corrected. A buzzer has been included as an audible warning for low battery power. The buzzer will begin to chirp once the battery needs to be charged and will increase in frequency as the battery gets lower. To recharge the battery, plug in the charger connector to the power board.

Please make sure that there are secure +5 Volts as well as Ground connections to your wired protoboard. Loose wires may cause damage to the components. Also, be careful when attaching the +12 Volts to the drive motor chip so that this connection is isolated from all other connections on your protoboard.

More Specifications on the C8051F020 EVB
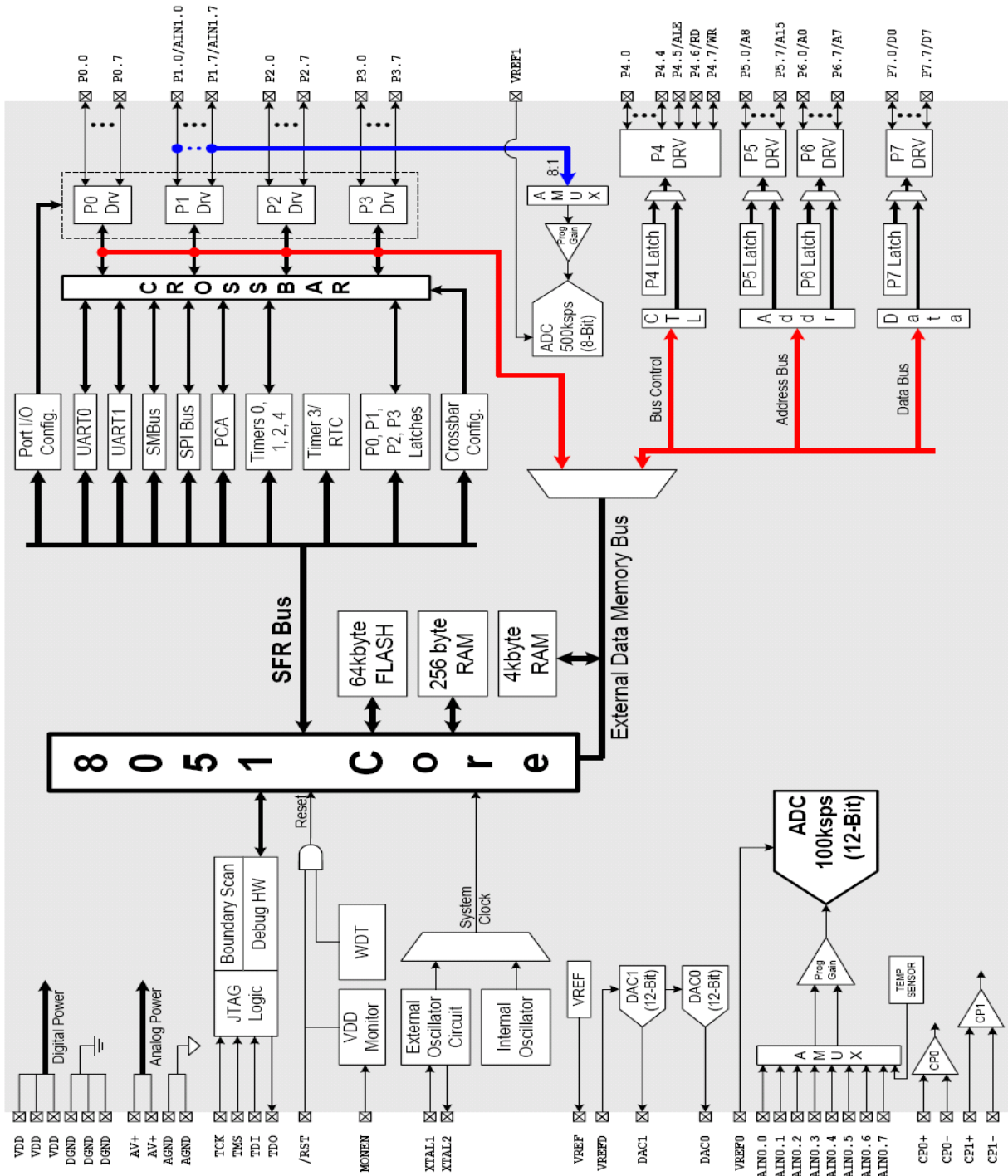
Figure B.2 - Block Diagram of the C8051F020 Evaluation Board

Frequently Asked Questions

1. **How do I create and enter a new project using the SiLabs IDE, compile it, execute it, etc.?**
   The LMS handout "Installing_SiLabs-SDCC-Drivers_Win" will always have the latest instructions for these tasks. Additionally, there are more troubleshooting hints towards the back of that document under "Common Problems".

