EMBEDDED SYSTEMS

MCA

Lesson Writer

A. HARI PRASAD REDDY
Assistant Professor
Vasireddy Venkatadri Institute of Technology
Nambur (P.O.), GUNTUR – Dt.

Editor & Advisor for the Course

Prof. E.SREENIVASA REDDY, M.Tech., Ph.D.
Principal
Vasireddy Venkatadri Institute of Technology
Nambur (P.O.), GUNTUR – Dt.

Director
Prof.V.CHANDRASEKHARA RAO, M.Com., Ph.D.

CENTRE FOR DISTANCE EDUCATION
ACHARAYA NAGARJUNA UNIVERSITY
NAGARJUNA NAGAR – 522 510

Ph: 0863-2293299,2293356,08645-211023,Cell:98482 85518
08645-21102 4 (Study Material)
Website: www.anucde.com, e-mail:anucde@yahoo.com

INDEX

UNIT – I

1. BASIC TERMINOLOGY

Objective

In today's world electronic gadgets are becoming part of our life style. For example to say mobiles, music players, iPods, navigation systems etc are possible with the advancement in technology the field of intelligent computing popularly known as embedded technology. In this text book we mainly learn about the basics of the embedded systems to understand what a basic embedded device consists.

At the end of this lesson the reader will understand what is a computer, differences between hardware and software, different parts of computer, what is operating system and types of operating systems.

Computer

Different types of definitions for what is computer are given below.

- Any device capable of processing information to produce a desired result. Computers typically perform their work in three well-defined steps: (1) accepting input, (2) processing the input data according to predefined rules (programs), and (3) producing output.
- A computer system comprises hardware and software used for executing different mathematical manipulations most perfectly within very less time.
- A functional unit that can perform substantial computations, including numerous arithmetic operations and logic operations without human intervention during a run. A computer may consist of a stand-alone unit or may consist of several interconnected units.
- An electronic machine that receives processes and presents data.

Types of computers

Computers are classified into different types depending on the resources available on them.

Super Computers

A supercomputer is a computer that has high speed and processing power. The most famous series of supercomputers were designed by the company founded and named after Seymour Cray. The Cray-1 was built in the 1976 and installed at Los Alamos National Laboratory. Supercomputers are used for extremely calculation-intensive tasks such simulating nuclear bomb detonations, aerodynamic flows, and global weather patterns. A supercomputer typically costs several million dollars. Figure1 represents the super computer CRAY-2 in around 1980's.

Figure -1

Main Frame Computer

A mainframe computer is a large, powerful computer that handles the processing for many users simultaneously (up to several hundred users). Users connect to the mainframe using terminals and submit their tasks for processing by the mainframe. A terminal is a device that has a screen and keyboard for input and output, but it does not do its own processing (they are also called dumb terminals since they can't process data on their own). The processing power of the mainframe is time-shared between all of the users.

Mainframes typically cost several hundred thousand dollars. They are used in situations where a company wants the processing power and information storage in a centralized location. Mainframes are also now being used as high-capacity server computers for networks with many client workstations. Figure2 shows the photo of IBM z-series computer which is about 6 feet tall.

Minicomputers

A minicomputer is a multi-user computer that is less powerful than a mainframe. This class of computers became available in the 1960’s when large scale integrated circuits made it possible to build a computer much cheaper than the then existing mainframes.

Work Stations/Servers

A workstation is a powerful, high-end microcomputer. They contain one or more microprocessor CPUs. They may be used by a single-user for applications requiring more power than a typical PC (rendering complex graphics, or performing intensive scientific calculations).

Alternately, workstation-class microcomputers may be used as server computers that supply files to client computers over a network. This class of powerful microcomputers can also be used to handle the processing for many users simultaneously who are connected via terminals in this respect, high-end workstations have essentially supplanted the role of minicomputers. Figure3 represents a workstation computer.

Figure-3

Desktop Computers

Desktop computers are not meant for portable usage. They usually sit in one place on a desk or table and are plugged into a wall outlet for power. The case of the computer holds the motherboard, drives, power supply, and expansion cards. This case may lay flat on the desk, or it may be a tower that stands vertically (on the desk or under it). The computer usually has a separate monitor (either a CRT or LCD) although some designs have a display built into the case. A separate keyboard and mouse allow the user to input data and commands. Figure4 represents a desktop computer.

Figure4

Laptop Computers

Laptop or notebook computers are small and lightweight enough to be carried around with the user. They run on battery power, but can also be plugged into a wall outlet. They typically have a built-in LCD display that folds down to protect the display when the computer is carried around. They also feature a built-in keyboard and some kind of built-in pointing device (such as a touch pad).

While some laptops are less powerful than typical desktop machines, this is not true in all cases. Laptops, however, cost more than desktop units of equivalent processing power because the smaller components needed to build laptops are more expensive. Figure5 represents a laptop computer photo.

Figure -5

PDA's

A Personal Digital Assistant (PDA) is a handheld microcomputer that trades off power for small size and greater portability. They typically use a touch-sensitive LCD screen for both output and input (the user draws characters and presses icons on the screen with a stylus). PDAs communicate with desktop computers and with each other either by cable connection, infrared (IR) beam, or radio waves. PDAs are normally used to keep track of appointment calendars, to-do lists, address books, and for taking notes. Figure6 represents the photo of personal Digital Assistance.

Figure -6

Palmtop/Handheld Computers

A palmtop or handheld PC is a very small microcomputer that also sacrifices power for small size and portability. These devices typically look more like a tiny laptop than a PDA, with a flip-up screen and small keyboard. They may use Windows CE or similar operating system for handheld devices. Some PDAs and palmtops contain wireless networking or cell phone devices so that users can check e-mail or surf the web on the move. Figure7 represent a photo of PDA.

Figure -7

Microprocessors Everywhere

Microprocessor chips are found in many electronic devices (in your iPod, in your DVD player, in your microwave, in your car, in your phone). These are special-purpose processing units that run programs to control the equipment and optimize its performance.

Parts of Computers

Figure8 represents the parts of a desktop computer.

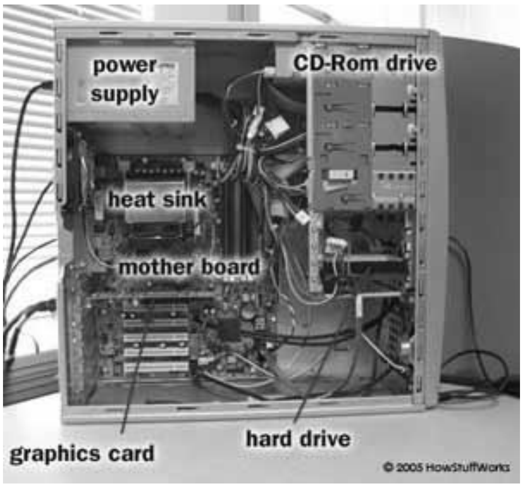


Figure -8

Let's take a look at the main components of a typical desktop computer:

- **Central processing unit (CPU)** - The "brain" of the computer system is called the central processing unit. It is a chip that holds a complete computational engine. It uses assembly language as its native language. Everything that a computer does is overseen by the CPU.

- **Memory** - This is very fast storage used to hold data. It has to be fast because it connects directly to the microprocessor. There are several specific types of memory in a computer:
- **Random-access memory (RAM)** - Used to temporarily store information with which the computer is currently working
- **Read-only memory (ROM)** - A permanent type of memory storage used by the computer for important data that doesn't change
- **Basic input/output system (BIOS)** - A type of ROM that is used by the computer to establish basic communication when the computer is first powered on
- **Caching** - The storing of frequently used data in extremely fast RAM that connects directly to the CPU
- **Virtual memory** - Space on a hard disk used to temporarily store data and swap it in and out of RAM as needed
- **Flash memory** - a solid state storage device, Flash memory requires no moving parts and retains data even after the computer powers off
- **Motherboard** - This is the main circuit board to which all of the other internal components connect. The CPU and memory are usually on the motherboard. Other systems may be found directly on the motherboard or connected to it through a secondary connection. For example, a sound card can be built into the motherboard or connected through an expansion slot.
- **Power supply** - An electrical transformer regulates the electricity used by the computer.
- **Hard disk** - This is large-capacity permanent storage used to hold information such as programs and documents. Traditional hard drives contain moving parts -- the drive has platters on which it stores data. The drive spins the platters to record and read data. But some newer hard drives are flash-based with no moving parts. These drives are called solid-state drives.
- **Operating system** - This is the basic software that allows the user to interface with the computer.
- **Integrated Drive Electronics (IDE) Controller** - This is the primary interface for the hard drive, CD-ROM and floppy disk drive.
- **Accelerated Graphics Port (AGP)** - This is a very high-speed connection used by the graphics card to interface with the computer.

- **Sound card** - This is used by the computer to record and play audio by converting analog sound into digital information and back again.
- **Graphics card** - This translates image data from the computer into a format that can be displayed by the monitor. Some graphics cards have their own powerful processing units (called a GPU -- graphics processing unit). The GPU can handle operations that normally would require the CPU.
- **Ports** - In computer hardware terms, a port is an interface that allows a computer to communicate with peripheral equipment. **Real-time clock** - Every PC has a clock containing a vibrating crystal. By referring to this clock, all the components in a computer can synchronize properly.
- **Complementary Metal-oxide Semiconductor** - The CMOS and CMOS battery allow a computer to store information even when the computer powers down. The battery provides uninterrupted power.
- **Fans, heat sinks and cooling systems** - The components in a computer generate heat. As heat rises, performance can suffer. Cooling systems keep computers from overheating.

SOFTWARE

Software is a general term for the various kinds of programs used to operate computers and related devices.

Software can be the variable part of a computer and hardware is the invariable part. Software is often divided into application software (programs that do work users are directly interested in) and system software (which includes operating systems and any program that supports application software). The term middleware is sometimes used to describe programming that mediates between application and system software or between two different kinds of application software (for example, sending a remote work request from an application in a computer that has one kind of operating system to an application in a computer with a different operating system).

An additional and difficult-to-classify category of software is the utility, which is a small useful program with limited capability. Some utilities come with operating systems. Like applications, utilities tend to be separately installable and capable of being used independently from the rest of the operating system.

Applets are small applications that sometimes come with the operating system as "accessories." They can also be created independently using the Java or other programming languages.

Software can be purchased or acquired as shareware (usually intended for sale after a trial period), liteware (shareware with some capabilities disabled), freeware (free software but with copyright restrictions), public domain software (free with no restrictions), and open source (software where the source code is furnished and users agree not to limit the distribution of improvements).

Software is often packaged on CD-ROMs and diskettes. Today, much purchased software, shareware, and freeware is downloaded over the Internet. A new trend is software that is made available for use at another site known as an application service provider.

Some general kinds of application software include:

- Productivity software, which includes word processors, spreadsheets, and tools for use by most computer users
- Presentation software
- Graphics software for graphic designers
- CAD/CAM software
- Specialized scientific applications

Firmware or microcode is programming that is loaded into a special area on a microprocessor or read-only memory on a one-time or infrequent basis so that thereafter it seems to be part of the hardware.

Operating systems

Operating system is defined as system software used as communication medium between user and hardware parts (electronic parts) of computer. It is the most important program that runs on a computer. Every general-purpose computer must have an operating system. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers.

For large systems, the operating system has even greater responsibilities and powers. It is like a traffic cop -- it makes sure those different programs and users running at the same time do not interfere with each other. The operating system is also responsible for security, ensuring that unauthorized users do not access the system. Operating systems can be classified as follows:

- **Multi-user**: Allows two or more users to run programs at the same time.Some operating systems permit hundreds or even thousands of concurrent users.
- **Multiprocessing**: Supports running a program on more than one CPU.
- **Multitasking**: Allows more than one program to run concurrently.

- **Multithreading**: Allows different parts of a single program to run concurrently.
- **Real time:** Responds to input instantly. General-purpose operating systems, such as DOS and UNIX, are not real-time.

Operating systems provide a software platform on top of which other programs called application programs run. The application programs must be written to run on top of a particular operating system. For PCs, the most popular operating systems are DOS, OS/2, and Windows, but others are available, such as Linux.

Summary

1. Embedded technology mainly deals with embedding intelligence in electronic devices.
2. Any device capable of processing information to produce a desired result. No matter how large or small they are, computers typically perform their work in three well-defined steps: (1) accepting input, (2) processing the input according to predefined rules (programs), and (3) producing output.
3. The different types of computers are
    a. Super computers
    b. Main frame computers
    c. Workstations/servers
    d. Desktop computers
    e. Laptop computers
    f. PDA’s
    g. Palm top/handheld computers.
4. The main parts of computer are
    a. Central processing Unit
    b. Random Access Memory
    c. Read Only Memory
    d. Hard Disk
    e. Mother Board
    f. Key Board.

5. Software is a general term for the various kinds of programs used to operate computers and related devices. It is of two types namely application software and system software.
6. Operating system is defined as system software used as communication medium between user and hardware parts (electronic parts) of computer.

Objective Questions

1. What are the different types of computers? Explain.
2. Briefly explain the different parts of a computer?
3. What is software? Classify different types of software.
4. What is operating system? Classify.

References

- www.howstuffworks.com
- www.wikipedia.com
- Embedded Software The Works – colin walls
- Vahid F & GIvargis T; Embedded System Design, John willey(2002)

2. EMBEDDED SYSTEMS - INTRODUCTION

OBJECTIVE

In this chapter we will study about basics of embedded systems and its terminology.

Introduction

An embedded system is a special-purpose computer designed to perform one or a few dedicated functions, often with real-time computing constraints. An embedded system has historically been defined as a single function product where the intelligence is embedded in the system. It is a system made with combination of hardware and software designed for a specific individual application. Embedded systems are usually programmed in high level language that is compiled (and/or assembled) into an executable ("machine") code. These are loaded into Read Only Memory (ROM) and called "firmware" or "microcode" or a "microkernel". The microprocessor can be from 8-bit tot 64-bit. The bit size refers to the amount of memory accessed by the processor at a time. The most advanced systems actually have a tiny, streamlined OS running the show, executing on a 32-bit or 64-bit processor. This is called RTOS. *Real-Time Systems* can be classified as:

- *Hard Real-Time Systems* - systems with severe constraints on the timeliness of the response.
- *Soft Real-Time Systems* - systems which tolerate small variations in response times.
- *Hybrid Real-Time Systems* - systems which exhibit both hard and soft constraints on its performance.

Embedded Hardware

All embedded systems contains a microprocessor or microcontroller for processing of information and execution of programs, memory in the form of ROM/RAM for storing embedded software programs and data, and I/O interfaces for external interface. Any additional requirement in an embedded system is dependent on the equipment it is controlling. Very often these systems have a standard serial port, a network interface, I/O interface, or hardware to interact with sensors and activators on the equipment.

Embedded Software

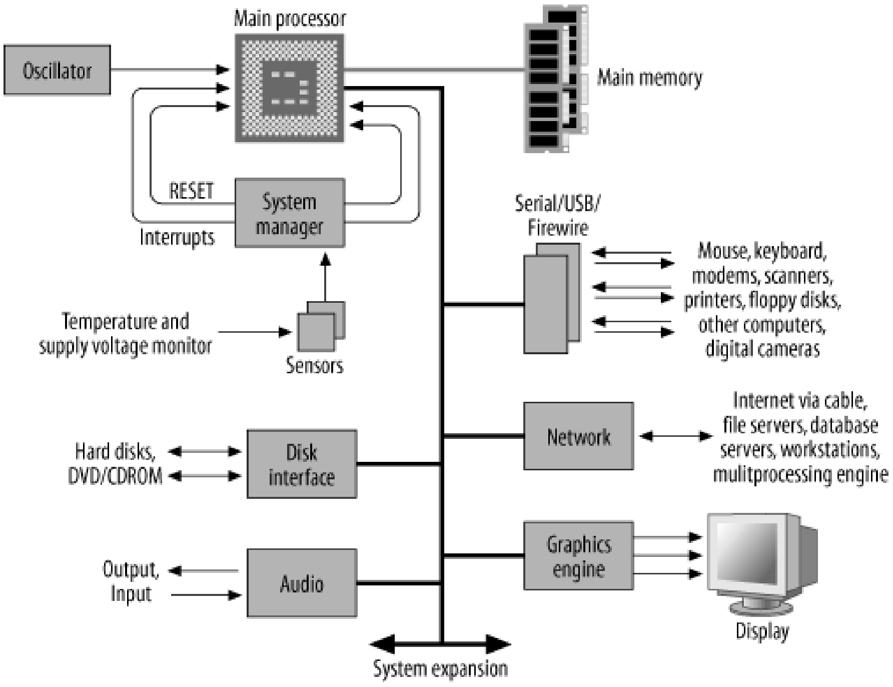
C has become the language of choice for embedded programmers, because it has the benefit of processor independence, which allows the programmer to concentrate on algorithms and applications, rather than on the details of processor architecture. However, many of its advantages apply equally to other high-level languages as well. Perhaps the greatest strength of C is that it gives embedded programmers an extraordinary degree of direct hardware control without sacrificing the benefits of high-level languages. Compilers and cross compilers are also available for almost every processor with C. Any source code written in C or C++ or Assembly language must be converted into an executable image that can be loaded onto a ROM chip. The process of converting the source code representation of your embedded software into an executable image involves three distinct steps, and the system or computer on which these processes are executed is called a host computer. First, each of the source files that make an embedded application must be compiled or assembled into distinct object files. Second, the entire object files that result from the first step must be linked into a final object file called the re-locatable program. Finally, the physical memory address must be assigned to the re-locatable program. The result of the third step is a file that contains an executable image that is ported on the ROM chip. This ROM chip, along with the processor and other devices and interfaces, makes an embedded system run. There are some very basic differences between conventional programming and embedded programming. First, each target platform is unique. Second, there is a difference in the development and debugging of applications.

Difference between computer and embedded device

An embedded system has a self-contained operating system on a "chip" thus embedded into the system and does not rely on having a hard disk with the operating system on it. An embedded system has historically been defined as a single function product where the intelligence is embedded in the system. It could be anything from a dishwasher to a hearing aid, if that product includes a microprocessor and software. A PC is designed to be a general purpose computing environment. Many of today's embedded systems are looking more like PCs with user interfaces, touch screens, displays, keypads and more. Still, these are not general function systems but are designed to perform very specific functions.

What a computer is used for, what tasks it must perform, and how it interacts with humans and other systems determine the functionality of the machine and, therefore, its architecture, memory, and I/O. It has a large main memory to hold the operating system, applications, and data, and an interface to mass storage devices (disks and DVD/CD-ROMs). It has a variety of I/O devices for user input (keyboard, mouse, and audio), user output (display interface and audio), and connectivity (networking and peripherals). The fast processor requires a system manager to monitor its core temperature and supply voltages, and to generate a system reset.

Large-scale embedded computers may also take the same form. For example, they may act as a network router or gateway, and so will require one or more network interfaces, large memory, and fast operation. They may also require some form of user interface as part of their embedded application and, in many ways, may simply be a conventional computer dedicated to a specific task. Thus, in terms of hardware, many high-performances embedded systems is not that much different from a conventional desktop machine? The diagram represents the basic components of general purpose computer.



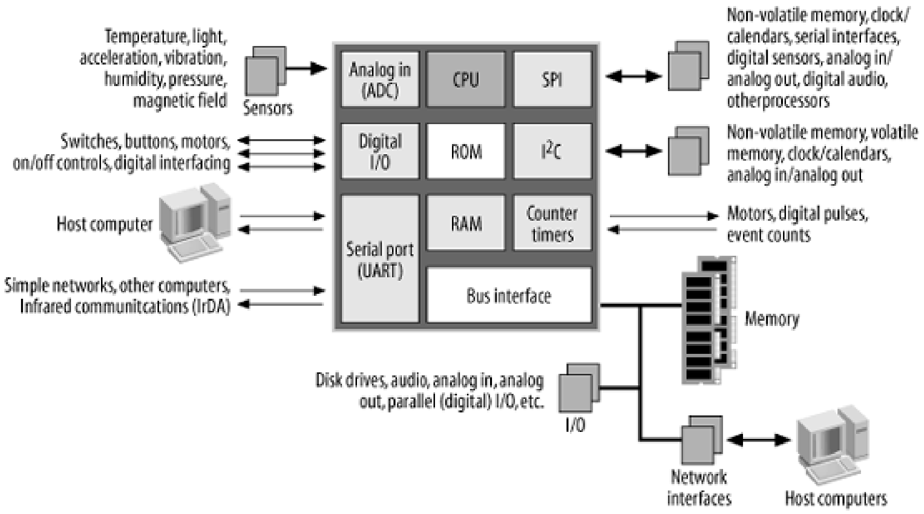
Smaller embedded systems use microcontrollers as their processor, with the advantage that this processor will incorporate much of the computer's functionality on a single chip. The microcontroller has, at a minimum, a CPU, a small amount of internal memory (ROM and/or RAM), and some form of I/O, which is implemented within a microcontroller as subsystem blocks. These subsystems provide the additional functionality for the processor and are common across many processors.

The most common I/O is digital I/O, commonly called general-purpose I/O, or GPIO. These are ports that may be configured by software, on a pin-by-pin basis, as either a digital input or digital output. As digital inputs, they may be used to read the state of switches or push buttons, or to read the digital status of another device. As outputs, they may be used to turn external devices on or off, or to convey status to an external device. For example, a digital output may be used to activate the control circuitry for a motor, turn a light on or off, or perhaps activate some other device such as a water valve for a garden-watering system. Used in combination, the digital inputs and outputs may be used to synthesize an interface and protocol to another chip. Most microcontrollers have other subsystems besides digital I/O but provide the ability to convert the other subsystems to general-purpose digital I/O if the functionality of the other subsystems is not required. This gives you great versatility as a system designer in how you use your microcontroller within your application.

Many microcontrollers also have analog inputs, allowing sensors to be sampled for monitoring or recording purposes. Thus, an embedded computer may measure light levels, temperature, vibration or acceleration, air or water pressure, humidity, or magnetic field, to name just some.

Alternatively, the analog inputs may be used to monitor simple voltages, perhaps to ensure the reliable operation of a larger system.

Some microcontrollers have serial ports, which enable the embedded computer to be interfaced to a host computer, a modem, another embedded system, or perhaps a simple network. Specialized forms of serial interface, such as SPI and I^{2}C, provide a simple way of expanding the microcontroller's functionality. They allow peripherals to be interfaced to the microcontroller, providing access to such devices as off-chip memories (for data or parameter storage), clock/calendar chips (for timekeeping), sensors with digital interfaces, external analog input or output, and even audio chips and other processors. Most microcontrollers have timers and counters. These may be used to generate internal interrupts at regular intervals for multitasking, to generate external triggers for off-chip systems, or to provide control pulses for motors. Alternatively, they may be used to count external triggers (pulses) from another system. A few microcontrollers also include network interfaces, such as USB, Ethernet, or CAN. In this book, we'll look at many of these peripheral subsystems in detail and see how to utilize them to increase an embedded computer's functionality. Diagram represents the block diagram of general embedded system.


