Unit 1 Embedded System

Introduction to Embedded System

An embedded system is an electronic system that has a software and is embedded in computer hardware. It is programmable or non-programmable depending on the application. An embedded system is defined as a way of working, organizing, performing single or multiple tasks according to a set of rules.

In an embedded system, all the units assemble and work together according to the program. Examples of embedded systems include numerous products such as microwave ovens, washing machine, printers, automobiles, cameras, etc. These systems use microprocessors, microcontrollers as well as processors like DSPs.

The important characteristics of an embedded system are speed, size, power, reliability, accuracy, adaptability. Therefore, when the embedded system performs the operations at high speed, then it can be used for real -time applications. The Size of the system and power consumption should be very low, then the system can be easily adaptable for different situations.

An Embedded system is a combination of computer hardware and software. As with any electronic system, this system requires a hardware platform and that is built with a microprocessor or microcontroller. The Embedded system hardware includes elements like user interface, Input/output interfaces, display and memory, etc. Generally, an embedded system comprises power supply, processor, memory, timers, serial communication ports and system application specific circuits.

2. Characteristics of an embedded system

The important characteristics of an embedded system are

Speed (bytes/sec) : should be high speed

Power (watts) : low power dissipation

Size and weight : as far as possible small in size and low weight

Accuracy (%error) : must be very accurate

Adaptability : high adaptability and

accessibility Reliability : must be reliable over

a long period of time

So an embedded system must perform the operations at a high speed so that it can be readily used for real time applications and its power consumption must be very low and the size of the system should be as far as possible small and the readings must be accurate with minimum error. The system must be easily adaptable for different situations.

3. Categories Of Embedded Systems:

Embedded systems can be classified into the following four categories based on their functional and performance requirements

Stand – alone embedded system

Real – time embedded system – hard real – time system and soft real – time system

Networked embedded system and

Mobile embedded system

Based on the performance of the microcontroller they are also classified into

Small scale embedded system

Medium scaled embedded system and

Large scaled embedded system

Stand – alone embedded system

A stand – alone embedded system works by itself. It is a self contained device which does not require any host system like a computer. It takes either digital or analog inputs from its input ports, calibrates, converts and process the data and outputs the resulting data to its attached output device, which either displays data or controls or drives the attached devices. Temperature measurement system, video game consoles, MP3 players, digital cameras and microwave ovens are the examples for this category

Real – time embedded system

An embedded system which gives the required output in a specified time or which strictly follows the deadlines for completion of a task is known as a real time embedded system

Soft real time embedded system

A real time system in which the violation of time constraints will cause only the degraded quality but the system can continue to operate is known as a soft real time system. In soft real time system the design focus is to offer a guaranteed bandwidth to each real – time task and to distribute the resources to the task. Examples a microwave oven, washing machine, tv remote etc.,

Hard – real time embedded System

A real time system in which the violation of time constraints will cause critical failure and loss of life or property damage or catastrophe is known as hard – real time system. These systems usually interact directly with physical hardware instead of through a human being. The hardware and software of hard – real time systems must allow a worst case execution (WCET) analysis is that guarantees the execution be completed within a strict dead-line. The chip selection and RTOS selection became important factors for hard- real time system design. Examples: deadline is a missile control embedded system., delayed alarm during a gas leakage, car air bag control system, a delay response in pace – makers, failure in RADAR functioning, etc.,

Networked embedded systems

The networked embedded systems are related to a network with network interfaces to access the resources. The connected network can be a local area network (LAN) or a wide area network (WAN) or the internet. The connection can be either wired or wireless.

The network embedded system is a fast growing area in an embedded system application. The embedded web server is such a system where all embedded device are connected to a web server and can be accessed and controlled by any web browser. Examples; a home security system is an example of a LAN networked embedded system where all sensors (e.g., monitor detectors, light sensors, or smoke sensors) are wired and running on the TCP/IP protocol.

Mobile embedded system

The portable embedded devices like mobile and cellular phones, digital cameras, MP3 players, PDA (Personal Digital Assistants) are the examples for mobile embedded systems. The basic limitations of these devices is the limitation of memory and other resources

Small scale embedded system: An embedded system supported by a single 8 – 16 bit microcontroller with on – chip RAM and ROM designed to perform simple task in a small scale embedded system

Medium scale embedded system: An embedded system supported by 16-32 bit microcontroller / microprocessor with external RAM and ROM that can perform more complex operations is a Medium scale embedded system

Large scale embedded system: An embedded system supported by 32 – 64 multiple chips which can perform distributed jobs is considered as a large scale embedded system

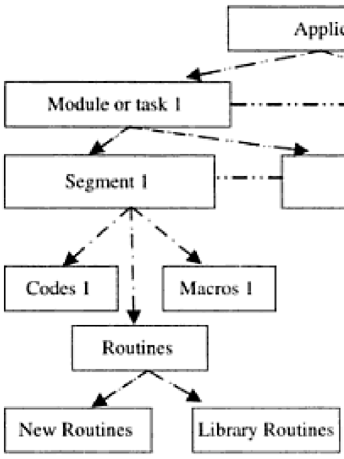
Applications of Embedded system

Integrated Development Environment (IDE) consists of software development tools for application-development. An IDE includes the project manager, editor, code-optimizing compiler, RTOS, macro assembler, library manager, linker/locator, object to hex converter, hex-file generator, simulator and debugger. The chapter describes development tools through the example of 8051 IDE from Keil Software (an ARM company).

We will also learn the hardware development tools— emulators, in-circuit emulators (ICE), target monitor-based target debugger and device-programmer.

Development phases of a microcontroller based system

Figure shows the contents of an application program. A module or task of an application is assumed to consist of several program segments. Each *segment* consists of the codes, macros and routines.



e Application Contents—Modules or Tasks, Segments, Codes, Macros, Routines (Functions) and Library Routines

A *macro* is a named entity. The entity corresponds to a set of codes within in a C function. Macro permits common sequence of codes to be developed only once and permits use of it as a software building block. A macro can be repeatedly used in different functions. The sequences of codes in the macros are given in the pre-processor statements. Within the functions, macro is used and the macros names are later replaced by the corresponding statements in preprocessor statements at the time of compilation.

A *routine* is called *function* in C and called *method* in Java. A routine can be a programmer's routine or a *library* routine.

A library-routine is a function called from a standard library provided during the compilation or linking so that the routine can be used directly. Just as a book is borrowed from a library for reading; a file of routines called *library* file is used. Segments and routines borrow the readily available routines for their use in the program. For example, standard library exists for mathematical functions.

Development for an application (product) consists of the following phases:

Phase 1: Analysis The programmer has to understand and analyse the requirements and specifications of the application to be developed. A listing of the requirements and required system specifications enable an appropriate design.

Phase 2: Design The design is done by assuming an application (program) to be consisting of modules or tasks. The module can be used in multiple applications or projects. A module is a set of functions. The set is independent of the results of the next module. A task is a set of instructions. The set performs some action or a set of actions in a system. The running of the task is controlled by systems software—An Operating System (OS) or Real-time Operating System (RTOS).

Firstly, the selection of appropriate software-development tools for the design is done. Selection of appropriate modules or tasks, program segments, macros, routines and library routines and their linkages is then done.

Selection of hardware for the design is based on requirements. Hardware consists of the microcontroller, needed external interfaces and expansion circuits. Selection of an appropriate target platform is done for development. An emulator circuit (Section 12.5) for a microcontroller helps in the test phase. A target board with a monitor (Section 12.6) helps in the tests at final stage when mcuruns at full speed.

Phase 3: Implementation (Coding as per Design) Each module or task is implemented (coded) using the segments, statements, macros and routines. Appropriate software development tools (Section 12.3) are employed for coding and using the macros and library routines.

Phase 4: Testing and Debugging A targeted system prototype is used during the development phase. A cycle starts by coding for application codes. The cycle is called *write–edit–embed and test* cycle. Design and implementation phases needs several cycles. The developed codes are embedded. The embedding is done in flash memory at the device. The codes are then tested. Each cycle is repeated over and over again till the testing of all segments and modules gives the results as per the requirements and specifications.

A system is integrated to give the targeted application. An *emulator* or target board provides a platform for hardware–software integrated testing. The debugging of developed codes is done. A cycle can be *write–edit–emulate and test* or can be *write–edit– debug by simulation and test*. After the design and implementation phases, the bugs and faults are traced. (Debugging means finding an error-causing or a source bug in the code that initially could not be taken into account. The bugs are discovered after the testing and simulating the actions of codes. Testing and simulating need appropriate tools.)

SOFTWARE DEVELOPMENT CYCLE AND APPLICATIONS

The process of software development consists of requirement and specification analysis, design, implementation and testing.

Software- Development Cycles of Implementation and T esting

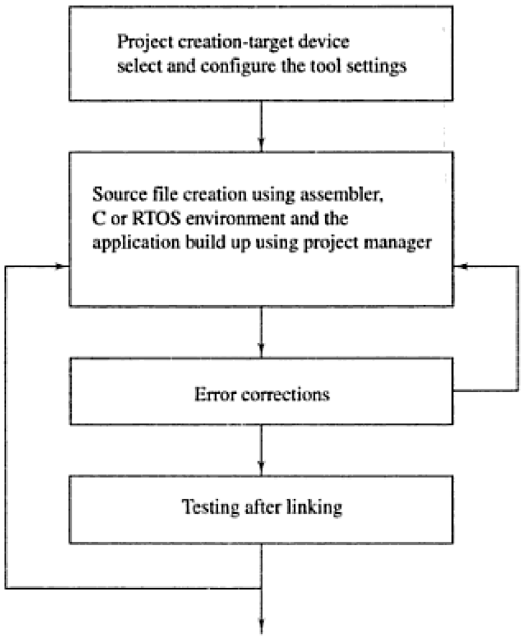
The software-development cycle consists of a source code development in C or assembly or using RTOS and creating an absolute object file for an application from the object and library files. An absolute file is a file after linking by a linker/locator. The device programmer directly uses a hex-file (a standard format file) from the absolute file. A debugger uses the absolute file and device simulator. Debugging can also be by a monitor in a target. It can also be done using an evaluation or demonstration board. Cycles of write–edit–embed and test give reliable and bug-free software.

Steps in Software- Development Process

Figure 12.2 shows the steps in a software project development process—Steps A to E.

Step A: A creation of a project starts with the selection of device (target microcontroller). The selection is done from *device database*. Then the process of device configuring is takes place. Then tools are set. Then they are used in the project.

Step B: A file called a *source file* is created for the project with the help of an editor.


Project creation-target device select and configure the tool settings
Source file creation using assembler, C or RTOS environment and the application build up using project manager
Error corrections
Testing after linking

Steps in a Software Project Development Process. (b) Development Cycle

The development is done in one of the following:

1. assembly and using an assembler or macro assembler, or
2. C and using a compiler, or
3. C and using an RTOS environment integrated with C compiler. This is done for a multi-tasking system,
4. Visual Basic using an environment integrated with Visual Basic compiler.
5. Visual C++ or .NET framework.

Step C: Other developed source files which have been previously developed can also be included in the project. Project manager helps in building the application from the source files.
Step D: Source-file errors are corrected.
Step E: The *source and library files* are linked and tested using (i) a suitable emulator or (ii) a suitable debugger/ simulator/ target debugger using a monitor.

Figure 12.3 shows a set of software tools during development cycle. These are used in each cycle of project development cycle and creating an *application.*

Software-development using IDE:

1. An IDE for C progamming provides an integrated development environment that involves managing, organizing, editing and an integrated *make* facility.

4. Using the linked codes in the hex-file[b] generated for each address in the device,[c] the debugger tests and debugs by simulating the target monitor (PROM) source codes either by using an emulator or a debugger in the IDE. Evaluation or demonstration board can also be used for testing and debugging.

SOFTWARE DEVELOPMENT TOOLS

Integrated Development Environment (IDE)

Software development tools are compilers, assemblers, library managers, linker/locators and debuggers and simulators. IDE provides an environment that integrates these. The IDE has a project manager along with the tools to develop an application.

An IDE gives a single focal point for the development because it enables the following:

1. Creation of source files through a project manager and rich featured source code editor and *make* facility.
2. Organization (through the manager) of the source files into a project that defines an application.
3. Provision of database for many devices (microcontrollers and peripherals) and therefore, provision of appropriate configuration of the development tool settings for the target device(s).
4. Automatically, (through the integrated *make* facility) (a) compiles, points errors and interactively corrects the errors, (b) assembles and links to application and (c) links to device data sheets, user guides and development-tool manuals.

Source File Make facility

An IDE eases the development efforts. IDE gives the *make* facility. There is an on-line help at each stage for help and interactive dialogs for all development tool settings. Dialog helps in interacting with the developer. The interaction is as per the device selected as target. IDE provides a window screen. It enables use of the mouse by dragging over text selected, double clicking a word for an action, selecting by bringing a pointer to a line beginning or selecting the multiple lines and hold a vertical block of text.

Menu Commands, Toolbars and Shortcuts

The screen shows the menu commands, toolbars and shortcuts as follows:

File menu and commands, edit menu and commands, view menu and commands, project menu and commands, peripherals menu, tools menu, software version control menu, window menu and help menu.

An IDE helps in creating, testing and debugging in application.

Using an IDE

(1) **Project manager**: A project manager helps in creating and organizing a project file. A project file enables developing application. An IDE helps in editing of the project source code. It helps in using a macro assembler, C and cross-compiler. A cross-compiler means program compiled for different processor or system for another microcontroller or processor of target system or vice versa.

The project menu in a development environment consists of the following: Create, Import, Open and Close.

It creates a new project, imports a previously created file, opens and closes existing project file, selects a device for the target, provides a target environment by including and removing files, including library files defining the path for tools chain, maintains target, groups and files, changes tool options for the target, selects file extensions, translates modified files and builds a target application, retranslates to build application and translates current file and stops current build process.

(2) **File types in a project**: During development there are different types of files. The extensions reflect a type. The extensions are as follows:

1. .C means ANSI-C ASCII text file, .C51 means C ASCII-text file in C51 compiler
2. .ASM means an assembler ASCII text file, .A51 means A51 macro assembler ASCII text file, .SRC means file from SRC directive to C51 compiler so that it creates an assembled ASCII text file.
3. .LST means listing object file by A51 or C51 to give documentation of the translation process. A listing has ASCII program text as well as the diagnostic information of the source module developed by A51 or C51.
4. .OBJ means relocatable (to different memory-start addresses for linking or other purposes) binary output called object file from A51 or C51. .ABS means absolute (non-relocatable addresses) binary executable output file. It is a complete program to run an 8051 application.
5. .HEX means hex-file using a converter of object to hex file. The hex-file is directly given as input to a device programmer. The device programmer consists of hardware to hold the device. The device programmer has software. The software is used to burn (program) the codes in flesh or PROM or microcontroller as per the hex-file. The device programmer can program many types of devices and microcontrollers. .H86 is also a file to act as input to device programmer.
6. .MAP and .M51 mean map file (listing from Linker) for list of memory usage information and other statistical information.

(3) **Editor:** An editor in a development environment helps in coding as text file. An editor prepares a text for the assembler or C compiler. The edit menu consists of the following:

An editor uses *ctrl-F* to find a word or text, and *replace* (ctrl-H) to replace specific text. An editor helps in *undoing* (ctrl-Z) the last operation, *redoing* last (ctrl-shift-Z), *cut* (ctrl-X selected text, ctrl-Y text of current line) and take to clip board, *copy* (ctrl-C) from clip board, *paste* (ctrl-V) from clip board, *indent* insert in the text at current cursor, *indent right, home to* move cursor at the line beginning, *end* to move the cursor at the line ending, *ctrl-home* to move the cursor at the file beginning, *ctrl-end* to move the cursor at the file ending, *ctrl-left arrow* to move the cursor at the beginning of one word on the left, *ctrl-right arrow* to move the cursor at the beginning of one word on the right, *ctrl-A* to select all of one line text. An editor provides for bookmarks, *go* to previous or next bookmark.

Open Source IDE and Tools

All open source tools through free download have the limitations. A commercial IDE provides full features and integration. For example, commercial IDE provides the file system, USB and TCP for networked embedded systems. File system is a set of functions used to manage (create, describe, read, write, secure and delete).

An open-source IDE for the 8051-based ATMEL microcontroller AT89S8253 and other microcontrollers is from source http://mcu8051ide.sf.net. It supports assembly and C. It has the simulator, assembler, editor and several other tools.

The open source tools are also available from a number of sources. One such source for the compiler is http://sdcc.Sourceforge.net/Open Source Development Network. Small Device C Compiler (SDCC) is a freeware/open source compiler for the 8051 and other 8-bit microcontroller platforms.

Open source RTOS is available from Sourceforge.net/Open Source Development Network. FreeRTOS™ is a mini Real Time Kernel. It is portable, open source and royalty free. FreeRTOS™ is the cross platform de facto standard for embedded microcontrollers.

RTOS

An RTOS is used when a system is considered as multiple tasking. RTOS is used for synchronizing and scheduling the tasks. The tasks have real-time constraints. RTOS is used to control their running. Chapter 11 describes the uses of RTOS in system development.

Compiler

C language has portability among different device numbers of a microcontroller. Features of C are as follows:

1. Statements for the data-type declarations;
2. macros;
3. functions;
4. loops;
5. decision blocks;
6. Easy inclusion of the modules in other source files and C library functions;
7. Data type checking by compiler when processing and gives in the output the errors when creating the object code file. It greatly reduces the chances of error;
8. Program development without knowledge of processor instructions. Automatically allocates and manages the memory addresses and registers for different data types and variables by the compiler.

A program can be adapted easily for another processor. A programming language is easier than assembly. Structures divide the program into separate functions. The structures can be used in this and other applications. They imbib features of functions reusability. They help in modular program development. C run-time library has standard functions ready, for example, for numeric conversion and formatted outputs.

A compiler creates an object code file. The compiler can be 8051 specific as well as chosen 8051 device specific. A compiler will thus have the features to support 8051, like the C51 compiler from Keil.

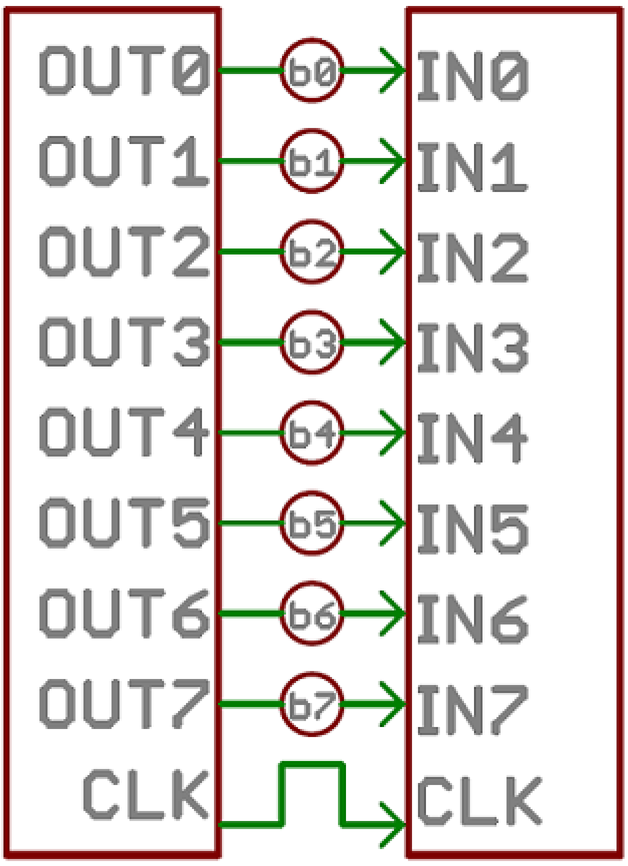
UNIT-II EMBEDDED SYSTEM INTERFACING

Serial Communication

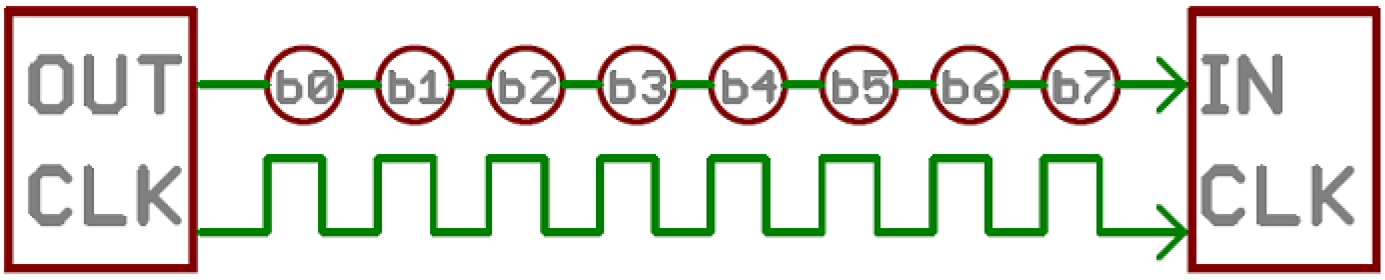
Embedded electronics is all about interlinking circuits (processors or other integrated circuits) to create a symbiotic system. In order for those individual circuits to swap their information, they must share a common communication protocol. Hundreds of communication protocols have been defined to achieve this data exchange, and, in general, each can be separated into one of two categories: parallel or serial.

Parallel vs. Serial

Parallel interfaces transfer multiple bits at the same time. They usually require **buses** of data - transmitting across eight, sixteen, or more wires. Data is transferred in huge, crashing waves of 1's and 0's.



An 8-bit data bus, controlled by a clock, transmitting a byte every clock pulse. 9 wires are used.Serial interfaces stream their data, one single bit at a time. These interfaces can operate on as little as one wire, usually never more than four.



Example of a serial interface, transmitting one bit every clock pulse. Just 2 wires required!