2. **How do I print my program?**
   Select *Print* from the "File" menu of the SiLabs IDE. Then select "Print" from the Print Document window after the appropriate selections are set. There are two printers in the LITEC Studio, and the print server will spool the document to either printer depending on current usage.

3. **I gave the command for printing out my file long ago, but the printer has not printed it yet.**
   If your computer is still processing the print file, a printer icon will appear in the toolbox in the lower right corner of the screen. Double-click on the printer icon to view the status of your print job. If the printer icon is not present, your print job was probably already sent to the print server and will soon be printed. **Do not issue multiple print commands at any time**. If your print job does not come out, please inform your TA of the problem. Also, inform the TA if the printer runs out of paper. Occasionally jobs get stuck in the printer queue and can take a while to come out.

4. **The EVB on my workbench does not respond.**
   Have you switched the power supply on? Have you reset the EVB before downloading your program? If not, do so and try downloading your code again. If this step fails even after two or three attempts, switch the power supply off, and check the connections of all cables, including the serial cables and the ribbon cables. Also, make sure that none of the fuses on the Smart Car connection board have blown. If this does not work, request assistance from a TA, but **DO NOT** switch EVBs between workbenches.

5. **Why is my car beeping?**
   The cars are designed to "chirp" when the battery supply is low. Make sure you keep your charger plugged in when working at your stations. If you plug the charger in and the car continues to "chirp", inform a TA.

6. **What is causing the car to beep rapidly when I turn it on.**
   The beep indicates a low voltage to the battery voltage sensor. If the beep is slowly beeping, the car just needs to be plugged in to the charger. If the car is beeping rapidly as soon as it is turned on, it is much more serious. The rapid beeping indicates a severe drain on the circuit. This is usually caused by a short (detectable by measuring the resistance between power and ground on the board with everything else detached). A wiring error that often causes this is reversing power and ground on one of the 74 series chips (or putting the chip in backwards). You can tell if this is the problem by touching the chips to see if they have heated up significantly.

7. **How do I reset the EVB?**
   The EVB can be reset by pressing by pressing the reset button on the C8051 board.

8. **Why doesn't my code compile?**
   Write your code in a 'top-down' fashion. You should compile the code after each small addition - NOT when you are ready to test the whole thing.

9. **What is wrong with my code?**
   For debugging a C program, use *printf* statements at various points in your program to determine the flow of control when the program is being executed.

10. **How do I get my code back?**
    It is difficult to retrieve files from a damaged disk. Be sure to save and back-up your files on a regular basis. If you have submitted a softcopy of your code in the past, your TA can retrieve that copy.

11. **Why is my time printing out as 0 all the time?**
    If you have declared seconds as a "long int" then it is a 32-bit number. If you are using "%d" (for printing 16 bit number) in your format string to print it, you will only print the first 16 bits of the number. So, any positive number less than 65536 will show up as 0. Correct this by changing the format string to "%ld" (see *printf* command in *Appendix A*).

12. **Why is my pulsewidth printing out as a negative number?**
    The pulsewidth is a 16 bit UNSIGNED number (0 - 65535). The "%d" format for printf assumes a 16 bit SIGNED number (-32768 to 32767). Above 32767, an unsigned number will appear as negative in signed notation. Correct this by changing the format string to "%u" (see *printf* command in *Appendix A*).

13. **Why does it take a while for changes to variables to show up on the screen?**
    The text that you are printing out takes a certain amount of time to be displayed on the screen. If more text is changing on the screen, it takes longer. If you are printing out information faster than it can be displayed on the screen, the screen will fall behind. To correct this, try using "\r" in the format string instead of "\n". This will cause the cursor to go back and let you re-write over the same line so only 1 line on the screen needs to be updated instead of the entire screen and the screen will update faster.

Appendix C - Course Syllabus & Policies

(Listed for convenience only; the updated current version is on LMS.)