Temperature, light, acceleration, vibration, humidity, pressure, magnetic field
Sensors
Analog in (ADC)
CPU
SPI
Non-volatile memory, clock/ calendars, serial interfaces, digital sensors, analog in/ analog out, digital audio, otherprocessors
Switches, buttons, motors, on/off controls, digital interfacing
Digital I/O
ROM
I2C
Non-volatile memory, volatile memory, clock/calendars, analog in/analog out
Host computer
Serial port (UART)
RAM
Counter timers
Motors, digital pulses, event counts
Simple networks, other computers, Infrared communitcations (IrDA)
Bus interface
Memory
Disk drives, audio, analog in, analog out, parallel (digital) I/O, etc.
I/O
Network interfaces
Host computers

Some of the larger microcontrollers also provide a bus interface, bringing the internal address, data, and control buses to the outside world. This allows the processor to be interfaced to a huge variety of possible peripherals in very much the same way as a conventional processor. All of the possible devices and interfaces described previously may also be implemented through the bus interface and the appropriately chosen peripheral. A bus interface provides enormous possibility.

The mix of I/O subsystems that microcontrollers may have varies considerably. Some microcontrollers are intended for simple digital control and may have only digital I/O. Others may be intended for industrial applications, and may have digital I/O, analog input, motor control, and networking. The choice of microcontroller (and there are literally thousands of subspecies available from dozens of manufacturers) depends on your processing needs and your interfacing requirements. Choose the one that best suits your purposes.

Downfalls of Embedded Computers

Embedded computers may be economical, but they are designed to specific problems. A PC computer may ship with a glitch in the software, and once discovered, a software patch can often be shipped out to fix the problem. An embedded system, however, is frequently programmed once, and the software cannot be patched. Even if it is possible to patch faulty software on an embedded system, the process is frequently far too complicated for the user. Another problem with embedded computers is that they are often installed in systems for which unreliability is not an option. For instance, the computer controlling the brakes in your car cannot be allowed to fail under any condition. The targeting computer in a missile is not allowed to fail and accidentally target friendly units. As such, many of the programming techniques used when throwing together production software cannot be used in embedded systems. Reliability must be guaranteed before the chip leaves the factory. This means that every embedded system needs to be tested and analyzed extensively. An embedded system will have very few resources when compared to full blown computing systems like a desktop computer, the memory capacity and processing power in an embedded system is limited. It is more challenging to develop an embedded system when compared to developing an application for a desktop system as we are developing a program for a very constricted environment. Some embedded systems run a scaled down version of operating system called an RTOS (real time operating system).

CHARACTERISTICS OF EMBEDDED SYSTEMS

Frequently, embedded systems are connected to the physical environment through sensors collecting information about that environment and actuators controlling that environment.

Embedded systems have to be dependable. Many embedded systems are safety-critical and therefore have to be dependable. Nuclear power plants are an example of extremely safety-critical systems that are at least partially controlled by software. Dependability encompasses the following aspects of a system:

- **Reliability:** Reliability is the probability that a system will not fail.
- **Maintainability**: Maintainability is the probability that a failing system can be repaired within a certain time-frame.
- **Availability**: Availability is the probability that the system is available. Both the reliability and the maintainability must be high in order to achieve a high availability.

- **Safety**: This term describes the property that a failing system will not cause any harm.
- **Security**: This term describes the property that confidential data remains confidential and that authentic communication is guaranteed.
- Embedded systems have to be efficient. The following metrics can be used for evaluating the efficiency of embedded systems:
- **Energy**: Many embedded systems are mobile systems obtaining their energy through batteries. Therefore, the available electrical energy must be used very efficiently.
- **Code-size**: All the code to be run on an embedded system has to be typically, there are no hard discs on which code can be stored. Dynamically adding additional code is still an exception and limited to cases such as Java-phones and set-top boxes. Due to all the other constraints, this means that the code-size should be as small as possible for the intended application. This is especially true for Systems On a Chip (SoCs), systems for which all the information processing circuits are included on a single chip stored with the system.
- **Run-time efficiency**: The minimum amount of resources should be used for implementing the required functionality.
- **Weight**: All portable systems must be of low weight. Low weight is frequently an important argument for buying a certain system.
- **Cost**: For high-volume embedded systems, especially in consumer electronics, competitiveness on the market is an extremely crucial issue, and efficient use of hardware components and the software development budget are required.

These systems are dedicated towards a certain application. For example, processors running control software in a car or a train will always run that software, and there will be no attempt to run a computer game or spreadsheet program on the same processor. There are mainly two reasons for this:

- Running additional programs would make those systems less dependable.
- Running additional programs is only feasible if resources such as memory are unused. No unused resources should be present in an efficient system.
- Most embedded systems do not use keyboards, mice and large computer monitors for their user-interface. Instead, there is a dedicated user-interface consisting of push-buttons, steering wheels, pedals etc.

- Many embedded systems must meet real-time constraints. Not completing computations within a given time-frame can result in a serious loss of the quality provided by the system.
- Many embedded systems are hybrid systems in the sense that they include analog and digital parts. Analog parts use continuous signal values in continuous time, whereas digital parts use discrete signal values in discrete time.

Typically, embedded systems are reactive systems. They can be defined as follows: A reactive system is one that is in continual interaction with its environment and executes at a pace determined by that environment.

Embedded Development Environment

The embedded system may not have a keyboard, a screen, a disk drive and other peripheral devices required for programming and development tasks. Therefore most of the programming for embedded systems is done on a host, which is a computer system with all the programming tools. Only after the program has been written, compiled, assembled and linked then it is moved to the target or the system that is shipped to the customers. After writing source file compiling, linking, relocating and porting the executable image into the ROM, you need to test and debug the application. Once you have an executable image stored as a file on the host computer, you need a way to download that image into a memory device on the target board or development board and execute it from there. And if you have the right tools at your disposal, it will be possible to set breakpoints in the program or set break points in the program or observe its execution. These various tools could be a remote debugger, simulator, emulator or an in-circuit emulator. A remote debugger can be used to download, execute, and debug embedded software over the serial port or network connection between the host and the target. In case of embedded systems, the debugger executes on two different computer systems – a remote debugger consists of two pieces of software. The front-end runs on the host computer and provides the human interface, and the hidden back-end runs on the target processor and communicates with the front-end over a communication link. The back-end provides low-level control of the target processor and is usually called debug monitor. The debug monitor resides in the ROM and is automatically started whenever the target processor is reset. It monitors the communication link to the host computer and responds to the request from the remote debugger running there. Remote debuggers are the most commonly used tools for downloading and testing tools during the development of embedded software – mainly because of their low cost. Remote debuggers are helpful in monitoring and controlling the state of embedded software, but only in in-circuit emulators (ICEs) allow you to examine the state of the processor on which that program is running. In fact an ICE actually takes the place of the processor on your target board, or in other words, emulates the work of the processor and provides the human interface with what exactly is happening on the board in real-time. This also allows the ICE to support powerful debugging features such as hardware breakpoints and real-time tracing.

Many other debugging tools – such as simulators, logic analyzers and oscilloscopes – are also used in embedded systems. A simulator is a completely host-based program that simulates the functionality and instruction set of the target processor. Although simulators have many disadvantages, they are quite valuable in the early stages of the project when there isn't as yet any actual hardware for the programmers to experiment with. The biggest disadvantage of a simulator is that it simulates only the processor. And embedded systems frequently contain one or more other peripherals. Interaction with these devices can only sometimes be imitated. You may not do much with the simulator once you have the actual embedded hardware available to you. Once the target hardware is available, you can use logic analyzers and oscilloscopes as debugging tools. These are very useful for debugging the interactions between the processor and other chips on the board. These tools only view signals that lie outside the processor, and cannot control the flow of execution of your software like debuggers or emulators can. A logic analyzer is equipment that is designed to find whether the electrical signal it is attached to is currently to logic level 1 or 0(zero). An oscilloscope so another piece of equipment for hardware debugging, and is used to examine any electrical signal, analogue signal, or digital signal on the hardware.

Design Requirements

Embedded systems typically have tight constraints on both functionality and implementation. In particular, they have must guarantee real time operation reactive to external events, conform to size and weight limits, budget, power and cooling consumption, satisfy safety and reliability requirements, and meet tight cost targets. Real time systems operation means that the correctness of a computation depends on the time at which it is delivered. In many cases the system design must take into account worst-case performance. The Signal Processing and Mission Critical example systems have a significant requirement for real time operation in order to meet external I/O and control stability requirements. Reactive computation means that the software executes in response to external events. These events may be periodic, in which case scheduling of events to guarantee performance may be possible. On the other hand, many events may be a periodic, in which case the maximum event arrival rate must be estimated in order to accommodate worst-case situations. Most embedded systems have a significant reactive component.

- **Small size, low weight:** Many embedded computers are physically located within small areas. In transportation and portable systems, weight may be critical for fuel economy or human endurance. Among the examples, the Mission Critical system has much more stringent size and weight requirements than the others because of its use in a flight vehicle, although all examples have restrictions of this type.

- **Safe and reliable:** Some systems have obvious risks associated with failure. In mission-critical applications such as aircraft flight control, severe personal injury or equipment damage could result from a failure of the embedded computer. Traditionally, such systems have employed multiply-redundant computers or distributed consensus protocols in order to ensure continued operation after an equipment failure However, many embedded systems that could cause personal or property damage cannot tolerate the added cost of redundancy in hardware or processing capacity needed for traditional fault tolerance techniques. This vulnerability is often resolved at the system level as discussed later.

- **Harsh environment**: Many embedded systems do not operate in a controlled environment. Excessive heat is often a problem, especially in applications involving combustion (e.g., many transportation applications). Additional problems can be caused for embedded computing by a need for protection from vibration, shock, lightning, power supply fluctuations, water, corrosion, fire, and general physical abuse. For example, in the Mission Critical example application the computer must function for a guaranteed, but brief, period of time even under non-survivable fire conditions.

- **Cost sensitivity**: Even though embedded computers have stringent requirements, cost is almost always an issue (even increasingly for military systems). Although designers of systems large and small may talk about the importance of cost with equal urgency, their sensitivity to cost changes can vary dramatically. A reason for this may be that the effect of computer costs on profitability is more a function of the proportion of cost changes compared to the total system cost, rather than compared to the digital electronics cost alone.

Application Areas

Embedded software is present in almost every electronic device you use today. There is embedded software inside your watch, cellular phone, automobile, thermostats, industrial control equipment, and scientific and medical equipment. Defense services use it to guide missiles and detect enemy aircrafts. Thus embedded systems cover such a broad range of products that generalization is difficult. Here are some broad categories:-

- **Aerospace and defense electronics (ADE)**: Astronomical research, flight safety and flight management, fire control, robotics, vehicular control.

- **Automotive electronics**: Modern cars can be sold only if they contain a significant amount of electronics. These include air bag control systems, engine control systems, anti-braking systems (ABS), air-conditioning, GPS systems, safety features, and many more.

- **Trains**: For trains, the situation is similar to the one discussed for cars and airplanes. Again, safety features contribute significantly to the total value of trains, and dependability is extremely important.

- **Broadcast and entertainment**: analogue and digital sound products, audio control systems, DVD players, digital TV, set-top boxes.

- **Telecommunication**: Mobile phones have been one of the fastest growing markets in the recent years. For mobile phones, radio frequency (RF) design, digital signal processing and low power design are key aspects.

- **Data communication**: Analogue modems, ATM broad band switches, cable modems.

- **Digital imaging**: Digital still camera, digital video cameras, fax machines, Printers, scanners.

- **Industrial measurement and control**: Building environmental control systems, industrial sensors, test & measurement devices, traffic management systems.

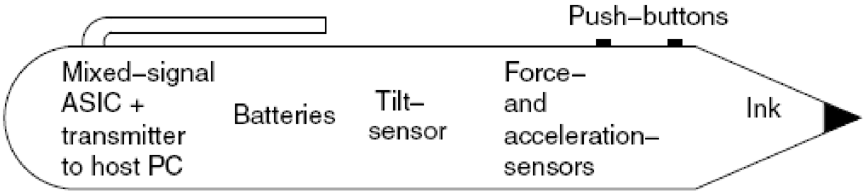
- **Medical electronics**: Cardiovascular devices, critical care systems, diagnostic devices, surgical devices.

- **Server I/O**: Embedded servers, LAN devices, supercomputing, server Management.

- **Mobile data infrastructures**: Mobile data terminals, satellites terminals, wireless LANs, pagers, wireless phones.

- **Military applications**: Information processing has been used in military equipment for many years. In fact, some of the very first computers analyzed military radar signals.

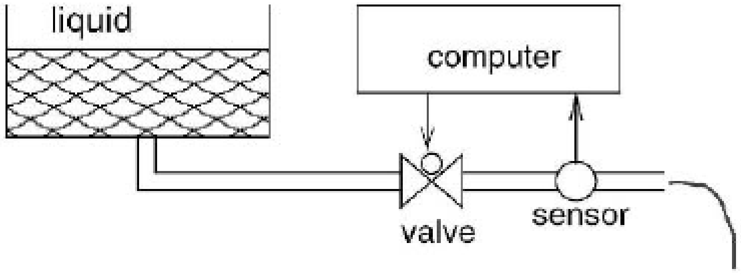
- **Authentication systems**: Embedded systems can be used for authentication purposes. For example, advanced payment systems can provide more security than classical systems. The SMART pen R_[IMEC, 1997] is an example of such an advanced payment system


Push–buttons
Mixed–signal ASIC + transmitter to host PC
Batteries
Tilt–sensor
Force– and acceleration–sensors
Ink

- The SMART pen is a pen-like instrument analyzing physical parameters while its user is signing. Physical parameters include the tilt, force and acceleration. These values are transmitted to a host PC and compared with information available about the user. As a result, it can be checked if both the image of the signature as well as the way it has been produced coincides with the stored information. Other authentication systems include finger print sensors or face recognition systems.

- **Consumer electronics**: Video and audio equipment is a very important sector of the electronics industry. The information processing integrated into such equipment is steadily growing. New services and better quality are implemented using advanced digital signal processing techniques. Many TV sets, multimedia phones, and game consoles comprise high performance processors and memory systems. They represent special cases of embedded systems.

- **Fabrication equipment**: Fabrication equipment is a very traditional area in which embedded systems have been employed for decades. Safety is very important for such systems; the energy consumption is less a problem. Below figure shows a container connected to a pipe. The pipe includes a valve and a sensor. Using the readout from the sensor, a computer may have to control the amount of liquid leaving the pipe.



- **Smart buildings**: Information processing can be used to increase the comfort level in buildings, can reduce the energy consumption within buildings, and can improve safety and security. Subsystems which traditionally were unrelated have to be connected for this purpose. There is a trend towards integrating air-conditioning, lighting, access control, accounting and distribution of information into a single system.

- **Robotics**: Robotics is also a traditional area in which embedded systems have been used. Mechanical aspects are very important for robots. Most of the characteristics described above also apply to robotics. Recently, some new kinds of robots, modeled after animals or human beings, have been designed. Below Figure shows such a robot.

- **Information Appliance**: In the past, embedded systems allowed information appliances to carry out simple and specific functions only. But with the penetration of the Internet into the homes of many ordinary families, it was realized that electric appliances could make human life easier and more convenient if they could access Internet information. Electric appliances can now access the Internet, compute and do what they were not able to do earlier. In other words, electric appliances are being transformed into information appliances (IA) or what may also be called ‘embedded IA’. Like the traditional embedded systems, the embedded information appliance needs only the least amount of hardware to operate. It can operate even without a hard disk, or with low power and small footprint. A product can be classified into four mainstream products:-
    - Set-Top Boxes (STB)
    - Personal Access Device (PAD)
    - Thin Client (TC)
    - Residential Gateway (or Home Gateway)

Most industrial appliances products may be derived, with little or some modifications, from these four types of products.

- **Set-Top Boxes:** The set-top box is driving the digital revolution right into your living room. Your fingertips now command a wealth of high quality digital information and digital entertainment, right from your favorite armchair. The set-top box revolutionizes home entertainment by providing vibrant television images with crystal clear sound, along with e-mail, Web surfing, along with customized information such as stock quotes, weather and traffic updates, on-line shopping, and video-on-demand, right through a traditional television.

- **Personal Access Devices**: Personal Access Devices (PADs) are web terminals that feature convenient Web browsing, email, and information access capabilities in a lightweight, mobile form.

- **Thin Client**: A thin client is an information access drive that provides users with remote access to applications and data that are maintained and executed on a central server. The thin-client computing environment consists of an application server, a network, and thin-client devices. By centralizing deployment and updates of corporate applications, thin clients allow for simplified Information Systems (IS) management with dramatically increased security.

- **Residential Gateway:** The RG mainly provides various kinds of interfaces that link all the electronic devices. The RG unlike the PC is a very small, slim and light piece of hardware and may soon be incorporated inside other popular electronic appliances. It will play the role of an information hub responsible for the exchange of information between all kinds of electronic devices in an ordinary home.

Conclusion

We are standing on the threshold of an exciting new age of information technology that will change our lives and the future forever. Soon we shall see more and more digitization of appliances, and these will be fuelled by human need. Embedded systems and Information Appliances have virtually entered every sphere of our life and they will truly change the way we live.

Summary

1. Embedded system is a combination of hardware and software designed for a specific application.

2. Operating systems used in embedded applications is real time operating systems.

3. RTOS are classified into 3 categories known as hard, soft and hybrid real time operating systems.

4. The main difference between embedded processors and other processors is availability of resources on chip.

5. The main disadvantage of an embedded system will have very few resources when compared to full blown computing systems like a desktop computer, the memory capacity and processing power in an embedded system is limited.

Questions:

1. What is an embedded system?
2. What is the difference between embedded device and a personal computer?
3. What are the characteristics of embedded systems?
4. What are the design requirements of embedded systems?
5. Explain the different fields of embedded systems?

References:

1. www.embedded.com
2. www.wikipedia.com
3. Embedded Software The Works – colin walls

3. Embedded System Technologies

Objective

In this chapter the reader will get an idea about different technologies that are used during the design of embedded systems.

Types of Technologies

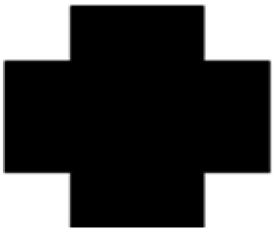
Technology is a manner of accomplishing a task, especially using technical processes, methods, or knowledge. Three key technologies for embedded systems are

- Processor technology
- IC technology
- Design technology

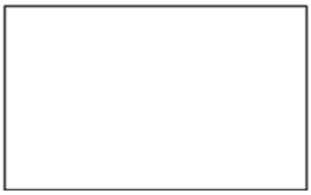
With the help of above three technologies engineers are enabled to design different types of processors those can be used for general purpose, customized and application specific processors.

Processor Technology

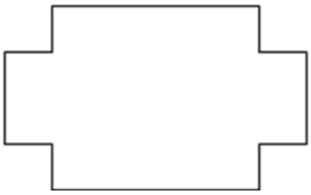
Generally processors are designed depend on their application areas. Processors can be divided into three categories depend on their type of usage. Those can be named as general purpose, single purpose, and customized application processors (For example DSP processors used generally in multimedia electronic gadgets, ARM processors used generally in SONY Ericson mobiles etc). Let us assume below diagram represents a block of required application.



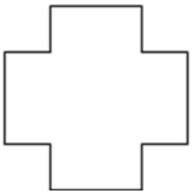
Then according to the processors selections below diagrams represents the general purpose, application specific and customized processors.



General purpose



Application Specific

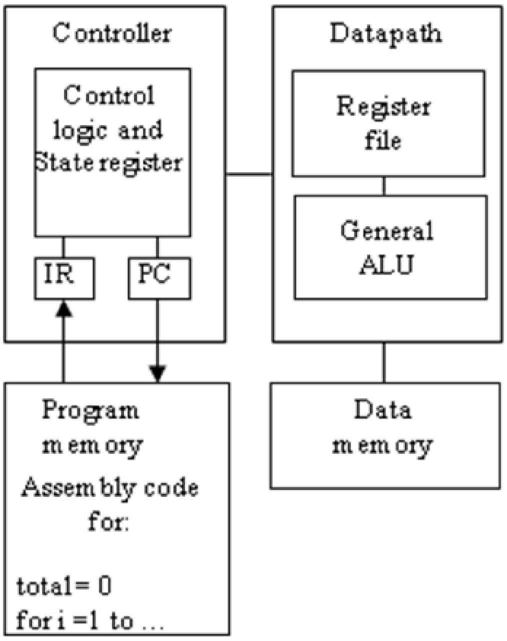


Customized

From above diagrams reader can conclude that general purpose processors are suitable for all types of applications where resources wastage is high. When come to application specific processors the resource wastage is low when compared with general purpose and the same is almost zero in customized processor.

General purpose Processors

Below Figure1 represents the Block diagram of general purpose processor.



General-purpose ("software")

Figure 2

The main parts of any processor include control unit, arithmetic logic unit (ALU), registers, data memory, program memory etc. In the above type of processors software used is general purpose software. These processors are known micro processors which can be used for many different application fields depending on program stored in program memory.

Features of these processors are

- Program memory
- General data path with large register file and general ALU

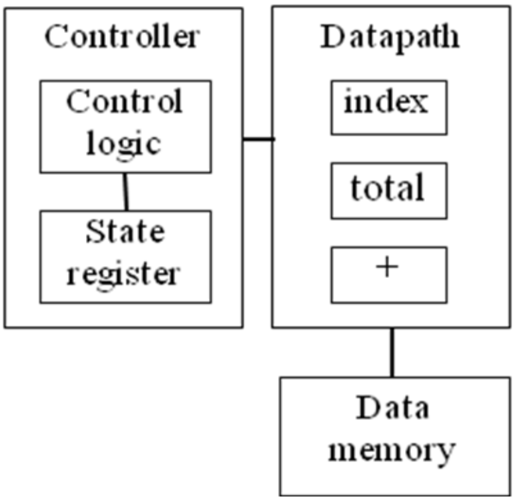
The benefits of this type of processors are

- Low time-to-market and NRE costs
- High flexibility

The examples of these type of processors are Intel Pentium processors (P-I, P-II, P-III, P-IV, CORE2), AMD processors (AMD ATHLON 64), MOTOROLA processors etc.

Single purpose Processors

Below Figure represents the Block diagram of general purpose processor.



The main parts of this type processor include control unit, arithmetic logic unit (ALU), registers, data memory etc. This is chip which contains a simple digital circuit designed to serve only one single user specified application.