Think of the two interfaces as a stream of cars: a parallel interface would be the 8+ lane mega-highway, while a serial interface is more like a two-lane rural country road. Over a set amount of time, the mega-highway potentially gets more people to their destinations, but that rural two-laner serves its purpose and costs a fraction of the funds to build.

Parallel communication certainly has its benefits. It's fast, straightforward, and relatively easy to implement. But it requires many more input/output (I/O) lines. If you've ever had to move a project from a basic Arduino Uno to a Mega, you know that the I/O lines on a microprocessor can be precious and few. So, we often opt for serial communication, sacrificing potential speed for pin real estate.

Asynchronous Serial

Over the years, dozens of serial protocols have been crafted to meet particular needs of embedded systems. USB (universal serial bus), and Ethernet, are a couple of the more well-known computing serial interfaces. Other very common serial interfaces include SPI, I^{2}C, and the serial standard we're here to talk about today. Each of these serial interfaces can be sorted into one of two groups: synchronous or asynchronous.

A synchronous serial interface always pairs its data line(s) with a clock signal, so all devices on a synchronous serial bus share a common clock. This makes for a more straightforward, often faster serial transfer, but it also requires at least one extra wire between communicating devices. Examples of synchronous interfaces include SPI, and I^{2}C.

Asynchronous means that data is transferred **without support from an external clock signal**. This transmission method is perfect for minimizing the required wires and I/O pins, but it does mean we need to put some extra effort into reliably transferring and receiving data. The serial protocol we'll be discussing in this tutorial is the most common form of asynchronous transfers. It is so common, in fact, that when most folks say "serial" they're talking about this protocol (something you'll probably notice throughout this tutorial).

- Synchronization bits,
- Parity bits,
- and Baud rate.

Synchronization bits

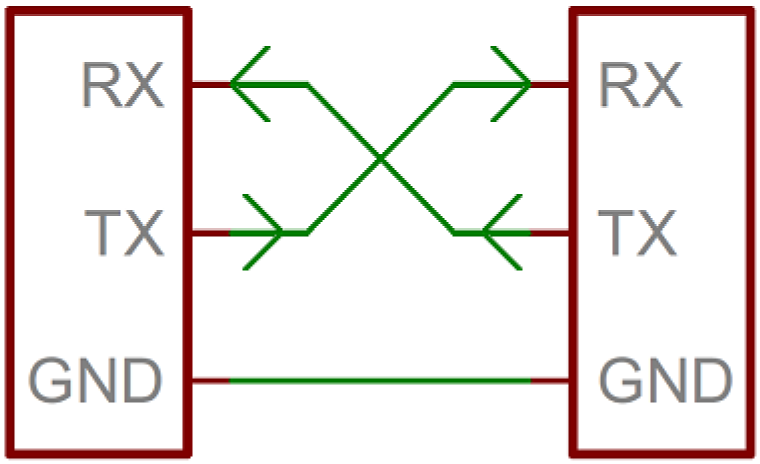
The synchronization bits are two or three special bits transferred with each chunk of data. They are the **start bit** and the **stop bit(s)**. True to their name, these bits mark the beginning and end of a packet. There's always only one start bit, but the number of stop bits is configurable to either one or two (though it's commonly left at one).

The start bit is always indicated by an idle data line going from 1 to 0, while the stop bit(s) will transition back to the idle state by holding the line at 1.

Parity bits

Wiring and Hardware

A serial bus consists of just two wires - one for sending data and another for receiving. As such, serial devices should have two serial pins: the receiver, **RX**, and the transmitter, **TX**.



It's important to note that those RX and TX labels are with respect to the device itself. So the RX from one device should go to the TX of the other, and vice-versa. It's weird if you're used to hooking up VCC to VCC, GND to GND, MOSI to MOSI, etc., but it makes sense if you think about it. The transmitter should be talking to the receiver, not to another transmitter.

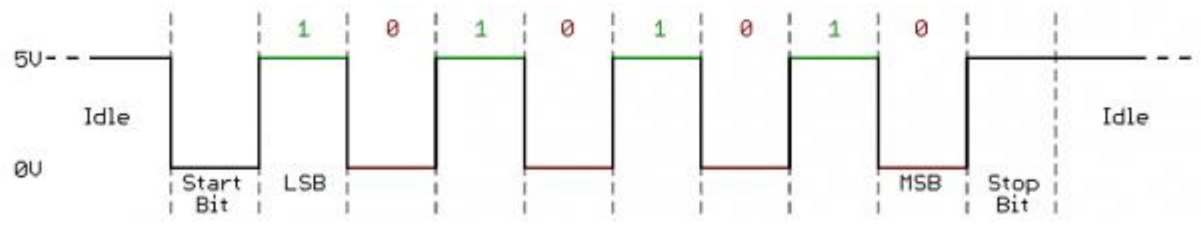
A serial interface where both devices may send and receive data is either **full-duplex** or **half-duplex**. Full-duplex means both devices can send and receive simultaneously. Half-duplex communication means serial devices must take turns sending and receiving.

Some serial busses might get away with just a single connection between a sending and receiving device. For example, our Serial Enabled LCDs are all ears and don't really have any data to relay back to the controlling device. This is what's known as **simplex** serial communication. All you need is a single wire from the master device's TX to the listener's RX line.

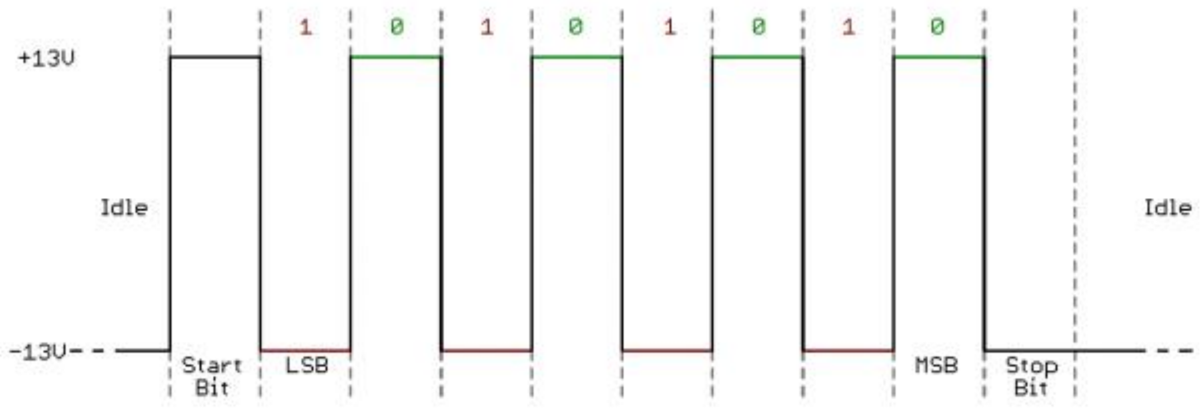
Hardware Implementation

We've covered asynchronous serial from a conceptual side. We know which wires we need. But how is serial communication actually implemented at a signal level? In a variety ways, actually. There are all sorts of standards for serial signaling. Let's look at a couple of the more popular hardware implementations of serial: logic-level (TTL) and RS-232.

When microcontrollers and other low-level ICs communicate serially they usually do so at a TTL (transistor-transistor logic) level. **TTL serial** signals exist between a microcontroller's voltage supply range - usually 0V to 3.3V or 5V. A signal at the VCC level (3.3V, 5V, etc.) indicates either an idle line, a bit of value 1, or a stop bit. A 0V (GND) signal represents either a start bit or a data bit of value 0.

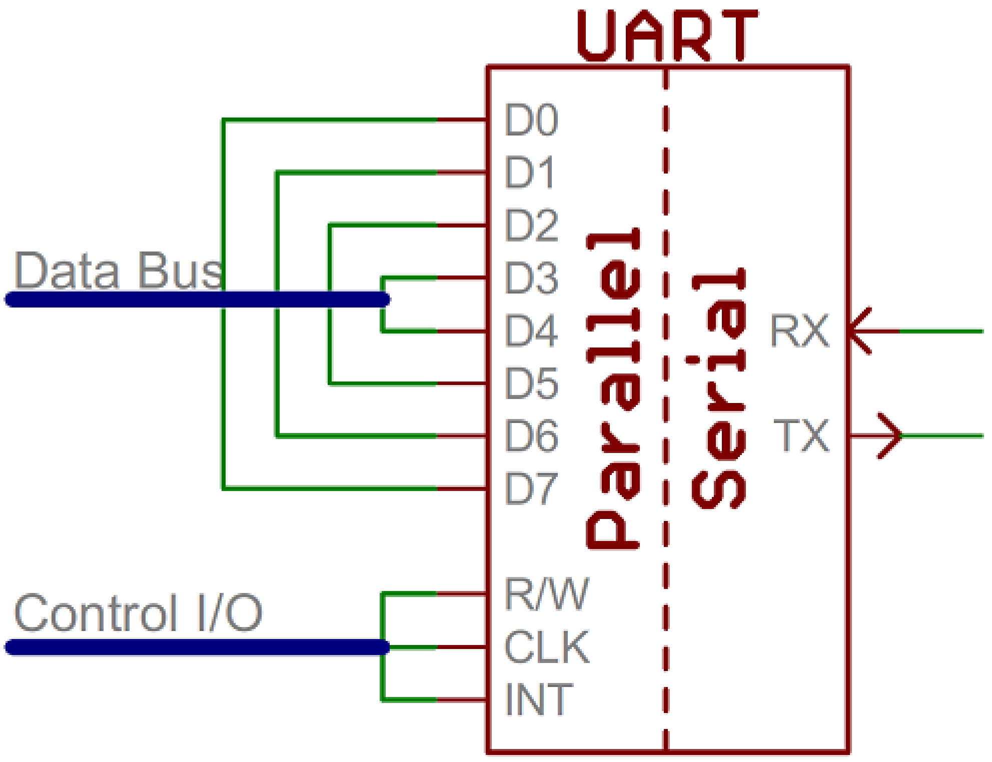


RS-232, which can be found on some of the more ancient computers and peripherals, is like TTL serial flipped on its head. RS-232 signals usually range between -13V and 13V, though the spec allows for anything from +/- 3V to +/- 25V. On these signals a low voltage (-5V, -13V, etc.) indicates the idle line, a stop bit, or a data bit of value 1. A high RS-232 signal means either a start bit, or a 0-value data bit. That's kind of the opposite of TTL serial.



Between the two serial signal standards, TTL is much easier to implement into embedded circuits. However the low voltage levels are more susceptible to losses across long transmission lines. RS-232, or more complex standards like RS-485, are better suited to long range serial transmissions.

When you’re connecting two serial devices together, it’s important to make sure their signal voltages match up. You can’t directly interface a TTL serial device with an RS-232 bus. You’ll have to shift those signals!



Super-simplified UART interface. Parallel on one end, serial on the other.

Internal UART block diagram (courtesy of the Exar ST16C550 datasheet)

Wireless Devices

Wireless technology describes electronic devices that communicate without cords using radio frequency signals. Wireless technology is used in a variety of modern device to provide convenience and greater mobility, and wireless devices play an important role in voice and Internet communications.

Wireless Router

A wireless router is a device that accepts an incoming Internet connection and sends data as RF signals to other wireless devices that are near the router. Wireless routers are used to connect wireless-enabled computers and other devices to the Internet. A network set up with a wireless router is sometimes called a wireless local area network (WLAN.). Many routers have built-in security features such as firewalls which help protect devices connected to the router against malicious data, such as computer viruses.

Wireless Adapters

Wireless adapters are hardware devices installed inside computers that enable wireless connectivity. If a computer does not have a wireless adapter, it will not be able to connect to a router in order to access the Internet. Some computers have wireless adapters built directly into the motherboard while it is also possible to install stand-alone wireless adapters to add wireless capability to a computer that did not come with an adapter built in.

Wireless Repeater

A wireless repeater is a wireless networking device that is used to extend the range of a router. A repeater receives wireless signals and then re-emits them with increased strength. By placing a repeater between a router and the computer connected to the router, signal strength can be boosted, resulting in faster connection speeds.

Wireless Phones

Cellular and cordless phones are two more examples of device that make use of wireless signals. Cordless phones have a limited range, but cell phones typically have a much larger range than local wireless networks, since cell phone providers use large telecommunication towers to provide cell phone coverage.Satellite phones make use of signals from satellites to communicate, similar to Global Positioning System (GSP) devices.

Introduction to Counter/Timers

Counter/timer hardware is a crucial component of most embedded systems. In some cases a timer is needed to measure elapsed time; in others we want to count or time some external events. Here's a primer on the hardware.

Counter/timer hardware is a crucial component of most embedded systems. In some cases, a timer measures elapsed time (counting processor clock ticks). In others, we want to count or time external events. The names counter and timer can be used interchangeably when talking about the hardware. The difference in terminology has more to do with how the hardware is used in a given application.

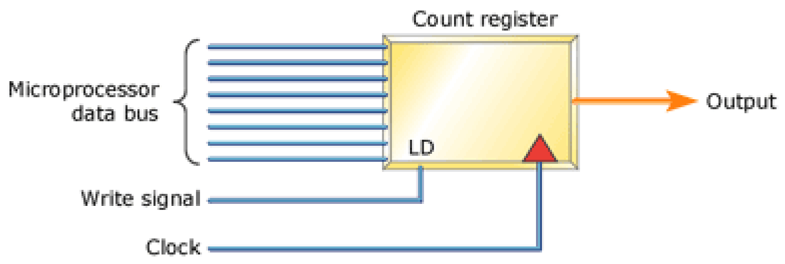


Figure A simple counter/timer

Figure 1 shows a simple timer similiar to those often included on-chip within a microcontroller. You could build something similar from a couple of 74HC161 counters or a programmable logic device. The timer shown consists of a loadable 8-bit count register, an input clock signal, and an output signal. Software loads the count register with an initial value between 0x00 and 0xFF. Each subsequent transition of the input clock signal increments that value.

When the 8-bit count overflows, the output signal is asserted. The output signal may thereby trigger an interrupt at the processor or set a bit that the processor can read. To restart the timer, software reloads the count register with the same or a different initial value.

If a counter is an up counter, it counts up from the initial value toward 0xFF. A down counter counts down, toward 0x00.

A typical counter will have some means to start the counter running once it is loaded, usually by setting a bit in a control register. This is not shown in the figure. A real counter would generally also provide a way for the processor to read the current value of the count register at any time, over the data bus.

Semi-automatic

A timer with automatic reload capability will have a latch register to hold the count written by the processor. When the processor writes to the latch, the count register is written as well. When the timer later overflows, it first generates an output signal. Then, it automatically reloads the contents of the latch into the count register. Since the latch still holds the value written by the processor, the counter will begin counting again from the same initial value.

Such a timer will produce a regular output with the same accuracy as the input clock. This output could be used to generate a periodic interrupt like a real-time operating system (RTOS) timer tick, provide a baud rate clock to a UART, or drive any device that requires a regular pulse.

A variation of this feature found in some timers uses the value written by the processor as the endpoint rather than the initial count. In this case, the processor writes into a terminal count register that is constantly compared with the value in the count register. The count register is always reset to zero and counts up. When it equals the value in the terminal count register, the output signal is asserted. Then the count register is reset to zero and the process repeats. The terminal count remains the same. The overall effect is the same as an overflow counter. A periodic signal of a pre-determined length will then be produced.

If a timer supports automatic reloading, it will often make this a software-selectable feature. To distinguish between a count that will not repeat automatically and one that will, the hardware is said to be in one of two modes: one-shot or periodic. The mode is generally controlled by a field in the timer's control register.

Input capture

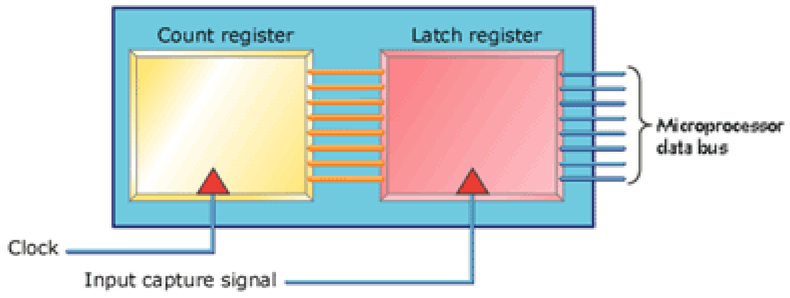


Figure: An input capture timer

An input capture timer, like the one shown in Figure 2, has a latch connected to the timer's count register. The timer is run at a constant clock rate (usually a derivative of the processor clock), so that the count registers is constantly incrementing (or decrementing, for a down counter). An external signal latches the value of the free-running timer into the processor-visible register and generates an output signal (typically an interrupt).

One use for an input capture timer is to measure the time between the leading edge of two pulses. By reading the value in the latch and comparing it with a previous reading, the software can determine how many clock cycles elapsed. In some cases, the timer's count register might be automatically reset just after its value is latched. If so, the software can directly interpret the value it reads as the number of clock ticks elapsed. An input capture pin can usually be programmed to capture on either the rising or falling edge of the input signal.

Options abound

Many timers provide a means to prescale the input clock signal. For example, the 8-bit timer in Atmel's AT90S8515 microcontroller can be incremented with every processor clock cycle, every 8th, every 64th, every 256th, or every 1,024th. Selecting a less frequent update cycle is called prescaling the input clock. Similarly, each increment could occur on either the rising or falling edge of some other signal entirely. In the Atmel part, these features are software-selectable.

Some timers can directly control a general-purpose I/O pin. When an overflow occurs, the pin can be automatically set to 1, reset to 0, or toggled. This can be useful in, for example, generating a PWM signal.1 Using two different initial or terminal count values and a one-shot timer that toggles the I/O pin on overflow, the pin could be set to 1 for a desired amount of time, then 0 for a different amount of time, then 1 again, and so on. The period of the PWM signal would be a function of the sum of the two timer lengths. The duty cycle would then be the length of time that the pin is set to 1 as a percentage of the period.

Introduction to Watchdog Timers

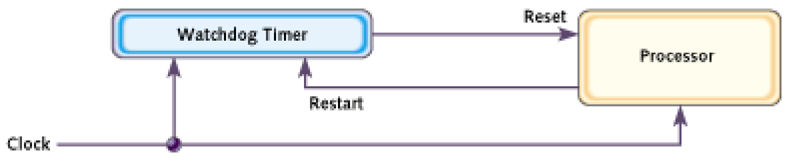
For those embedded systems that can't be constantly watched by a human, watchdog timers may be the solution.

Most embedded systems need to be self-reliant. It's not usually possible to wait for someone to reboot them if the software hangs. Some embedded designs, such as space probes, are simply not accessible to human operators. If their software ever hangs, such systems are permanently disabled. In other cases, the speed with which a human operator might reset the system would be too slow to meet the uptime requirements of the product.

A watchdog timer is a piece of hardware that can be used to automatically detect software anomalies and reset the processor if any occur. Generally speaking, a watchdog timer is based on a counter that counts down from some initial value to zero. The embedded software selects the counter's initial value and periodically restarts it. If the counter ever reaches zero before the software restarts it, the software is presumed to be malfunctioning and the processor's reset signal is asserted. The processor (and the embedded software it's running) will be restarted as if a human operator had cycled the power.

Figure 1 shows a typical arrangement. As shown, the watchdog timer is a chip external to the processor. However, it could also be included within the same chip as the CPU. This is done in many microcontrollers. In either case, the output from the watchdog timer is tied directly to the processor's reset signal.

Figure 1: A typical watchdog setup



Kicking the dog

The process of restarting the watchdog timer's counter is sometimes called "kicking the dog." The appropriate visual metaphor is that of a man being attacked by a vicious dog. If he keeps kicking the dog, it can't ever bite him. But he must keep kicking the dog at regular intervals to avoid a bite. Similarly, the software must restart the watchdog timer at a regular rate, or risk being restarted.

A simple example is shown in Listing 1. Here we have a single infinite loop that controls the entire behavior of the system. This software architecture is common in many embedded systems with low-end processors and behaviors based on a single operational frequency. The hardware implementation of this watchdog allows the counter value to be set via a memory-mapped register.

Listing 1: Kicking the dog

```
uint16 volatile * pWatchdog =
   (uint16 volatile *) 0xFF0000;

main(void)
{
      hwinit();

      for (;;)
      {
         *pWatchdog = 10000;
         read_sensors();
         control_motor();
         display_status();
      }
}
```

Suppose that the loop must execute at least once every five milliseconds. (Say the motor must be fed new control parameters at least that often.) If the watchdog timer's counter is initialized to a value that corresponds to five milliseconds of elapsed time, say 10,000, and the software has no bugs, the watchdog timer will never expire; the software will always restart the counter before it reaches zero.

Software anomalies

A watchdog timer can get a system out of a lot of dangerous situations. However, if it is to be effective, resetting the watchdog timer must be considered within the overall software design. Designers must know what kinds of things could go wrong with their software, and ensure that the watchdog timer will detect them, if any occur.

Systems hang for any number of reasons. A logical fallacy resulting in the execution of an infinite loop is the simplest. Suppose such a condition occurred within the read_sensors() call in Listing 1. None of the other software (except ISRs, if interrupts are still enabled) would get a chance to run again.

Another possibility is that an unusual number of interrupts arrives during one pass of the loop. Any extra time spent in ISRs is time not spent executing the main loop. A dangerous delay in feeding the motor new control instructions could result.

When multitasking kernels are used, deadlocks can occur. For example, a group of tasks might get stuck waiting on each other and some external signal that one of them needs, leaving the whole set of tasks hung indefinitely.

If such faults are transient, the system may function perfectly for some length of time after each watchdog-induced reset. However, failed hardware could lead to a system that constantly resets. For this reason it may be wise to count the number of watchdog-induced resets, and give up trying after some fixed number of failures.