Staff

    Dr. Russell Kraft, Course Instructor and Coordinator,   kraftr2@rpi.edu
    Dr. Jeff Braunstein, Course Instructor,   braunj4@rpi.edu
    Dr. Kyle Wilt, Course Instructor,   wiltk2@rpi.edu
    Dr. Paul Schoch, Course Instructor & Advisor,   schocp@rpi.edu
    Dr. Paul Moon, Course Instructor (as needed),   moonp@rpi.edu
    Dr. Mark Embrechts, Course Instructor (as needed),   embrem@rpi.edu

Course Administration Office Hours

By appointment (send email to the administrative TA: see LMS). These office hours are primarily for grading concerns, course comments and administration problems. Instructor office hours are posted on the course website and in the open shop schedule.

Open Shop Hours

The Open Shop schedule will be posted in JEC 4201 and on the Embedded Control course LMS website. No open shop hours will be held the first week of class.

Location

JEC (JONSSN) 4201 - Core Engineering Studio Laboratory

Course Objectives

On completion of this course, the student will:

    1. Be able to design interface hardware and software to sense, display, command, and control simple engineering processes.

    2. Be able to write software for a real-time embedded control system.

    3. Be able to prototype, implement, and debug the hardware for an embedded control system.

    4. Understand the basic operation of microcontrollers and their role in embedded control systems.

    5. Understand the role of sensors and actuators and how they are applied to systems.

    6. Understand and be able to use the laboratory voltmeters and logic probes required to **build and troubleshoot** these systems.

Course Format

The course follows a combined lecture/laboratory studio format. Not all classes have lectures and some lectures take up more of the class time than others with whatever remaining time devoted to lab exercises. During lectures students are expected to halt all other activities and keep their laptops closed. This closed-computer policy for lectures will be strictly followed.

Texts

– Required: (Note: Any part of the manual may be printed from the Rensselaer LMS site.) Check LMS or the handouts for possible errata sheets with last-minute manual updates.

Embedded Control Lab Manual (available on the course LMS site)

SiLabs C8051F020 Reference Manual (free download on LMS or www.silabs.com)

– Recommended

Programming With C, Schaum's Outlines.

C, The Pocket Reference

– Optional for reference (available as down load from the course LMS site)

SDCC *Compiler Manual*

Grading Criteria

You must submit ALL of the laboratory reports and complete all laboratories to receive a passing grade for the course. The grade weighting is posted on the course RPILMS web site and below. The chart outlines the relative contribution of each component for your overall grade in this class. Components in bold are individual grades. The others are team grades (note that some these assessments have individually graded components.)

Grade Component	**Contribution to Average (out of 100%)**
Exam 1	**23%**
Exam 2	**23%**
5 Quizzes	**16%**
12 Homework Assignments	**11%**[1]
6 Laboratories (pre-lab, check-off, questions)	12%
Lab Preparation/Participation/Performance	**6%** (ranges from -3% to +6%)
Reports (Compass/Ranger, Accelerometer, Gondola)	9%[2] (2%, 3%, 4%)

[1] The lowest score of assignments will be dropped.

[2] No reports will be accepted if the labs required for the reports have not been completed fully and checked off. Lab related grades are scaled to equalize between TA grading variations within a section. Sections are assigned grades independently from each other.

Late Policy

– Homework assignments are due by 10:00am on the due date as indicated by the course calendar for your section and are considered late if not submitted by then. No credit is given for late submissions.

– The LITEC Calendar on the web page indicates the laboratory due dates (for preparation & check-off). The penalty for late submission for these components is also listed on the calendar.

– All Lab Reports are due by the date shown on the schedule for your particular section. All late lab reports will be discounted 20% per school day for lateness.

Homework

All homework assignments are contained and submitted on LMS. The due date for each homework is listed on the class calendar. The due time for each homework is by10:00am on that date. **Programming assignments containing compiler and/or linker errors will not be accepted** (i.e. given a grade of 0). Each individual is responsible for ensuring this and submitting their file with their name, section number and side in the program header comments. Note that the file required to be submitted must be the C source file: extension of *.c, NOT *.wsp, *.asm, etc. Open shop and class time prior to the due date and time may be used to obtain help with the assignments and to ensure it operates correctly. All homework assignments are considered to be individual assignments, whether C coding or assessments however Homework 10 may be completed as a team. Homework assignments 8 and 12 are teammate evaluations. The lowest graded homework score is dropped.