Features of these processors are

- Contains only the components needed to execute a single program
- No program memory

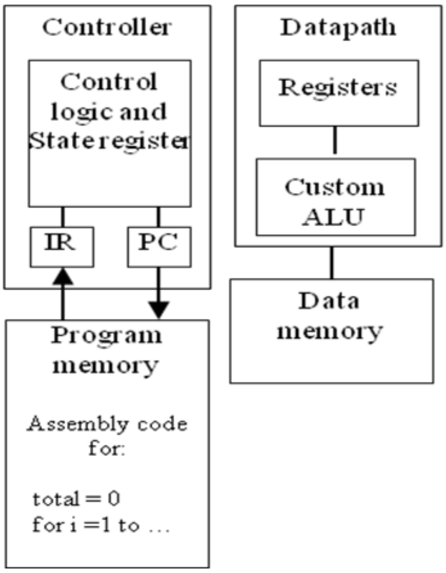
The benefits of this type of processors are

- Fast
- Low power
- Small size

MAX232 is an IC (Integrated Chip) used only for serial communication between different devices either in synchronous or asynchronously. This IC can’t be used for any other application. This type of chips / Processors designed to serve only one single purpose is known as single purpose processors.

Application Specific Processors

Below Figure represents the Block diagram of general purpose processor.



The main parts of this type processor include control unit, arithmetic logic unit (ALU), registers, data memory and program memory like general purpose. These processors are a part of general purpose processors. But this type processor is designed in such a way that these can be used only for some specific applications only.

Features of these processors are

- Program memory
- Optimized data path
- Special functional units

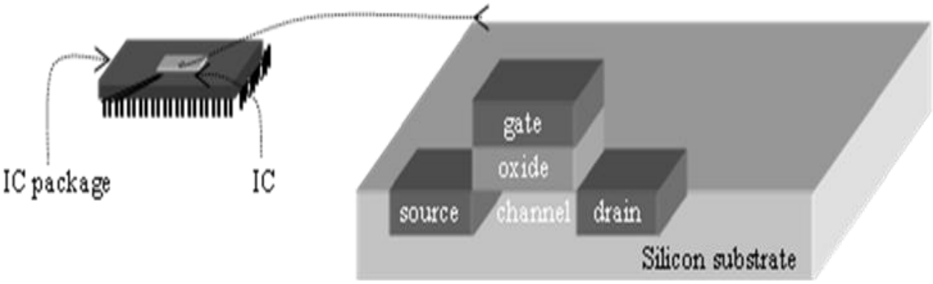
The benefits of this type of processors are

- Some flexibility
- good performance
- size and power

Digital signal processors are used mainly in signal processing very fastly and effectively. These are mainly used in graphics cards, data processing units etc. This type of chips / Processors designed to serve only one specific type of application field.

IC technology

The technology that deals with the gate level implementation and mapping of the chips is IC Technology. The full form of IC is integrated circuit or integrated chip. There exist different IC technologies depending on IC customization. A single IC may consist of single layer on more than one. The number of layers depends on the complexity, design process of the IC. IC technologies differ with respect to who builds each layer and when it was made, how many transistors are used for each layer etc. figure represents the very basic IC design methodology.



There exists mainly three type of IC technologies are present. Those are

- Full-custom/VLSI
- Semi-custom ASIC (gate array and standard cell)
- PLD (Programmable Logic Device)

Full-custom/VLSI

In this method each and every part of the IC is designed by the designer according to the application requirement. In this technology every feature of processor like number of transistors on the chip, number of layers etc are designed to serve the required task. In this design all layers are optimized for an embedded system’s particular digital implementation like

- Placing transistors
- Sizing transistors
- Routing wires

Benefits of this technology are

- Excellent performance
- small size
- low power

The main drawback of this system is its initial cost and long time to market.

Semi-custom/VLSI

In this method layers are pre designed and wiring has to be done according to the application requirement. In this technology lower layers of the IC are fully or partially build. Designers are left with routing of wires and maybe placing some blocks on the layers of IC. Advantage with this technology is

- Good performance
- Good size
- Less initial cost than a full-custom implementation

The main drawback of this technology is time requirement for its development.

Programmable Logic Devices

In this technology all layers are exists in the IC. Designer either creates or destroys the connections on IC to get desired functionality. In these programmable logic devices FPGA (Field-Programmable Gate Arrays) are most popular.

- Benefits
  - Low NRE costs, almost instant IC availability
- Drawbacks
  - Bigger, expensive (perhaps $30 per unit), power hungry, slower

Moore’s Law

In 1959, Calvin Moore’s, one of the pioneers of Information Retrieval, set forth what he called a "contradictory principle" of the fledgling science, and attached his own name to it:

Moore’s Law: An information retrieval system will tend not to be used whenever it is more painful and troublesome for a customer to have information than for him not to have it.

An interesting thing has happened to Moore’s’ Law, however, along the way to acceptance: the law that is becoming widely held as true by information professionals is not the same one that Mr. Moore’s proposed.

The difference between the actual law and its mutation centers specifically upon a misinterpretation of the word "have", a misinterpretation which perhaps results primarily from reading the law excerpted from the original article in which it appeared, and applying it to a concern that is distinctly separate from the one Mr. Moore’s was attempting to address.

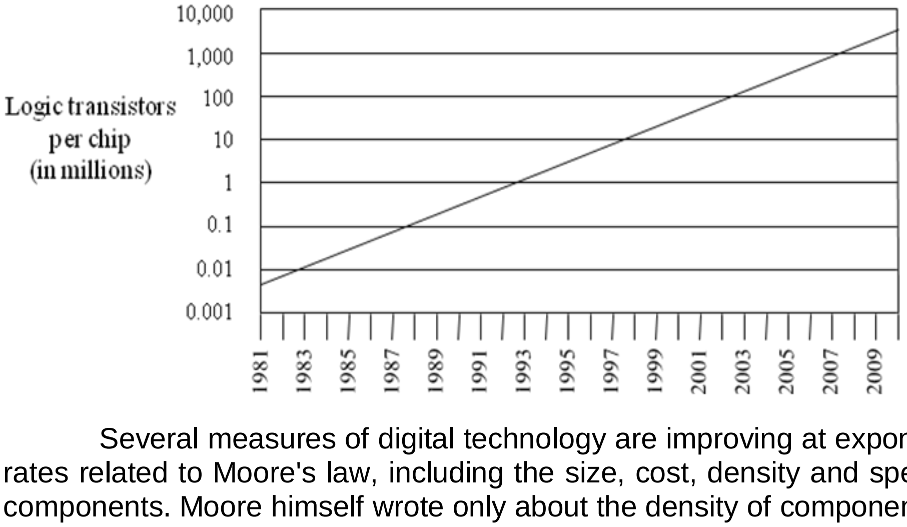
Moore’s’ Law Expanded

It seems that Moore’s, in spite of the attention he focuses upon environments in which IR systems tend not to be used, was unaware of the rest of the scale; his observations about "the best of the chemical or pharmaceutical laboratories" acknowledge the opposite extreme, while the very study in which his law is first mentioned seems to exist primarily to address the needs of those environments that are in between. Indeed, early on in this Seven Model Systems study, he makes a statement that very closely resembles what his law has now become: "If the burden on the users of the information becomes too high, either in the retrieval process or in the labor of delineating new material, the users will give the system up and try to get along without it" (1959a, p.6). Clearly, though, this statement has no meaning within either of the environments which lie at the extreme ends of the scale; it makes sense only if applied to the middle. Thus, it would seem that an expansion of Mooers’ original law would be in order:

Moore’s’ 1st Law: In an environment in which it is more painful and troublesome for a customer to have information in hand than for him not to have it, an IR system will tend not to be used.

Moore’s’ 2nd Law: In an environment in which it is absolutely critical for a customer to have information, an IR system, no matter how poorly designed, will tend to be used.

Moore’s’ 3rd Law: In an environment in which the trouble of having information versus that of not having it are fairly evenly balanced, system design and performance tend to be the deciding factors in whether or not an IR system will be used.



Several measures of digital technology are improving at exponential rates related to Moore's law, including the size, cost, density and speed of components. Moore himself wrote only about the density of components (or transistors) at minimum cost. Moore's law has been the name given to everything that changes exponentially.

Transistors per integrated circuit. The most popular formulation is of the doubling of the number of transistors on integrated circuits every two years. At the end of the 1970s, Moore's law became known as the limit for the number of transistors on the most complex chips. Recent trends show that this rate has been maintained into 2007.

Density at minimum cost per transistor. This is the formulation given in Moore's 1965 paper. It is not about just the density of transistors that can be achieved, but about the density of transistors at which the cost per transistor is the lowest. As more transistors are put on a chip, the cost to make each transistor decreases, but the chance that the chip will not work due to a defect increases. In 1965, Moore examined the density of transistors at which cost is minimized, and observed that, as transistors were made smaller through advances in photolithography, this number would increase at "a rate of roughly a factor of two per year".

Cost per transistor. As the size of transistors has decreased, the cost per transistor has decreased as well. However, the *manufacturing cost per unit area* has only increased over time, since materials and energy expenditures per unit area have only increased with each successive technology node.

Computing performance per unit cost. Also, as the size of transistors shrinks, the speed at which they operate increases. It is also common to cite Moore's law to refer to the rapidly continuing advance in computing performance per unit cost, because increase in transistor count is also a rough measure of computer processing performance. On this basis, the performance of computers per unit cost—or more colloquially, "bang per buck"—doubles every 24 months.

Power consumption. The power consumption of compute nodes doubles every 18 months.

Hard disk storage cost per unit of information. A similar law (sometimes called Kryder's Law) has held for hard disk storage cost per unit of information. The rate of progression in disk storage over the past decades has actually sped up more than once, corresponding to the utilization of error correcting codes, the magneto resistive effect and the giant magneto resistive effect. The current rate of increase in hard drive capacity is roughly similar to the rate of increase in transistor count. Recent trends show that this rate has been maintained into 2007.

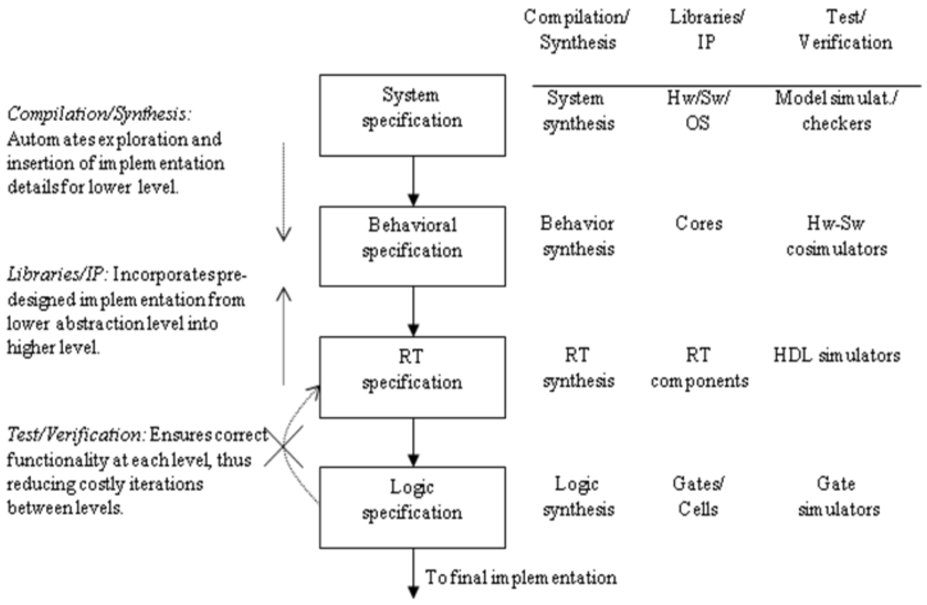
RAM storage capacity. Another version states that RAM storage capacity increases at the same rate as processing power.

Network capacity According to Gerry/Gerald Butters,[21][22] the former head of Lucent's Optical Networking Group at Bell Labs, there is another version, called Butter's Law of Photonics, a formulation which deliberately parallels Moore's law. Butter's law says that the amount of data coming out of an optical fiber is doubling every nine months. Thus, the cost of transmitting a bit over an optical network decreases by half every nine months. The availability of wavelength-division multiplexing (sometimes called "WDM") increased the capacity that could be placed on a single fiber by as much as a factor of 100. Optical networking and DWDM is rapidly bringing down the cost of networking, and further progress seems assured. As a result, the wholesale price of data traffic collapsed in the dot-com bubble. Nielsen's Law says that the bandwidth available to users increases by 50% annually.

Pixels per dollar. Similarly, Barry Hendy of Kodak Australia has plotted the "pixels per dollar" as a basic measure of value for a digital camera, demonstrating the historical linearity (on a log scale) of this market and the opportunity to predict the future trend of digital camera price and resolution.

Design Technology

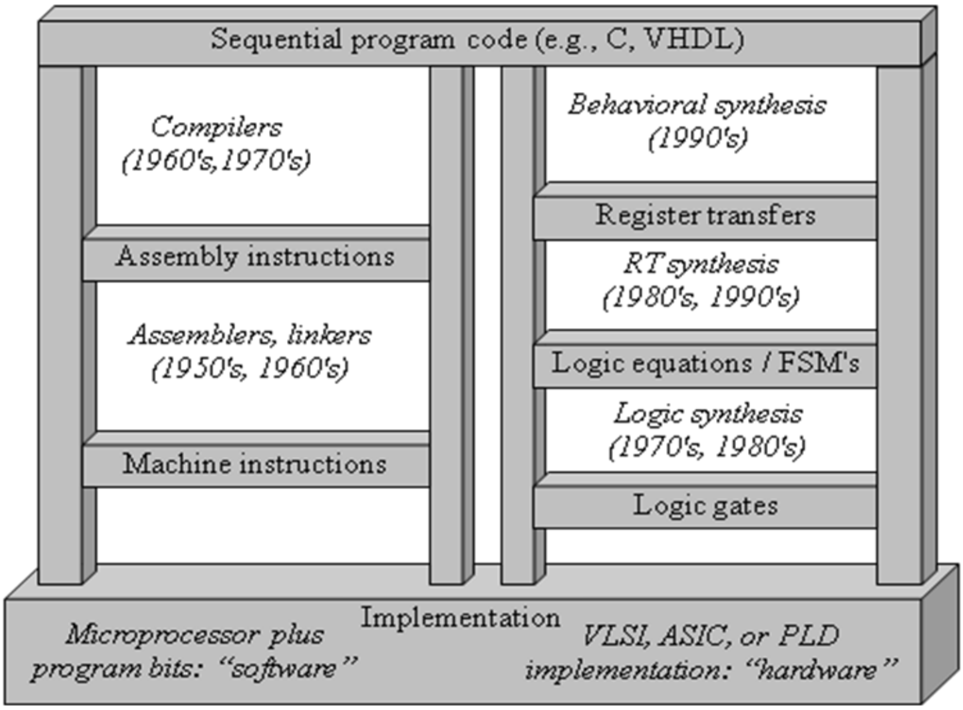
In this technology reader will understand how to convert user specified application into integrated chip. Below diagram represents the entire process of conversion.



The co-design ladder

- In the past:
  - Hardware and software design technologies were very different
  - Recent maturation of synthesis enables a unified view of hardware and software
- Hardware/software “co design”

Below diagram shows the comparison of IC design methodology.



Summary:

- Embedded systems are everywhere
- Key challenge: optimization of design metrics
  - Design metrics compete with one another
- A unified view of hardware and software is necessary to improve productivity
- Three key technologies
  - Processor: general-purpose, application-specific, single-purpose
  - IC: Full-custom, semi-custom, PLD
  - Design: Compilation/synthesis, libraries/IP, test/verification
  -

Review Questions

1. Explain key technologies used in embedded systems each in around 500 words.

2. What is the difference between single prpose and multipurpose processors.

3. Write a brief notes on moore,s law?

References:

1. Designing embedded hardware, john catsoulis.
2. Embedded_Controller_Hardware_Design,ken arnold.
3. www.embedded.com
4. First Steps with Embedded Systems.

4. Custom single-purpose processors: Hardware

Introduction

As mentioned in the previous chapter, a single-purpose processor is a digital system intended to solve a specific computation task. While a manufacturer builds a standard single-purpose processor for use in a variety of applications, we build a custom single purpose processor to execute a specific task within our embedded system. An embedded system designer choosing to use a custom single-purpose, rather than a general-purpose, processor to implement part of a system's functionality may achieve several benefits, similar to some of those of the previous chapter.

First, performance may be fast, due to fewer clock cycles resulting from a customized data path, and due to shorter clock cycles resulting from simpler functional units, less multiplexors, or simpler control logic. Second, size may be small, due to a simpler data path and no program memory. In fact, the processor may be faster and smaller than a standard one implementing the same functionality, since we can optimize the implementation for our particular task.

However, because we probably won't manufacture as many of the custom processor as a standard processor, we may not be able to invest as much NRE, unless the embedded system we are building will be sold in large quantities or does not have tight cost constraints. This fact could actually penalize performance and size.

In this chapter, we describe basic techniques for designing custom processors. We start with a review of combinational and sequential design, and then describe a method for converting programs to custom single-purpose processors.

Combinational logic design

A transistor is the basic electrical component of digital systems. Combinations of transistors form more abstract components called logic gates, which designers primarily use when building digital systems. Thus, we begin with a short description of transistors before discussing logic design.

A transistor acts as a simple on/off switch. One type of transistor (CMOS -- Complementary Metal Oxide Semiconductor) is shown in Figure 4.1(a). The *gate* (not to be confused with logic gate) controls whether or not current flows from the *source* to the *drain*. When a high voltage (typically +5 Volts, which we'll refer to as logic 1) is applied to the gate, the transistor conducts so current flows. When low voltage (which we'll refer to as logic 0, typically ground, which is drawn as several horizontal lines of decreasing width) is applied to the gate, the transistor does not conduct. We can also build a transistor with the opposite functionality, illustrated in Figure 4.1(b). When logic 0 is applied to the gate, the transistor conducts, and when logic 1 is applied, the transistor does not conduct. Given these two basic transistors, we can easily build a circuit whose output inverts its gate input, as shown in in Figure 4.1(c). When the input *x* is logic 0, the top transistor conducts (and the bottom does not), so logic 1 appears at the output *F*. We can also easily build a circuit whose output is logic 1 when at least one of its inputs is logic 0, as shown in Figure 4.1(d). When at least one of the inputs *x* and *y* is logic 0, then at least one of the top transistors conducts (and the bottom transistors do not), so logic 1 appears at *F*. If both inputs are logic 1, then neither of the top transistors conducts, but both of the bottom ones do, so logic 0 appears at *F*. Likewise, we can easily build a circuit whose output is logic 1 when both of its inputs are logic 0, as illustrated in Figure 4.1(e). The three circuits shown implement three basic logic gates: an inverter, a NAND gate, and a NOR gate.

Figure 4.1: CMOS transistor implementations of some basic logic gates: (a) nMOS transistor, (b) pMOS transistor, (c) inverter, (d) NAND gate, (e) NOR gate.

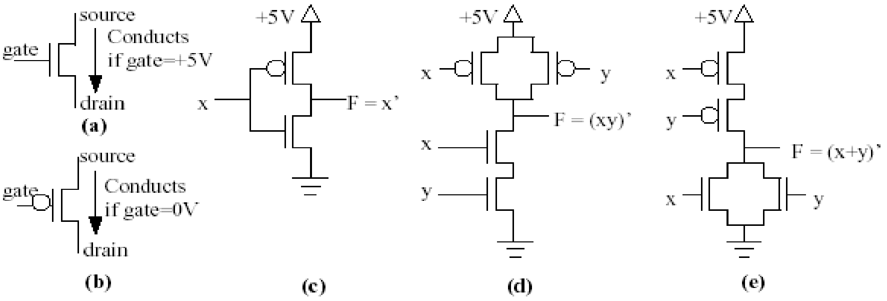
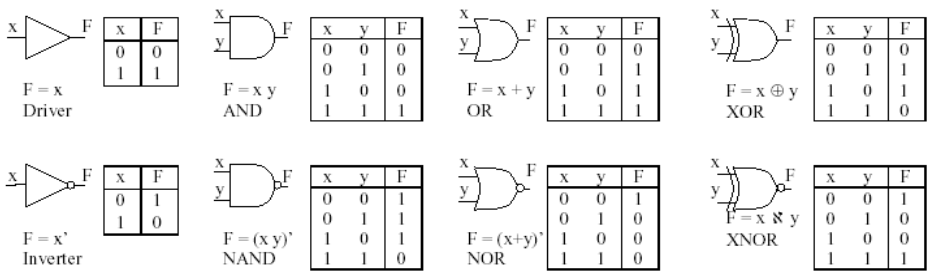


Figure 4.2: Basic logic gates



x	F
0	0
1	1

x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

x	F
0	1
1	0

x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

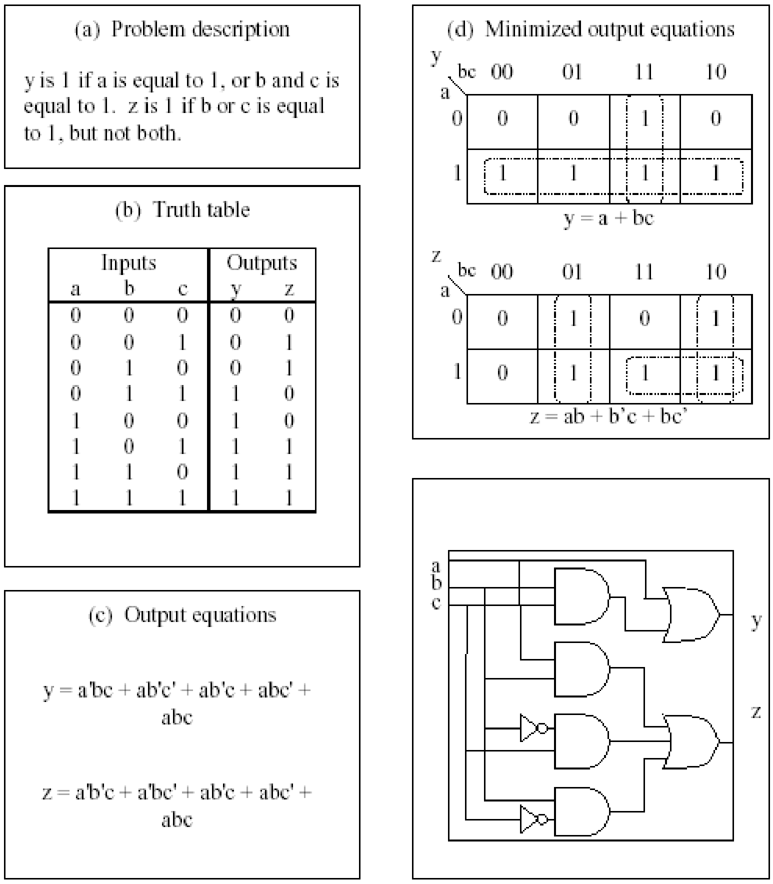
x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

Digital system designers usually work with logic gates, not transistors. Figure 4.2 describes 8 basic logic gates. Each gate is represented symbolically, with a Boolean equation, and with a truth table. The truth table has inputs on the left, and output on the right. The AND gate outputs 1 if and only if both inputs are 1. The OR gate outputs 1 if and only if at least one of the inputs is 1. The XOR (exclusive-OR) gate outputs 1 if and only if exactly one of its two inputs is 1. The NAND, NOR, and XNOR gates output the complement of AND, OR, and XOR, respectively. As you might have noticed from our transistor implementations, the NAND and NOR gates are actually simpler to build than AND and OR gates.