Karate lessons

An actual watchdog implementation would usually have an interface to the software that is more complex than the one in Listing 1. When the set of instructions required to reset the watchdog is very simple, it's possible that buggy software could perform this action by accident. Consider a bug that causes the value 10,000 to be written to every location in memory, over and over again. This code would regularly restart the watchdog counter, and the watchdog might never bite. To prevent this, many watchdog implementations require that a complex sequence of two or more consecutive writes be used to restart the watchdog timer.

If the watchdog is built into your microcontroller, it may not be enabled automatically when the device resets. You must be sure to enable it during hardware initialization. To provide protection against a bug accidentally disabling the watchdog, the hardware design usually makes it impossible to disable the watchdog timer once it has been enabled.

If your software can do a complete loop faster than the watchdog period, the structure in Listing 1 may work fine for you. It gets more challenging if some part of your software takes a long time to complete. Say you have a loop that waits for an element to heat to a certain temperature before returning. Many watchdog timers have a maximum period of around two seconds. If you are going to delay for more than that length of time, you may have to kick the dog from within the waiting loop. If there are many such places in your software, control of the watchdog can become problematic.

System initialization is a part of the code that often takes longer than the watchdog timer's maximum period. Perhaps a memory test or ROM to RAM data transfer slows this down. For this reason, some watchdogs can wait longer for their first kick than they do for subsequent kicks.

As threads of control are added to software (in the form of ISRs and software tasks), it becomes ineffective to have just one place in the code where the watchdog is kicked.

Choosing a proper kick interval is also an important issue, one that can only be addressed in a system-specific manner. These and other issues of greater complexity are discussed in the references listed at the end of this article.

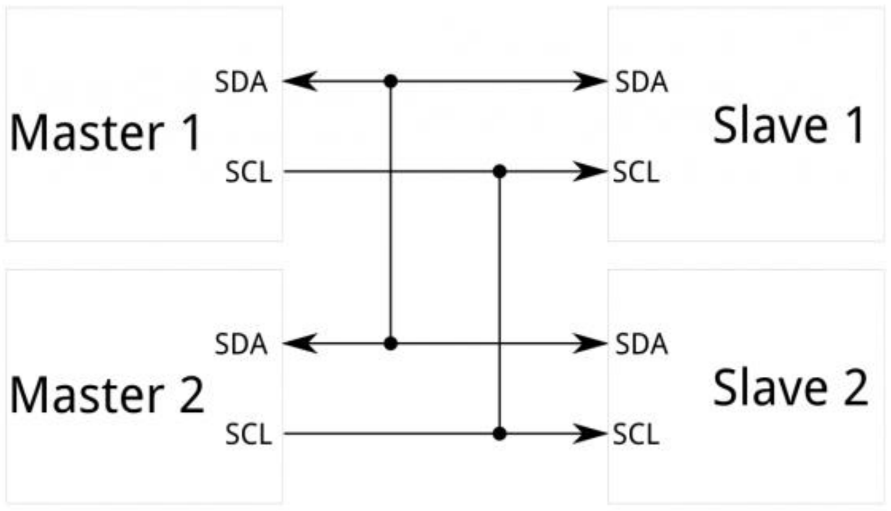
Dog days

A watchdog timer is useful tools in helping your system recover from transient failures. Since it is so common to find watchdogs built into modern microcontrollers, the technique is effectively free. If you are working on a mission-critical system, then either common sense or a regulatory body will insist that you use a watchdog. It's always a good idea to make your systems more self-reliant.

I^{2}C

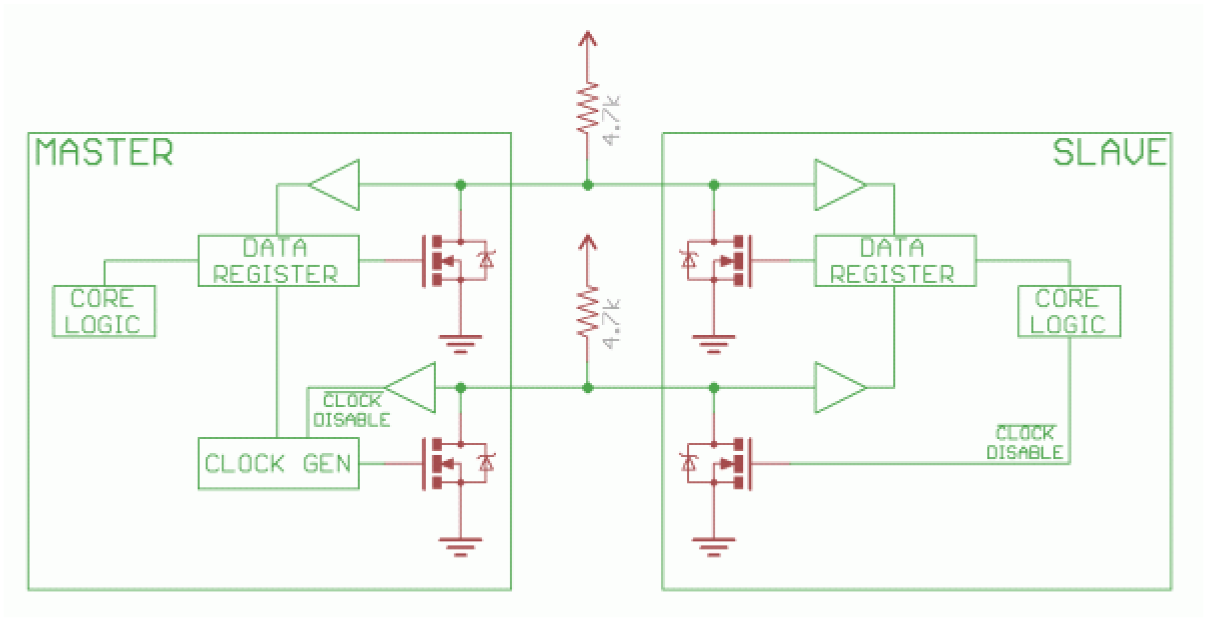
The Inter-integrated Circuit (I^{2}C) Protocol is a protocol intended to allow multiple “slave” digital integrated circuits (“chips”) to communicate with one or more “master” chips. Like the Serial Peripheral Interface (SPI), it is only intended for short distance communications within a single device. Like Asynchronous Serial Interfaces (such as RS-232 or UARTs), it only requires two signal wires to exchange information.

1. Enter I^{2}C - The Best of Both Worlds!



ach I^{2}C bus consists of two signals: SCL and SDA. SCL is the clock signal, and SDA is the data signal. The clock signal is always generated by the current bus master; some slave devices may force the clock low at times to delay the master sending more data (or to require more time to prepare data before the master attempts to clock it out). This is called "clock stretching" and is described on the protocol page.

Unlike UART or SPI connections, the I^{2}C bus drivers are "open drain", meaning that they can pull the corresponding signal line low, but cannot drive it high. Thus, there can be no bus contention where one device is trying to drive the line high while another tries to pull it low, eliminating the potential for damage to the drivers or excessive power dissipation in the system. Each signal line has a pull-up resistor on it, to restore the signal to high when no device is asserting it low.



Notice the two pull-up resistors on the two communication lines.

Resistor selection varies with devices on the bus, but a good rule of thumb is to start with 4.7k and adjust down if necessary. I^{2}C is a fairly robust protocol, and can be used with short runs of wire (2-3m). For long runs, or systems with lots of devices, smaller resistors are better.

CAN Logger is a control unit which allows to filter and to memorize all CAN frames of the bus on which is connected on. Beside the system can be directly interfaced to PC through USB or RS232 interfaces.

The unit is easy to use and to configure, thanks to the provided software for Windows XP, Vista and 7.

CAN Logger can be used immediately with any CAN BUS, because it is completely configurable.
Customizable parameters are:
- High-speed / Low-speed.
- Baudrate.
- Address.

CAN Logger is available in **two versions**, CAN Logger-Flash and CAN Logger-SD:

CAN Logger-Flash can record up to 9,000 can messages on a on-board flash memory readable with the provided software.
CAN Logger-SD allows to record messages form CAN bus directly on a Micro-SD card: that SD card can be read from any card reader, beside there is the possibility to connect the logger directly to a PC using the provided software.

USB,

short for **Universal Serial Bus**, is an industry standard that defines cables, connectors and communications protocols for connection, communication, and power supply between computers and devices.[3]

USB was designed to standardize the connection of computer peripherals (including keyboards, pointing devices, digital cameras, printers, portable media players, disk drives and network adapters) to personal computers, both to communicate and to supply electric power. It has largely replaced a variety of earlier interfaces, such as serial portsand parallel ports, as well as separate power chargers for portable devices – and has become commonplace on a wide range of devices.[4]

In general, there are three basic formats of USB connectors: the default or standard format intended for desktop or portable equipment (for example, on USB flash drives), the miniintended for mobile equipment (now deprecated except the Mini-B, which is used on many cameras), and the thinner micro size, for low-profile mobile equipment (most modern mobile phones). Also, there are 5 modes of USB data transfer, in order of increasing bandwidth: Low Speed (from 1.0), Full Speed (from 1.0), High Speed (from 2.0), SuperSpeed(from 3.0), and SuperSpeed+ (from 3.1); modes have differing hardware and cabling requirements. USB devices have some choice of implemented modes, and USB version is not a reliable statement of implemented modes. Modes are identified by their names and icons, and the specifications suggests that plugs and receptacles be colour-coded (SuperSpeed is identified by blue).

Unlike other data buses (e.g., Ethernet, HDMI), USB connections are directed, with both upstream and downstream ports emanating from a single host. This applies to electrical power, with only downstream facing ports providing power; this topology was chosen to easily prevent electrical overloads and damaged equipment. Thus, USB cables have different ends: A and B, with different physical connectors for each. Therefore, in general, each different format requires four different connectors: a plug and receptacle for each of the A and B ends. USB cables have the plugs, and the corresponding receptacles are on the computers or electronic devices. In common practice, the A end is usually the standard format, and the B side varies over standard, mini, and micro. The mini and micro formats also provide for USB On-The-Go with a hermaphroditic AB receptacle, which accepts either an A or a B plug. On-the-Go allows USB between peers without discarding the directed topology by choosing the host at connection time; it also allows one receptacle to perform double duty in space-constrained applications.

There are cables with A plugs on both ends, which may be valid if the cable includes, for example, a USB host-to-host transfer device with 2 ports, but they could also be non-standard and erroneous and should be used carefully.[5]

The micro format is the most durable from the point of view of designed insertion lifetime. The standard and mini connectors have a design lifetime of 1,500 insertion-removal cycles,[6] the improved Mini-B connectors increased this to 5,000. The micro connectors were designed with frequent charging of portable devices in mind, so have a design life of

10,000 cycles[6] and also place the flexible contacts, which wear out sooner, on the easily replaced cable, while the more durable rigid contacts are located in the receptacles. Likewise, the springy component of the retention mechanism, parts that provide required gripping force, were also moved into plugs on the cable side.[7]

UNIT-3 ARM PROCESSOR-7

The ARM architecture has been designed to allow very small, yet high-performance implementations. The architectural simplicity of ARM processors leads to very small implementations, and small implementations allow devices with very low power consumption.
The ARM is a Reduced Instruction Set Computer (RISC), as it incorporates

These typical RISC architecture features:

A large uniform register file
A load/store architecture, where data-processing operations only operate on Register contents, not directly on memory contents Simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only
Uniform and fixed-length instruction fields, to simplify instruction
Decode.
In addition, the ARM architecture gives you:
Control over both Arithmetic Logic Unit (ALU) and shifter in every data-
Processing instruction to maximize the use of an ALU and a shifter Load and Store multiple to maximize data throughput.

These enhancements to a basic RISC architecture allow ARM processors to
Achieve a good balance of high performance, low code size and low power
Consumption.

ARM Block diagram

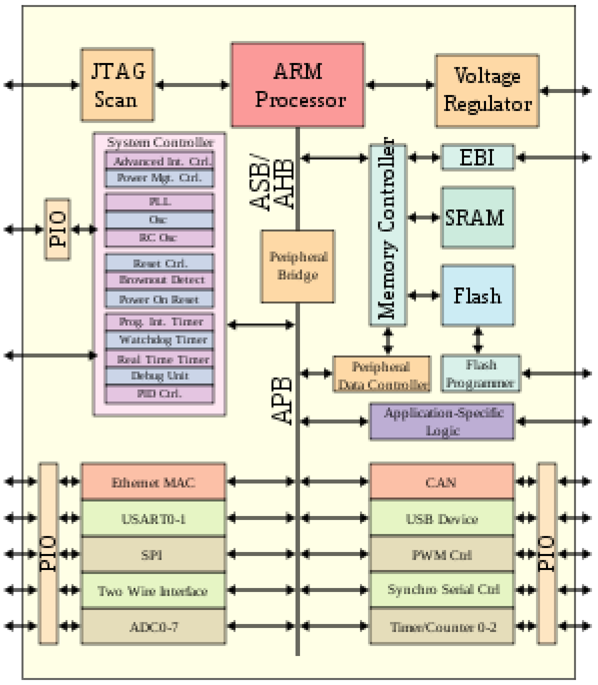
The main parts of the ARM processor are:

1. Register file: The processor has a total of 37 registers made up of
   31 general 32 bit registers and 6 status registers
2. Booth Multiplier
3. Barrel shifter
4. Arithmetic Logic Unit (ALU)
5. Control Unit.

Microprocessor-based system on a chip

Constants in Assembly for Arm Architecture

Arm is a 32-bit CPU architecture where every instruction is 32 bits long. Any constants which are part of an instruction must be encoded within the 32 bits of the given instruction and this naturally limits the range of constants that can be represented in one instruction. This post will show you how we can deal with these limitations and how the latest revision of the Arm architecture (Armv7) provides a simple and efficient solution



Most arithmetic and logical Arm instructions accept 3 parameters:

- The destination: always a register.
- Operand 1: always a register.
- Operand 2: a register, an immediate constant value or a shifted register. We'll cover shifted registers in a future post. For now, we're only interested in the constants. Examples of such instructions are:

```
add   r0, r1, r2    @ r0 = r1 + r2
sub   r0, r1, #3    @ r0 = r1 - 3
```

An Operand 2 immediate must obey the following rule to fit in the instruction: an 8-bit value rotated right by an even number of bits between 0 and 30 (inclusive). This allows for constants such as 0xFF (0xFF rotated right by 0), 0xFF00 (0xFF rotated right by 24) or 0xF000000F (0xFF rotated right by 4).

Operand 2 immediates are also valid immediates for mov instructions, making it possible to move constant values into registers without performing any other computation:

```
mov    r0, #0xFF0    @ r0 = 0xFF0
```

In software - especially in languages like C - constants tend to be small. When they are not small they tend to be bit masks. Operand 2 immediates provide a reasonable compromise between constant coverage and encoding space; most common constants can be encoded directly.

Loading a Constant from the Instruction Stream: Armv7 way

As mentioned earlier, there are other ways to load a constant. In the latest version of the Arm architecture, Armv7, two new instructions were introduced to improve the situation:

- movw, or move wide, will move a 16-bit constant into a register, implicitly zeroing the top 16 bits of the target register.
- movt, or move top, will move a 16-bit constant into the top half of a given register without altering the bottom 16 bits. Now moving an arbitrary 32-bit constant is as simple as this:

```
movw      r0, #0x5678      @ r0 = 0x00005678
movt      r0, #0x1234      @ r0 = (r0 & 0x0000FFFF) | 0x12340000 (=0x12345678)
```

Note that the order matters since movw will zero the upper 16 bits. Here again the GNU assembler provides some syntactic sugar: the prefixes :upper16: and :lower16: allow you to extract the corresponding half from a 32-bit constant:

```
.equ    label, 0x12345678
movw    r0, #:lower16:label
movt    r0, #:upper16:label
```

While this approach takes two instructions, it does not require any extra space to store the constant so both the movw/movt method and the ldr method will end up using the same amount of memory. Memory bandwidth is precious in and the movw/movt approach avoids an extra read on the data side, not to mention the read could have missed the cache.

If you know you can use it, movw/movt is the recommended way to load a 32-bit constant. However, if it is possible to encode the 32-bit constant using an 8-bit immediate and if necessary rotated right, try to use Operand 2 directly, and avoid the need to use an extra register.

The Instruction Set

We now know what the ARM provides by way of memory and registers, and the sort of instructions to manipulate them. This chapter describes those instructions in great detail.

As explained in the previous chapter, all ARM instructions are 32 bits long. Here is a typical one:

10101011100101010010100111101011

Fortunately, we don't have to write ARM programs using such codes. Instead we use assembly language. We saw at the end of Chapter One a few typical ARM mnemonics. Usually, mnemonics are followed by one or more operands which are used to completely describe the instruction.

An example mnemonic is **ADD**, for 'add two registers'. This alone doesn't tell the assembler which registers to add and where to put the result. If the left and right hand side of the addition are R1 and R2 respectively, and the result is to go in R0, the operand part would be written R0,R1,R2. Thus the complete add instruction, in assembler format, would be:

ADD R0, R1, R2 ;R0 = R1 + R2

Most ARM mnemonics consist of three letters, e.g. **SUB**, **MOV**, **STR**, **STM**. Certain 'optional extras' may be added to slightly alter the affect of the instruction, leading to mnemonics such as **ADCNES** and **SWINE**.

The mnemonics and operand formats for all of the ARM's instructions are described in detail in the sections below. At this stage, we don't explain how to create programs, assemble and run them. There are two main ways of assembling ARM programs - using the assembler built-in to BBC BASIC, or using a dedicated assembler. The former method is more convenient for testing short programs, the latter for developing large scale projects. Chapter Four covers the use of the BASIC assembler.

3.1 Condition codes

The property of conditional execution is common to all ARM instructions, so its representation in assembler is described before the syntax of the actual instructions.

As mentioned in chapter two, there are four bits of condition encoded into an instruction word. This allows sixteen possible conditions. If the condition for the current instruction is true, the execution goes ahead. If the condition does not hold, the instruction is ignored and the next one executed.

The result flags are altered mainly by the data manipulation instructions. These instructions only affect the flags if you explicitly tell them to. For example, a **MOV** instruction which copies the contents of one register to another. No flags are affected. However, the **MOVS** (move with **S**et) instruction additionally causes the result flags to be set. The way in which each instruction affects the flags is described below.

To make an instruction conditional, a two-letter suffix is added to the mnemonic. The suffixes, and their meanings, are listed below.

AL Always

An instruction with this suffix is always executed. To save having to type '**AL**' after the majority of instructions which are unconditional, the suffix may be omitted in this case. Thus **ADDAL** and **ADD** mean the same thing: add unconditionally.

NV Never

All ARM conditions also have their inverse, so this is the inverse of always. Any instruction with this condition will be ignored. Such instructions might be used for 'padding' or perhaps to use up a (very) small amount of time in a program.

EQ Equal

This condition is true if the result flag Z (zero) is set. This might arise after a compare instruction where the operands were equal, or in any data instruction which received a zero result into the destination.

NE Not equal

This is clearly the opposite of **EQ**, and is true if the Z flag is cleared. If Z is set, and instruction with the **NE** condition will not be executed.

VS Overflow set

This condition is true if the result flag V (overflow) is set. Add, subtract and compare instructions affect the V flag.

VC Overflow clear

The opposite to **VS**.

MI Minus

Instructions with this condition only execute if the N (negative) flag is set. Such a condition would occur when the last data operation gave a result which was negative. That is, the N flag reflects the state of bit 31 of the result. (All data operations work on 32-bit numbers.)

PL Plus

This is the opposite to the **MI** condition and instructions with the **PL** condition will only execute if the N flag is cleared.

The next four conditions are often used after comparisons of two unsigned numbers. If the numbers being compared are n1 and n2, the conditions are n1>=n2, n1<n2, n1>n2 and n1<=n2, in the order presented.

CS Carry set

This condition is true if the result flag C (carry) is set. The carry flag is affected by arithmetic instructions such as **ADD**, **SUB** and **CMP**. It is also altered by operations involving the shifting or rotation of operands (data manipulation instructions).

When used after a compare instruction, **CS** may be interpreted as 'higher or same', where the operands are treated as unsigned 32-bit numbers. For example, if the left hand operand of **CMP** was 5 and the right hand operand was 2, the carry would be set. You can use **HS** instead of **CS** for this condition.

CC Carry clear

This is the inverse condition to **CS**. After a compare, the **CC** condition may be interpreted as meaning 'lower than', where the operands are again treated as unsigned numbers. An synonym for **CC** is **LO**.

HI Higher

This condition is true if the C flag is set and the Z flag is false. After a compare or subtract, this combination may be interpreted as the left hand operand being greater than the right hand one, where the operands are treated as unsigned.

LS Lower or same

This condition is true if the C flag is cleared or the Z flag is set. After a compare or subtract, this combination may be interpreted as the left hand operand being less than or equal to the right hand one, where the operands are treated as unsigned.

The next four conditions have similar interpretations to the previous four, but are used when signed numbers have been compared. The difference is that they take into account the state of the V (overflow) flag, whereas the unsigned ones don't.

Again, the relationships between the two numbers which would cause the condition to be true are n1>=n2, n1<n2, n1>n2, n1<=n2.

GE Greater than or equal

This is true if N is cleared and V is cleared, or N is set and V is set.

LT Less than

This is the opposite to **GE** and instructions with this condition are executed if N is set and V is cleared, or N is cleared and V is set.

GT Greater than

This is the same as **GE**, with the addition that the Z flag must be cleared too.

LE Less than or equal

This is the same as **LT**, and is also true whenever the Z flag is set.

Note that although the conditions refer to signed and unsigned numbers, the operations on the numbers are identical regardless of the type. The only things that change are the flags used to determine whether instructions are to be obeyed or not.

The flags may be set and cleared explicitly by performing operations directly on R15, where they are stored.

3.2 Group one - data manipulation

This group contains the instructions which do most of the manipulation of data in ARM programs. The other groups are concerned with moving data between the processor and memory, or changing the flow of control.