Quizzes

On-line LMS Quizzes will be given over the course of the semester as noted on the course calendar. Students are allowed to use **printed** references (lecture notes, worksheets, & lab manual) and calculators, but no sharing of material is allowed. Also no other computer programs may be running on the laptop other than the LMS browser window, and no copies of old quizzes are permitted. Pertinent information will be given by the instructor prior to the quiz. All quizzes start at the beginning of class. Solutions will be available after all sections have completed the quiz and make-ups have been administered. Students may view their quizzes by asking their instructor or TA. Late arrivals should not expect to be given extra time to complete quizzes. The quizzes end typically 30 minutes after the start of class so that all students may start lab work or lectures may begin on time.

Lab Preparation/Participation/Performance:

Students will be assigned a grade for this component based on several factors: attendance, observation of professionalism, class commitment, and engagement from the instructor and TAs, and evaluation from teammates. Do not assume that high lab scores indicate good lab performance. While all partners will receive the same grades for lab reports, lab performance is based on individual contribution to materials and coursework and ability to debug/troubleshoot. You will evaluate your teammates twice during the semester. Note that fully 75% of your grade (comprised of items in bold) is dependent on your individual performance. For this reason, teams with successful projects may find that individual members receive very different final grades.

Attendance Policy

Attendance is required for all scheduled sections unless you are told otherwise. If for any officially approved reason you cannot make it to lab, you should inform your TA and lab partner(s) in advance, and make arrangements to work in the lab at another time. You will have to provide documentation of this fact to your TA (e.g., a note from your doctor, dean of students, etc.).

Lab Equipment

− ID or other identification will not be required to sign-out equipment (logic probe, voltmeter) for use during the lab session. This may change at any time if students are irresponsible.

− Please be considerate of others by taking good care of the lab equipment provided for you. Make sure all tools are returned and the voltmeter is **turned off**. Carelessness that results in equipment damage can affect the grade received in this course.

− This policy requiring an ID and TA verification to sign out and return tools may be instituted during the semester if deemed necessary.

General Lab Guidelines and Lab Check off Policies

The following guidelines are minimum recommended guidelines for LITEC. Students should consult their TA or Professor for more specific requirements.

General

− Only pick **your own** protoboard and equipment from the cabinet.

− If there is something wrong with your protoboard or you need more parts, **DO NOT TAKE SOMEONE ELSE'S PROTOBOARD OR PARTS**.

− In case your protoboard is damaged or you need more parts, ask the TA to give you a new protoboard or parts.

− Start returning your board, parts, toolkits etc. **AT LEAST 5 MINUTES** before the end of class. Make sure you remove shared items from your protoboard and return them to the center table. Limited supply items (speakers, accelerometers, LCD/Keypads, RF link) should NEVER be left on protoboards because they are needed by all classes and there are not enough for everyone. Protoboards or part bags found containing such parts will be confiscated and stripped down.

− Start cleaning up your desktop and wires 5 minutes before the end of your class and do not wait until the class time has ended. We need to get out of the way for the next class.

Pre-Lab

− Carefully read the lab description available on LMS.

− Make sure that you understand the requirements of the lab and what is the eventual outcome.

− If you find any discrepencies in lab description or feel that the lab description is missing some information, contact your professor or TA and ask them for clarification. **DO NOT** assume anything because you may end up working on different aspects of the lab that were never required.

− Pseudocode is due when a new lab begins for a section by the course calendar. It must be turned in on time for the first 2 points of the lab check-off. Use the pseudocode when you start writing your C-code.

Lab Check off

− Read the lab check-off grade sheet given to you at the start of the class (Front and Back) for all the information regarding lab grading, lab check-off etc.

− Your preparation must be checked off and scores written in the grade sheet before you ask the TA to check off your lab. Prelab worksheets must be submitted to Gradescope by the due dates specified on the course calendar.

− Make sure your lab is working according to the lab description provided on LMS before you ask the TA to check you off.

− If you ask the TA to check you off but your lab is not working according to the lab description or you have not completed all the requirements, the **TA WILL DEDUCT POINTS** and you will be asked to redo the lab to make your lab compatible with the lab description.

− In order to obtain good scores on lab check-off your circuit must be wired very neatly, the software must be well commented and you must answer all the questions asked by the TA.