A *combinational* circuit is a digital circuit whose output is purely a function of its current inputs; such a circuit has no memory of past inputs.

We can apply a simple technique to design a combinational circuit using our basic logic gates, as illustrated in Figure 4.3. We start with a problem description, which describes the outputs in terms of the inputs. We translate that description to a truth table, with all possible combinations of input values on the left, and desired output values on the right. For each output column, we can derive an output equation, with one term per row. However, we often want to minimize the logic gates in the circuit. We can minimize the output equations by algebraically manipulating the equations. Alternatively, we can use Karnaugh maps, as shown in the figure. Once we've obtained the desired output equations (minimized or not), we can draw the circuit diagram.

Figure 4.3: Combinational logic design.



(a) Problem description

y is 1 if a is equal to 1, or b and c is equal to 1. z is 1 if b or c is equal to 1, but not both.

(b) Truth table

Inputs			Outputs	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

(c) Output equations

y = a'bc + ab'c' + ab'c + abc' + abc

z = a'b'c + a'bc' + ab'c + abc' + abc

(d) Minimized output equations

y: a \ bc	00	01	11	10
0	0	0	1	0
1	1	1	1	1

y = a + bc

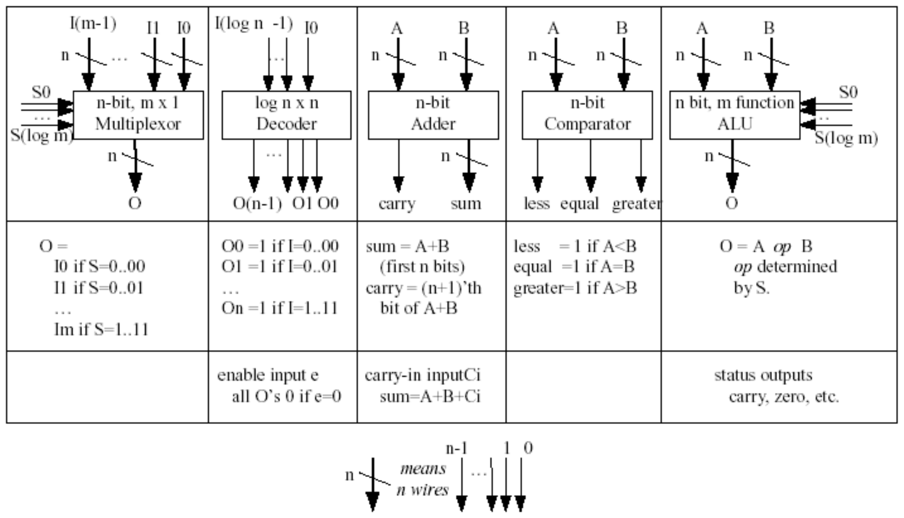
z: a \ bc	00	01	11	10
0	0	1	0	1
1	0	1	1	1

z = ab + b'c + bc'

Although we can design all combinational circuits in the above manner, large circuits would be very complex to design. For example, a circuit with 16 inputs would have 2^{16}, or 64K, rows in its truth table. One way to reduce the complexity is to use components that are more abstract than logic gates. Figure 4.4 shows several such combinational components. We now describe each briefly.

A *multiplexor*, sometimes called a selector, allows only one of its data inputs *Im* to pass through to the output *O*. Thus, a multiplexor acts much like a railroad switch, allowing only one of multiple input tracks to connect to a single output track. If there are *m* data inputs, then there are *log₂(m)* select lines *S*, and we call this an m-by-1 multiplexor (*m* data inputs, one data output). The binary value of *S* determines which data input passes through; 00...00 means *I0* may pass, 00...01 means *I1* may pass, 00...10 means *I2* may pass, and so on. For example, an 8x1 multiplexor has 8 data inputs and thus 3 select lines. If those three select lines have values of 110, then *I6* will pass through to the output. So if *I6* is 1, then the output would be 1; if *I6* is 0, then the output would be 0. We commonly use a more complex device called an *n*-bit multiplexor, in which each data input, as well as the output, consists of *n* lines. Suppose the previous example used a 4-bit 8x1 multiplexor. Thus, if *I6* is 0110, then the output would be 0110. Note that *n* does not affect the number of select lines.

Figure 4.4: Combinational components.



O =
I0 if S=0..00
I1 if S=0..01
…
Im if S=1..11
	O0 =1 if I=0..00
	O1 =1 if I=0..01
	…
	On =1 if I=1..11
		sum = A+B
		(first n bits)
		carry = (n+1)'th
		bit of A+B
			less = 1 if A<B
			equal =1 if A=B
			greater=1 if A>B
				O = A *op* B
				op determined
				by S.

	enable input e all O's 0 if e=0	carry-in inputCi sum=A+B+Ci		status outputs carry, zero, etc.

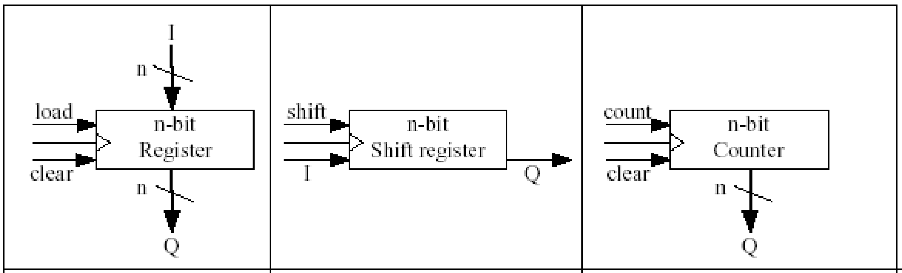
A *decoder* converts its binary input *I* into a one-hot output *O*. "One-hot" means that exactly one of the output lines can be 1 at a given time. Thus, if there are *n* outputs, then there must be *log₂(n)* inputs. We call this a *log₂(n)* × *n* decoder. For example, a 3x8 decoder has 3 inputs and 8 outputs. If the input is 000, then the output *O0* will be 1. If the input is 001, then the output *O1* would be 1, and so on. A common feature on a decoder is an extra input called *enable.* When enable is 0, all outputs are 0. When enable is 1, the decoder functions as before.

An adder adds two n-bit binary inputs A and B, generating an n-bit output sum along with an output carry. For example, a 4-bit adder would have a 4-bit A input, a 4-bit B input, a 4-bit sum output, and a 1-bit carry output. If A is 1010 and B is 1001, then sum would be 0011 and carry would be 1.

A comparator compares two n-bit binary inputs A and B, generating outputs that indicate whether A is less than, equal to, or greater than B. If A is 1010 and B is 1001, then less would be 0, equal would be 0, and greater would be 1.

An *ALU* (arithmetic-logic unit) can perform a variety of arithmetic and logic functions on its n-bit inputs *A* and *B.* The select lines *S* choose the current function; if there are *m* possible functions, then there must be at least $log_2(m)$ select lines. Common functions include addition, subtraction, AND, and OR.

Figure 4.5: Sequential components.



Q = 0 if clear=1, I if load=1 and clock 8, I(prev) else.		Q = 0 if clear=1, Q(prev)+1 if count=1 and clock 8.

Sequential logic design

A sequential circuit is a digital circuit whose outputs are a function of the current as well as previous input values. In other words, sequential logic possesses memory. One of the most basic sequential circuits is the flip-flop. A flip-flop stores a single bit. The simplest type of flip-flop is the D flip-flop. It has two inputs: D and clock. When clock is 1, the value of D is stored in the flip-flop, and that value appears at an output Q. When clock is 0, the value of D is ignored; the output Q maintains its value. Another type of flip-flop is the SR flip-flop, which has three inputs: S, R and clock. When clock is 0, the previously stored bit is maintained and appears at output Q. When clock is 1, the inputs S and R are examined. If S is 1, a 1 is stored. If R is 1, a 0 is stored. If both are 0, there's no change. If both are 1, behavior is undefined. Thus, S stands for set and R for reset. Another flip-flop type is a JK flip-flop, which is the same as an SR flip-flop except that when both J and K are 1, the stored bit toggles from 1 to 0 or 0 to 1. To prevent unexpected behavior from signal glitches, flip-flops are typically designed to be edge triggered, meaning they only pay attention to their non-clock inputs when the clock is rising from 0 to 1, or alternatively when the clock is falling from 1 to 0.

Just as we used more abstract combinational components to implement complex combinational systems, we also use more abstract sequential components for complex sequential systems. Figure 4.5 illustrates several sequential components, which we now describe.

A register stores n bits from its n-bit data input I, with those stored bits appearing at its output O. A register usually has at least two control inputs, clock and load. For a rising-edge-triggered register, the inputs I are only stored when load is 1 and clock is rising from 0 to 1. The clock input is usually drawn as a small triangle, as shown in the figure. Another common register control input is clear, which resets all bits to 0, regardless of the value of I. Because all n bits of the register can be stored in parallel, we often refer to this type of register as a parallel-load register, to distinguish it from a shift register, which we now describe.

A shift register stores n bits, but these bits cannot be stored in parallel. Instead, they must be shifted into the register serially, meaning one bit per clock edge. A shift register has a one-bit data input I, and at least two control inputs clock and shift. When clock is rising and shift is 1, the value of I is stored in the (n)'th bit, while the (n)'th bit is stored in the (n-1)'th bit, and likewise, until the second bit is stored in the first bit. The first bit is typically shifted out, meaning it appears over an output Q.

A counter is a register that can also increment (add binary 1) to its stored binary value. In its simplest form, a counter has a clear input, which resets all stored bits to 0, and a count input, which enables incrementing on the clock edge. A counter often also has a parallel load data input and associated control signal. A common counter feature is both up and down counting (incrementing and decrementing), requiring an additional control input to indicate the count direction.

The control inputs discussed above can be either synchronous or asynchronous. A synchronous input's value only has an effect during a clock edge. An asynchronous input's value affects the circuit independent of the clock. Typically, clear control lines are asynchronous.

Sequential logic design can be achieved using a straightforward technique, whose steps are illustrated in Figure 4.1. We again start with a problem description. We translate this description to a state diagram. We describe state diagrams further in a later chapter. Briefly, each state represents the current "mode" of the circuit, serving as the circuit's memory of past input values. The desired output values are listed next to each state. The input conditions that cause a transition from one state to another are shown next to each arc. Each arc condition is implicitly AND'ed with a rising (or falling) clock edge. In other words, all inputs are synchronous. State diagrams can also describe asynchronous systems, but we do not cover such systems in this book, since they are not common.

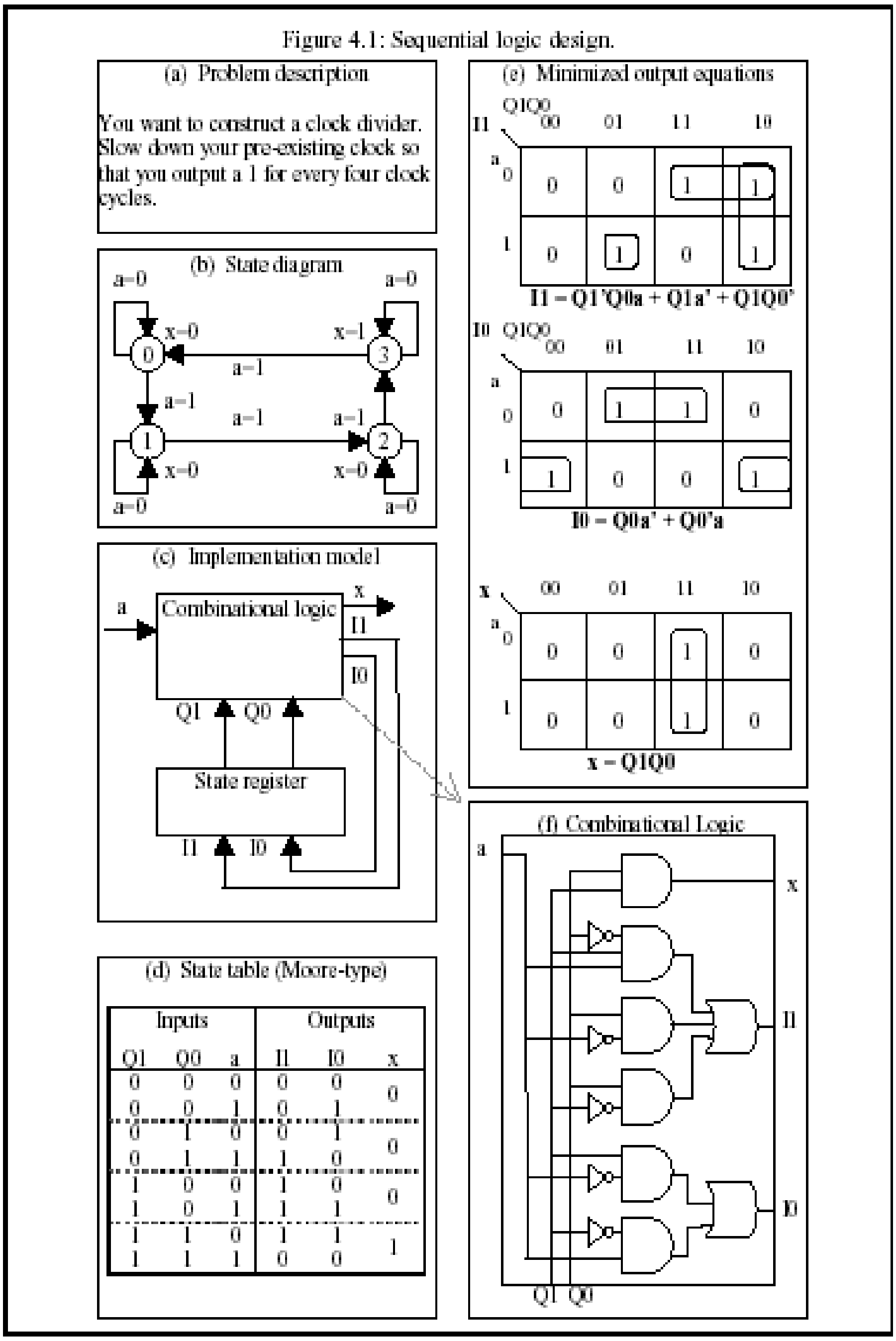
We will implement this state diagram using a register to store the current state, and combinational logic to generate the output values and the next state. We assign each state with a unique binary value, and we then create a truth table for the combinational logic. The inputs for the combinational logic are the state bits coming from the state register, and the external inputs, so we list all combinations of these inputs on the left side of the table. The outputs for the combinational logic are the state bits to be loaded into the register on the next clock edge (the next state), and the external output values, so we list desired values of these outputs for each input combination on the right side of the table. Because we used a state diagram for which outputs were functions of the current state only, and not of the inputs, we list an external output value only for each possible state, ignoring the external input values. Now that we have a truth table, we proceed with combinational logic design as described earlier, by generating minimized output equations, and then drawing the combinational logic circuit.

Custom single-purpose processor design

We can apply the above combinational and sequential logic design techniques to build data path components and controllers. Therefore, we have nearly all the knowledge we need to build a custom single-purpose processor for a given program, since a processor consists of a controller and a data path. We now describe a technique for building such a processor.

We begin with a sequential program we must implement. Figure provides a example based on computing a greatest common divisor (GCD). Figure 4.3(a) shows a black-box diagram of the desired system, having *x_i* and *y_i* data inputs and a data output *d_i*. The system's functionality is straightforward: the output should represent the GCD of the inputs. Thus, if the inputs are 12 and 8, the output should be 4. If the inputs are 13 and 5, the output should be 1. Figure 4.3(b) provides a simple program with this functionality. The reader might trace this program's execution on the above examples to verify that the program does indeed compute the GCD.

Figure 4.1: Sequential logic design.



(a) Problem description

You want to construct a clock divider. Slow down your pre-existing clock so that you output a 1 for every four clock cycles.

(b) State diagram

(c) Implementation model

(d) State table (Moore-type)

Inputs			Outputs		
Q1	Q0	a	I1	I0	x
0	0	0	0	0	0
0	0	1	0	1	
0	1	0	0	1	0
0	1	1	1	0	
1	0	0	1	0	0
1	0	1	1	1	
1	1	0	1	1	1
1	1	1	0	0	

(e) Minimized output equations

I1 a \ Q1Q0	00	01	11	10
0	0	0	1	1
1	0	1	0	1

I1 = Q1'Q0a + Q1a' + Q1Q0'

I0 a \ Q1Q0	00	01	11	10
0	0	1	1	0
1	1	0	0	1

I0 = Q0a' + Q0'a

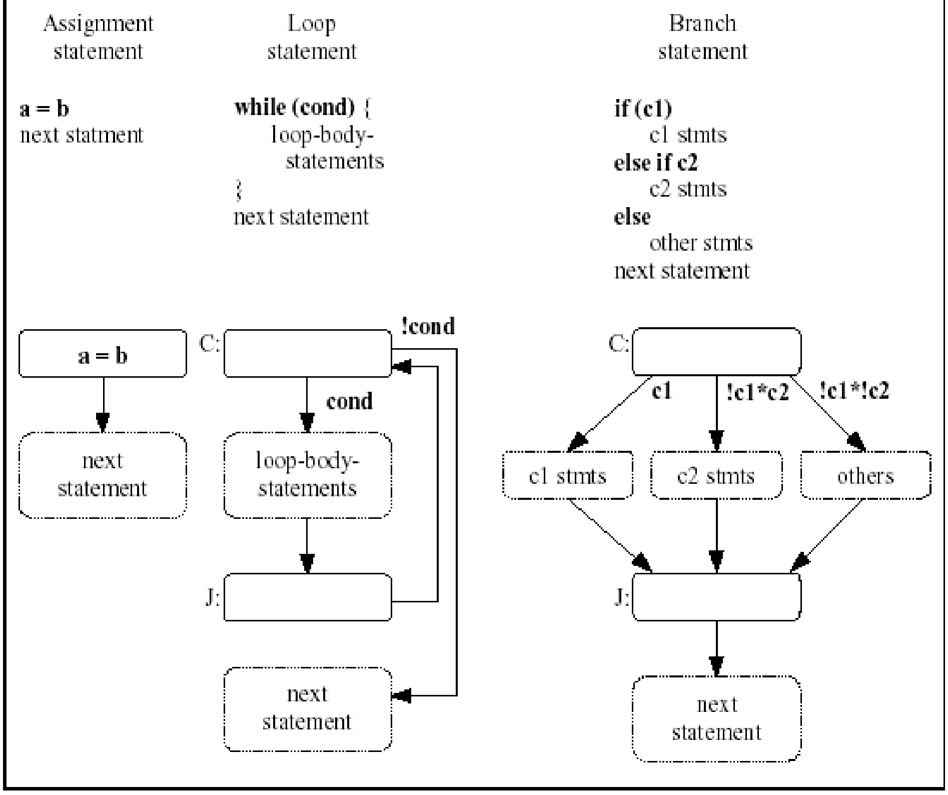
x a \	00	01	11	10
0	0	0	1	0
1	0	0	1	0

x = Q1Q0

(f) Combinational Logic

To begin building our single-purpose processor implementing the GCD program, we first convert our program into a complex state diagram, in which states and arcs may include arithmetic's expressions, and these expressions may use external inputs and outputs or variables. In contrast, our earlier state diagrams only included Boolean expressions, and these expressions could only use external inputs and outputs, not variables. Thus, these more complex state diagram looks like a sequential program in which statements have been scheduled into states.

Figure 4.2: Templates for creating a state diagram from a program.



```
a = b
next statment
```

```
while (cond) {
    loop-body-
      statements
}
next statement
```

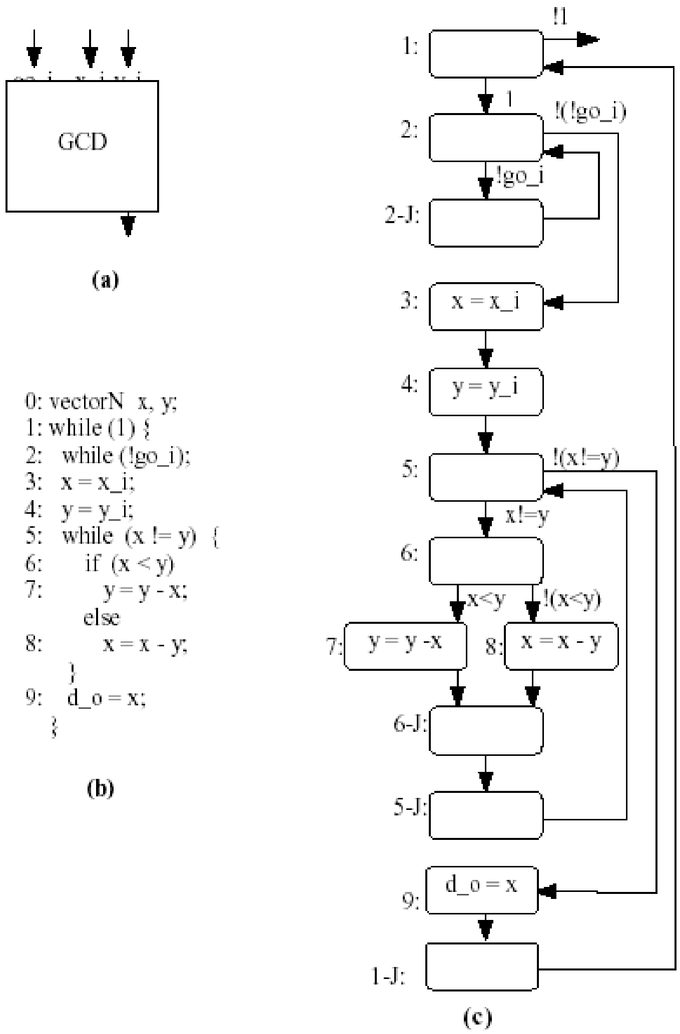
```
if (c1)
    c1 stmts
else if c2
    c2 stmts
else
    other stmts
next statement
```

We can use templates to convert a program to a state diagram, as illustrated in Figure 4.2. First, we classify each statement as an assignment statement, loop statement, or branch (if-then-else or case) statement. For an assignment statement, we create a state with that statement as its action. We add an arc from this state to the state for the next statement, whatever type it may be. For a loop statement, we create a condition state *C* and a join state *J*, both with no actions. We add an arc with the loop's condition from the condition state to the first statement in the loop body. We add a second arc with the complement of the loop's condition from the condition state to the next statement after the loop body. We also add an arc from the join state back to the condition state. For a branch statement, we create a condition state *C* and a join state *J*, both with no actions.