The group comprises sixteen distinct instructions. All have a very similar format with respect to the operands they take and the 'optional extras'. We shall describe them generically using **ADD**, then give the detailed operation of each type.

Assembler format

ADD has the following format:

ADD{cond}{S} <dest>, <lhs>, <rhs>

The parts in curly brackets are optional. **Cond** is one of the two-letter condition codes listed above. If it is omitted, the 'always' condition **AL** is assumed. The **S**, if present, causes the instruction to affect the result flags. If there is no **S**, none of the flags will be changed. For example, if an instruction **ADDS** É yields a result which is negative, then the N flag will be set. However, just **ADD** É will not alter N (or any other flag) regardless of the result.

After the mnemonic are the three operands. **<dest>** is the destination, and is the register number where the result of the **ADD** is to be stored. Although the assembler is happy with actual numbers here, e.g. 0 for R0, it recognises R0, R1, R2 etc. to stand for the register numbers. In addition, you can define a name for a register and use that instead. For example, in BBC BASIC you could say:-

iac = 0

where **iac** stands for, say, integer accumulator. Then this can be used in an instruction:-

ADD iac, iac, #1

The second operand is the left hand side of the operation. In general, the group one instructions act on two values to provide the result. These are referred to as the left and right hand sides, implying that the operation determined by the mnemonic would be written between them in mathematics. For example, the instruction:

ADD R0, R1, R2

has R1 and R2 as its left and right hand sides, and R0 as the result. This is analogous to an assignment such as **R0=R1+R2** in BASIC, so the operands are sometimes said to be in 'assignment order'.

The **<lhs>** operand is always a register number, like the destination. The **<rhs>** may either be a register, or an immediate operand, or a shifted or rotated register. It is the versatile form that the right hand side may take which gives much of the power to these instructions.

If the **<rhs>** is a simple register number, we obtain instructions such as the first **ADD** example above. In this case, the contents of R1 and R2 are added (as signed, 32-bit numbers) and the result stored in R0. As there is no condition after the instruction, the **ADD** instruction will always be executed. Also, because there was no **S**, the result flags would not be affected.

The three examples below all perform the same **ADD** operation (if the condition is true):

ADDNE R0, R0, R2

ADDS R0, R0, R2

ADDNES R0, R0, R2

They all add R2 to R0. The first has a **NE** condition, so the instruction will only be executed if the Z flag is cleared. If Z is set when the instruction is encountered, it is ignored. The second one is unconditional, but has the **S** option. Thus the N, Z, V and C flags will be altered to reflect the result. The last example has the condition and the **S**, so if Z is cleared, the **ADD** will occur and the flags set accordingly. If Z is set, the **ADD** will be skipped and the flags remain unaltered.

Immediate operands

Immediate operands are written as a # followed by a number. For example, to increment R0, we would use:

ADD R0, R0, #1

Now, as we know, an ARM instruction has 32 bits in which to encode the instruction type, condition, operands etc. In group one instructions there are twelve bits available to encode immediate operands. Twelve bits of binary can represent numbers in the range 0..4095, or -2048..+2047 if we treat them as signed.

The designers of the ARM decided not to use the 12 bits available to them for immediate operands in the obvious way just mentioned. Remember that some of the status bits are stored in bits 26..31 of R15. If we wanted to store an immediate value there using a group one instruction, there's no way we could using the straightforward twelve-bit number approach.

To get around this and related problems, the immediate operand is split into two fields, called the position (the top four bits) and the value (stored in the lower eight bits). The value is an eight bit number representing 256 possible combinations. The position is a four bit field which determines where in the 32-bit word the value lies. Below is a diagram showing how the sixteen values of the position determine where the value goes. The bits of the value part are shown as 0, 1, 2 etc.

The way of describing this succinctly is to say that the value is rotated by 2*position bits to the right within the 32-bit word. As you can see from the diagram, when position=&03, all of the status bits in R15 can be reached.

b31	***b0***	***Pos***
........................76543210		**&00**
10........................765432		**&01**
3210........................7654		**&02**
543210........................76		**&02**
76543210........................		**&04**
..76543210......................		**&05**
....76543210....................		**&06**
......76543210..................		**&07**
........76543210................		**&08**
..........76543210..............		**&09**
............76543210............		**&0A**
..............76543210..........		**&0B**
................76543210........		**&0C**
..................76543210......		**&0D**
....................76543210....		**&0E**
......................76543210..		**&0F**

The sixteen immediate shift positions

When using immediate operands, you don't have to specify the number in terms of position and value. You just give the number you want, and the assembler tries to generate the appropriate twelve-bit field. If you specify a value which can't be generated, such as &101 (which would require a nine-bit value), an error is generated. The **ADD** instruction below adds 65536 (&1000) to R0:

ADD R0, R0, #&1000

To get this number, the assembler might use a position value of 8 and value of 1, though other combinations could also be used.

Shifted operands

If the **<rhs>** operand is a register, it may be manipulated in various ways before it is used in the instruction. The contents of the register aren't altered, just the value given to the ALU, as applied to this operation (unless the same register is also used as the result, of course).

The particular operations that may be performed on the **<rhs>** are various types of shifting and rotation. The number of bits by which the register is shifted or rotated may be given as an immediate number, or specified in yet another register.

Shifts and rotates are specified as left or right, logical or arithmetic. A left shift is one where the bits, as written on the page, are moved by one or more bits to the left, i.e. towards the more significant end. Zero-valued bits are shifted in at the right and the bits at the left are lost, except for the final bit to be shifted out, which is stored in the carry flag.

Left shifts by n bits effectively multiply the number by 2^n, assuming that no significant bits are 'lost' at the top end.

A right shift is in the opposite direction, the bits moving from the more significant end to the lower end, or from left to right on the page. Again the bits shifted out are lost, except for the last one which is put into the carry. If the right shift is logical then zeros are shifted into the left end. In arithmetic shifts, a copy of bit 31 (i.e. the sign bit) is shifted in.

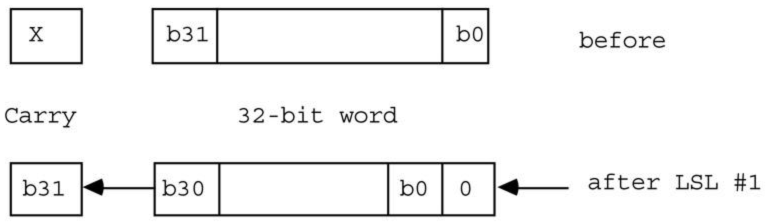
Right arithmetic shifts by n bits effectively divide the number by 2^n, rounding towards minus infinity (like the BASIC **INT** function).

A rotate is like a shift except that the bits shifted in to the left (right) end are those which are coming out of the right (left) end.

Here are the types of shifts and rotates which may be used:

LSL #n Logical shift left immediate

n is the number of bit positions by which the value is shifted. It has the value 0..31.
An **LSL** by one bit may be pictured as below:



After n shifts, n zero bits have been shifted in on the right and the carry is set to bit 32-n of the original word.

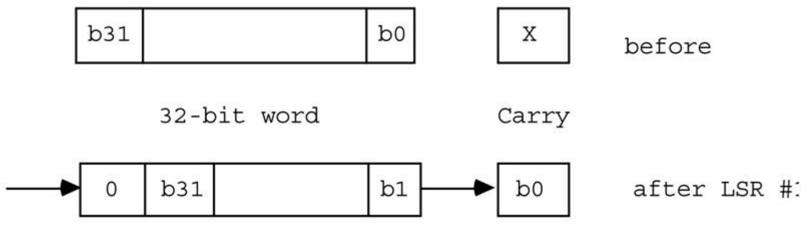
Note that if there is no shift specified after the **<rhs>** register value, **LSLÊ#0** is used, which has no effect at all.

ASL #n Arithmetic shift left immediate

This is a synonym for **LSL #n** and has an identical effect.

LSR #n Logical shift right immediate

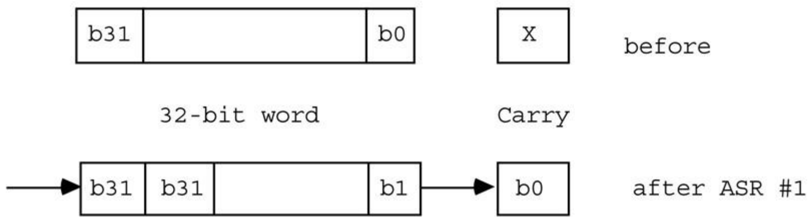
n is the number of bit positions by which the value is shifted. It has the value 1...32. An **LSR** by one bit is shown below:



After n of these, n zero bits have been shifted in on the left, and the carry flag is set to bit n-1 of the original word.

ASR #n Arithmetic shift right immediate

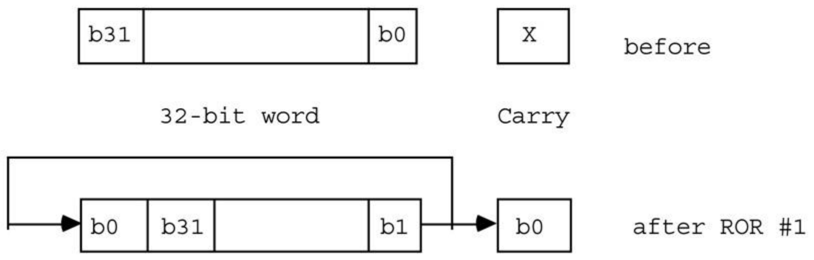
n is the number of bit positions by which the value is shifted. It has the value 1..32. An **ASR** by one bit is shown below:



If ' sign' is the original value of bit 31 then after n shifts, n 'sign' bits have been shifted in on the left, and the carry flag is set to bit n-1 of the original word.

ROR #n Rotate right immediate

n is the number of bit positions to rotate in the range 1..31. A rotate right by one bit is shown below:

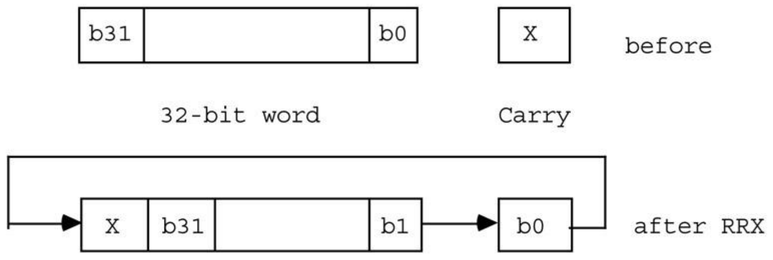


After n of these rotates, the old bit n is in the bit 0 position; the old bit (n-1) is in bit 31 and in the carry flag.

Note that a rotate left by n positions is the same as a rotate right by (32-n). Also note that there is no rotate right by 32 bits. The instruction code which would do this has been reserved for rotate right with extend (see below).

RRX rotate right one bit with extend

This special case of rotate right has a slightly different effect from the usual rotates. There is no count; it always rotates by one bit only. The pictorial representation of **RRX** is:



The old bit 0 is shifted into the carry. The old content of the carry is shifted into bit 31.

Note that there is no equivalent **RLX** rotate. However, the same effect may be obtained using the instruction:

ADCS R0, R0, R0

PC relative addressing

The assembler will accept a special form of pre-indexed address in the **LDR** instruction, which is simply:

LDR <dest>,<expression>

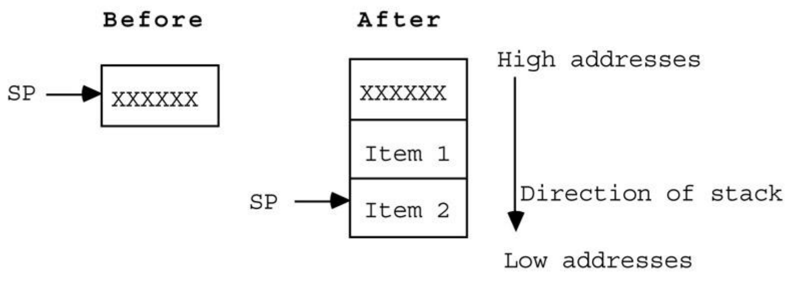
Where **<expression>** yields an address. In this case, the instruction generated will use R15 (i.e. the program counter) as the base register, and calculate the immediate offset automatically. If the address given is not in the correct range (-4095 to +4095) from the instruction, an error is given.

An example of this form of instruction is:

LDR R0, default

(We assume that **default** is a label in the program. Labels are described more fully in the next chapter, but for now suffice is to say that they are set to the address of the point in the program where they are defined.)

As the assembler knows the value of the PC when the program is executed, it can calculate the immediate offset required to access the location **default**. This must be within the range -4095 to +4095 of course. This form of addressing is used frequently to access constants embedded in the program.



Software interrupt

The final group is the most simple, and the most complex. It is very simple because it contains just one instruction, **SWI**, whose assembler format has absolutely no variants or options.

The general form of **SWI** is:

SWI {cond} <expression>

It is complex because depending on **<expression>**, **SWI** will perform tasks as disparate as displaying characters on the screen, setting the auto-repeat speed of the keyboard and loading a file from the disc.

SWI is the user's access to the operating system of the computer. When a **SWI** is executed, the CPU enters supervisor mode, saves the return address in R14_SVC, and jumps to location 8. From here, the operating system takes over. The way in which the **SWI** is used depends on **<expression>**. This is encoded as a 24-bit field in the instruction. The operating system can examine the instruction using, for example:.

```
STMFD R13!,{R0-R12}   ;Save user's registers

                      BIC R14, R14,#&FC000003 ;Mask status bits

                      LDR R13,[R14,#-4]   ;Load SWI instruction
```

To find out what **<expression>** is.

Since the interpretation of **<expression>** depends entirely on the system in which the program is executing, we cannot say much more about **SWI** here. However, as practical programs need to use operating system functions, the examples in later chapters will use a 'standard' set that you could reasonably expect. Two of the most important ones are called **WriteC** and **ReadC**. The former sends the character in the bottom byte of R0 to the screen, and the latter reads a character from the keyboard and returns it in the bottom byte of R0.

Note: The code in the example above will be executed in **SVC** mode, so the accesses to R13 and R14 are actually to R13_SVC and R14_SVC. Thus the user's versions of these registers do not have to be saved.

3.7 Instruction timings

Instructions which are skipped due to the condition failing always execute in 1s cycle.

Group one

MOV, **ADD** etc. 1 s-cycle. If **<rhs>** contains a shift count in a register (i.e. not an immediate shift), add 1 s-cycle. If **<dest>** is R15, add 1 s + 1 n-cycle.

Group one A

MUL, **MLA**. 1 s + 16 i- cycles worst-case.

Group two

LDR. 1 s + 1 n + 1 i-cycle. If **<dest>** is R15, add 1 s + 1n-cycle.

STR. 2 n-cycles.

Group three

LDM. (regs-1) s + 1 n + 1 i-cycle. Regs is the number of registers loaded. Add 1 s + 1 n-cycles if R15 is loaded.

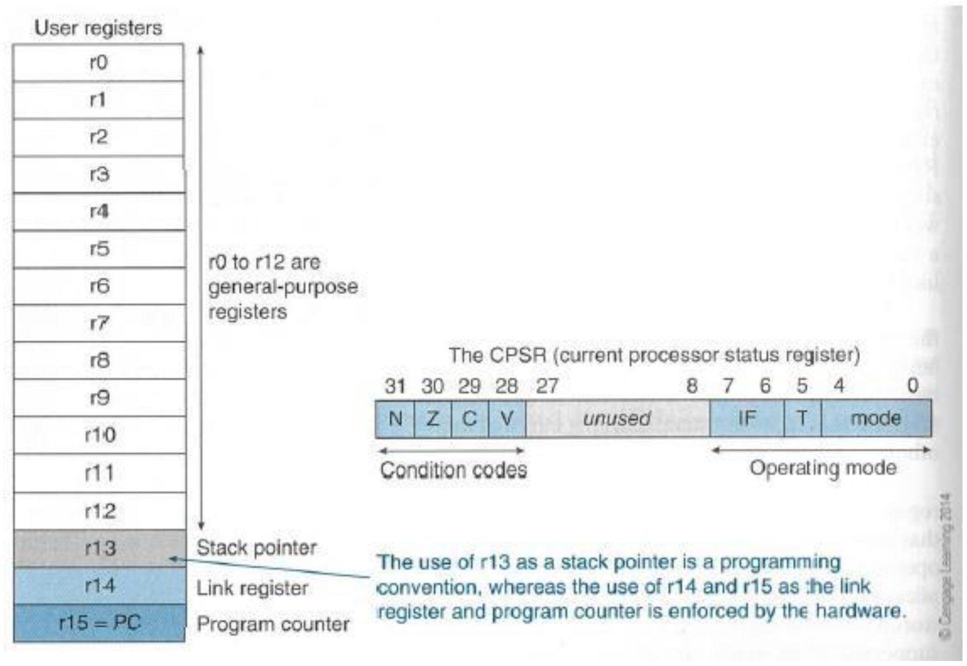
STM. 2 n + (regs-1) s-cycles.

Group four

B, **BL**. 2 s + 1 n-cycles.

Group five

SWI. 2 s + 1 n-cycles.


User registers
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13
r14
r15 = PC
r0 to r12 are general-purpose registers
Stack pointer
Link register
Program counter
The CPSR (current processor status register)
31 30 29 28 27 8 7 6 5 4 0
N Z C V unused IF T mode
Condition codes
Operating mode
The use of r13 as a stack pointer is a programming convention, whereas the use of r14 and r15 as the link register and program counter is enforced by the hardware.
© Cengage Learning 2014

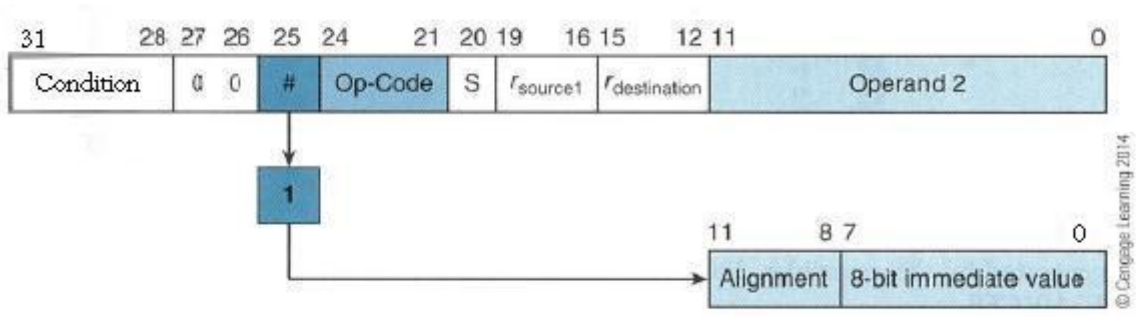
R0 to R12 are the general-purpose registers.
R13 is reserved for the programmer to use it as the stack pointer.
R14 is the link register which stores a subroutine return address.
R15 contains the program counter and is accessible by the programmer.

Conditonion code flags in CPSR:
N - Negative or less than flag
Z - Zero flag
C - Carry or bowrrow or extendedflag
V - Overflow flag
The least-significant 8-bit of the CPSR are the control bits of the system.
The other bits are reserved.

ARM addressing Modes

There are different ways to specify the address of the operands for any given operations such as load, add or branch. The different ways of determining the address of the operands are called addressing modes. In this lab, we are going to explore different addressing modes of ARM processor and learn how all instructions can fit into a single word (32 bits).

Literal Addressing



Register Indirect Addressing

Register indirect addressing means that the location of an operand is held in a register. It is also called indexed addressing or base addressing.

Register indirect addressing mode requires three read operations to access an operand. It is very important because the content of the register containing the pointer to the operand can be modified at runtime. Therefore, the address is a vaiable that allows the access to the data structure like arrays.