− You should only answer the question asked to you.

− You are allowed to look at anything including the lab manual, lectures, worksheets, etc. to answer a question but you should not start browsing through the material and start reading it all over. The only purpose of allowing you to refer to the material is that you can get to the page and instructions related to the question and refresh your memory. It does not mean that you should start understanding that material after the question was asked because you are required to develop that understanding before lab check-off and not after the lab check-off.

− You **SHOULD NOT TRY TO ANSWER ANY QUESTION DIRECTED AT YOUR PARTNER(S)**.

− If the TA has asked your partner some question, you should not give your partner any hint or show him the part of code. If you do so, the **TA CAN TAKE OFF POINTS FOR THAT QUESTION**.

− In order to be checked off, you **must** ask the TA to check you off **AT LEAST 15 MINUTES** before the end of the class or open shop. The check-off procedure can take up to 15 minutes and in order to be checked off before the end of class or open shop, **THE DEADLINE FOR CHECK-OFF REQUEST IS 15 MINUTES** before end of class or open shop.

− If you ask the TA to check you off and there is not enough time (15 minutes) to complete the check-off before the end of class or open shop, the TA can deny you the check-off request and you will have to be checked off in the next class or next available open shop hours. This also applies for labs with check-off and report deadlines.

Reports

− You are required to submit 3 brief summary reports in the class:

    Car Compass/Ranger report (Lab 4), **Car Accelerometer report** (Lab 5) and **Gondola report** (Lab 6).

− Make sure that you read the grading rubric and grading scheme for these reports on LMS. You must include all the items mentioned in the grading rubric. If you miss some item you

will not be given a chance to resubmit it and you will get a **ZERO FOR THE MISSING ITEM**.
  – Make sure to consult your TA regarding the report format and what to write in the report.
  – If you do not submit your report on time, you will **LOSE 20 POINTS** for every school day late.

Computers

Personal laptop computers will be used to complete the laboratory exercises in this course. Quizzes and parts of homework are also done on your personal computers. You are expected to **bring your computer to every class**.

  – You will start the semester working in groups of 3, and stay that way for the entire semester. Make sure that everybody has working course software on his or her laptop. Make sure that everybody has the latest version of the C-code available to them. If a partner is missing, you are still required to do the session assignments, so you need to have your computer, the code and the course software.

Always make backups of your programs & any other files, and save your work frequently. Make sure that you and your partners each have all the program files. Lost code will have to be regenerated.

Questions / Problems

Teaching assistants are in the lab to guide you in the right direction and to clarify the assignments if necessary. They are NOT a substitute or alternative to completion of the reading assignments, multimedia tutorials, and attendance of the sections. Nor are they to do your lab work for you. If you have a question, first look in the lab and course resource information, and if you still cannot find the answer, then ask the TA. Be prepared to describe the problem and what diagnostics you have already tried. Saying "It doesn't work." is of little value. If you have a problem that your TA cannot resolve or have a disagreement with the TA, then you should bring it to the attention of the course instructors. Remember, there are lists of common problems and solutions in Appendix D of the Lab Manual and on the LMS main page in the **Installing_SiLabs-SDCC-Drivers_Win** handout. Becoming familiar with these fixes to the most frequently occurring problems can save a lot of time compared to waiting for TA help!

Lab Partners

You are responsible for finding lab partners in your section. This is a very important choice for you to make in the course. It should be made on the basis of complementary backgrounds, talents and commitment. Failure of a partner to do their share can be harmful to all partners. Irresolvable problems with your partners should be brought to the attention of your grading TA and instructor.

Academic Dishonesty

Academic dishonesty is a very serious matter, and we suggest that you read the remainder of this statement carefully:

Student-teacher relationships are built upon trust. For example, students must trust that teachers have made appropriate decisions about the structure and content of the courses they teach, and teachers must trust that the assignments that students turn in are their own. Acts that violate this trust undermine the educational process.

The *Rensselaer Handbook* defines various forms of Academic Dishonesty and procedures for responding to them. All forms are violations of the trust between students and teachers. Students should familiarize themselves with this portion of the *Rensselaer Handbook* and should note that the penalties for plagiarism and other forms of cheating can be quite harsh.