We add an arc with the first branch's condition from the condition state to the branch's first statement. We add another arc with the complement of the first branch's condition AND'ed with the second branches condition from the condition state to the branches first statement. We repeat this for each branch. Finally, we connect the arc leaving the last statement of each branch to the join state, and we add an arc from this state to the next statement's state.

Using this template approach, we convert our GCD program to the complex state diagram of Figure 4.3(c). We are now well on our way to designing a custom single-purpose processor that executes the GCD program. Our next step is to divide the functionality into a data path part and a controller part, as shown in Figure 4.4. The data path part should consist of an interconnection of combinational and sequential components. The controller part should consist of a basic state diagram, i.e., one containing only Boolean actions and conditions.

Figure 4.3: Example program -- Greatest Common Divisor (GCD): (a) black-box view, (b) desired functionality, (c) state diagram



```
0: vectorN  x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while  (x != y)  {
6:       if  (x < y)
7:          y = y - x;
         else
8:          x = x - y;
       }
9:   d_o = x;
   }
```

(b)

We construct the data path through a four-step process:

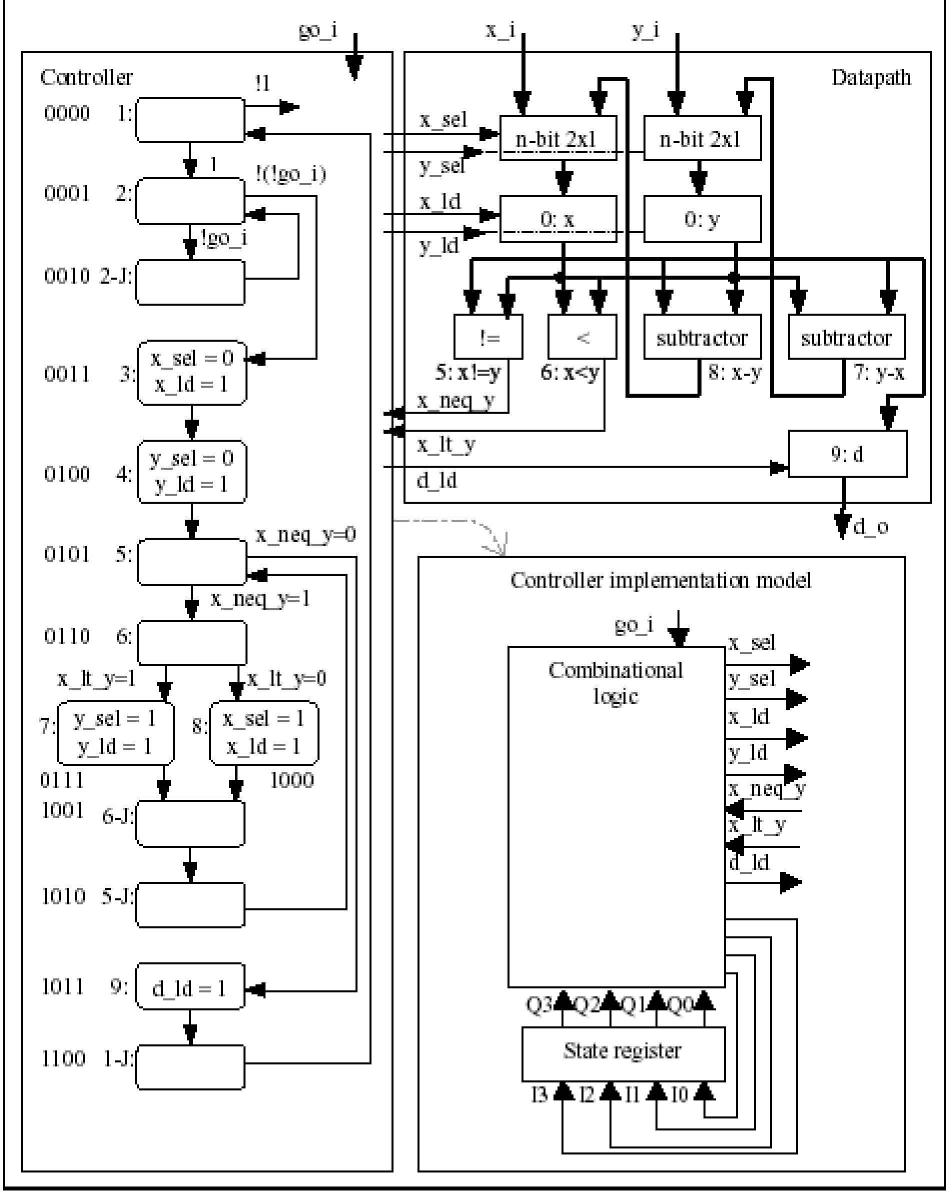
1. First, we create a register for any declared variable. In the example, these are *x* and *y*. We treat an output port as having an implicit variable, so we create a register *d* and connect it to the output port. We also draw the input and output ports.

2. Second, we create a functional unit for each arithmetic operation in the state diagram. In the example, there are two subtractions, one comparison for less than, and one comparison for inequality, yielding two subtractors and two comparators, as shown in the figure.

3. Third, we connect the ports, registers and functional units. For each write to a variable in the state diagram, we draw a connection from the writer's source (an input port, a functional unit, or another register) to the variable's register. For each arithmetic and logical operation, we connect sources to an input of the operation's corresponding functional unit. When more than one source is connected to a register, we add an appropriately-sized multiplexor.

4. Finally, we create a unique identifier for each control input and output of the data path components.

Figure 4.4: Example after splitting into a controller and a datapath.

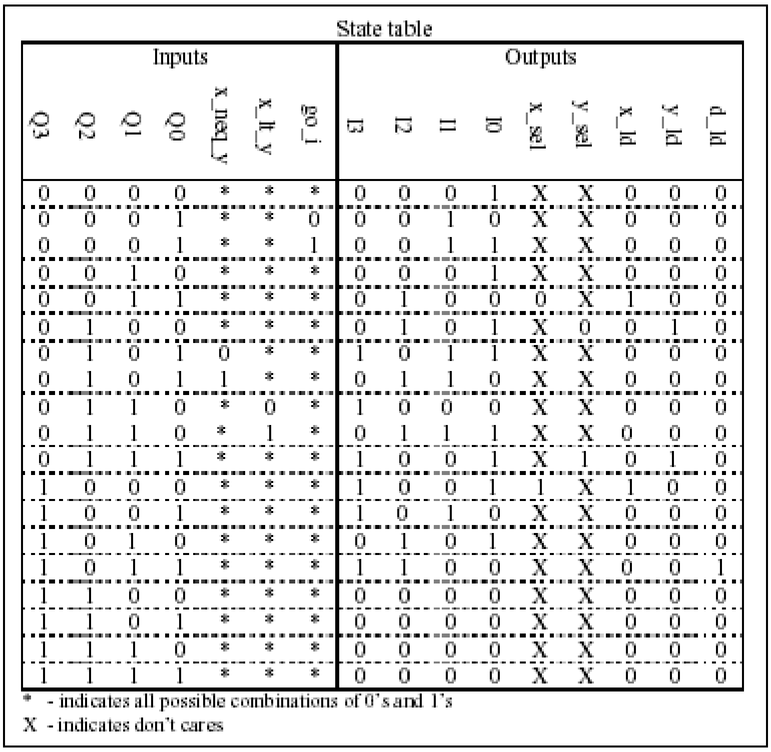


Now that we have a complete data path, we can build a state diagram for our controller. The state diagram has the same structure as the complex state diagram.

However, we replace complex actions and conditions by Boolean ones, making use of our data path. We replace every variable write by actions that set the select signals of the multiplexor in front of the variable's register's such that the writer's source passes through, and we assert the load signal of that register. We replace every logical operation in a condition by the corresponding functional unit control output.

We can then complete the controller design by implementing the state diagram using our sequential design technique described earlier. Figure 4.4 shows the controller implementation model, and Figure 4.5 shows a state table. Note that there are 7 inputs to the controller, resulting in 128 rows for the table. We reduced rows in the state table by using don't cares for some input combinations, but we can still see that optimizing the design can still see that optimizing the design using hand techniques could be quite tedious. For this reason, computer-aided design (CAD) tools that automate the combinational as well as sequential logic design can be very helpful; we'll introduce such CAD tools.

Figure 4.5: State table for the GCD example.



State table

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	d_ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
0	0	0	1	*	*	1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

* - indicates all possible combinations of 0's and 1's
X - indicates don't cares

Also, note that we could perform significant amounts of optimization to both the data path and the controller. For example, we could merge functional units in the data path, resulting in fewer units at the expense of more multiplexors. We could also merge states in the data path.

Remember that we could alternatively implement the GCD program by programming a microcontroller, thus eliminating the need for this design process, but possibly yielding a slower and bigger design.

Summary

- Designing a custom single- purpose processor for a given program requires an understanding of various aspects of digital design.
- Design of a circuit to implement Boolean functions requires combinational design, which consists of building a truth table with all possible inputs and desired outputs, optimizing, and drawing a circuit.
- Design of a circuit to implement a state a circuit to implement a state diagram requires sequential design, which consists of drawing an implementation model with a state register and a combinational logic block, assigning a binary encoding to each state, drawing a state table with inputs and outputs, and repeating our combinational design process for this table.
- Finally, design of a single purpose processor circuit to implement a program requires us to first schedule the program's statements into a complex state diagram, construct a data path from the diagram, create a new state diagram that replaces complex actions and conditions by data path control operations, and then design a controller circuit for the new state diagram using sequential design. Because processors can be complex, CAD tools would be a great designer's aid.

Questions:

1. Build a 3-input NAND gate using a minimum number of CMOS transistors.

2. Design a 2-bit comparator (compares two 2-bit words) with a single output "less than," using the combinational design technique described in the chapter. Start from a truth table, use K-maps to minimize logic, and draw the final circuit.

3. Design a 3-bit counter that counts the following sequence: 1, 2, 4, 5, 7, 1, 2, This counter has an output "odd" that is one when the current count value is odd. Use the sequential design technique of the chapter. Start from a state diagram, draw the state table, minimize logic, and draw the final circuit.

4. Compare the GCD custom-processor implementation to a software implementation. (a) Compare the performance. Assume a 100 ns clock for the microcontroller, and a 20 ns clock for the custom processor. Assume the microcontroller uses two operand instructions, and each instruction

requires 4 clock cycles. Estimates for the microcontroller are fine. (b) Estimate the number of gates for the custom design, and compare this to 10,000 gates for a simple 8-bit microcontroller. (c) Compare the custom GCD with the GCD running on a 300 MHz processor with 2-operand instructions and 1 clock cycle per instruction (advanced processors use parallelism to meet or exceed 1 cycle per instruction). (d) Compare the estimated gates with 200,000 gates, a typical number of gates for a modern 32-bit processor.

5. Design a custom single-purpose processor implementing the following program, using the technique of the chapter. Start with a complex state diagram, construct a data path and a simplified state diagram, and draw the truth table for the controller, but do not complete the design for the controller beyond the truth table.

```
input_port U;

int V;

for (int i=0; i<32; i++)

V = V + U*V;
```

References and further reading

- Gajski, Daniel D. Principles of Digital Design. New Jersey: Prentice-Hall, 1997. ISBN 0-13-301144-5. Describes combinational and sequential logic design, with a focus on optimization techniques, CAD, and higher-levels of design.

- Katz, Randy. Contemporary Logic Design. Redwood City, California: Benjamin/Cummings, 1994. ISBN 0-8053-2703-7. Describes combinational and sequential logic design, with a focus on logic and sequential optimization and CAD.

UNIT – II

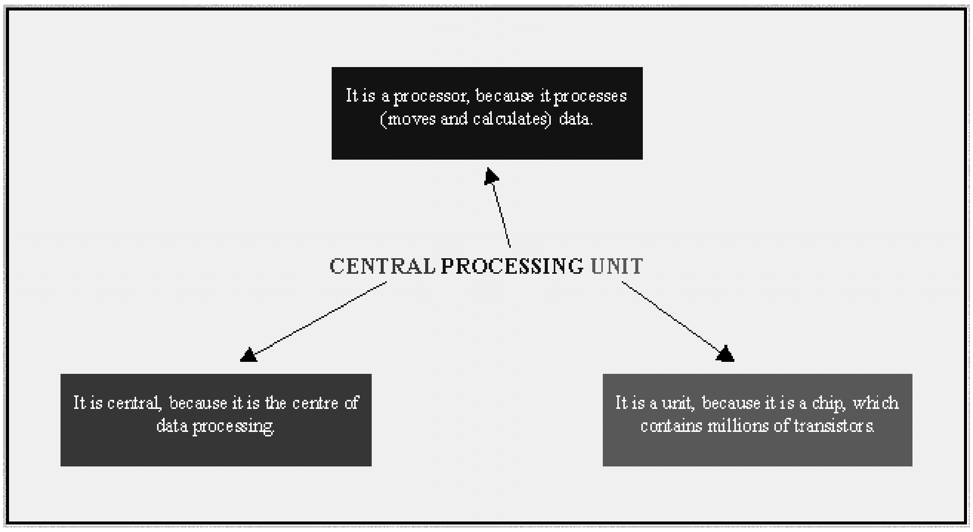
5. General purpose processor software

Objective:

This chapter discusses history of the 80x 86 CPU families and the major improvements occurring along the line. The historical background will help you better understand the design compromises they made as well as under-stand the legacy issues surrounding the CPU s design. This chapter also discusses the major advances in computer architecture that Intel employed while improving the x861.

Introduction to the Central Processing Unit

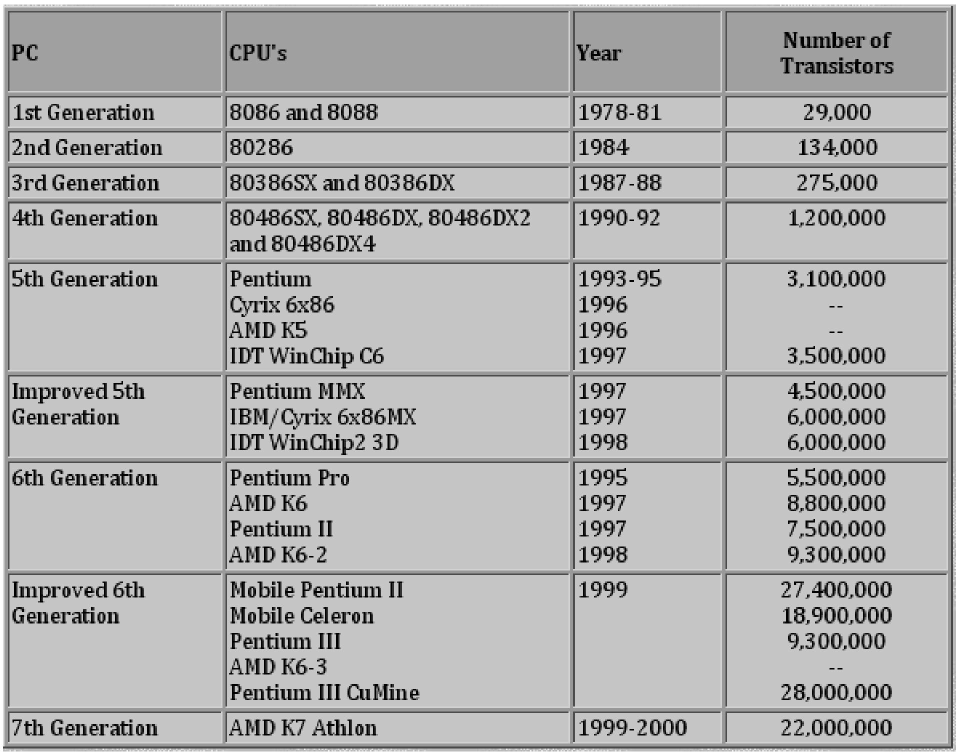
In order to work, a computer needs some sort of "brain" or "calculator". At the core of every computer is a device roughly the size of a large postage stamp. This device is known as the central processing unit or CPU for short. This is the "brain" of the computer; it reads and executes program instructions, performs calculations, and makes decisions. The CPU is responsible for storing and retrieving information on disks and other media. It also handles information on from one part of the computer to another like a central switching station that directs the flow of traffic throughout the computer system.


It is a processor, because it processes (moves and calculates) data.
CENTRAL PROCESSING UNIT
It is central, because it is the centre of data processing.
It is a unit, because it is a chip, which contains millions of transistors.

History of the Central Processing Unit

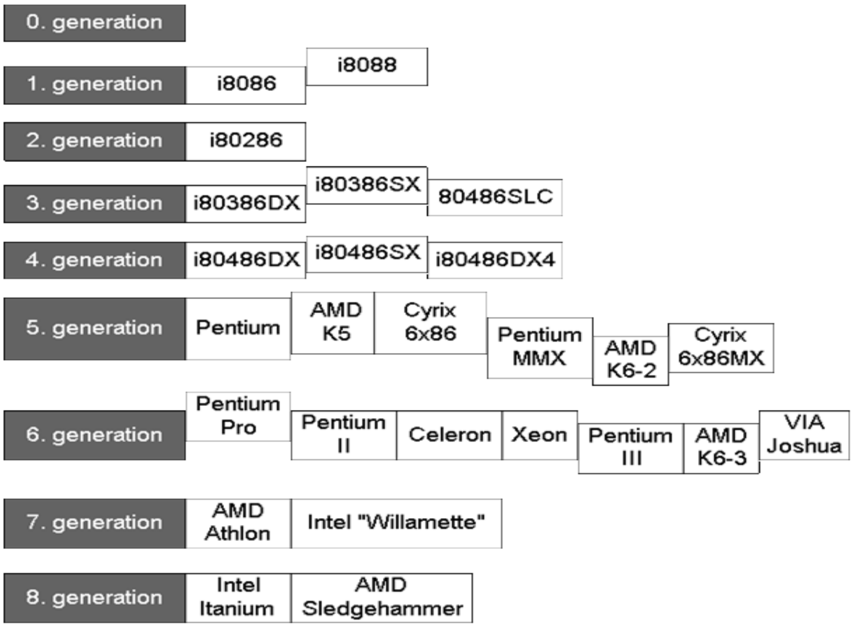
CPU history starts in 1971, when a small unknown company, Intel, for the first time combined multiple transistors to form a *central processing unit* - a chip called Intel 4044. However, it was 8 years before the first Personal Computer was constructed.

PC's are designed around different CPU generations. Intel is not the only company manufacturing CPU's, but by far the leading one. The following table shows the different CPU *generations*. They are predominantly Intel chips, but in the 5th generation we see alternatives.



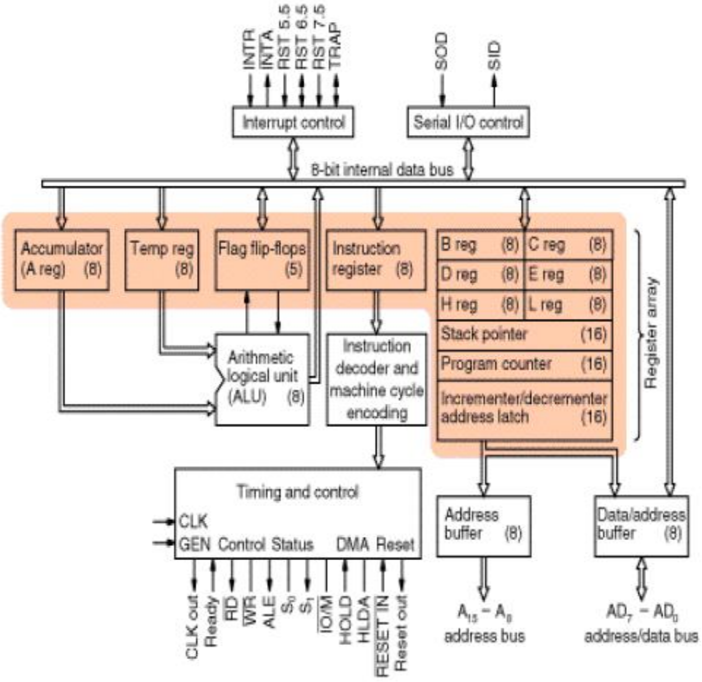
PC	CPU's	Year	Number of Transistors
1st Generation	8086 and 8088	1978-81	29,000
2nd Generation	80286	1984	134,000
3rd Generation	80386SX and 80386DX	1987-88	275,000
4th Generation	80486SX, 80486DX, 80486DX2 and 80486DX4	1990-92	1,200,000
5th Generation	Pentium Cyrix 6x86 AMD K5 IDT WinChip C6	1993-95 1996 1996 1997	3,100,000 -- -- 3,500,000
Improved 5th Generation	Pentium MMX IBM/Cyrix 6x86MX IDT WinChip2 3D	1997 1997 1998	4,500,000 6,000,000 6,000,000
6th Generation	Pentium Pro AMD K6 Pentium II AMD K6-2	1995 1997 1997 1998	5,500,000 8,800,000 7,500,000 9,300,000
Improved 6th Generation	Mobile Pentium II Mobile Celeron Pentium III AMD K6-3 Pentium III CuMine	1999	27,400,000 18,900,000 9,300,000 -- 28,000,000
7th Generation	AMD K7 Athlon	1999-2000	22,000,000

There are CPU's of many brand names (IBM, Texas, Cyrix, AMD), and often they make models which overlap two generations. This can make it difficult to keep track of CPU's. Here is an attempt to identify the various CPU's according to generation –



0. generation							
1. generation	i8086	i8088					
2. generation	i80286						
3. generation	i80386DX	i80386SX	80486SLC				
4. generation	i80486DX	i80486SX	i80486DX4				
5. generation	Pentium	AMD K5	Cyrix 6x86	Pentium MMX	AMD K6-2	Cyrix 6x86MX	
6. generation	Pentium Pro	Pentium II	Celeron	Xeon	Pentium III	AMD K6-3	VIA Joshua
7. generation	AMD Athlon	Intel "Willamette"					
8. generation	Intel Itanium	AMD Sledgehammer					

Internal Architecture of 8085 Microprocessor



Control Unit

Generates signals within Micro processor to carry out the instruction, which has been decoded. In reality causes certain connections between blocks of the Micro processor to be opened or closed, so that data goes where it is required, and so that ALU operations occur.

Arithmetic Logic Unit

The ALU performs the actual numerical and logic operation such as 'add', 'subtract', 'AND', 'OR', etc. Uses data from memory and from Accumulator to perform arithmetic operations. Always stores result of operation in Accumulator.