- Read the instruction to find the pointer register
- Read the pointer register to find the oprand address
- Read memory at the operand address to find the operand

Register Indirect Addressing with an Offset

ARM supports a memory-addressing mode where the effective address of an operand is computed by adding the content of a register and a literal offset coded into load/store instruction. For example,

```
        Instruction                      Effective Address
-------------------------------------------------------------------------------
        LDR R0, [R1, #20]            R1 + 20    ; loads R0 with the word pointed at by
R1+20
-------------------------------------------------------------------------------
```

ARM's Autoindexing Pre-indexed Addressing Mode

This is used to facilitate the reading of sequential data in structures such as arrays, tables, and vectors. A pointer register is used to hold the base address. An offset can be added to achieve the effective address. For example,

```
        Instruction                      Effective Address
-------------------------------------------------------------------------------
        LDR R0, [R1, #4]!       R1 + 4     ; loads R0 with the word pointed at by R1+4
                                                    ; then update the pointer by adding 4 to
R1
-------------------------------------------------------------------------------
```

ARM's Auto indexing Post-indexing Addressing Mode

This is similar to the above, but it first accesses the operand at the location pointed by the base register, then increments the base register. For example,

```
        Instruction                      Effective Address
-------------------------------------------------------------------------------
        LDR R0, [R1], #4        R1        ; loads R0 with the word pointed at by R1
                                                    ; then update the pointer by adding 4 to
R1
-------------------------------------------------------------------------------
```

Program Counter Relative (PC Relative) Addressing Mode

Register R15 is the program counter. If you use R15 as a pointer register to access operand, the resulting addressing mode is called PC relative addressing. The operand is specified with respect to the current code location. Please look at this example,

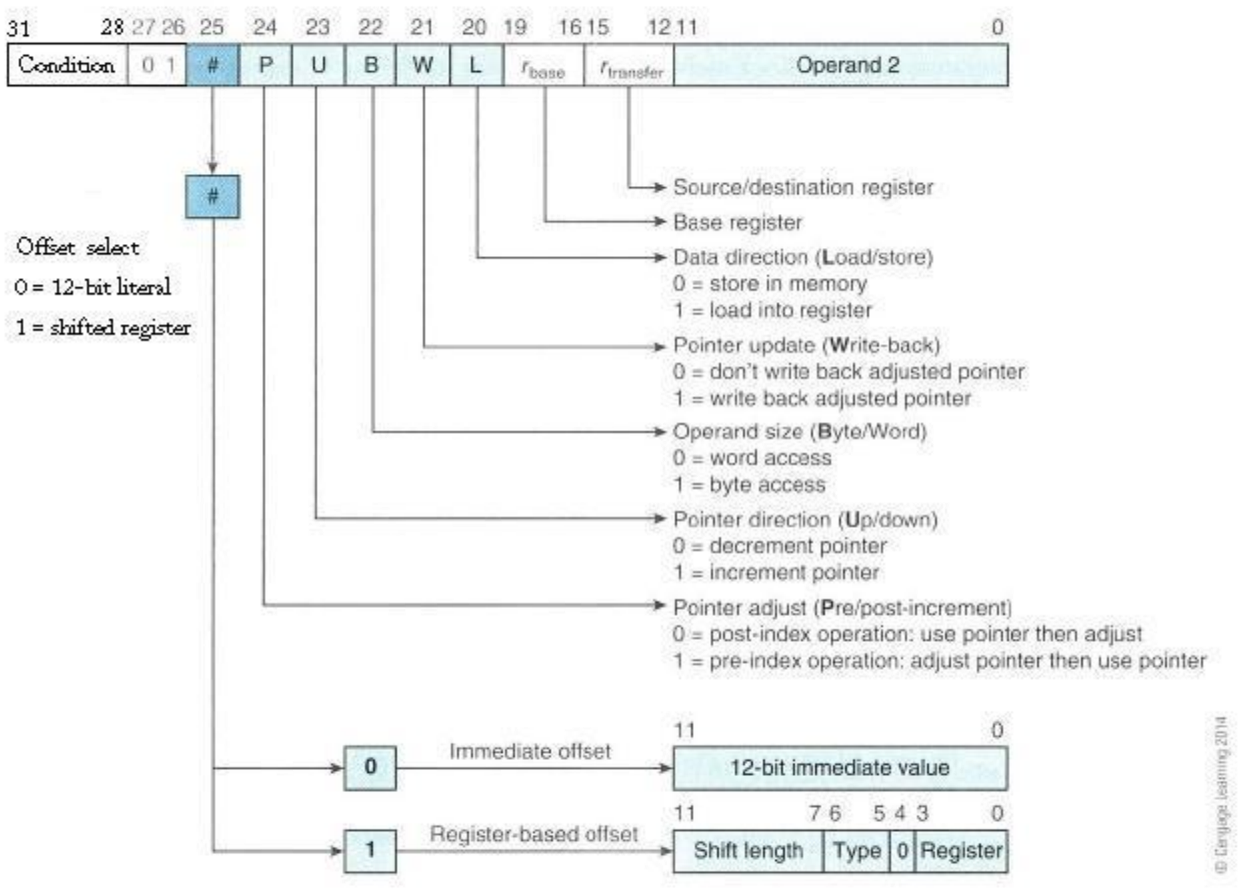
```
        Instruction                      Effective Address
--------------------------------------------------------------------------------
        LDR R0, [R15, #24]      R15 + 24; loads R0 with the word pointed at by R1+24
--------------------------------------------------------------------------------
```

ARM's Load and Store Encoding Format

The following picture illustrates the encoding format of the ARM's load and store instructions, which is included in the lab material for your reference. Memory access operations have a conditional execution field in bit 31, 03, 29, and 28. The load and store instructions can be conditionally executed depending on a condition specified in the instruction. Now look at the following examples:

```
        CMP      R1, R2
------------------------------------------------------------
        LDREQ   R3, [R4]
------------------------------------------------------------
        LDRNE  R3, [R5]
```

Encoding Format of ARM's load and store instructions



UNITIV –PIC CONTROLLER

Salient features of PIC 16F877A Microcontroller

High-Performance RISC CPU:

• Only 35 single-word instructions to learn

• All single-cycle instructions except for program branches, which are two-cycle

• Operating speed: DC – 20 MHz clock input DC – 200 ns instruction cycle

• Up to 8K x 14 words of Flash Program Memory, Up to 368 x 8 bytes of Data Memory (RAM),
Up to 256 x 8 bytes of EEPROM Data Memory

• Pinout compatible to other 28-pin or 40/44-pin PIC16CXXX and PIC16FXXX microcontrollers

Peripheral Features:

• Timer0: 8-bit timer/counter with 8-bit prescaler

• Timer1: 16-bit timer/counter with prescaler, can be incremented during Sleep via external crystal/clock

• Timer2: 8-bit timer/counter with 8-bit period register, prescaler and postscaler

• Two Capture, Compare, PWM modules

- Capture is 16-bit, max. resolution is 12.5 ns
- Compare is 16-bit, max. resolution is 200 ns
- PWM max. resolution is 10-bit

• Synchronous Serial Port (SSP) with SPI™ (Master mode) and I2C™ (Master/Slave)

• Universal Synchronous Asynchronous Receiver Transmitter (USART/SCI) with 9-bit address
Detection

• Parallel Slave Port (PSP) – 8 bits wide with external RD, WR and CS controls (40/44-pin only)

• Brown-out detection circuitry for Brown-out Reset (BOR)

Analog Features:

• 10-bit, up to 8-channel Analog-to-Digital Converter (A/D)

• Brown-out Reset (BOR)

• Analog Comparator module with: - Two analog comparators - Programmable on-chip voltage reference (VREF) module

- Programmable input multiplexing from device inputs and internal voltage reference - Comparator outputs are externally accessible

Special Microcontroller Features:

• 100,000 erase/write cycle Enhanced Flash program memory typical• 1,000,000 erase/write
Cycle Data EEPROM
Memory typical
• Data EEPROM Retention > 40 years

• Self-reprogrammable under software control

• In-Circuit Serial Programming™ (ICSP™) via two pins

• Single-supply 5V In-Circuit Serial Programming

• Watchdog Timer (WDT) with its own on-chip RC oscillator for reliable operation • Programmable code protection

• Power saving Sleep mode • Selectable oscillator options

• In-Circuit Debug (ICD) via two pins

CMOS Technology:

• Low-power, high-speed Flash/EEPROM technology

• Fully static design

• Wide operating voltage range (2.0V to 5.5V)

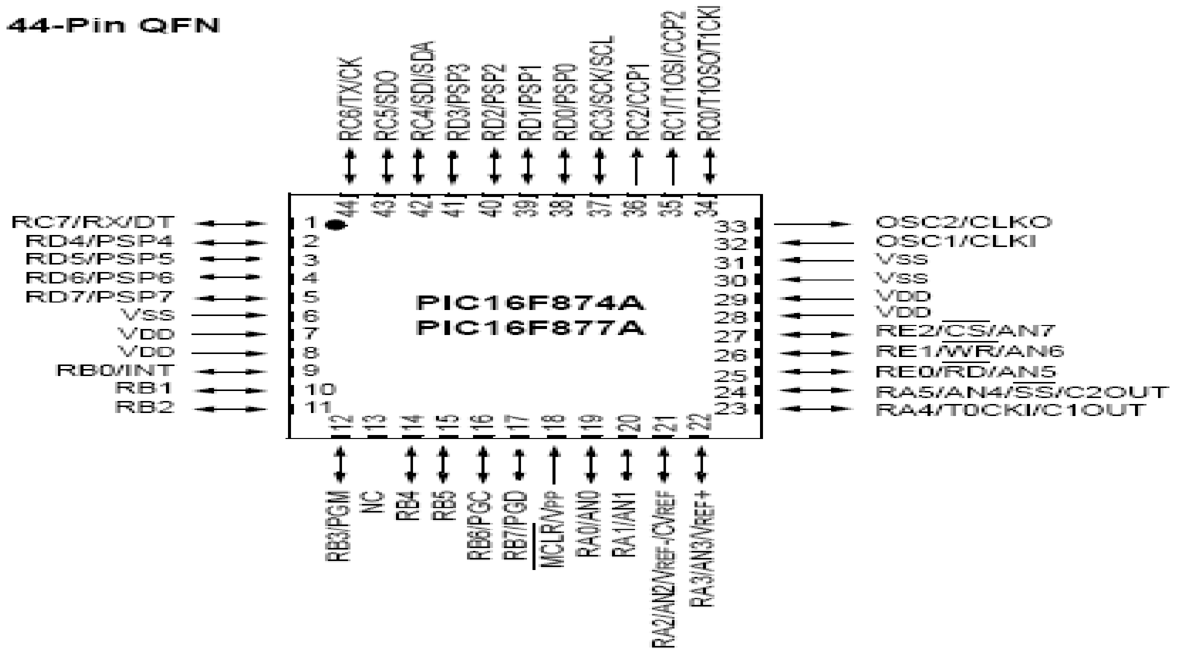
• Commercial and Industrial temperature ranges

• Low-power consumption

TABLE 1-1: PIC16F87XA DEVICE FEATURES

Key Features	PIC16F873A	PIC16F874A	PIC16F876A	PIC16F877A
Operating Frequency	DC – 20 MHz	DC – 20 MHz	DC – 20 MHz	DC – 20 MHz
Resets (and Delays)	POR, BOR (PWRT, OST)	POR, BOR (PWRT, OST)	POR, BOR (PWRT, OST)	POR, BOR (PWRT, OST)
Flash Program Memory (14-bit words)	4K	4K	8K	8K
Data Memory (bytes)	192	192	368	368
EEPROM Data Memory (bytes)	128	128	256	256
Interrupts	14	15	14	15
I/O Ports	Ports A, B, C	Ports A, B, C, D, E	Ports A, B, C	Ports A, B, C, D, E
Timers	3	3	3	3
Capture/Compare/PWM modules	2	2	2	2
Serial Communications	MSSP, USART	MSSP, USART	MSSP, USART	MSSP, USART
Parallel Communications	—	PSP	—	PSP
10-bit Analog-to-Digital Module	5 input channels	8 input channels	5 input channels	8 input channels
Analog Comparators	2	2	2	2
Instruction Set	35 Instructions	35 Instructions	35 Instructions	35 Instructions
Packages	28-pin PDIP 28-pin SOIC 28-pin SSOP 28-pin QFN	40-pin PDIP 44-pin PLCC 44-pin TQFP 44-pin QFN	28-pin PDIP 28-pin SOIC 28-pin SSOP 28-pin QFN	40-pin PDIP 44-pin PLCC 44-pin TQFP 44-pin QFN

PIC 16F877A Pin Diagram and Description



Pin Name	PDIP Pin#	PLCC Pin#	TQFP Pin#	QFN Pin#	I/O/P Type	Buffer Type	Description
OSC1/CLKI	13	14	30	32		ST/CMOS[4]	Oscillator crystal or external clock input.
OSC1					I		Oscillator crystal input or external clock source input. ST buffer when configured in RC mode; otherwise CMOS.
CLKI					I		External clock source input. Always associated with pin function OSC1 (see OSC1/CLKI, OSC2/CLKO pins).
OSC2/CLKO	14	15	31	33		—	Oscillator crystal or clock output.
OSC2					O		Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode.
CLKO					O		In RC mode, OSC2 pin outputs CLKO, which has 1/4 the frequency of OSC1 and denotes the instruction cycle rate.
$\overline{\text{MCLR}}$/VPP	1	2	18	18		ST	Master Clear (input) or programming voltage (output).
$\overline{\text{MCLR}}$					I		Master Clear (Reset) input. This pin is an active low Reset to the device.
VPP					P		Programming voltage input.
							PORTA is a bidirectional I/O port.
RA0/AN0	2	3	19	19		TTL	
RA0					I/O		Digital I/O.
AN0					I		Analog input 0.
RA1/AN1	3	4	20	20		TTL	
RA1					I/O		Digital I/O.
AN1					I		Analog input 1.
RA2/AN2/VREF-/CVREF	4	5	21	21		TTL	
RA2					I/O		Digital I/O.
AN2					I		Analog input 2.
VREF-					I		A/D reference voltage (Low) input.
CVREF					O		Comparator VREF output.
RA3/AN3/VREF+	5	6	22	22		TTL	
RA3					I/O		Digital I/O.
AN3					I		Analog input 3.
VREF+					I		A/D reference voltage (High) input.
RA4/T0CKI/C1OUT	6	7	23	23		ST	
RA4					I/O		Digital I/O – Open-drain when configured as output.
T0CKI					I		Timer0 external clock input.
C1OUT					O		Comparator 1 output.
RA5/AN4/$\overline{\text{SS}}$/C2OUT	7	8	24	24		TTL	
RA5					I/O		Digital I/O.
AN4					I		Analog input 4.
$\overline{\text{SS}}$					I		SPI slave select input.
C2OUT					O		Comparator 2 output.

Legend: I = input O = output I/O = input/output P = power
— = Not used TTL = TTL input ST = Schmitt Trigger input

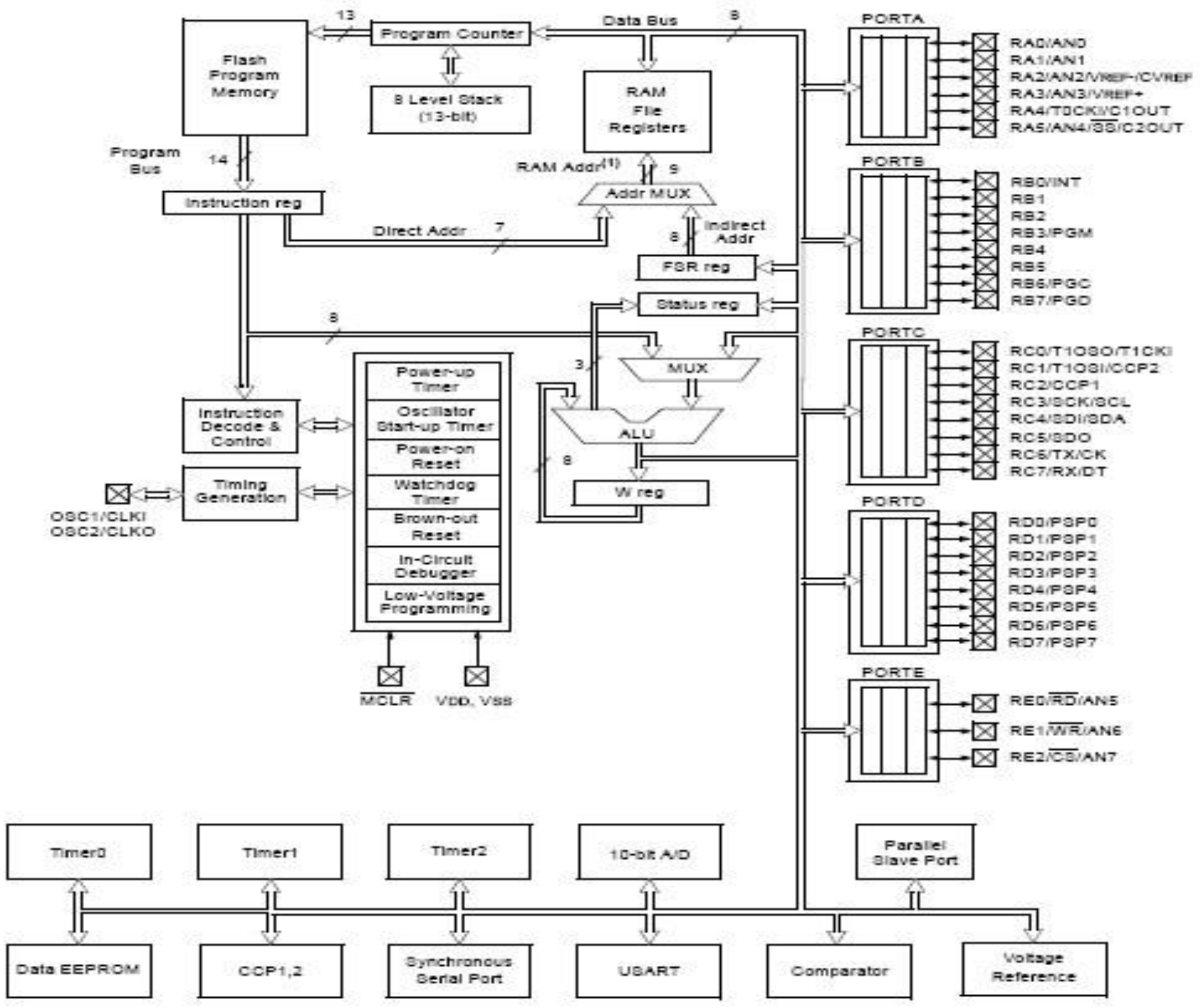
Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.
2: This buffer is a Schmitt Trigger input when used in Serial Programming mode.
3: This buffer is a Schmitt Trigger input when configured in RC Oscillator mode and a CMOS input otherwise.

Pin Name	PDIP Pin#	PLCC Pin#	TQFP Pin#	QFN Pin#	I/O/P Type	Buffer Type	Description
							PORTB is a bidirectional I/O port. PORTB can be software programmed for internal weak pull-up on all inputs.
RB0/INT	33	36	8	9		TTL/ST(1)	
RB0					I/O		Digital I/O.
INT					I		External interrupt.
RB1	34	37	9	10	I/O	TTL	Digital I/O.
RB2	35	38	10	11	I/O	TTL	Digital I/O.
RB3/PGM	36	39	11	12		TTL	
RB3					I/O		Digital I/O.
PGM					I		Low-voltage ICSP programming enable pin.
RB4	37	41	14	14	I/O	TTL	Digital I/O.
RB5	38	42	15	15	I/O	TTL	Digital I/O.
RB6/PGC	39	43	16	16		TTL/ST(2)	
RB6					I/O		Digital I/O.
PGC					I		In-circuit debugger and ICSP programming clock.
RB7/PGD	40	44	17	17		TTL/ST(2)	
RB7					I/O		Digital I/O.
PGD					I/O		In-circuit debugger and ICSP programming data.

Legend: I = input O = output I/O = input/output P = power
— = Not used TTL = TTL input ST = Schmitt Trigger input

Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.
2: This buffer is a Schmitt Trigger input when used in Serial Programming mode.
3: This buffer is a Schmitt Trigger input when configured in RC Oscillator mode and a CMOS input otherwise.

PIC 16F877A Block Diagram (Architecture)



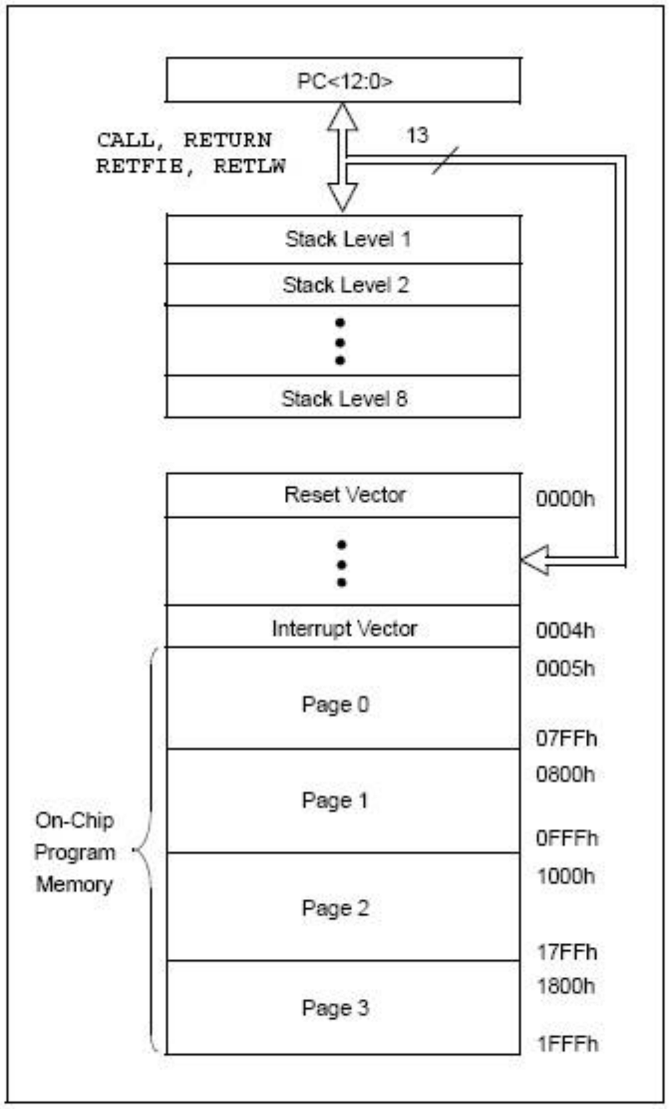
Device	Program Flash	Data Memory	Data EEPROM
PIC16F874A	4K words	192 Bytes	128 Bytes
PIC16F877A	8K words	368 Bytes	256 Bytes

Note 1: Higher order bits are from the Status register.

Memory organization

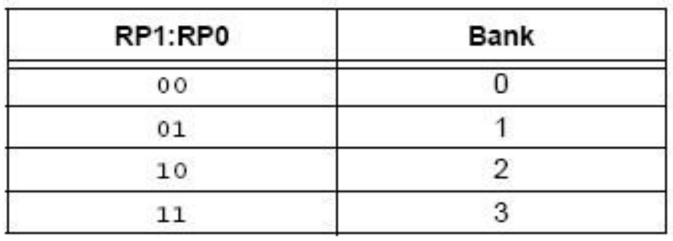
The program memory and data memory have separate buses so that concurrent access can occur
Program memory map

The PIC16F87XA devices have a 13-bit program counter capable of addressing an 8K word x 14 bit.program memory space. The PIC16F876A/877Adevices have 8K words x 14 bits of Flash programmemory. The Reset vector is at 0000h and the interrupt vector is at 0004h.