Any portion of work handed in that is not your own, should cite the author. Just as you would not write a history paper by copying text from the encyclopedia, you should not take credit for another person's engineering work. See *A few words on plagiarism:* on page 164 for more information.

Collaboration on assignments is both allowed and encouraged between lab partners. However, having one partner always work on hardware aspects and another on the software will be detrimental to all partners. Each partner should understand and participate in all aspects of the lab exercises in order to learn the necessary topics that will be required for lab check-off and covered on the exams. Collaboration on assignments is not allowed *between* lab groups, either within or between lab sections. Turning in similar out-of-class assignments (homeworks or lab reports), which suggest that copying (in part or in total) has taken place, will be considered as academic dishonesty.

The material presented in the course and the equipment and components used in the labs may change each semester. If you receive help from another student who previously took this course, make sure that the information is current and applicable to the work that you are required to perform. DO NOT use or copy assignments from previous semesters. Using out-of-date materials will be considered as academic dishonesty, whether you copied it or were told by a previous student. Please consider any help you receive from outside sources critically and check the information against that in this manual. Also, if you are taking this course again, make sure any work you use from the previous semester is updated to reflect changes in the course. It may be mistaken as academic dishonesty.

Cheating on an exam or quiz will be considered as academic dishonesty and will generally result in a failing grade for the course. In particular, all mobile devices (cell/smart phones, computers, pagers, etc.) must be stored securely away during exams and are not to be used unless specifically directed otherwise by the instructor. Use of (or **<u>ANY</u>** interaction with) a mobile device during an exam without explicit permission of the instructor will be interpreted as the illicit transfer of exam data, will be considered an act of cheating and will be treated as such.

At all times, we reserve the right to take formal action against anyone engaging in academic dishonesty. This action may range from failing an assignment to failing the course, or to being reported to the Dean of Students. If you have any questions about these rules or how they apply to any specific assignment or exam, discuss it with one of the instructors or course administrators.

Monitoring Homework and Lab Reports

Due to our extensive database of previous assignments and advances in software programs for document and code comparison, the number of identified cases of plagiarism has grown tremendously. We reserve the right to utilize all available methods that verify that documents are the original work of the submitter(s) and if necessary change grades retroactively when discrepancies are discovered.

As a Final Note:

This policy and the statements included should not be interpreted as an assumption on our part that every student will commit acts of academic dishonesty. We strive to treat every student fairly and make LITEC an enjoyable and valuable class. As such, every effort must be made to uphold the integrity of Rensselaer degrees by discouraging acts of academic dishonesty.

A few words on plagiarism:

Plagiarism is defined as: "1: a piece of writing that has been copied from someone else and is presented as being your own work 2: the act of plagiarizing; taking someone's words or ideas as if they were your own" (*WordNet ® 1.6, © 1997 Princeton University*).

Plagiarism is a serious breach of honesty in any venue; in an academic setting it will not only have a significant negative impact on your grade, but it will prevent you from learning writing (both code and report) skills that will be needed in future academic and professional work. In a professional setting, it can be grounds for dismissal from a job or from professional societies, as well as casting doubt on the integrity of your work as a whole.

Some examples of plagiarism include:

- Copying sentences, parts of sentences, paragraphs, or longer pieces of text from the lab manual, the online tutorials, books, articles, or other people's writing in any form, and using them in your writing assignments without attribution. Changing some of the words, or rearranging the sentences or portions of sentences still constitutes plagiarism.

- Copying schematics, graphics, illustrations, charts, graphs, tables, or other illustrative material from the lab manual, the online tutorials, books, articles, or other people's writing in any form, and using them in your writing assignments without attribution.

Avoiding plagiarism is not difficult, and will benefit you in the long run. The following guidelines may help:

1. Do not share your group's produced code or accept code from someone else.
2. In the course of your lab work, try to make notes on what you are doing, what you have tried, problems encountered.
3. When you go to write your lab reports or other technical writing, use these notes as a starting point for describing **in your own words** the goals, concepts, techniques, results, and other components of the development process.
4. If there are certain things that you feel are best described by text from the lab manual or other source, make sure you reference that material. A word-for-word copy should be placed in quotation marks, and a footnote or endnote reference placed at the end of the quoted portion. A paraphrase or idea-by-idea copy should have a footnote or endnote reference placed at the end of that portion of text.