Registers

The 8085/8080A-programming model includes six registers, one accumulator, and one flag register, as shown in Figure. In addition, it has two 16-bit registers: the stack pointer and the program counter. They are described briefly as follows. The 8085/8080A has six general-purpose registers to store 8-bit data; these are identified as B,C,D,E,H, and L as shown in the figure. They can be combined as register pairs - BC, DE, and HL - to perform some 16-bit operations. The programmer can use these registers to store or copy data into the registers by using data copy instructions.

Accumulator

The accumulator is an 8-bit register that is a part of arithmetic/logic unit (ALU). This register is used to store 8-bit data and to perform arithmetic and logical operations. The result of an operation is stored in the accumulator. The accumulator is also identified as register A.

Flags

The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers. They are called Zero(Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags; they are listed in the Table and their bit positions in the flag register are shown in the Figure below. The most commonly used flags are Zero, Carry, and Sign. The microprocessor uses these flags to test data conditions. For example, after an addition of two numbers, if the sum in the accumulator id larger than eight bits, the flip-flop uses to indicate a carry -- called the Carry flag (CY) – is set to one. When an arithmetic operation results in zero, the flip-flop called the Zero(Z) flag is set to one. The first Figure shows an 8-bit register, called the flag register, adjacent to the accumulator. However, it is not used as a register; five bit positions out of eight are used to store the outputs of the five flip-flops. The flags are stored in the 8-bit register so that the programmer can examine these flags (data conditions) by accessing the register through an instruction. These flags have critical importance in the decision-making process of the microprocessor. The conditions (set or reset) of the flags are tested through the software instructions. For example, the instruction JC

(Jump on Carry) is implemented to change the sequence of a program when CY flag is set. The thorough understanding of flag is essential in writing assembly language programs.

Program Counter (PC)

This 16-bit register deals with sequencing the execution of instructions. This register is a memory pointer. Memory locations have 16-bit addresses, and that is why this is a 16-bit register. The microprocessor uses this register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location.

Stack Pointer (SP)

The stack pointer is also a 16-bit register used as a memory pointer. It points to a memory location in R/W memory, called the stack. The beginning of the stack is defined by loading 16-bit address in the stack pointer. The stack concept is explained in the chapter "Stack and Subroutines."

Instruction Register/Decoder

Temporary storage for the current instruction of a program. Latest instruction sent here from memory prior to execution. Decoder then takes instruction and ‘decodes’ or interprets the instruction. Decoded instruction then passed to next stage.

Memory Address Register

Holds address, received from PC, of next program instruction. Feeds the address bus with address of locations of the program under execution.

Control Generator

Generates signals within Micro processor to carry out the instruction which has been decoded. In reality causes certain connections between blocks of the Micro processor to be opened or closed, so that data goes where it is required, and so that ALU operations occur.

Register Selector

This block controls the use of the register stack in the example. Just a logic circuit which switches between different registers in the set will receive instructions from Control Unit.

General Purpose Registers

Micro processor requires extra registers for versatility and can be used to store additional data during a program. More complex processors may have a variety of differently named registers. Microprogramming how does the Micro processor knows what an instruction means, especially when it is only a binary number? The micro program in a Micro processor/ micro controller is written by the chip designer and tells the Micro processor/ micro controller the meaning of each instruction Micro processor/micro controller can then carry out operation.

8085 System Bus

Typical system uses a number of busses, collection of wires, which transmit binary numbers, one bit per wire. A typical microprocessor communicates with memory and other devices (input and output) using three busses: Address Bus, Data Bus and Control Bus.

Address Bus

One wire for each bit, therefore 16 bits = 16 wires. Binary number carried alerts memory to ‘open’ the designated box. Data (binary) can then be put in or taken out. The Address Bus consists of 16 wires, therefore 16 bits. Its "width" is 16 bits. A 16 bit binary number allows 216 different numbers, or 32000 different numbers, ie 0000000000000000 up to 1111111111111111. Because memory consists of boxes, each with a unique address, the size of the address bus determines the size of memory, which can be used. To communicate with memory the microprocessor sends an address on the address bus, eg 0000000000000011 (3 in decimal), to the memory. The memory the selects box number 3 for reading or writing data. Address bus is unidirectional, ie numbers only sent from microprocessor to memory, not other way.

Question?: If you have a memory chip of size 256 kilobytes (256 x 1024 x 8 bits), how many wires does the address bus need, in order to be able to specify an address in this memory? Note: the memory is organized in groups of 8 bits per location, therefore, how many locations must you be able to specify?

Data Bus

Data Bus: carries ‘data’, in binary form, between µP and other external units, such as memory. Typical size is 8 or 16 bits. Size determined by size of boxes in memory and µP size helps determine performance of µP. The Data Bus typically consists of 8 wires. Therefore, 28 combinations of binary digits. Data bus used to transmit "data", ie information, results of arithmetic, etc, between memory and the microprocessor. Bus is bi-directional. Size of the data bus determines what arithmetic can be done. If only 8 bits wide then largest number is 11111111 (255 in decimal). Therefore, larger number have to be broken down into chunks of 255. This slows microprocessor. Data Bus also carries instructions from memory to the microprocessor. Size of the bus therefore limits the number of possible instructions to 256, each specified by a separate number.

Control Bus

Control Bus is various lines which have specific functions for coordinating and controlling uP operations. Eg: Read/Not write line, single binary digit. Control whether memory is being ‘written to’ (data stored in memory) or ‘read from’ (data taken out of memory) 1 = Read, 0 = Write. May also include clock line(s) for timing/synchronizing, ‘interrupts’, ‘reset’ etc. Typically µP has 10 control lines. Cannot function correctly without these vital control signals. The Control Bus carries control signals partly unidirectional, partly bi-directional. Control signals are things like "read or write". This tells memory that we are either reading from a location, specified on the address bus, or writing to a location specified, various other signals to control and coordinate the operation of the system. Modern day microprocessors, like 80386, 80486 have much larger busses. Typically 16 or 32 bit busses, which allow larger number of instructions, more memory location, and faster arithmetic. Microcontrollers organized along same lines, except: because microcontrollers have memory etc inside the chip, the busses may all be internal. In the microprocessor the three busses are external to the chip (except for the internal data bus). In case of external busses, the chip connects to the busses via buffers, which are simply an electronic connection between external bus and the internal data bus.

8085 Pin description.

Properties

Single + 5V Supply

4 Vectored Interrupts (One is Non Maskable)

Serial In/Serial Out Port

Decimal, Binary, and Double Precision Arithmetic

Direct Addressing Capability to 64K bytes of memory

The Intel 8085A is a new generation, complete 8 bit parallel central processing unit (CPU). The 8085A uses a multiplexed data bus. The address is split between the 8bit address bus and the 8bit data bus.

Pin Description

The following describes the function of each pin:

A6 - A1s (Output 3 State)

Address Bus; The most significant 8 bits of the memory address or the 8 bits of the I/0 address,3 stated during Hold and Halt modes.

AD0 - 7 (Input/Output 3state)

Multiplexed Address/Data Bus; Lower 8 bits of the memory address (or I/0 address) appear on the bus during the first clock cycle of a machine state.

It then becomes the data bus during the second and third clock cycles. 3 stated during Hold and Halt modes.

ALE (Output)

Address Latch Enable: It occurs during the first clock cycle of a machine state and enables the address to get latched into the on chip latch of peripherals. The falling edge of ALE is set to guarantee setup and hold times for the address information. ALE can also be used to strobe the status information. ALE is never 3stated.

SO, S1 (Output)

Data Bus Status. Encoded status of the bus cycle:

S1	S0	Description
O	O	HALT
0	1	WRITE
1	0	READ
1	1	FETCH

S1 can be used as an advanced R/W status.

RD (Output 3state)

READ; indicates the selected memory or 1/0 device is to be read and that the Data Bus is available for the data transfer.

WR (Output 3state)

WRITE; indicates the data on the Data Bus is to be written into the selected memory or 1/0 location. Data is set up at the trailing edge of WR. 3stated during Hold and Halt modes.

READY (Input)

If Ready is high during a read or write cycle, it indicates that the memory or peripheral is ready to send or receive data. If Ready is low, the CPU will wait for Ready to go high before completing the read or write cycle.

HOLD (Input)

HOLD; indicates that another Master is requesting the use of the Address and Data Buses. The CPU, upon receiving the Hold request. will relinquish the use of buses as soon as the completion of the current machine cycle. Internal processing can continue. The processor can regain the buses only after the Hold is removed. When the Hold is acknowledged, the Address, Data, RD, WR, and IO/M lines are 3stated.

HLDA (Output)

HOLD ACKNOWLEDGE; indicates that the CPU has received the Hold request and that it will relinquish the buses in the next clock cycle. HLDA goes low after the Hold request is removed. The CPU takes the buses one half clock cycle after HLDA goes low.

INTR (Input)

INTERRUPT REQUEST; is used as a general purpose interrupt. It is sampled only during the next to the last clock cycle of the instruction. If it is active, the Program Counter (PC) will be inhibited from incrementing and an INTA will be issued. During this cycle a RESTART or CALL instruction can be inserted to jump to the interrupt service routine. The INTR is enabled and disabled by software. It is disabled by Reset and immediately after an interrupt is accepted.

INTA (Output)

INTERRUPT ACKNOWLEDGE; is used instead of (and has the same timing as) RD during the Instruction cycle after an INTR is accepted. It can be used to activate the 8259 Interrupt chip or some other interrupt port.

RST 5.5

RST 6.5 - (Inputs)

RST 7.5

RESTART INTERRUPTS; These three inputs have the same timing as I NTR except they cause an internal RESTART to be automatically inserted.

RST 7.5 - Highest Priority

RST 5.5 - Lowest Priority

The priority of these interrupts is ordered as shown above. These interrupts have a higher priority than the INTR.

TRAP (Input)

Trap interrupt is a nonmaskable restart interrupt. It is recognized at the same time as INTR. It is unaffected by any mask or Interrupt Enable. It has the highest priority of any interrupt.

RESET IN (Input)

Reset sets the Program Counter to zero and resets the Interrupt Enable and HLDA flipflops. None of the other flags or registers (except the instruction register) are affected The CPU is held in the reset condition as long as Reset is applied.

RESET OUT (Output)

Indicates CPIJ is being reset. Can be used as a system RESET. The signal is synchronized to the processor clock.

X1, X2 (Input)

Crystal or R/C network connections to set the internal clock generator X1 can also be an external clock input instead of a crystal. The input frequency is divided by 2 to give the internal operating frequency.

CLK (Output)

Clock Output for use as a system clock when a crystal or R/ C network is used as an input to the CPU. The period of CLK is twice the X1, X2 input period.

IO/M (Output)

IO/M indicates whether the Read/Write is to memory or l/O Tristated during Hold and Halt modes.

SID (Input)

Serial input data line The data on this line is loaded into accumulator bit 7 whenever a RIM instruction is executed.

SOD (output)

Serial output data line. The output SOD is set or reset as specified by the SIM instruction.

Vcc

+5 volt supply.

Vss

Ground Reference.

8085 Functional Description

The 8085A is a complete 8 bit parallel central processor. It requires a single +5 volt supply. Its basic clock speed is 3 MHz thus improving on the present 8080's performance with higher system speed. Also it is designed to fit into a minimum system of three IC's: The CPU, a RAM/ IO, and a ROM or PROM/IO chip.

The 8085A uses a multiplexed Data Bus. The address is split between the higher 8bit Address Bus and the lower 8bit Address/Data Bus. During the first cycle the address is sent out. The lower 8bits are latched into the peripherals by the Address Latch Enable (ALE). During the rest of the machine cycle the Data Bus is used for memory or I/O data.

The 8085A provides RD, WR, and lO/Memory signals for bus control. An Interrupt Acknowledge signal (INTA) is also provided. Hold, Ready, and all Interrupts are synchronized. The 8085A also provides serial input data (SID) and serial output data (SOD) lines for simple serial interface.

In addition to these features, the 8085A has three maskable, restart interrupts and one non-maskable trap interrupt. The 8085A provides RD, WR and IO/M signals for Bus control.

Status Information

Status information is directly available from the 8085A. ALE serves as a status strobe. The status is partially encoded, and provides the user with advanced timing of the type of bus transfer being done. IO/M cycle status signal is provided directly also. Decoded So, S1 Carries the following status information.

HALT, WRITE, READ, FETCH:

S1 can be interpreted as R/W in all bus transfers. In the 8085A the 8 LSB of address are multiplexed with the data instead of status. The ALE line is used as a strobe to enter the lower half of the address into the memory or peripheral address latch. This also frees extra pins for expanded interrupt capability.

Interrupt and Serial l/O :

The8085A has5 interrupt inputs: INTR, RST5.5, RST6.5, RST 7.5, and TRAP. INTR is identical in function to the 8080 INT. Each of the three RESTART inputs, 5.5, 6.5. 7.5, has a programmable mask. TRAP is also a RESTART interrupt except it is non maskable.

The three RESTART interrupts cause the internal execution of RST (saving the program counter in the stack and branching to the RESTART address) if the interrupts are enabled and if the interrupt mask is not set. The non-maskable TRAP causes the internal execution of a RST independent of the state of the interrupt enable or masks.

The interrupts are arranged in a fixed priority that determines which interrupt is to be recognized if more than one is pending as follows: TRAP highest priority, RST 7.5, RST 6.5, RST 5.5, INTR lowest priority This priority scheme does not take into account the priority of a routine that was started by a higher priority interrupt. RST 5.5 can interrupt a RST 7.5 routine if the interrupts were re-enabled before the end of the RST 7.5 routine. The TRAP interrupt is useful for catastrophic errors such as power failure or bus error. The TRAP input is recognized just as any other interrupt but has the highest priority. It is not affected by any flag or mask. The TRAP input is both edge and level sensitive.

Basic System Timing

The 8085A has a multiplexed Data Bus. ALE is used as a strobe to sample the lower 8bits of address on the Data Bus. Figure 2 shows an instruction fetch, memory read and I/ O write cycle (OUT). Note that during the I/O write and read cycle that the I/O port address is copied on both the upper and lower half of the address. As in the 8080, the READY line is used to extend the read and write pulse lengths so that the 8085A can be used with slow memory. Hold causes the CPU to stop the bus when it Is through with it by floating the Address and Data Buses.

System Interface:

8085A family includes memory components, which are directly compatible to the 8085A CPU. For example, a system consisting of the three chips, 8085A, 8156, and 8355 will have the following features:

2K Bytes ROM

256 Bytes RAM

1 Timer/Counter

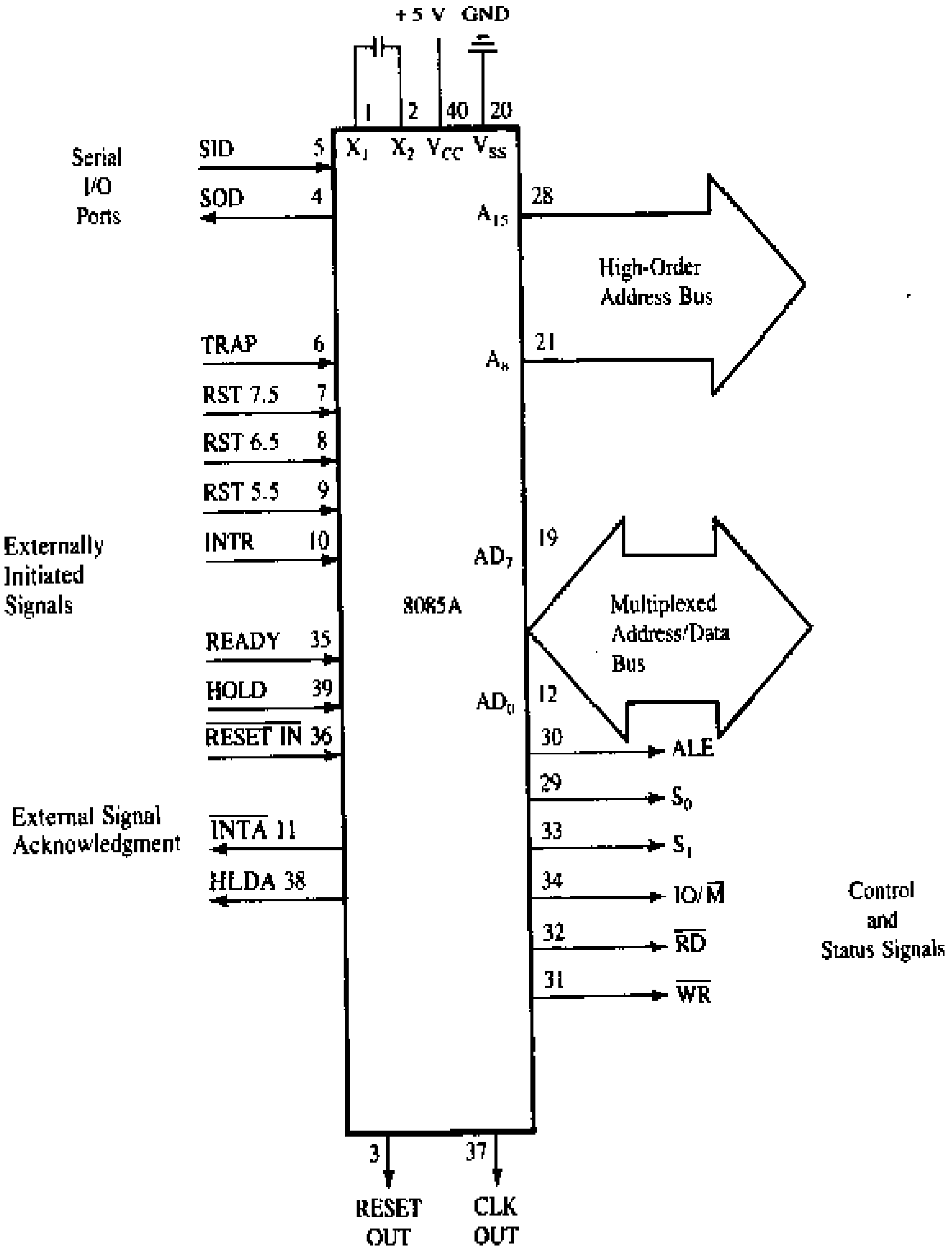
4 8bit I/O Ports

1 6bit I/O Port

4 Interrupt Levels

Serial In/Serial Out Ports

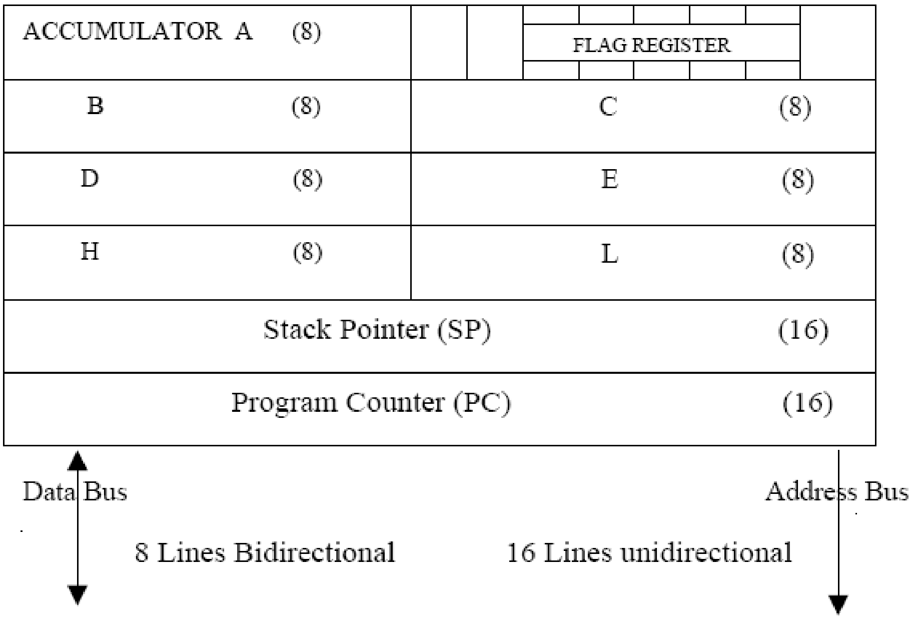
In addition to standard I/O, the memory mapped I/O offers an efficient I/O addressing technique. With this technique, an area of memory address space is assigned for I/O address, thereby, using the memory address for I/O manipulation. The 8085A CPU can also interface with the standard memory that does not have the multiplexed address/data bus.



The 8085 Programming Model

In the previous tutorial we described the 8085 microprocessor registers in reference to the internal data operations. The same information is repeated here briefly to provide the continuity and the context to the instruction set and to enable the readers who prefer to focus initially on the programming aspect of the microprocessor.

The 8085 programming model includes six registers, one accumulator, and one flag register, as shown in Figure. In addition, it has two 16-bit registers: the stack pointer and the program counter. They are described briefly as follows.



Registers

The 8085 has six general-purpose registers to store 8-bit data; these are identified as B,C,D,E,H, and L as shown in the figure. They can be combined as register pairs - BC, DE, and HL - to perform some 16-bit operations. The programmer can use these registers to store or copy data into the registers by using data copy instructions.

Accumulator

The accumulator is an 8-bit register that is a part of arithmetic/logic unit (ALU). This register is used to store 8-bit data and to perform arithmetic and logical operations. The result of an operation is stored in the accumulator. The accumulator is also identified as register A.

Flags

The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers. They are called Zero(Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags; their bit positions in the flag register are shown in the Figure below. The most commonly used flags are Zero, Carry, and Sign. The microprocessor uses these flags to test data conditions.

D7	D6	D5	D4	D3	D2	D1	D0
S	**Z**		**AC**		**P**		**CY**

For example, after an addition of two numbers, if the sum in the accumulator id larger than eight bits, the flip-flop uses to indicate a carry -- called the Carry flag (CY) – is set to one. When an arithmetic operation results in zero, the flip-flop called the Zero(Z) flag is set to one. The first Figure shows an 8-bit register, called the flag register, adjacent to the accumulator. However, it is not used as a register; five bit positions out of eight are used to store the outputs of the five flip-flops. The flags are stored in the 8-bit register so that the programmer can examine these flags (data conditions) by accessing the register through an instruction. These flags have critical importance in the decision-making process of the microprocessor. The conditions (set or reset) of the flags are tested through the software instructions. For example, the instruction JC (Jump on Carry) is implemented to change the sequence of a program when CY flag is set. The thorough understanding of flag is essential in writing assembly language programs.

Program Counter (PC)

This 16-bit register deals with sequencing the execution of instructions. This register is a memory pointer. Memory locations have 16-bit addresses, and that is why this is a 16-bit register. The microprocessor uses this register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location

Stack Pointer (SP)

The stack pointer is also a 16-bit register used as a memory pointer. It points to a memory location in R/W memory, called the stack. The beginning of the stack is defined by loading 16-bit address in the stack pointer. This programming model will be used in subsequent tutorials to examine how these registers are affected after the execution of an instruction.