Data Memory Organization

The data memory is partitioned into multiple banks which contain the General Purpose Registers and the Special Function Registers. Bits RP1 (Status<6>) and RP0 (Status<5>) are the bank select bits. Each bank extends up to 7Fh (128 bytes). The lower locations of each bank are reserved for the Special Function Registers. Above the Special Function Registers are General Purpose Registers, implemented as static RAM. All implemented banks contain Special Function Registers.



RP1:RP0	Bank
00	0
01	1
10	2
11	3

Data eeprom and flash program memory

The data EEPROM and Flash program memory is readable and writable during normal operation (over the full VDD range). This memory is not directly mapped in the register file space. Instead, it is indirectly addressed through the Special Function Registers. There are six SFRs used to read and write this memory:

- EECON1
- EECON2
- EEDATA
- EEDATH
- EEADR
- EEADRH

GENERAL PURPOSE REGISTER FILE
The register file can be accessed either directly, or indirectly, through the File Select Register (FSR).

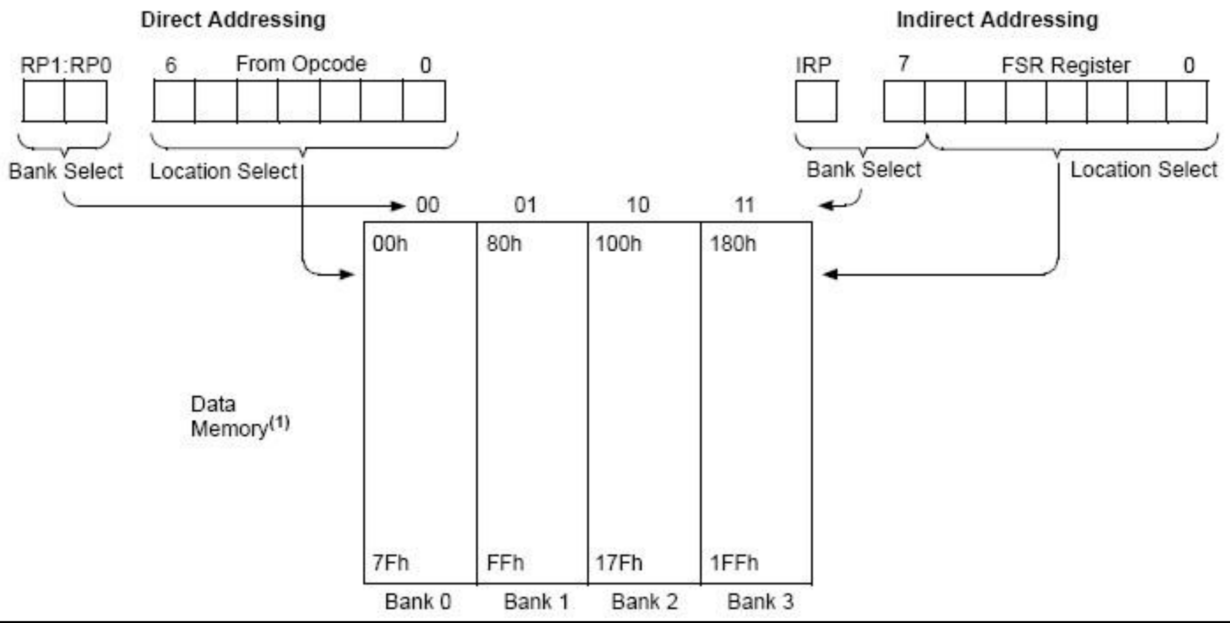
SPECIAL FUNCTION REGISTERS

The Special Function Registers are registers used by the CPU and peripheral modules for controlling the desired operation of the device. These registers are implemented as static RAM. The Special Function Registers can be classified into two sets: core (CPU) and peripheral. Some examples are Status Register the Status register contains the arithmetic status of the ALU, the Reset status and the bank select bits for data memory.

DIRECT/INDIRECT ADDRESSING

Indirect Addressing, INDF and FSR Registers

The INDF register is not a physical register. Addressing the INDF register will cause indirect addressing.Indirect addressing is possible by using the INDF register.Any instruction using the INDF register actually accesses the register pointed to by the File Select Register, FSR. Reading the INDF register itself, indirectly (FSR = 0) will read 00h. Writing to the INDF register indirectly results in a no operation (although status bits may be affected). An effective 9-bit address is obtained by concatenating the 8-bit FSR register and the IRP bit

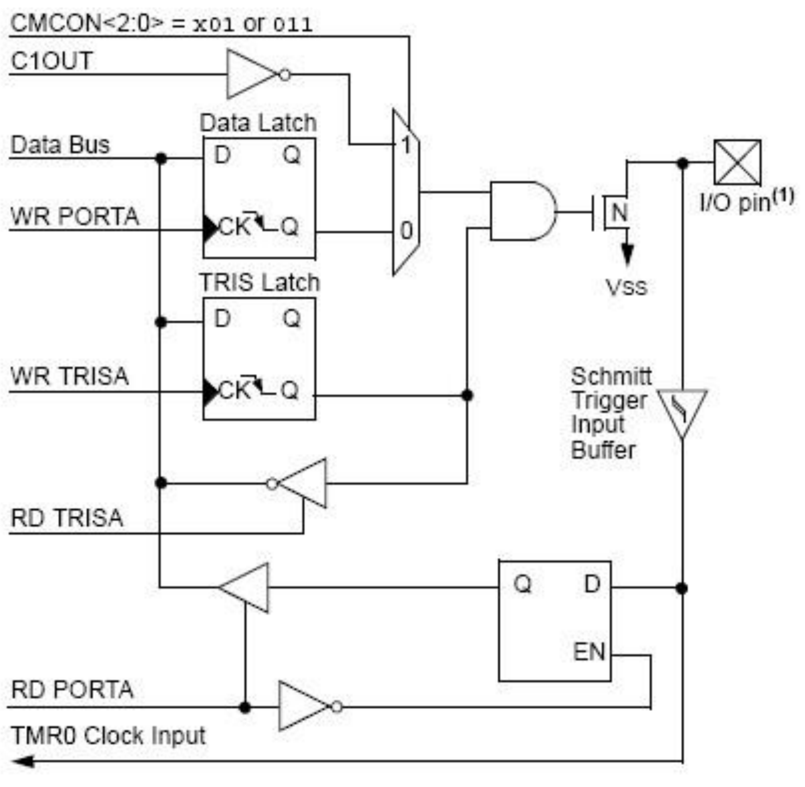


<u>I/O PORTS</u>

Some pins for these I/O ports are multiplexed with an alternate function for the peripheral features on the device. In general, when a peripheral is enabled, that pin may not be used as a general purpose I/O pin.

PORTA and the TRISA Register

PORTA is a 6-bit wide, bidirectional port. The corresponding data direction register is TRISA. Setting a TRISA bit (= 1) will make the corresponding PORTA pin an input (i.e., put the corresponding output driver in a High-Impedance mode). Clearing a TRISA bit (= 0) will make the corresponding PORTA pin an output (i.e., put the contents of the output latch on the selected pin).Reading the PORTA register reads the status of the pins, whereas writing to it will write to the port latch. All write operations are read-modify-write operations. Therefore, a write to a port implies that the port pins are read; the value is modified and then written to the port data latch. The TRISA register controls the direction of the port pins even when they are being used as analog inputs.



Note 1: I/O pin has protection diodes to Vss only.

Similarly for other ports : PORTB and the TRISB Register ,PORTC and the TRISC Register, PORTD and TRISD Registers ,PORTE and TRISE Register.

<u>TIMER0 MODULE</u>

The Timer0 module timer/counter has the following features:

• 8-bit timer/counter

• Readable and writable

• 8-bit software programmable prescaler

• Internal or external clock select

• Interrupt on overflow from FFh to 00h

• Edge select for external clock

The Timer1 module is a 16-bit timer/counter consisting of two 8-bit registers (TMR1H and TMR1L) which are readable and writable. Timer1 can operate in one of two modes:

• As a Timer

• As a Counter

Timer2 is an 8-bit timer with a prescaler and a postscaler. It can be used as the PWM time base for the PWM mode of the CCP module(s). The TMR2 register is readable and writable and is cleared on any device Reset.

CAPTURE/COMPARE/PWM MODULES

Each Capture/Compare/PWM (CCP) module contains
a 16-bit register which can operate as a:

- 16-bit Capture register
- 16-bit Compare register
- PWM Master/Slave Duty Cycle register

CCP1 Module:
Capture/Compare/PWM Register 1 (CCPR1) is comprised of two 8-bit registers: CCPR1L (low byte) and CCPR1H (high byte). The CCP1CON register controls the operation of CCP1. The special event trigger is generated by a compare match and will reset Timer1.

CCP2 Module:
Capture/Compare/PWM Register 2 (CCPR2) is comprised of two 8-bit registers: CCPR2L (low byte) and CCPR2H (high byte). The CCP2CON register controls the operation of CCP2. The special event trigger is generated by a compare match and will reset Timer1 and start an A/D conversion (if the A/D module is enabled).

Capture Mode

In Capture mode, CCPR1H:CCPR1L captures the 16-bit value of the TMR1 register when an event occurs on pin RC2/CCP1. An event is defined as one of the following:

- Every falling edge
- Every rising edge
- Every 4th rising edge
- Every 16th rising edge

The type of event is configured by control bits, CCP1M3:CCP1M0 (CCPxCON<3:0>).

Compare Mode

In Compare mode, the 16-bit CCPR1 register value is constantly compared against the TMR1 register pair value. When a match occurs, the RC2/CCP1 pin is:

• Driven high

• Driven low

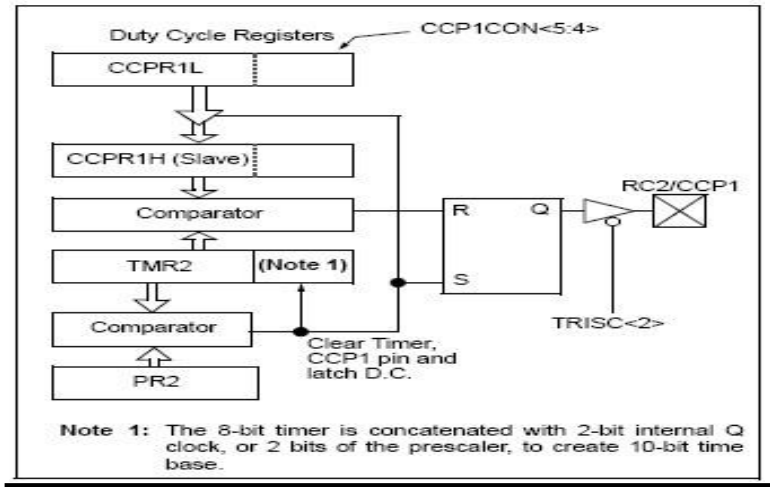
• Remains unchanged

The action on the pin is based on the value of control bits, CCP1M3:CCP1M0 (CCP1CON<3:0>).

PWM Mode (PWM)

In Pulse Width Modulation mode, the CCPx pin produces up to a 10-bit resolution PWM output.

PWM BLOCK DIAGRAM



Note 1: The 8-bit timer is concatenated with 2-bit internal Q clock, or 2 bits of the prescaler, to create 10-bit time base.

Master SSP (MSSP) Module

The Master Synchronous Serial Port (MSSP) module is a serial interface, useful for communicating with other peripheral or microcontroller devices. These peripheral devices may be serial EEPROMs, shift registers,display drivers, A/D converters, etc. The MSSP module can operate in one of two modes:

• Serial Peripheral Interface (SPI)

• Inter-Integrated Circuit (I2C)

- Full Master mode

- Slave mode (with general address call)

The I2C interface supports the following modes in hardware:

• Master mode

• Multi-Master mode

• Slave mode

COMPARATOR MODULE

The comparator module contains two analog comparators.The inputs to the comparators are multiplexed with I/O port pins RA0 through RA3, while the outputs are multiplexed to pins RA4 and RA5.

Reset

The PIC16F87XA differentiates between various kinds
of Reset:

• Power-on Reset (POR)
• MCLR Reset during normal operation
• MCLR Reset during Sleep
• WDT Reset (during normal operation)
• WDT Wake-up (during Sleep)
• Brown-out Reset (BOR)0

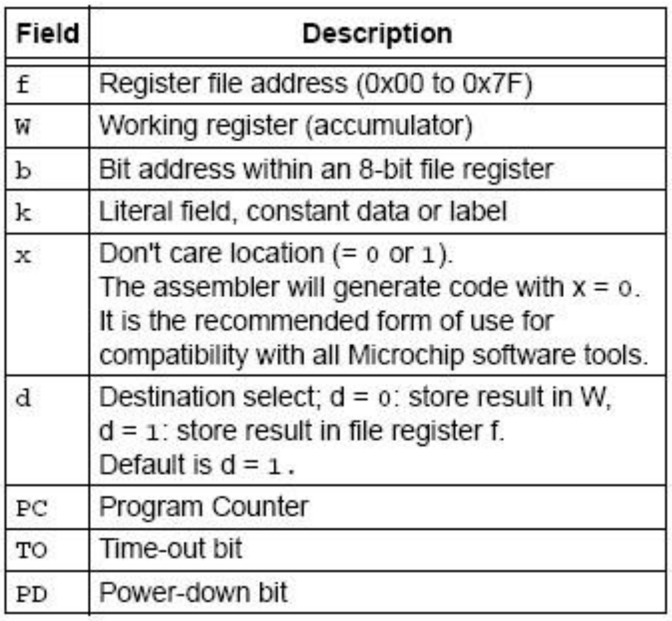
INSTRUCTION SET SUMMARY

The PIC16 instruction set is highly orthogonal and is comprised of three basic categories:

• **Byte-oriented** operations
• **Bit-oriented** operations
• **Literal and control** operations.

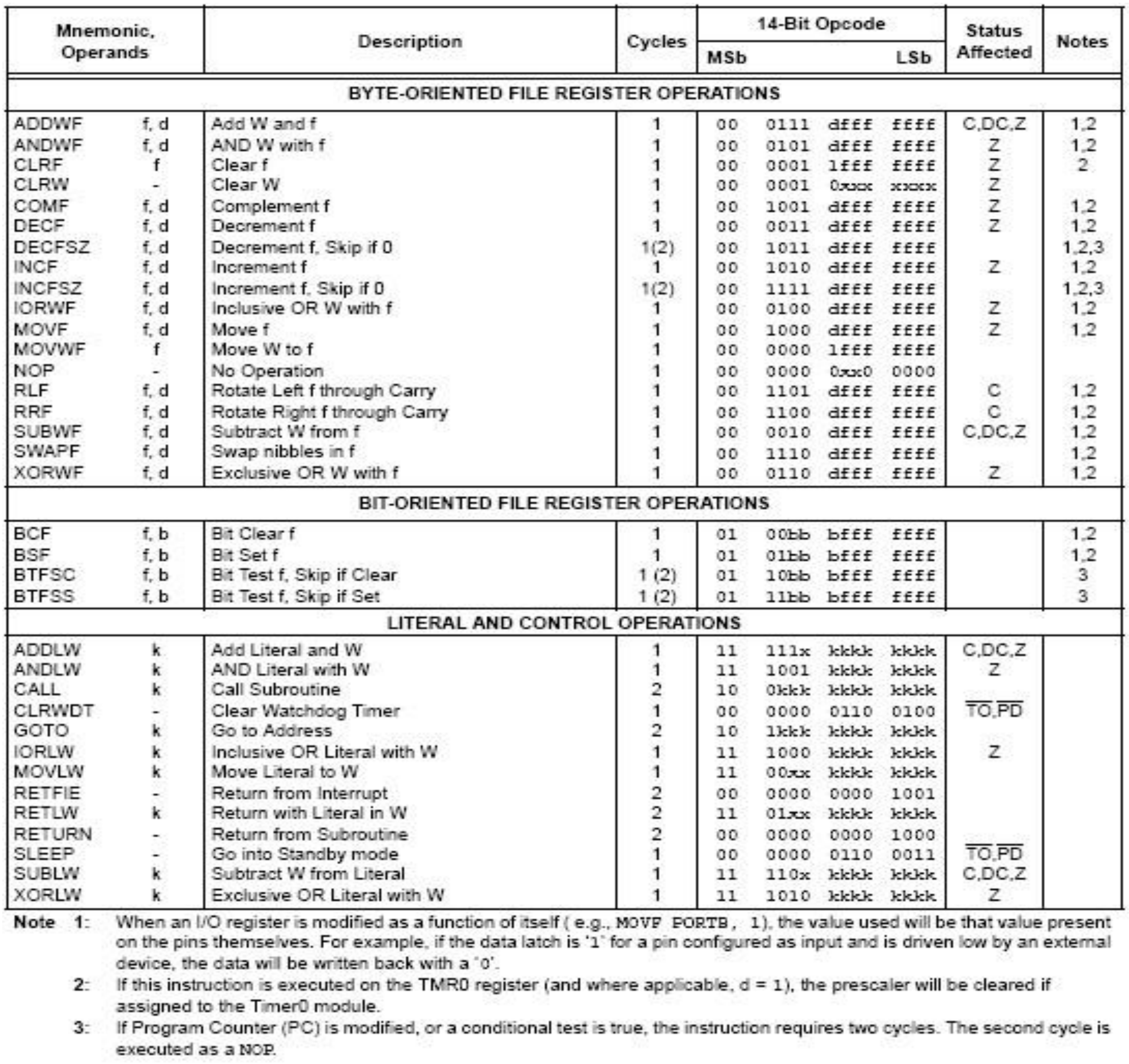
Each PIC16 instruction is a 14-bit word divided into an **opcode** which specifies the instruction type and one or more **operands** which further specify the operation of the instruction.

OPCODE FIELD DESCRIPTIONS



Field	Description
f	Register file address (0x00 to 0x7F)
W	Working register (accumulator)
b	Bit address within an 8-bit file register
k	Literal field, constant data or label
x	Don't care location (= 0 or 1). The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.
d	Destination select; d = 0: store result in W, d = 1: store result in file register f. Default is d = 1.
PC	Program Counter
TO	Time-out bit
PD	Power-down bit

PIC16F87XA INSTRUCTION SET



Mnemonic, Operands		Description	Cycles	14-Bit Opcode				Status Affected	Notes
				MSb			LSb		
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	lfff	ffff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	lfff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add Literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND Literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call Subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to Address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR Literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move Literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	-	Return from Interrupt	2	00	0000	0000	1001		
RETLW	k	Return with Literal in W	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	-	Go into Standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from Literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR Literal with W	1	11	1010	kkkk	kkkk	Z	

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 module.

3: If Program Counter (PC) is modified, or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

ADDLW	**Add Literal and W**
Syntax:	[*label*] ADDLW k
Operands:	$0 \le k \le 255$
Operation:	$(W) + k \rightarrow (W)$
Status Affected:	C, DC, Z
Description:	The contents of the W register are added to the eight-bit literal 'k' and the result is placed in the W register.

ADDWF	**Add W and f**
Syntax:	[*label*] ADDWF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	$(W) + (f) \rightarrow$ (destination)
Status Affected:	C, DC, Z
Description:	Add the contents of the W register with register 'f'. If 'd' is '0', the result is stored in the W register. If 'd' is '1', the result is stored back in register 'f'.

ANDLW	**AND Literal with W**
Syntax:	[*label*] ANDLW k
Operands:	$0 \le k \le 255$
Operation:	(W) .AND. (k) $\rightarrow$ (W)
Status Affected:	Z
Description:	The contents of W register are AND'ed with the eight-bit literal 'k'. The result is placed in the W register.

ANDWF	**AND W with f**
Syntax:	[*label*] ANDWF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(W) .AND. (f) $\rightarrow$ (destination)
Status Affected:	Z
Description:	AND the W register with register 'f'. If 'd' is '0', the result is stored in the W register. If 'd' is '1', the result is stored back in register 'f'.

BCF	**Bit Clear f**
Syntax:	[*label*] BCF f,b
Operands:	$0 \le f \le 127$ $0 \le b \le 7$
Operation:	$0 \rightarrow$ (f<b>)
Status Affected:	None
Description:	Bit 'b' in register 'f' is cleared.

BSF	**Bit Set f**
Syntax:	[*label*] BSF f,b
Operands:	$0 \le f \le 127$ $0 \le b \le 7$
Operation:	$1 \rightarrow$ (f<b>)
Status Affected:	None
Description:	Bit 'b' in register 'f' is set.

BTFSS	**Bit Test f, Skip if Set**
Syntax:	[*label*] BTFSS f,b
Operands:	$0 \le f \le 127$ $0 \le b < 7$
Operation:	skip if (f<b>) = 1
Status Affected:	None
Description:	If bit 'b' in register 'f' is '0', the next instruction is executed. If bit 'b' is '1', then the next instruction is discarded and a NOP is executed instead, making this a 2 TCY instruction.

BTFSC	**Bit Test, Skip if Clear**
Syntax:	[*label*] BTFSC f,b
Operands:	$0 \le f \le 127$ $0 \le b \le 7$
Operation:	skip if (f<b>) = 0
Status Affected:	None
Description:	If bit 'b' in register 'f' is '1', the next instruction is executed. If bit 'b' in register 'f' is '0', the next instruction is discarded and a NOP is executed instead, making this a 2 TCY instruction.

CALL	Call Subroutine
Syntax:	[*label*] CALL k
Operands:	$0 \le k \le 2047$
Operation:	(PC)+ 1$\rightarrow$ TOS, k $\rightarrow$ PC<10:0>, (PCLATH<4:3>) $\rightarrow$ PC<12:11>
Status Affected:	None
Description:	Call Subroutine. First, return address (PC+1) is pushed onto the stack. The eleven-bit immediate address is loaded into PC bits <10:0>. The upper bits of the PC are loaded from PCLATH. `CALL` is a two-cycle instruction.