Plagiarism, even if inadvertent, will have serious consequences on your future career - now is the time to learn the skill of writing about your own work, and referencing others' writing appropriately.

Appendix D - Lab Assignments

Six Laboratories will be assigned during the semester.

1. <u>Lab 1 Digital input/output, Timer overflows</u> - An introduction on how to configure the Port pins for input/output operation, initializing and reading from special function registers, using timer registers to measure periods of time, application of the A/D capabilities of the microprocessor

2. <u>Lab 2 Interactive Game</u> - Further sophistication in the software and hardware developed in the previous Laboratory to design a game that reacts to user inputs

3. <u>Lab 3 PCA, Pulse width modulation, System Bus</u> - Using the Programmable Counter Array to generate a pulse width modulated signal, implementing the System Bus to communicate with an external device, control the speed or direction of the car

4. <u>Lab 4 Steering the car</u> - Merge codes to communicate with multiple devices, control both the speed and direction of the car, simple linear control

5. <u>Lab 5 Control Systems</u> - Develop more sophisticated responses to new accelerometer sensors to detect the tilt of a slope and change the heading of the car to drive up the slope at the steepest angle and stop at the top when the slope levels off

6. <u>Lab 6 Gondola Control</u> - Modifying software for the car to drive the gondola and develop more sophisticated responses to correct heading and position of the gondola on a turntable, exploring the effects of different feedback control responses

See LMS for details on the Laboratory assignments for specific details regarding each laboratory.

Index

V

variable, 121
variable resistor, 115
VCC, 121
void keyword, 122
volt, 122
voltage, 122
voltage reference high, 122
voltage regulator, 97, 122
VRL, 122

W

waveform, 122
wavelength, 122
`while loop`, 18
while statement, 122
Windows 2000, 123
wiring diagram, 123
wiring methods, 12, 69

ASCII Table

HEX	DEC	Char	HEX	DEC	Char	HEX	DEC	Char	HEX	DEC	Char
0x00	000	NUL	0x20	032	Space	0x40	064	@	0x60	096	`
0x01	001	SOH	0x21	033	!	0x41	065	A	0x61	097	a
0x02	002	STX	0x22	034	"	0x42	066	B	0x62	098	b
0x03	003	ETX	0x23	035	#	0x43	067	C	0x63	099	c
0x04	004	EOT	0x24	036	$	0x44	068	D	0x64	100	d
0x05	005	ENQ	0x25	037	%	0x45	069	E	0x65	101	e
0x06	006	ACK	0x26	038	&	0x46	070	F	0x66	102	f
0x07	007	BEL	0x27	039	'	0x47	071	G	0x67	103	g
0x08	008	BS	0x28	040	(	0x48	072	H	0x68	104	h
0x09	009	TAB	0x29	041	)	0x49	073	I	0x69	105	i
0x0A	010	LF	0x2A	042	*	0x4A	074	J	0x6A	106	j
0x0B	011	VT	0x2B	043	+	0x4B	075	K	0x6B	107	k
0x0C	012	FF	0x2C	044	,	0x4C	076	L	0x6C	108	l
0x0D	013	CR	0x2D	045	-	0x4D	077	M	0x6D	109	m
0x0E	014	SO	0x2E	046	.	0x4E	078	N	0x6E	110	n
0x0F	015	SI	0x2F	047	/	0x4F	079	O	0x6F	111	o
0x10	016	DLE	0x30	048	0	0x50	080	P	0x70	112	p
0x11	017	DC1	0x31	049	1	0x51	081	Q	0x71	113	q
0x12	018	DC2	0x32	050	2	0x52	082	R	0x72	114	r
0x13	019	DC3	0x33	051	3	0x53	083	S	0x73	115	s
0x14	020	DC4	0x34	052	4	0x54	084	T	0x74	116	t
0x15	021	NAK	0x35	053	5	0x55	085	U	0x75	117	u
0x16	022	SYN	0x36	054	6	0x56	086	V	0x76	118	v
0x17	023	ETB	0x37	055	7	0x57	087	W	0x77	119	w
0x18	024	CAN	0x38	056	8	0x58	088	X	0x78	120	x
0x19	025	EM	0x39	057	9	0x59	089	Y	0x79	121	y
0x1A	026	SUB	0x3A	058	:	0x5A	090	Z	0x7A	122	z
0x1B	027	ESC	0x3B	059	;	0x5B	091	[	0x7B	123	{
0x1C	028	FS	0x3C	060	<	0x5C	092	\	0x7C	124		
0x1D	029	GS	0x3D	061	=	0x5D	093	]	0x7D	125	}
0x1E	030	RS	0x3E	062	>	0x5E	094	^	0x7E	126	~
0x1F	031	US	0x3F	063	?	0x5F	095	_	0x7F	127	DEL