The 8085 Addressing Modes

The instructions MOV B, A or MVI A, 82H are to copy data from a source into a destination. In these instructions the source can be a register, an input port, or an 8-bit number (00H to FFH). Similarly, a destination can be a register or an output port. The sources and destination are operands. The various formats for specifying operands are called the ADDRESSING MODES. For 8085, they are:

1. Immediate addressing.

2. Register addressing.

3. Direct addressing.

4. Indirect addressing.

Immediate addressing

Data is present in the instruction. Load the immediate data to the destination provided. Example: MVI R,data

Register addressing

Data is provided through the registers.

Example: MOV Rd, Rs

Direct addressing

Used to accept data from outside devices to store in the accumulator or send the data stored in the accumulator to the outside device. Accept the data from the port 00H and store them into the accumulator or Send the data from the accumulator to the port 01H.

Example: IN 00H or OUT 01H

Indirect Addressing

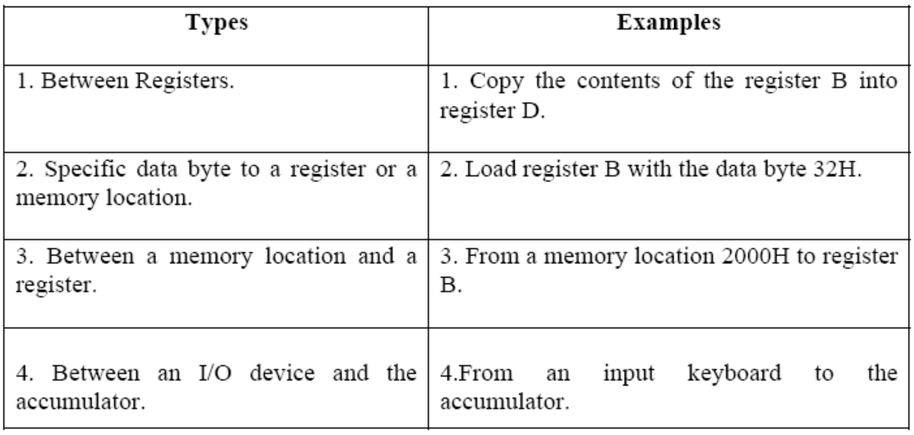
This means that the Effective Address is calculated by the processor. And the contents of the address (and the one following) is used to form a second address. The second address is where the data is stored. Note that this requires several memory accesses; two accesses to retrieve the 16-bit address and a further access (or accesses) to retrieve the data which is to be loaded into the register.

Instruction Set Classification

An instruction is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions, called the instruction set, determines what functions the microprocessor can perform. These instructions can be classified into the following five functional categories: data transfer (copy) operations, arithmetic operations, logical operations, branching operations, and machine-control operations.

Data Transfer (Copy) Operations

This group of instructions copy data from a location called a source to another location called a destination, without modifying the contents of the source. In technical manuals, the term *data transfer* is used for this copying function. However, the term *transfer* is misleading; it creates the impression that the contents of the source are destroyed when, in fact, the contents are retained without any modification. The various types of data transfer (copy) are listed below together with examples of each type:



Types	Examples
1. Between Registers.	1. Copy the contents of the register B into register D.
2. Specific data byte to a register or a memory location.	2. Load register B with the data byte 32H.
3. Between a memory location and a register.	3. From a memory location 2000H to register B.
4. Between an I/O device and the accumulator.	4.From an input keyboard to the accumulator.

Arithmetic Operations

These instructions perform arithmetic operations such as addition, subtraction, increment, and decrement.

Addition - Any 8-bit number, or the contents of a register or the contents of a memory location can be added to the contents of the accumulator and the sum is stored in the accumulator. No two other 8-bit registers can be added directly (e.g., the contents of register B cannot be added directly to the contents of the register C). The instruction DAD is an exception; it adds 16-bit data directly in register pairs.

Subtraction - Any 8-bit number, or the contents of a register, or the contents of a memory location can be subtracted from the contents of the accumulator and the results stored in the accumulator. The subtraction is performed in 2's compliment, and the results if negative, are expressed in 2's complement. No two other registers can be subtracted directly.

Increment/Decrement - The 8-bit contents of a register or a memory location can be incremented or decrement by 1. Similarly, the 16-bit contents of a register pair (such as BC) can be incremented or decrement by 1. These increment and decrement operations differ from addition and subtraction in an important way; i.e., they can be performed in any one of the registers or in a memory location.

Logical Operations

These instructions perform various logical operations with the contents of the accumulator.

AND, OR, Exclusive-OR - Any 8-bit number, or the contents of a register, or of a memory location can be logically ANDed, Ored, or Exclusive-ORed with the contents of the accumulator. The results are stored in the accumulator.

Rotate- Each bit in the accumulator can be shifted either left or right to the next position.

Compare- Any 8-bit number, or the contents of a register, or a memory location can be compared for equality, greater than, or less than, with the contents of the accumulator.

Complement - The contents of the accumulator can be complemented. All 0s are replaced by 1s and all 1s are replaced by 0s.

Branching Operations

This group of instructions alters the sequence of program execution either conditionally or unconditionally.

Jump - Conditional jumps are an important aspect of the decision-making process in the programming. These instructions test for a certain conditions (e.g., Zero or Carry flag) and alter the program sequence when the condition is met. In addition, the instruction set includes an instruction called *unconditional jump*.

Call, Return, and Restart - These instructions change the sequence of a program either by calling a subroutine or returning from a subroutine. The conditional Call and Return instructions also can test condition flags.

Machine Control Operations

These instructions control machine functions such as Halt, Interrupt, or do nothing. The microprocessor operations related to data manipulation can be summarized in four functions:

1. copying data

2. performing arithmetic operations

3. performing logical operations

4. testing for a given condition and alerting the program sequence

Some important aspects of the instruction set are noted below:

1. In data transfer, the contents of the source are not destroyed; only the contents of the destination are changed. The data copy instructions do not affect the flags.

2. Arithmetic and Logical operations are performed with the contents of the accumulator, and the results are stored in the accumulator (with some expectations). The flags are affected according to the results.

3. Any register including the memory can be used for increment and decrement.

4. A program sequence can be changed either conditionally or by testing for a given data condition.

Instruction Format

An instruction is a command to the microprocessor to perform a given task on a specified data. Each instruction has two parts: one is task to be performed, called the operation code (opcode), and the second is the data to be operated on, called the operand. The operand (or data) can be specified in various ways. It may include 8-bit (or 16-bit) data, an internal register, a memory location, or 8-bit (or 16-bit) address. In some instructions, the operand is implicit.

Instruction word size

The 8085 instruction set is classified into the following three groups according to word size:

1. One-word or 1-byte instructions

2. Two-word or 2-byte instructions

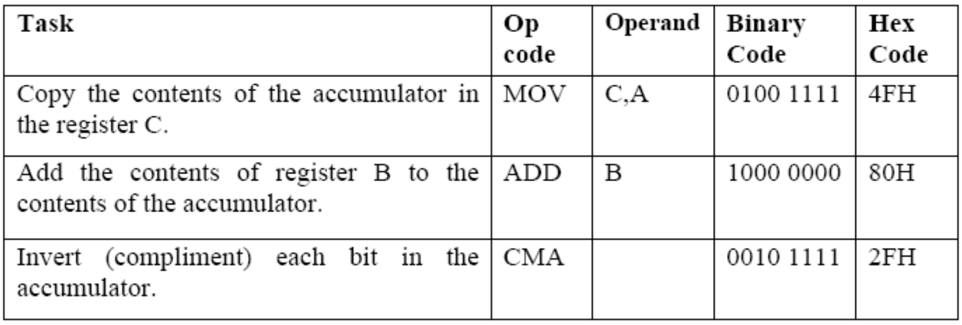
3. Three-word or 3-byte instructions

In the 8085, "byte" and "word" are synonymous because it is an 8-bit microprocessor. However, instructions are commonly referred to in terms of bytes rather than words.

One-Byte Instructions

A 1-byte instruction includes the opcode and operand in the same byte. Operand(s) are internal register and are coded into the instruction.

For example:



Task	Op code	Operand	Binary Code	Hex Code
Copy the contents of the accumulator in the register C.	MOV	C,A	0100 1111	4FH
Add the contents of register B to the contents of the accumulator.	ADD	B	1000 0000	80H
Invert (compliment) each bit in the accumulator.	CMA		0010 1111	2FH

These instructions are 1-byte instructions performing three different tasks. In the first instruction, both operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8- bit binary format in memory; each requires one memory location.

MOV rd, rs

rd <-- rs copies contents of rs into rd.

Coded as 01 ddd sss where ddd is a code for one of the 7 general registers which is the destination of the data, sss is the code of the source register.

Example: MOV A,B

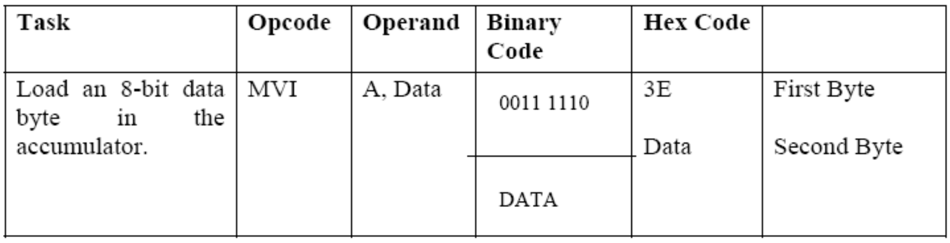
Coded as 01111000 = 78H = 170 octal (octal was used extensively in instruction design of such processors).

ADD r

A <-- A + r

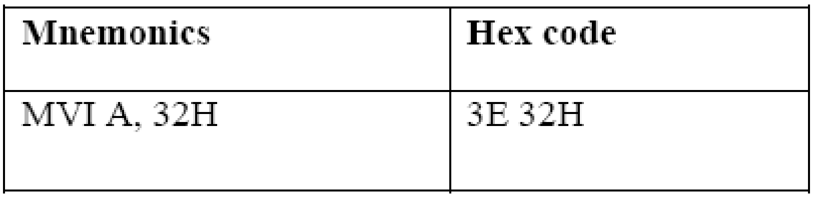
Two-Byte Instructions

In a two-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. Source operand is a data byte immediately following the opcode. For example:



Task	Opcode	Operand	Binary Code	Hex Code	
Load an 8-bit data byte in the accumulator.	MVI	A, Data	0011 1110	3E	First Byte
			DATA	Data	Second Byte

Assume that the data byte is 32H. The assembly language instruction is written as



Mnemonics	Hex code
MVI A, 32H	3E 32H

The instruction would require two memory locations to store in memory.

MVI r,data

r <-- data

Example: MVI A,30H coded as 3EH 30H as two contiguous bytes. This is an example of immediate addressing.

ADI data

A <-- A + data

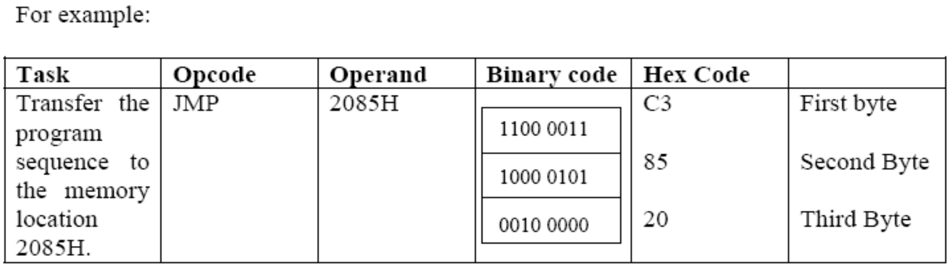
OUT port

where port is an 8-bit device address. (Port) <-- A. Since the byte is not the data but points directly to where it is located this is called direct addressing.

Three-Byte Instructions

In a three-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third byte is the high-order address.

opcode + data byte + data byte



For example:

Task	Opcode	Operand	Binary code	Hex Code	
Transfer the program sequence to the memory location 2085H.	JMP	2085H	1100 0011	C3	First byte
			1000 0101	85	Second Byte
			0010 0000	20	Third Byte

This instruction would require three memory locations to store in memory. Three byte instructions - opcode + data byte + data byte

LXI rp, data16

rp is one of the pairs of registers BC, DE, HL used as 16-bit registers. The two data bytes are 16-bit data in L H order of significance.

rp <-- data16

Example:

LXI H,0520H coded as 21H 20H 50H in three bytes. This is also immediate addressing.

LDA addr

A <-- (addr) Addr is a 16-bit address in L H order. Example: LDA 2134H coded as 3AH 34H 21H. This is also an example of direct addressing.

Sample Programs

Write an assembly program to add two numbers

Program

MVI D, 8BH

MVI C, 6FH

MOV A, C

ADD D

OUT PORT1

```
HLT
```

Write an assembly program to multiply a number by 8 Program

```
MVI A, 30H
RRC
RRC
RRC
OUT PORT1
HLT
```

Write an assembly program to find greatest between two numbers

Program

```
MVI B, 30H
MVI C, 40H
MOV A, B
CMP C
JZ EQU
JC GRT
OUT PORT1
HLT
EQU: MVI A, 01H
OUT PORT1
HLT
GRT: MOV A, C
OUT PORT1
HLT
```

Summary:

- Micro processor contains ALU, Interrupt controller, Serial I/O controller etc.
- The 8085/8080A-programming model includes six registers, one accumulator, and one flag register.
- The accumulator is an 8-bit register that is a part of arithmetic/logic unit (ALU).
- The flags in 8085 are Zero(Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags.
- This 16-bit register deals with sequencing the execution of instructions.
- The stack pointer is also a 16-bit register used as a memory pointer.
- Data Bus: carries 'data', in binary form, between µP and other external units, such as memory.

QUESTIONS:

- Explain the pin description of 8085?
- Explain the architecture of 8085?
- Explain functional description of 085 along with types of busses in 8085?
- What are the different addressing modes of 8085?
- Explain about instructions in 8085?

References:

- C Programming for Embedded Systems – KIRK ZURELL
- Design with 8051- FRONTLINE ELECTRONICS
- Embedded Controller Hardware Design - Ken Arnold
- Embedded Software The Works – colin walls
- Embedded Systems Firmware Demystified - Ed Sutter
- Embedded_Controller_Hardware_Design – KEN ARNOLD

6. Standard single purpose processors: peripherals

Objective:

In this chapter we will learn about the different timers and their technologies. Different technologies include timers, watchdog timers etc.,

Introduction:

Embedded device consists of a intelligent device namely processor/ controller and remaining different hardware devices which are communicating with that intelligent device. But here we get a problem? What is that is speed of communications of controlling devices and hardware interfaces. Generally control units are very much faster than the hardware devices which are used for interfacing. So inorder to provide proper communication between the control unit and hardware device the programmer has to make the control unit to wait unitill it gets response from that respective hardware device. In order to provide these waiting times for the cpu designers will use timer, counters, watchdog timers etc to achive the goal of communication.

Single purpose processors are the processors designed for Performs specific computation task. The examples of single purpose processors are

- “Off-the-shelf” -- pre-designed for a common task
- a.k.a., peripherals
- serial transmission
- analog/digital conversions

Timers:

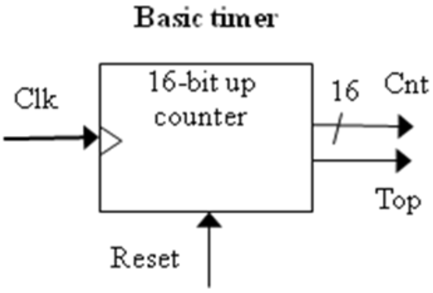
Timers are the electonic devices used for producing the required amounts of time intervels for proper communication between control unit and the hardware device. Generally these timers are present on the chip for micro controllers. The timers may be 8-bit, 16-bit depend on its size. The timers depend on the machine cycles for producing required amount of timeperiods. Machine cycle is the smallest time period that a processor require to execute a single byte instruction. The machine cycle is defined as the time required by the processor for the execution of 1byte instructon. It is mainly depend on the frequency of operation of the processor. The formula used is

$$Machine\ Cycle = \frac{12}{Frequecy\ of\ operation\ of\ control\ unit}$$

For example suppose a control unit is designed to operate at 12Mhz frequency. Then the machine cycle is calculated as

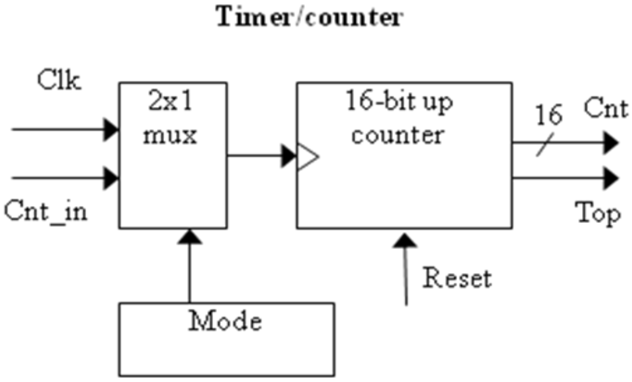
$$Machine\ cycle = \frac{12}{12x10^6} = 1\mu sec.$$

A 16-bit timer can count upto $2^{16} - 1$ =65,535 pulses. Each pulse require 1 machine cycle for its execution. The below diagram represents figure of the a 16-bit timer.



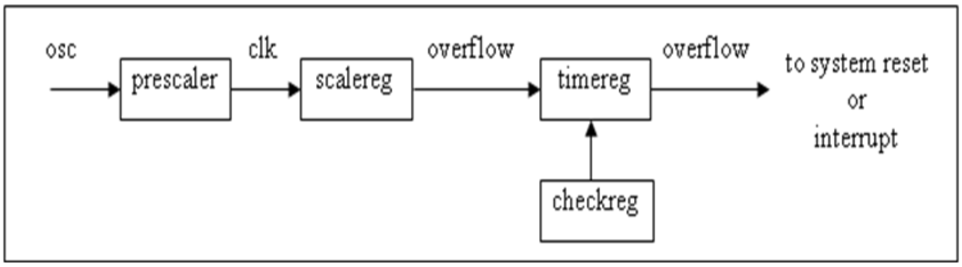
Counter:

Counter is also same as timer but it countes the number of external pulses rather than the generating/counting internal pulses. The below digram shows a counter which contain a mltiflexer and a counter.

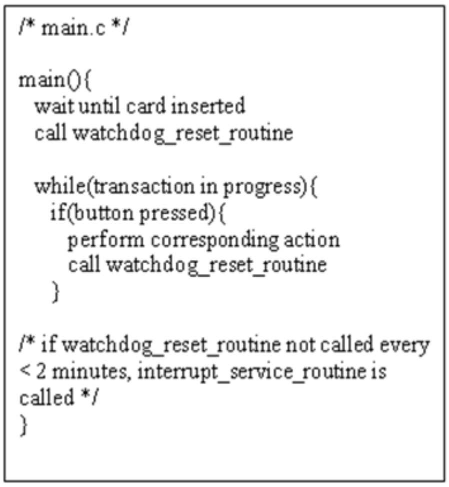


Watchdog timer:

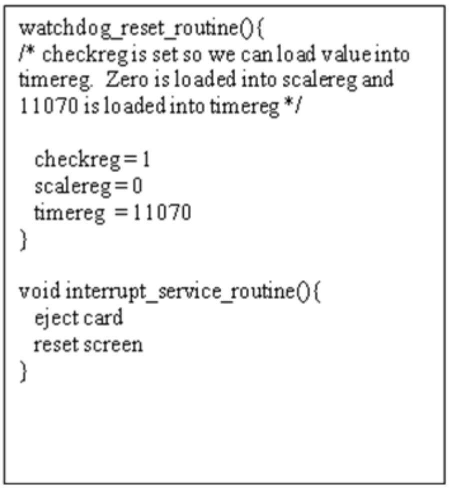
This is also a special time of timer. The operation of watchdog timer is a token generated with required time. A countdown timer is started when this time reaches to zero, It enables to execute user specified task. The block diagram of the watchdog timer is as below



The subroutine for watch dog timer is as below.



```
/* main.c */

main(){
  wait until card inserted
  call watchdog_reset_routine

  while(transaction in progress){
    if(button pressed){
      perform corresponding action
      call watchdog_reset_routine
    }

/* if watchdog_reset_routine not called every
< 2 minutes, interrupt_service_routine is
called */
}
```



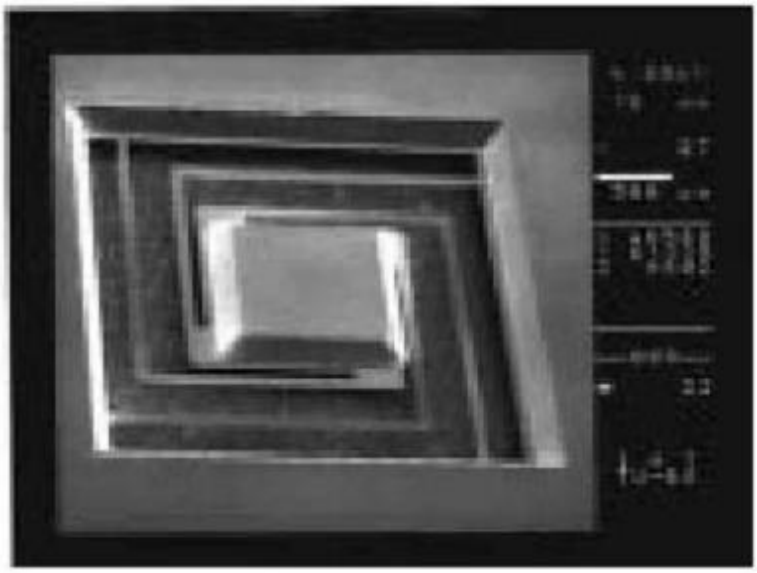
```
watchdog_reset_routine(){
/* checkreg is set so we can load value into
timereg. Zero is loaded into scalereg and
11070 is loaded into timereg */

  checkreg = 1
  scalereg = 0
  timereg  = 11070
}

void interrupt_service_routine(){
  eject card
  reset screen
}
```

Sensors

We start with a brief discussion of sensors. Sensors can be designed for virtually every physical quantity. There are sensors for weight, velocity, acceleration, electrical current, voltage, temperatures etc. A large amount of physical effects can be used for constructing sensors [Elsevier B.V., 2003a]. Examples include the law of induction (generation of voltages in an electric field), or light-electric effects. Also, there are sensors for chemical substances [Elsevier B.V., 2003b].

In recent years, a huge amount of sensors has been designed and much of the progress in designing smart systems can be attributed to modern sensor technology. Hence, it is impossible to cover this subset of embedded hardware technology comprehensively and we can only give characteristic examples:

Acceleration sensors: Fig.shows a small sensor manufactured using microsystem technology. The sensor contains a small mass in its center. When accelerated, the mass will be displaced from its standard position, thereby changing the resistance of the tiny wires connected to the mass.