CLRF	Clear f
Syntax:	[*label*] CLRF f
Operands:	$0 \le f \le 127$
Operation:	00h $\rightarrow$ (f) 1 $\rightarrow$ Z
Status Affected:	Z
Description:	The contents of register 'f' are cleared and the Z bit is set.

CLRW	Clear W
Syntax:	[*label*] CLRW
Operands:	None
Operation:	00h $\rightarrow$ (W) 1 $\rightarrow$ Z
Status Affected:	Z
Description:	W register is cleared. Zero bit (Z) is set.

CLRWDT	Clear Watchdog Timer
Syntax:	[*label*] CLRWDT
Operands:	None
Operation:	00h $\rightarrow$ WDT 0 $\rightarrow$ WDT prescaler, 1 $\rightarrow$ $\overline{\text{TO}}$ 1 $\rightarrow$ $\overline{\text{PD}}$
Status Affected:	$\overline{\text{TO}}$, $\overline{\text{PD}}$
Description:	`CLRWDT` instruction resets the Watchdog Timer. It also resets the prescaler of the WDT. Status bits, $\overline{\text{TO}}$ and $\overline{\text{PD}}$, are set.

COMF	Complement f
Syntax:	[*label*] COMF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	$(\bar{f}) \rightarrow$ (destination)
Status Affected:	Z
Description:	The contents of register 'f' are complemented. If 'd' is '`0`', the result is stored in W. If 'd' is '`1`', the result is stored back in register 'f'.

DECF	Decrement f
Syntax:	[*label*] DECF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f) - 1 $\rightarrow$ (destination)
Status Affected:	Z
Description:	Decrement register 'f'. If 'd' is '`0`', the result is stored in the W register. If 'd' is '`1`', the result is stored back in register 'f'.

DECFSZ	**Decrement f, Skip if 0**
Syntax:	[*label*] DECFSZ f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f) - 1 → (destination); skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are decremented. If 'd' is '0', the result is placed in the W register. If 'd' is '1', the result is placed back in register 'f'. If the result is '1', the next instruction is executed. If the result is '0', then a NOP is executed instead, making it a 2 TCY instruction.

GOTO	**Unconditional Branch**
Syntax:	[*label*] GOTO k
Operands:	$0 \le k \le 2047$
Operation:	k → PC<10:0> PCLATH<4:3> → PC<12:11>
Status Affected:	None
Description:	GOTO is an unconditional branch. The eleven-bit immediate value is loaded into PC bits <10:0>. The upper bits of PC are loaded from PCLATH<4:3>. GOTO is a two-cycle instruction.

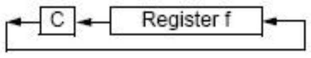
INCF	**Increment f**
Syntax:	[*label*] INCF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f) + 1 → (destination)
Status Affected:	Z
Description:	The contents of register 'f' are incremented. If 'd' is '0', the result is placed in the W register. If 'd' is '1', the result is placed back in register 'f'.

INCFSZ	**Increment f, Skip if 0**
Syntax:	[*label*] INCFSZ f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f) + 1 → (destination), skip if result = 0
Status Affected:	None
Description:	The contents of register 'f' are incremented. If 'd' is '0', the result is placed in the W register. If 'd' is '1', the result is placed back in register 'f'. If the result is '1', the next instruction is executed. If the result is '0', a NOP is executed instead, making it a 2 TCY instruction.

IORLW	**Inclusive OR Literal with W**
Syntax:	[*label*] IORLW k
Operands:	$0 \le k \le 255$
Operation:	(W) .OR. k → (W)
Status Affected:	Z
Description:	The contents of the W register are OR'ed with the eight-bit literal 'k'. The result is placed in the W register.

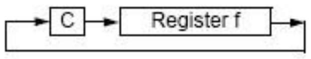
IORWF	**Inclusive OR W with f**
Syntax:	[*label*] IORWF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(W) .OR. (f) → (destination)
Status Affected:	Z
Description:	Inclusive OR the W register with register 'f'. If 'd' is '0', the result is placed in the W register. If 'd' is '1', the result is placed back in register 'f'.

RLF	**Rotate Left f through Carry**
Syntax:	[*label*] RLF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	See description below
Status Affected:	C
Description:	The contents of register 'f' are rotated one bit to the left through the Carry flag. If 'd' is '0', the result is placed in the W register. If 'd' is '1', the result is stored back in register 'f'.



RETURN	**Return from Subroutine**
Syntax:	[*label*] RETURN
Operands:	None
Operation:	TOS $\rightarrow$ PC
Status Affected:	None
Description:	Return from subroutine. The stack is POPed and the top of the stack (TOS) is loaded into the program counter. This is a two-cycle instruction.

RRF	**Rotate Right f through Carry**
Syntax:	[*label*] RRF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	See description below
Status Affected:	C
Description:	The contents of register 'f' are rotated one bit to the right through the Carry flag. If 'd' is '0', the result is placed in the W register. If 'd' is '1', the result is placed back in register 'f'.



SLEEP	
Syntax:	[*label*] SLEEP
Operands:	None
Operation:	00h $\rightarrow$ WDT, 0 $\rightarrow$ WDT prescaler, 1 $\rightarrow$ $\overline{TO}$, 0 $\rightarrow$ $\overline{PD}$
Status Affected:	$\overline{TO}$, $\overline{PD}$
Description:	The power-down status bit, $\overline{PD}$, is cleared. Time-out status bit, $\overline{TO}$, is set. Watchdog Timer and its prescaler are cleared. The processor is put into Sleep mode with the oscillator stopped.

SUBLW	**Subtract W from Literal**
Syntax:	[*label*] SUBLW k
Operands:	$0 \le k \le 255$
Operation:	k - (W) $\rightarrow$ (W)
Status Affected:	C, DC, Z
Description:	The W register is subtracted (2's complement method) from the eight-bit literal 'k'. The result is placed in the W register.

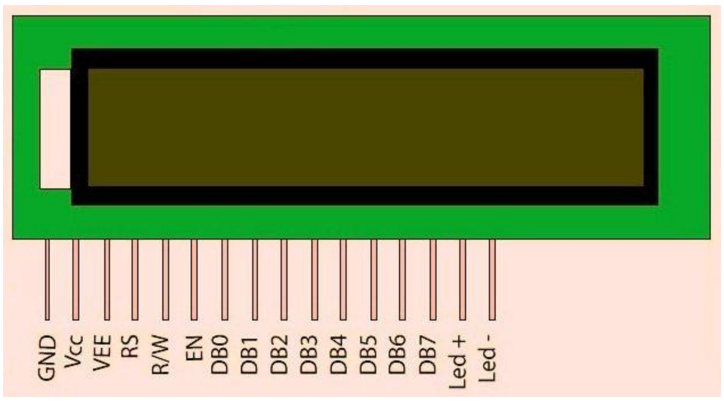
SUBWF	**Subtract W from f**
Syntax:	[*label*] SUBWF f,d
Operands:	$0 \le f \le 127$ $d \in [0,1]$
Operation:	(f) - (W) $\rightarrow$ (destination)
Status Affected:	C, DC, Z
Description:	Subtract (2's complement method) W register from register 'f'. If 'd' is '0', the result is stored in the W register. If 'd' is '1', the result is stored back in register 'f'.

UNIT-V INTERFACING CASE STUDY

Interfacing LCD with PIC Microcontroller

16×2 Character LCD is a very basic LCD module which is commonly used in electronics projects and products. It contains 2 rows that can display 16 characters. Each character is displayed using 5×8 or 5×10 dot matrix. It can be easily interfaced with a microcontroller. In this tutorial we will see how to write data to an LCD with PIC Microcontroller using Hi-Tech C Compiler. Hi-Tech C has no built in LCD libraries so we require the hardware knowledge of LCD to control it. Commonly **used**

LCD Displays uses HD44780 compliant controllers.



16×2 LCD Pin Diagram

This is the pin diagram of a 16×2 Character LCD display. As in all devices it also has two inputs to give power Vcc and GND. Voltage at VEE determines the Contrast of the display. A 10K potentiometer whose fixed ends are connected to Vcc, GND and variable end is connected to VEE can be used to adjust contrast. A microcontroller needs to send two informations to operate this LCD module, Data and Commands. Data represents the ASCII value (8 bits) of the character to be displayed and Command determines the other operations of LCD such as position to be displayed. Data and Commands are send through the same data lines, which are multiplexed using the RS (Register Select) input of LCD. When it is HIGH, LCD takes it as data to be displayed and when it is LOW, LCD takes it as a command. Data Strobe is given using E (Enable) input of the LCD. When the E (Enable) is HIGH, LCD takes it as valid data or command. The input signal R/W (Read or Write) determines whether data is written to or read from the LCD. In normal cases we need only writing hence it is tied to GROUND in circuits shown below.

The interface between this LCD and Microcontroller can be 8 bit or 4 bit and the difference between them is in how the data or commands are send to LCD. In the 8 bit mode, 8 bit data and commands are send through the data lines DB0 – DB7 and data strobe is given through E input of the LCD. But 4 bit mode uses only 4 data lines. In this 8 bit data and commands are splitted into 2 parts (4 bits each) and are sent sequentially through data lines DB4 – DB7 with its own data strobe through E input. The idea of 4 bit communication is introduced to save pins of a microcontroller. You may think that 4 bit mode will be slower than 8 bit. But the speed difference is only minimal. As LCDs are slow speed devices, the tiny speed difference between these modes is not significant. Just remember that microcontroller is operating at high speed in the range of MHz and we are viewing LCD with our eyes. Due to Persistence of Vision of our eyes we will not even feel the speed difference.

Hope that you got rough idea about how this LCD Module works. Actually you need to read the datasheet of HD44780 LCD driver used in this LCD Module to write a Hi-Tech C program for PIC. But we solved this problem by creating a header file lcd.h which includes all the commonly used functions. Just include it and enjoy.

FUNCTIONS IN LCD.H

Lcd8_Init() & Lcd4_Init() : These functions will initialize the LCD Module connected to the following defined pins in 8 bit and 4 bit mode respectively.

8 Bit Mode :

```
#define RS RB6
#define EN RB7
#define D0 RC0
#define D1 RC1
#define D2 RC2
#define D3 RC3
#define D4 RC4
#define D5 RC5
#define D6 RC6
#define D7 RC7
```

These connections must be defined for the working of LCD library.

Lcd8_Clear() & Lcd4_Clear() : Calling these functions will clear the LCD Display when interfaced in 8 Bit and 4 Bit mode respectively.
Lcd8_Set_Cursor() & Lcd4_Set_Cursor() : These functions set the row and column of the cursor on the LCD Screen. By using this we can change the position of the character being displayed by the following functions.

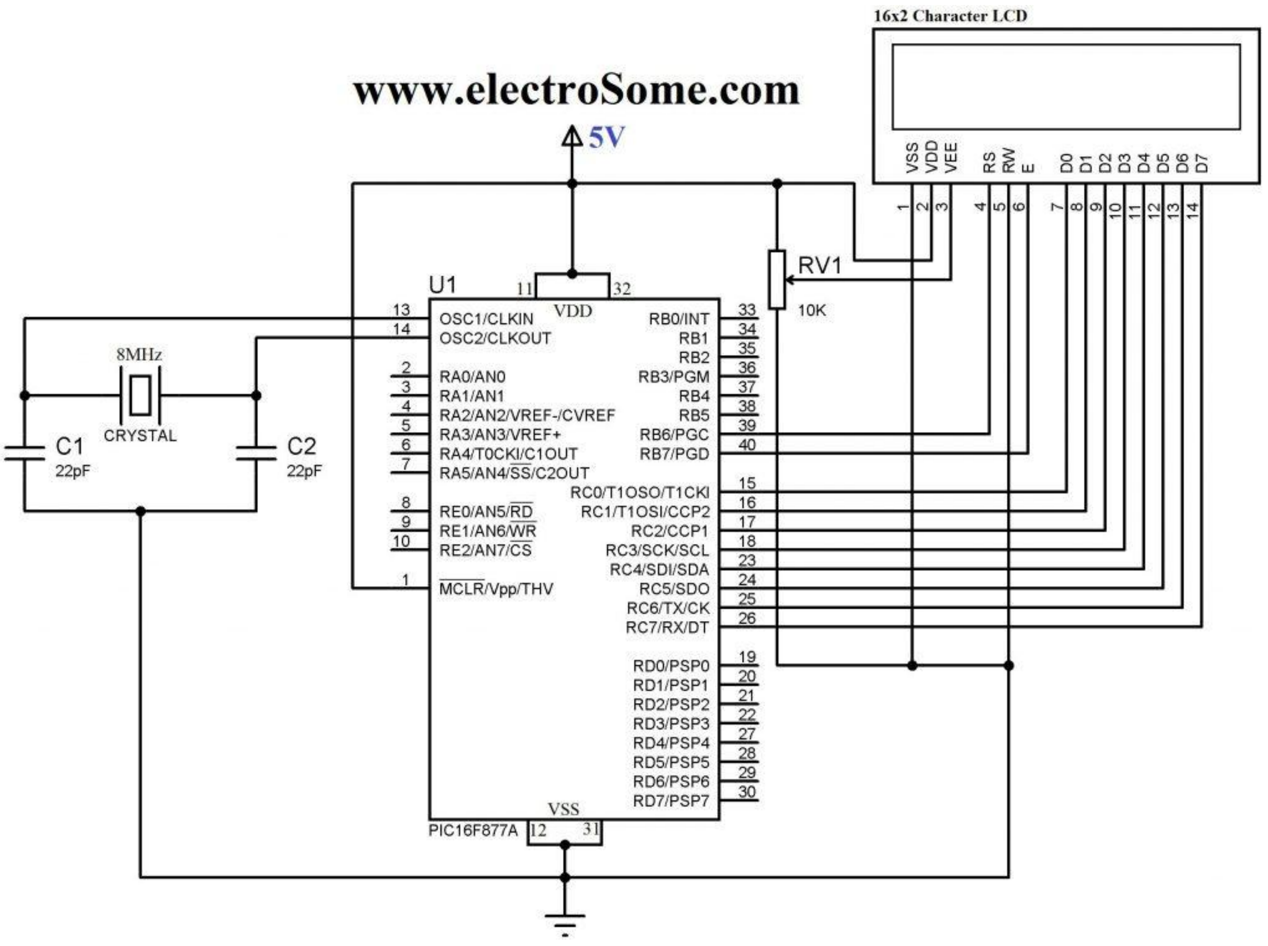
Lcd8_Write_Char() & Lcd4_Write_Char() : These functions will write a character to the LCD Screen when interfaced through 8 Bit and 4 Bit mode respectively.

Lcd8_Write_String() & Lcd4_Write_String() : These functions are used to write strings to the LCD Screen.

Lcd8_Shift_Left() & Lcd4_Shift_Left() : These functions are used to shift the content on the LCD Display left without changing the data in the display RAM.

Lcd8_Shift_Right() & Lcd4_Shift_Right() : Similar to above functions, these are used to shift the content on the LCD Display right without changing the data in the display RAM.

8 Bit Mode-Circuit Diagram



Interfacing LCD with PIC Microcontroller – 8 Bit Mode

```
#include<htc.h>

#include<pic.h>


#define RS RB6

#define EN RB7

#define D0 RC0

#define D1 RC1

#define D2 RC2

#define D3 RC3

#define D4 RC4

#define D5 RC5

#define D6 RC6

#define D7 RC7
```

}

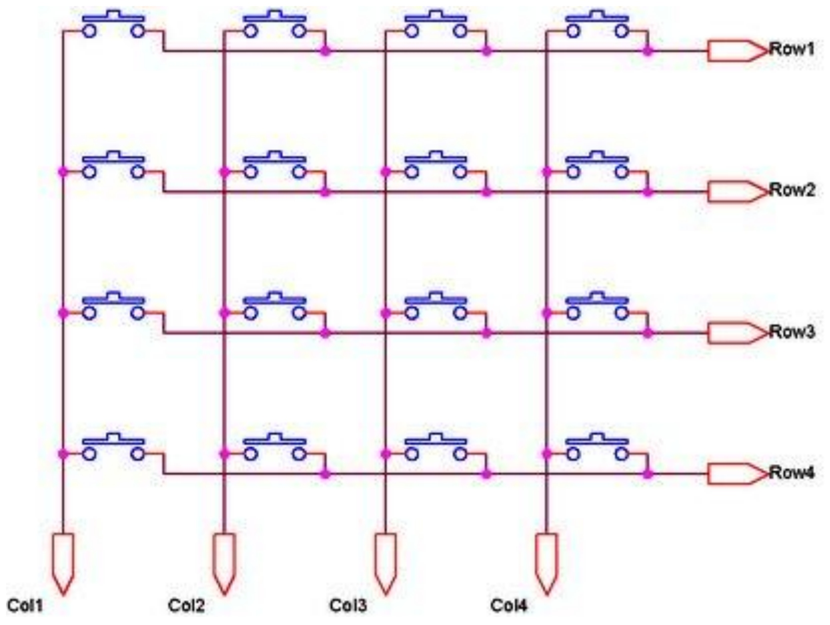
```
#define _XTAL_FREQ 8000000

#include "lcd.h"

void main()
{
  int i;
  TRISB = 0x00;
  TRISC = 0x00;
  Lcd8_Init();
  while(1)
  {
    Lcd8_Set_Cursor(1,1);
    Lcd8_Write_String("electroSome LCD Hello World");
    for(i=0;i<15;i++)
    {
      __delay_ms(1000);
      Lcd8_Shift_Left();
    }
    for(i=0;i<15;i++)
    {
      __delay_ms(1000);
      Lcd8_Shift_Right();
    }
    Lcd8_Clear();
    Lcd8_Set_Cursor(2,1);
    Lcd8_Write_Char('e');
    Lcd8_Write_Char('S');
    __delay_ms(2000);
  }
}
```

KEY BOARD

Matrix Keypad is a very useful and userfriendly when we want to design certain applications like Calculator, Telephone etc. Matrix Keypad is made by arranging push button switches in rows and columns. Just imagine, if you want to interface a 4*4 (16 keys) matrix keypad with a microcontroller. In the straight forward way, you will need 16 pins of a microcontroller for that, but by using a simple technique we can reduce it to 8 pins. In the matrix keypad switches are connected in a special manner a shown in the figure below.



4×4-Matrix-Keypad

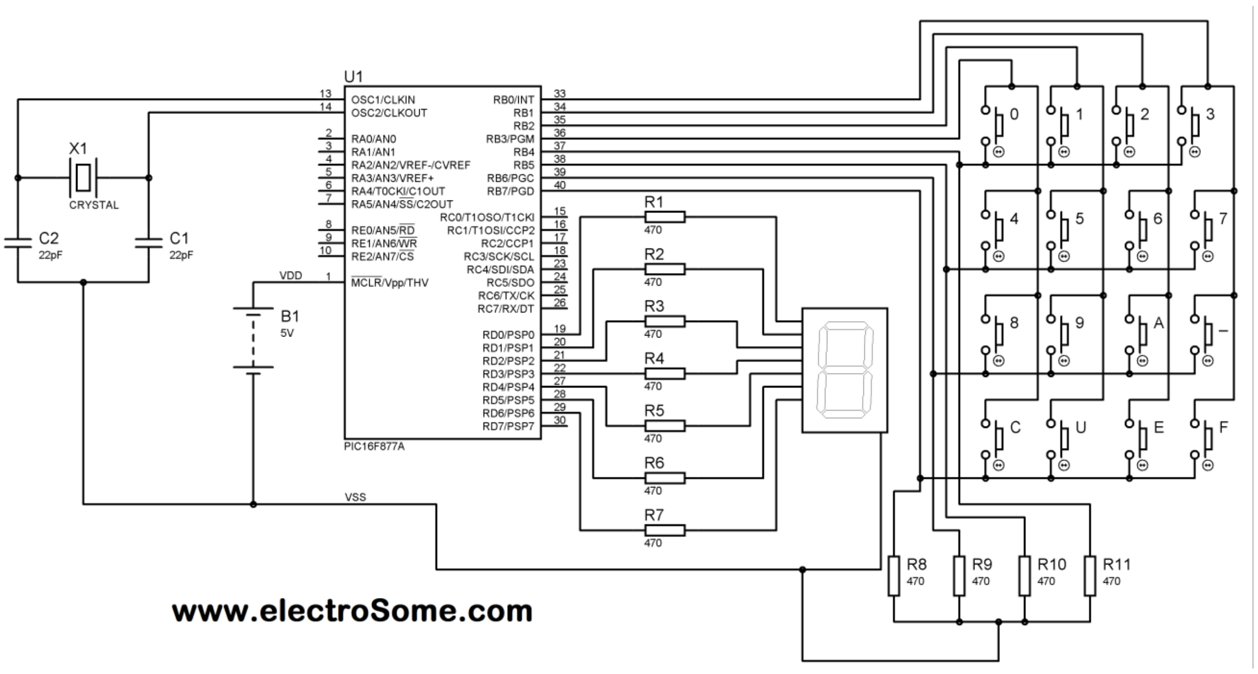
Pressed keys can be detected by Scanning. For the sake of explanation, lets assume all column connections (Col1 – Col4) are input pins and all row connections (Row1 – Row4) are output pins. In the normal case (not scanning) all column inputs where in LOW (GND) state. For scanning keypad,

1. A Logic HIGH signal is given to Col1 of column inputs.
2. Then each Row output (row1 – row4) is scanned one by one. If any of the key belongs to first column is pressed, the Logic high signal from the Col1 will pass to that row. Through we can detect the key.
3. This process is repeated for all columns if we want to detect multiple keys.

In this post I am explaining only about detecting one key at a time. For explaining the working I am using a 4*4 matrix keypad and the result is displayed in a Seven Segment Display. Matrix Keypad scanning is stopped as soon as any key press is detected and the Scanning is restarted if we need more inputs.