Crossbar on the C8051

Priority Crossbar Decode Table

EVB Pin: 20 21 18 19 16 17 14 15 12 13 10 11 6 9 4 5 29 30 27 28 24 25 22 23

Signal	P0 / P1 / P2 / P3 (PIN I/O 0–7 each)	Crossbar Register Bits
TX0 / RX0		UART0EN: XBR0.2
SCK / MISO / MOSI / NSS		SPI0EN: XBR0.1
SDA / SCL		SMB0EN: XBR0.0
TX1 / RX1		UART1EN: XBR2.2
CEX0 / CEX1 / CEX2 / CEX3 / CEX4		PCA0ME: XBR0.[5:3]
ECI		ECI0E: XBR0.6
CP0		CP0E: XBR0.7
CP1		CP1E: XBR1.0
T0		T0E: XBR1.1
/INT0		INT0E: XBR1.2
T1		T1E: XBR1.3
/INT1		INT1E: XBR1.4
T2		T2E: XBR1.5
T2EX		T2EXE: XBR1.6
T4		T4E: XBR2.3
T4EX		T4EXE: XBR2.4
/SYSCLK		SYSCKE: XBR1.7
CNVSTR		CNVSTE: XBR2.0

	7	6	5	4	3	2	1	0
XBR0 Register	CP0E Comparator 0 Output Enable	ECI0E PCA0 External Counter Input Enable	PCA0ME PCA0 Module I/O Enable			UART0EN UART0 I/O Enable	SPI0EN SPI0 Bus I/O Enable	SMB0EN SMBus I/O Enable
Priority Order	7	6	5			1	2	3

5	4	3	
0	0	0	No CEXn
0	0	1	CEX0
0	1	0	CEX0, CEX1
0	1	1	CEX0, CEX1, CEX2
1	0	0	CEX0, CEX1, CEX2, CEX3
1	0	1	CEX0, CEX1, CEX2, CEX3, CEX4

XBR0: Port I/O Crossbar Register 0 and Priority Order

C8051 EVB Port Connector

DGND	1	● ●	● ●	2		
	3	● ●	● ●	4	P1.6	
P1.7	5	● ●	● ●	6	P1.4	
+3.3V	7	● ●	● ●	8		PORT 1
P1.5	9	● ●	● ●	10	P1.2	
P1.3	11	● ●	● ●	12	P1.0	
P1.1	13	● ●	● ●	14	P0.6	
P0.7	15	● ●	● ●	16	P0.4	
P0.5	17	● ●	● ●	18	P0.2	PORT 0
P0.3	19	● ●	● ●	20	P0.0 *	
* P0.1	21	● ●	● ●	22	P2.6	
P2.7	23	● ●	● ●	24	P2.4	
P2.5	25	● ●	● ●	26		PORT 2
P2.2	27	● ●	● ●	28	P2.3	
P2.0	29	● ●	● ●	30	P2.1	
P3.6	31	● ●	● ●	32	P3.7	
P3.4	33	● ●	● ●	34	P3.5	PORT 3
P3.2	35	● ●	● ●	36	P3.3	
P3.1	37	● ●	● ●	38	P3.0	
/RST	39	● ●	● ●	40	DAC0	
DAC1	41	● ●	● ●	42	CP1-	
CP1+	43	● ●	● ●	44	CP0-	
CP0+	45	● ●	● ●	46	VREF	
AIN0	47	● ●	● ●	48	AIN1	
AIN2	49	● ●	● ●	50	AIN3	
AIN4	51	● ●	● ●	52	AIN5	
AIN6	53	● ●	● ●	54	AIN7	
AGND	55	● ●	● ●	56		
	57	● ●	● ●	58		
	59	● ●	● ●	60		

PORT 1 (left, pins 5–13)
PORT 0 (left, pins 15–21)
PORT 2 (left, pins 23–29)
PORT 3 (left, pins 31–37)