Interfacing with PIC Microcontroller

Circuit Diagram



Matrix Keypad interfacing with PIC Microcontroller

Note: VDD and VSS of the pic microcontroller is not shown in the circuit diagram. VDD should be connected to +5V and VSS to GND.

Matrix Keypad is connected to the PORTB of the PIC Microcontroller. Each column of the Matrix Keypad is connected to RB0 – RB3 of the PIC Microcontroller, which are configured as output pins. While each row of the Matrix Keypad is connected to RB4 – RB7 of the PIC Microcontroller, which are configured as input pins.

Here I am using 4*4 matrix keypad, having characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, -, C, U, E, F. ‘B’ is replaced by ‘-‘ and ‘D’ is replaced by ‘U’ because Seven Segment Display is used for displaying characters. ‘B’ will be similar to ‘8’ and ‘D’ will be similar to ‘0’ when displayed in Seven Segment Display. For reading data for the Matrix Keypad, each column is made high and rows are scanned as I said above.

MikroC Programming

We use the function readKeyboard() to scan the Matrix Keypad and findKey() to find the pressed key. You can edit the function findKey() to change the character corresponds to each key of the Matrix Keypad.

Function to Scan Keypad

```
char readKeyboard()
{
 unsigned int i = 0;
 for(i=0;i<4;i++)
 {
  if(i == 0)
  PORTB = 1;
  else if(i == 1)
  PORTB = 2;
  else if(i == 2)
  PORTB = 4;
  else if(i == 3)
  PORTB = 8;

  if(PORTB.F4)
    return findKey(i,0);
  if(PORTB.F5)
   return findKey(i,1);
  if(PORTB.F6)
   return findKey(i,2);
  if(PORTB.F7)
   return findKey(i,3);
 }
 return ' ';
}
```

This function initiates the keypad scanning and returns the character corresponds to the pressed key when a key press is detected. It uses the function findKey() to find the character corresponds to a particular row and column. In this function space (‘ ‘) is used as the null character, which is returned when no key is pressed, you may change this according to your needs.

Function to Find Keys

```
char findKey(unsigned short a, unsigned short b)
{
 if(b == 0)
 {
   if(a == 3)
    return '0';
   else if(a == 2)
    return '1';
   else if(a == 1)
    return '2';
   else if(a == 0)
    return '3';
 }
 else if(b == 1)
 {
   if(a == 3)
    return '4';
   else if(a == 2)
    return '5';
   else if(a == 1)
    return '6';
   else if(a == 0)
    return '7';
 }
 else if(b == 2)
 {
   if(a == 3)
    return '8';
   else if(a == 2)
    return '9';
   else if(a == 1)
    return 'A';
   else if(a == 0)
    return '-';
 }
 else if(b == 3)
 {
   if(a == 3)
    return 'C';
   else if(a == 2)
    return 'U';
   else if(a == 1)
    return 'E';
   else if(a == 0)
    return 'F';
 }
}
```

This function returns the character corresponding to a particular row and column. You may change characters corresponding to each key according to your need by editing this function.

Seven Segment Decoding Function

```
unsigned int sevenSegmentDecoder(char a)
{
 switch(a)
 {
   case '0': return 0x3F;
   case '1': return 0x06;
   case '2': return 0x5B;
   case '3': return 0x4F;
   case '4': return 0x66;
   case '5': return 0x6D;
   case '6': return 0x7D;
   case '7': return 0x07;
   case '8': return 0x7F;
   case '9': return 0x6F;
   case '0': return 0x3F;
   case 'A': return 0x77;
   case '-': return 0x40;
   case 'C': return 0x39;
   case 'U': return 0x3E;
   case 'E': return 0x79;
   case 'F': return 0x71;
   case ' ': return 0;
 }
}
```

This function decodes the given character to display it in Seven Segment Display.

PARALLEL AND SERIAL ADC

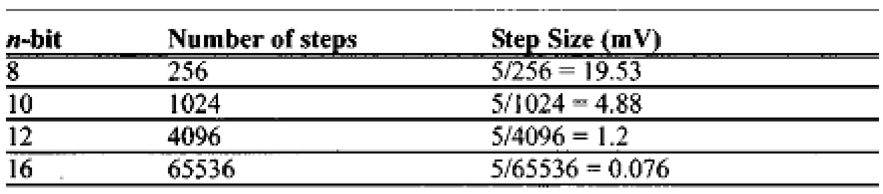
This chapter explores some more real-world devices such as ADCs (analog-to-digital converters), DACs (digital-to-analog converters), and sensors. We will also explain how to interface the 8051 to these devices. In Section 13.1, we describe analog-to-digital converter (ADC) chips. We will study the 8-bit parallel ADC chips ADC0804, ADC0808/0809, and ADC0848 We will also look at the serial ADC chip MAX1112. The characteristics of DAC chips are discussed in Section 13.2. In Section 13.3, we show the interfacing of sensors and discuss the issue of signal conditioning.

ADC devices

Analog-to-digital converters are among the most widely used devices for data acquisition. Digital computers use binary (discrete) values, but in the physical world everything is analog (continuous). Temperature, pressure (wind or liquid), humidity, and velocity are a few examples of physical quantities that we deal with every day. A physical quantity is converted to electrical (voltage, current) signals using a device called a *transducer*. Transducers are also referred to as *sensors*. Sensors for temperature, velocity, pressure, light, and many other natural quantities produce an output that is voltage (or current). Therefore, we need an analog-to-digital converter to translate the analog signals to digital numbers so that the microcontroller can read and process them. An ADC has n-bit resolution where *n* can be 8, 10, 12, 16 or even 24 bits.

The higher-resolution ADC provides a smaller step size, where *step size* is the smallest change that can be discerned by an ADC. This is shown in Table 13-1. In this chapter we examine several 8-bit ADC chips. In addition to resolution, conversion time is another major factor in judging an ADC. *Conversion time* is defined as the time it takes the ADC to convert the analog input to a digital (binary) number. The ADC chips are either parallel or serial. In parallel ADC, we have 8 or more pins dedicated to bringing out the binary data, but in serial ADC we have only one pin for data out. Serial ADCs are discussed at the end of this section.

Resolution vs. Step Size for ADC



n-bit	Number of steps	Step Size (mV)
8	256	5/256 = 19.53
10	1024	5/1024 = 4.88
12	4096	5/4096 = 1.2
16	65536	5/65536 = 0.076

Notes: **V_{cc} = 5 V**

Step size (resolution) is the smallest change that can be discerned by an ADC.

ADC0804 chip

The ADC0804 1C is an 8-bit parallel ADC in the family of the ADC0800 series from National Semiconductor (www.national.com). It is also available from many other manufacturers. It works with +5 volts and has a resolution of 8 bits. In the ADC0804, the conversion time varies depending on the clocking signals applied to the CLK IN pin, but it cannot be faster than 110 p.s. The following is the ADC0804 pin description.

CS

Chip select is an active low input used to activate the ADC0804 chip. To access the ADC0804, this pin must be low.

RD (read)

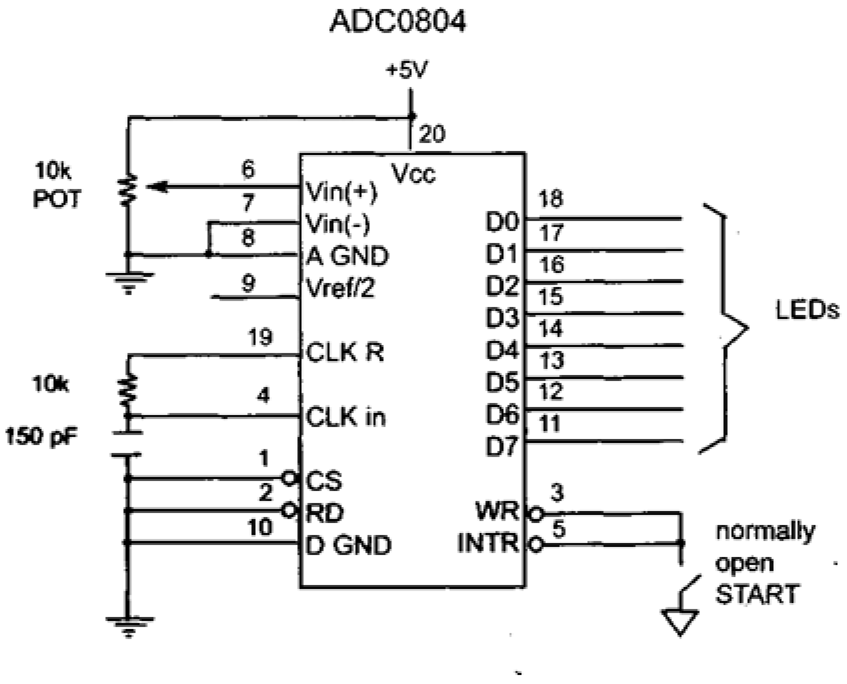
This is an input signal and is active low. The ADC converts the analog input to its binary equivalent and holds it in an internal register. RD is used to get the converted data out of the ADC0804 chip. When CS = 0, if a high-to-low pulse is applied to the RD pin, the 8-bit digital output shows up at the DO – D7 data pins. The RD pin is also referred to as output enable (OE).

WR (write; a better name might be "start conversion")

This is an active low input used to inform the ADC0804 to start the conversion process. If CS = 0 when WR makes a low-to-high
transition,

ADC0804 Chip (testing ADC0804 in free running mode)

Amount of time it takes to convert varies depending on the CLK IN and CLK R values explained below. When the data conversion is complete, the INTR pin is forced low by the ADC0804.



CLK IN and CLK R

CLK IN is an input pin connected to an external clock source when an external clock is used for timing. However, the 804 has an internal clock generator. To use the internal clock generator (also called self-clocking) of the ADC0804, the CLK IN and CLK R pins are connected to a capacitor and a resistor, as shown in Figure 13-1. In that case the clock frequency is determined by the equation:

$$f = \frac{1}{1.1\,RC}$$

Typical values are R = 10K ohms and C = 150 pF.

Substituting in the above equation, we get/= 606 kHz. In that case, the conversion time is 110 us.

INTR (interrupt; a better name might be "end of conversion")

This is an output pin and is active low. It is a normally high pin and when the conversion is finished, it goes low to signal the CPU that the converted data is ready to be picked up. After INTR goes low, we make CS = 0 and send a high-to-low pulse to the RD pin to get the data out of the ADC0804 chip.

V_{in} (+) and V_{in} (-)

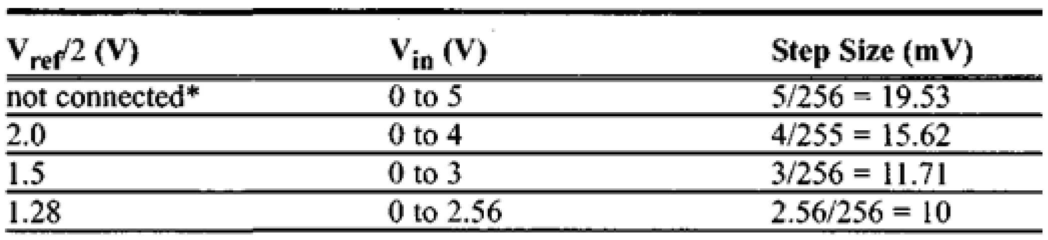
These are the differential analog inputs where Vj_n = Vj_n (+) – Vj_n (-). Often the Vj_n (-) pin is connected to ground and the Vj_n (+) pin is used as the analog input to be converted to digital.

Vcc

This is the +5 volt power supply. It is also used as a reference voltage when the $V_{ref}/2$ input (pin 9) is open (not connected). This is discussed next.

$V_{ref}/2$

Pin 9 is an input voltage used for the reference voltage. If this pin is open (not connected), the analog input voltage for the ADC0804 is in the range of 0 to 5 volts (the same as the V_{cc} pin). However, there are many applications where the analog input applied to V_{in}needs to be other than the 0 to +5 V range. V_{ref}*12*is used to implement analog input voltages other than 0 to 5 V. For example, if the analog input range needs to be 0 to 4 volts, $V_{rej}/2$ is connected to 2 volts.



$V_{ref}/2$ (V)	V_{in} (V)	Step Size (mV)
not connected*	0 to 5	5/256 = 19.53
2.0	0 to 4	4/255 = 15.62
1.5	0 to 3	3/256 = 11.71
1.28	0 to 2.56	2.56/256 = 10

***Notes:* V_{cc} = 5 V**

* When not connected (open), $V_{ret}/2$ is measured at 2.5 volts for V_{cc} = 5 V.

Step Size (resolution) is the smallest change that can be discerned by an ADC.

DO-D7

DO – D7 (D7 is the MSB) are the digital data output pins since ADC0804 is a parallel ADC chip. These are tri-state buffered and the converted data is accessed only when CS = 0 and RD is forced low. To calculate the output voltage, use the following formula.

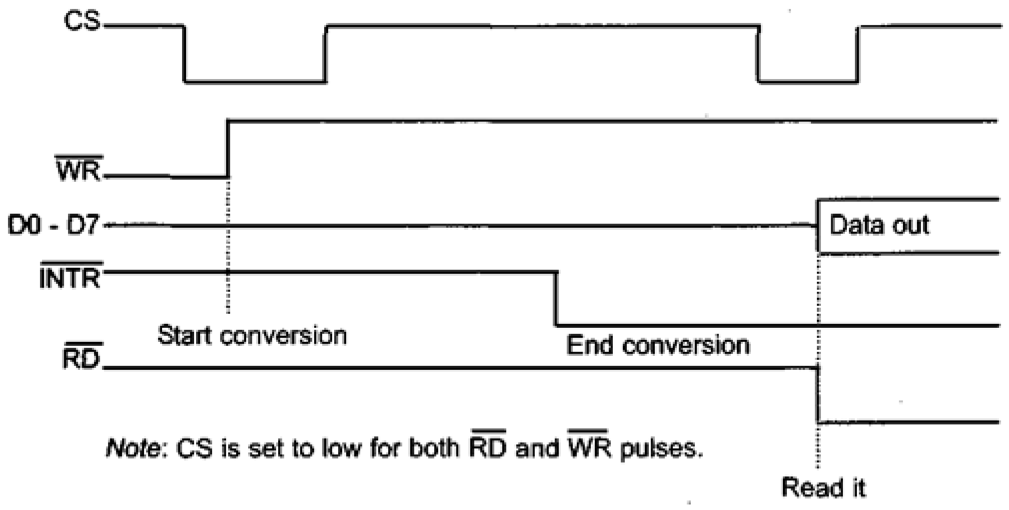
$$D_{out} = \frac{Vin}{step\ size}$$

Where D_{out} = digital data output (in decimal), V_{in} = analog input voltage, and step size (resolution) is the smallest change, which is (2 x $V_{ref}/2$)/256 for ADC0804.

Analog ground and digital ground

These are the input pins providing the ground for both the analog signal and the digital signal. Analog ground is connected to the ground of the analog V_{in} while digital ground is connected to the ground of the V_{cc} pin. The reason that we have two ground pins is to isolate the analog V_{in} signal from transient voltages caused by digital switching of the output DO – D7. Such isolation contributes to the accuracy of the digital data output. In our discussion, both are connected to the same ground; however, in the real world of data acquisition the analog and digital grounds are handled separately.

From this discussion we conclude that the following steps must be followed for data conversion by the ADC0804 chip.



Read and Write Timing for ADC0804

Make CS = 0 and send a low-to-high pulse to pin WR to start the conversion.

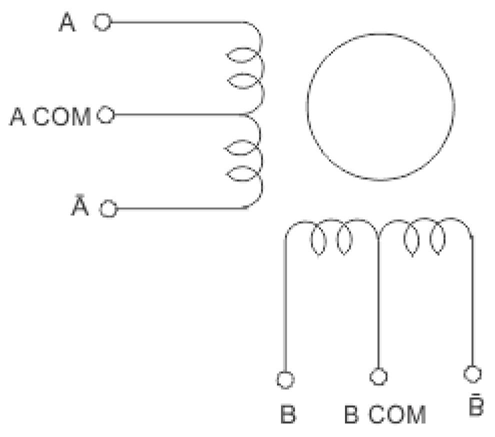
Keep monitoring the INTR pin. If INTR is low, the conversion is finished and we can go to the next step. If INTR is high, keep polling until it goes low.

After the INTR has become low, we make CS = 0 and send a high-to-low pulse to the RD pin to get the data out of the ADC0804 1C chip.

INTERFACING OF STEPPER MOTOR

A stepper motor is a type of DC motor that rotates in steps. When electrical signal is applied to it, the motor rotates in steps and the speed of rotation depends on the rate at which the electrical signals are applied and the direction of rotation is dependent on the pattern of pulses that is followed.

A **stepper motor** is made up of a rotor, which is normally a permanent magnet and it is, as the name suggests the rotating component of the motor. A stator is another part which is in the form of winding. In the diagram below, the center is the rotor which is surrounded by the stator winding. This is called as four phase winding.



Stepper Motor

Working of Stepper Motor

The centre tap on the stator winding allows the current in the coil to change direction when the winding are grounded. The magnetic property of the stator changes and it will selectively attract and repel the rotor, thereby resulting in a stepping motion for the motor.

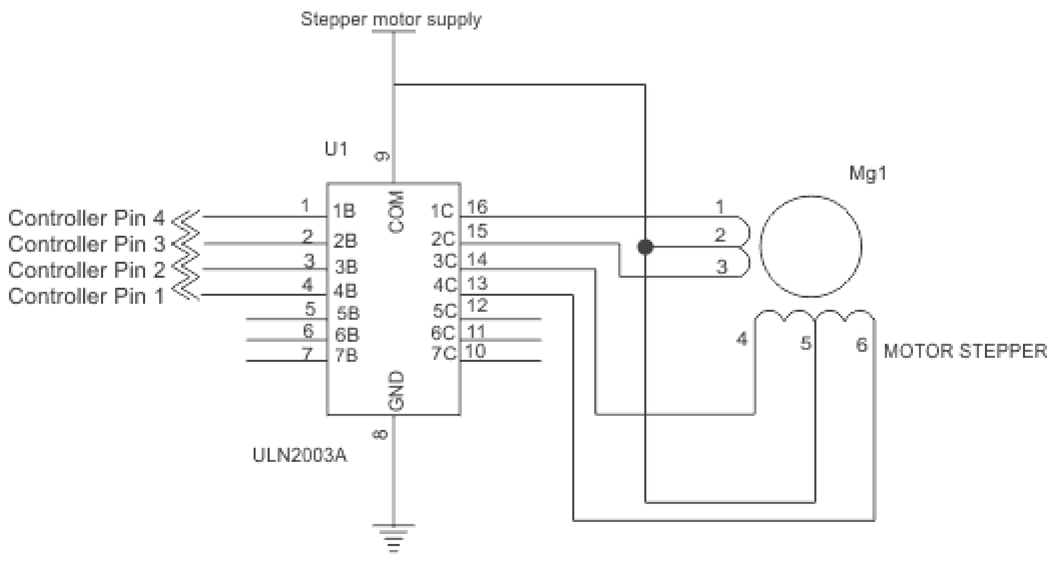
Stepping Sequence

In order to get correct motion of the motor, a stepping sequence has to be followed. This stepping sequence gives the voltage that must be applied to the stator phase. Normally a 4 step sequence is followed. When the sequence is followed from step 1 to 4, we get a clock wise rotation and when it is followed from step 4 to 1, we get a counter clockwise rotation.

Step No	**A**	**A**	**B**	**B**
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1

Interfacing Diagram

The diagram below shows the **interfacing of stepper motor** to a micro-controller. This is general diagram and can be applied to any micro-controller family like PIC micro-controller, AVR or 8051 micro-controller.



General Interfacing diagram of a Stepper with ULN2003A driver Since, the micro-controller cannot provide enough current to run the motor, a driver like a ULN2003 is used to drive the motor. Similarly, individual transistors or any other driver ICcan also be used to drive the motor. See to it that if required, the external pull up resistors is connected to pins depending on the micro-controller you use. The motor must never be directly connected to the controller pins. The motor voltage depends on the size of the motor. A typical 4 phase uni-polar stepper motor has 5 terminals. 4 phase terminals and one common terminal of the center tap that is connected to ground.

The programming algorithm for continuous rotation in clockwise mode is given below-

1. Initialize the port pins used for the motor as outputs
2. Write a common delay program of say 500 ms
3. Output first sequence-0 × 09 on the pins
4. Call delay function
5. Output second sequence-0 × 0 c on the pins
6. Call delay function
7. Output third sequence-0 × 06 on the pins
8. Call delay function
9. Output fourth sequence-0 × 03 on the pins
10. Call delay function
11. Go to step 3

Step Angle

The number of steps required to complete one full rotation depends on the step angle of the stepper motor. The step angle can vary from 0.72 degrees to 15 degrees per step. Depending on that 500 to 24 steps may be required to complete one rotation. In position control applications the selection on motor should be based on the minimum degree of rotation that is required per step.

Half Stepping

Stepper motors can be used at half the actual step angle. This is called half stepping. Suppose a motor is rated for 15 degrees per step, then it can be programmed in such a way that it rotates at 7.5 degrees per step by applying a special half stepping sequence to it.

Step No	A	A	B	B
1	1	0	0	1
2	1	0	0	0
3	1	1	0	0
4	0	1	0	0
5	0	1	1	0
6	0	0	1	0
7	0	0	1	1
8	0	0	0	1

C Code for 8051 Micro-controller

```
#include
#define out P1 //motor connected on Port 1 lower
#define step 50 //one revolution for 1.8 degree motor
unsigned char i;
void delay (unsigned char k);
void main()
{
for( i = 0; i<="" br="" style="margin: 0px; padding: 0px;"> {
out=0x09;
delay();
out=0x0c;
delay();
out=0x06;
delay();
out=0x03();
delay();
}
void delay(unsigned char k)
{
unsigned int j;
for(;k>>0;k--)
{
for(j = 0; j<<40000; j++);
}
```