Rain sensors: In order to remove distraction from drivers, some recent high end cars contain rain sensors. Using these, the speed of the wipers can be automatically adjusted to the amount of rain.

Image sensors: There are essentially two kinds of image sensors: charge coupled devices (CCDs) and CMOS sensors. In both cases, arrays of light sensors are used. The architecture of CMOS sensor arrays is similar to that of standard memories: individual pixels can be randomly addressed and read out. CMOS sensors use standard CMOS technology for integrated circuits [Dierickx, 2000]. Due to this, sensors and logic circuits can be integrated on the same chip. This allows some preprocessing to be done already on the sensor chip, leading to so-called smart sensors. CMOS sen sors require only a single standard supply voltage and interfacing in general is easy. Therefore, CMOS-based sensors can be cheap. In contrast, CCD technology is optimized for optical applications. In CCD technology, charges have to be transfered from one pixel to the next until they can finally be read out at an array boundary. This sequential charge transfer also gave CCDs their name. Images generated with CCDs can be of higher quality than those generated using CMOS sensors, since they generate less noise. However, interfacing is more complex. As a result, CMOS sensors are appropriate for applications requiring low or medium costs and low or medium image quality. CCD sensors are more adequate for high quality, expensive image sensors.

Bio-metrical sensors: Demands for higher security standards as well as the need to protect mobile and removable equipment have led to an increased interest in authentication. Due to the limitations of password based security (e.g. stolen and lost passwords), smartcards, bio-metrical sensors and bio-medical authentication receive significant attention. Bio-medical authentication tries to identify whether or not a certain person is actually the person she or he claims to be. Methods for bio-medical authentication include iris scans, finger print sensors and face recognition.

Finger print sensors are typically fabricated using the same CMOS technology which is used for manufacturing integrated circuits. Possible applications include notebooks which grant access only if the user's finger print is recognized. CCD and CMOS image sensors described above are used for face recognition. False accepts as well as false rejects are an inherent problem of bio-medical authentication. In contrast to password based authentication, exact matches are not possible.

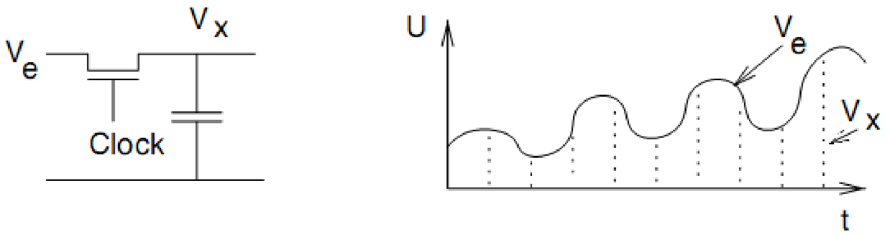
Artifical eyes: Artificial eye projects have received significant attention. While some projects attempt to actually affect the eye, others try to provide vision in an indirect way. The Dobelle Institute is experimenting with a setup in which a little camera is attached to glasses. This camera is connected to a computer translating these patterns into electrical pulses. These pulses are then sent directly to the brain, using a direct contact through an electrode. Currently (2003), the resolution is in the order of 128 by 128 pixels, enabling blind persons to drive a car in controlled areas.

Other sensors: Other common sensors include: pressure sensors, proximity sensors, engine control sensors, Hall effect sensors, and many more.

Sample-and-hold circuits

All known digital computers work in the discrete time domain. This means they can process discrete sequences of values. Hence, values in the continuous domain have to be converted to the discrete domain. This is the purpose of sample-and-hold circuits. Fig.(left) shows a simple sample-and-hold-circuit. In essence, the circuit consists of a clocked transistor and a capacitor. The transistor operates like a switch. Each time the switch is closed by the clock signal, the capacitor is charged so that its voltage is practically the same as the incoming voltage Ve. After opening the switch again, this voltage remain essentially unchanged until the switch is closed again. Each of the values stored on the capacitor can be considered as an element of a discrete sequence of values Vx, generated from a continuous sequence Ve.

Sample-and-hold-circuit

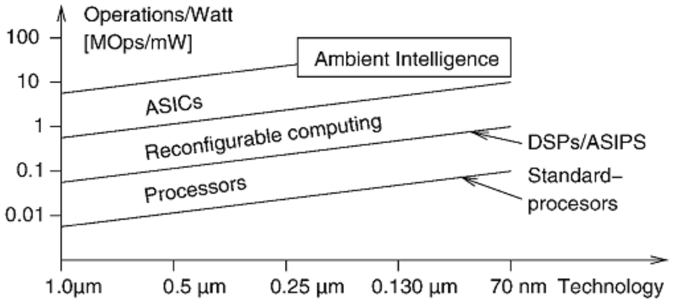


An ideal sample-and-hold circuit would be able to change the voltage at the capacitor in an arbitrarily short amount of time. This way, the input voltage at a particular instance in time could be transfered to the capacitor and each element in the discrete sequence would correspond to the input voltage at a particular point in time. In practice, however, the transistor has to be kept closed for a short time window in order to really charge or discharge the capacitor. The voltage stored on the capacitor will then correspond to a voltage averaged over that short time window.

Processing Units

For information processing, we will consider ASICs (application-specific integrated circuits) using hardwired multiplexed designs, reconfigurable logic, and processors. These three technologies are quite different, for example, as far as their energy efficiency is concerned. Fig.shows the number of operations per Watt that can be achieved with a certain hardware technology.

Hardware Efficiency



Obviously, the number of operations per Watt is increasing as technology advances to smaller and smaller feature sizes of integrated circuits. However, for any given technology, the number of operations per Watt is largest for application specific hardwired circuits. For reconfigurable logic, this value is about one order of magnitude lower. For programmable processors, it is about two orders of magnitude lower. On the other hand, processors offer the largest amount of flexibility, resulting from the flexibility of software. There is also some flexibility for reconfigurable logic, but it is limited to the size of applications that can be mapped to such logic. For hardwired designs, there is no flexibility. This observation also applies for processors: For processors optimized for the application domain, such as processors optimized for digital signal processing (DSP processors), power-efficiency values approach those of reconfigurable logic. For general standard microprocessors, the values for this figure of merit are the worst. The energy E for a certain application is closely related to the power P required per operation, since

$$E = \int P dt$$

Hence, reducing the power consumption also decreases the energy consumption, provided that the integral is taken over the same period of time. In some cases, however, a slightly increased power consumption might lead to a drastic reduction in the execution time and, hence, might lead to a minimized energy consumption. So, in some cases a minimized power consumption also corresponds to a minimized energy consumption, but this is not necessarily always true.

Minimization of power and energy consumption are both important. Power consumption has an effect on the size of the power supply, the design of the voltage regulators, the dimensioning of the interconnect, and short term cooling. Minimizing the energy consumption is required especially for mobile applications, since battery technology is only slowly improving , and since the cost of energy may be quite high. Also, a reduced energy consumption decreases cooling requirements and improves the reliability. Fig. reflects the efficiency/flexibility conflict of currently available hardware technologies: if we want to aim at very power- and energy-efficient designs, we should not use flexible designs based on processors or re-programmable logic and if we go for excellent flexibility, we cannot be power-efficient.We will consider ASICs first.

Application-Specific Circuits (ASICs)

For high-performance applications and for large markets, application-specific integrated circuits (ASICs) can be designed. However, the cost of designing and manufacturing such chips is quite high. For example, the cost of the mask which is used for transferring geometrical patterns onto the chip can cost about 10^5 Euros or dollars. Therefore, ASICs are appropriate only if either maximum energy efficiency is needed and if the market accepts the costs or if a large number of such systems can be sold.

Processors

The key advantage of processors is their flexibility. With processors, the overall behavior of embedded systems can be changed by just changing the software running on those processors. Changes of the behavior may be required in order to correct design errors, to update the system to a new or changed standard or in order to add features to the previous system. Because of this, processors have become very popular. This popularity has also been stressed in the public press:

At the chip level, embedded chips include micro-controllers and microprocessors. Micro-controllers are the true workhorses of the embedded family. They are the original ’embedded chips’ and include those first employed as controllers in elevators and thermostats. Embedded processors have to be efficient and they do not need to be instruction set compatible with commonly used personal computers (PCs). Therefore, their architectures may be different from those processors found in PCs.

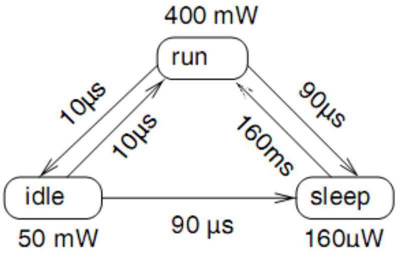
Energy-efficiency:

Architectures have to be optimized for their energy efficiency and we have to make sure that we are not loosing efficiency in the software generation process. For example, compilers generating 50% overhead in terms of the number of cycles will take us further away from the efficiency of ASICs, possibly by even more than 50%, if the supply voltage and the clock frequency have to be increased in order to meet deadlines.

There is a large amount of techniques available that can make processors energy efficient and energy efficiency should be considered at various levels of abstraction, from the design of the instruction set down to the design of the chip manufacturing process [Burd and Brodersen, 2003]. Gated clocking is an example of such a technique. With gated clocking, parts of the processor are decoupled from the clock during idle periods. For example, no clock is applied to the multiplier if no multiplications are executed. Also, there are attempts, to get rid of the clock for major parts of the processor altogether. There are two contrasting approaches: globally synchronous, locally asynchronous processors and globally asynchronous, locally synchronous processors (GALS) [Iyer and Marculescu, 2002]. Two techniques can be applied at a rather high level of abstraction:

Dynamic power management (DPM):

With this approach, processors have several power saving states in addition to the standard operating state. Each power saving state has a different power consumption and a different time for transitions into the operating state. Fig. shows the three states for the StrongArm SA 1100 processor.



The processor is fully operational in the run state. In the idle state, it is just monitoring the interrupt inputs. In the sleep state, all on-chip activity is shutdown. Note the large difference in the power consumption between the sleep state and the other states, and note also the large delay for transitions from the sleep to the run state.

Dynamic voltage scaling (DVS):

This approach exploits the fact that the energy consumption of CMOS processors increases with the supply voltage Vdd . The power consumption P of CMOS circuits is given by

$$P = \alpha C_L V_{dd}^2 f$$

where α is the switching activity, CL is the load capacitance, Vdd is the supply voltage and f is the clock frequency. The delay of CMOS circuits can be approximated as

$$\tau = k \cdot C_L \cdot \frac{V_{dd}}{(V_{dd} - V_t)^2}$$

where k is a constant, and Vt is the threshold voltage. Vt has an impact on the transistor input voltage required to switch the transistor on. For example, for a maximum supply voltage of Vdd,max=3.3 volts, Vt may be in the order of 0.8 volts. Consequently, the maximum clock frequency is a function of the supply voltage. However, decreasing the supply voltage reduces the power quadratically, while the run-time of algorithms is only linearly increased (ignoring the effects of the memory system). This can be exploited in a technique called dynamic voltage scaling (DVS). For example, the CrusoeTM processor by Transmeta provides 32 voltage levels between 1.1 and 1.6 volts, and the clock can be varied between 200 MHz and 700 MHz in increments of 33 MHz. Transitions from one voltage/frequency pair to the next takes about 20 ms. Design issues for DVS-capable processors are described in a paper by Burd and Brodersen. According to the same paper, potential power savings will exist even for future technologies with a decreased maximum Vdd , since the threshold voltages will also be decreased (unfortunately, this will lead to increased leakage currents, increasing the standby power consumption). Two different speed/voltage pairs are provided with the Intel SpeedStepTM technology for the Mobile Pentium III.

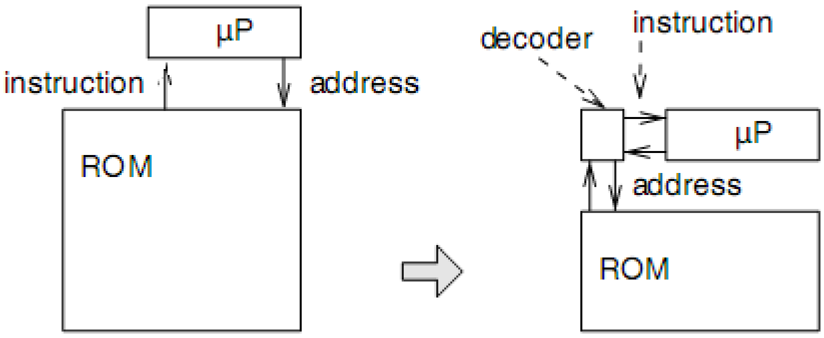
Code-size efficiency:

Minimizing the code size is very important for embedded systems, since hard disc drives are typically not available and since the capacity of memory is typically also very limited. This is even more pronounced for systems on a chip (SOCs). For SOCs, the memory and processors are implemented on the same chip. In this particular case, memory is called embedded memory. Embedded memory may be more expensive to fabricate than separate memory chips, since the fabrication processes for memories and processors have to be compatible. Nevertheless, a large percentage of the total chip area may be consumed by the memory. There are several techniques for improving the code-size efficiency:

– CISC machines: Standard RISC processors have been designed for speed, not for code-size efficiency. Earlier Complex Instruction Set Processors (CISC machines) were actually designed for code-size efficiency, since they had to be connected to slow memories and caches were not frequently used. Therefore, “old-fashioned” CISC processors are finding applications in embedded systems. Motorola’s ColdFire processors, which are based on the Motorola 68000 family of CISC processors are an example of this.

– Compression techniques: In order to reduce the amount of silicon needed for storing instructions as well as in order to reduce the energy needed for fetching these instructions, instructions are frequently stored in the memory in compressed form. This reduces both the area as well as the energy necessary for fetching instructions. Due to the reduced bandwidth requirements, fetching can also be faster. A (hopefully small and fast) decoder is placed between the processor and the (instruction) memory in order to generate the original instructions on the fly (see fig. 3.11, right). Instead of using a potentially large memory of uncompressed instructions, we are storing the instructions in a compressed format.

Decompression of compressed instructions



The goals of compression can be summarized as follows:

∗ We would like to save ROM and RAM areas, since these may be more expensive than the processors themselves.

∗ We would like to use some encoding technique for instructions and possibly also for data with the following properties:

· There should be little or no run-time penalty for these techniques.

· Decoding should work from a limited context (it is, for example, impossible to read the entire program to find the destination of a branch instruction).

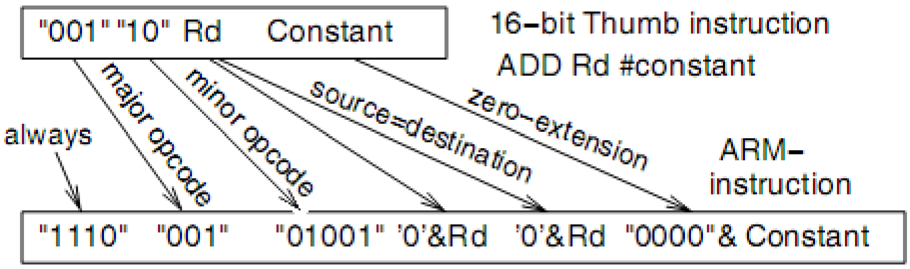
· Word-sizes of the memory, of instructions and addresses have to be taken into account.

· Branch instructions branching to arbitrary destination addresses have to be supported.

· Fast encoding is only required if writable data is encoded. Otherwise, fast decoding is sufficient.

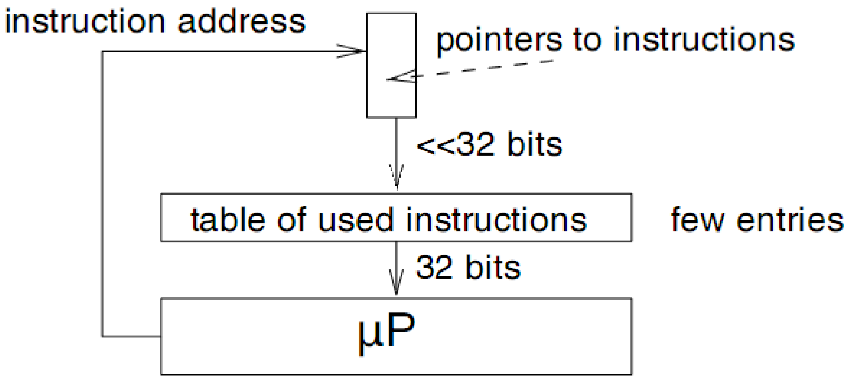
There are several variations of this scheme:

∗ For some processors, there is a second instruction set. This second instruction set has a narrower instruction format. An example of this is the ARM processor family. The ARM instruction set is a 32 bit instruction set. The ARM instruction set includes predicated execution. This means an instruction is executed if and only if a certain condition is met (see page 113). This condition is encoded in the first four bits of the instruction format. Most ARM processors also provide a second instruction set, with 16 bit wide instructions, called THUMB instructions. THUMB instructions are shorter, since they do not support predication, use shorter and less register fields and use shorter immediate fields. (see fig.).



THUMB instructions are dynamically converted into ARMinstructions while programs are running. THUMB instructions can use only half the registers in arithmetic instructions. Therefore, register fields of THUMB instructions are concatenated with a '0'-bit. In the THUMB instruction set, source and destination registers are identical and the length of constants that can be used, is reduced by 4 bits. During decoding, pipelining is used to keep the run-time penalty low. Similar techniques also exist for other processors. The disadvantage of this approach is that the tools (compilers, assemblers, debuggers etc.) have to be extended to support a second instruction set. Therefore, this approach can be quite expensive in terms of software development cost.

∗ A second approach is the use of dictionaries. With this approach, each instruction pattern is stored only once. For each value of the program counter, a look-up table then provides a pointer to the corresponding instruction in the instruction table, the dictionary (see fig.).



This approach relies on the idea that only very few different instruction patterns are used. Therefore, only few entries are required for the the instruction table. Correspondingly, the bit width of the pointers can be quite small. Many variations of this scheme exist. Some are called two-level control store.

Run-time efficiency:

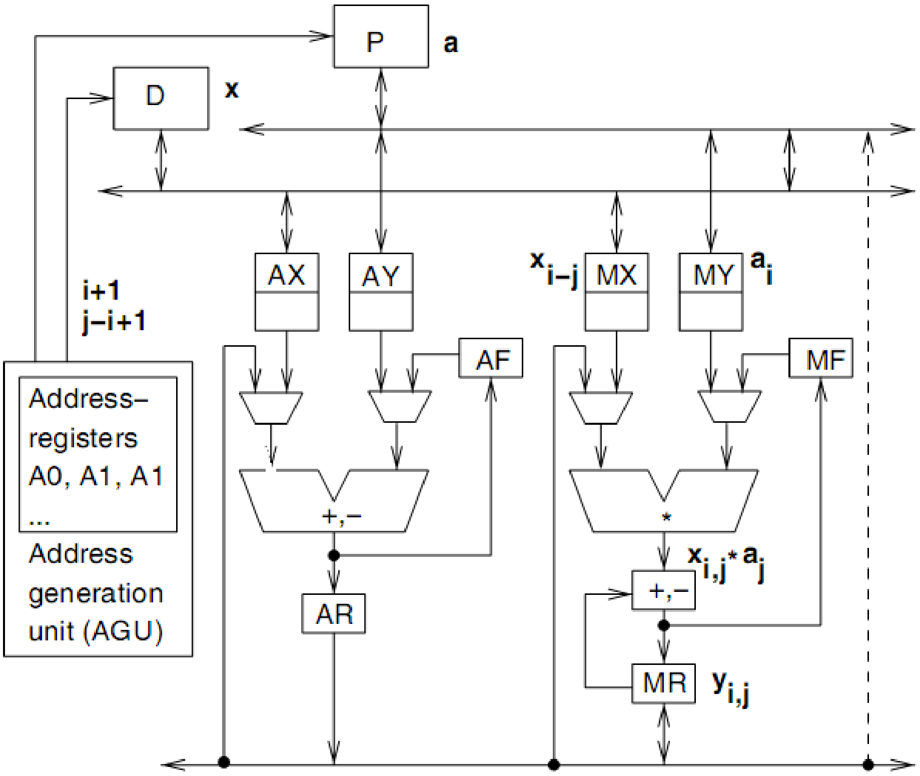
In order to meet time constraints without having to use high clock frequencies, architectures can be customized to certain application domains, such as digital signal processing (DSP). One can even go one step further and design application specific instruction set processors(ASIPs). As an example of domain-specific processors, we will consider processors for DSP. In digital signal processing, digital filtering is a very frequent operation. Equation 3.3 describes a digital filter generating an output sequence y =(y^0,y^1, ...) from an input sequence x =(x^0,x^1, ...).

$$y_i = \sum_{j=0}^{n-1} x_{i-j} * a_j$$

A certain output element yi corresponds to a weighted average over the last n sequence elements of x and can be computed iteratively using following equations.

$$\begin{aligned} y_{i,j} &= y_{i,j-1} + x_{i-j} * a_j \\ \text{where} &: y_{i,-1} = 0 \\ \text{and} &: y_i = y_{i,n-1} \end{aligned}$$

DSPs are designed such that each iteration can be encoded as a single instruction. Let us consider an example. Fig.shows the internal architecture of an ADSP 2100 DSP processor.



The processor has two memories, called D and P. A special address generating unit (AGU) can be used to provide the pointers for accessing these memories. There are separate units for additions and multiplications, each with their own argument registers AX, AY, AF, MX, MY and MF. The multiplier is connected to a second adder in order to compute series of multiplications and additions quickly.

For this processor, the update of the partial sum is essentially performed in a single cycle. For this purpose, the two memories are allocated to hold the two arrays x and a and address registers are allocated such that relevant pointers can be easily updated in the AGU. Partial sums yi,j are stored in MR. The pipelined computation involves registers A1, A2, MX and MY,as can be seen from the following implementation of the filter.

```
MR:=0;  A1:=1;  A2:=n-2;  MX:=x[n-1];  MY:=a[0];
for (j=1;  j<=n; j++)
  {MR:=MR + MX * MY;   MX:=x[A2];   MY:=a[A1];
   A1++; A2-- }
```

A single instruction encodes the loop body, comprising the following operations:

– reading of two arguments from argument registers MX and MY, multiplying them and adding the product to register MR storing values yi,j, - fetching the next elements of arrays a and x from memories P and D and storing them in argument registers MX and MY,

– updating pointers to the next arguments, stored in address registers A1

and A2,

– testing for the end of the loop.

This way, each iteration requires just a single instruction. In order to achieve this, several operations are performed in parallel. For given computational requirements, this (limited) form of parallelism leads to relatively low clock frequencies. Furthermore, the registers in this architecture perform different functions. They are said to be heterogeneous. Heterogeneous register files are a common characteristic for DSP processors. In order to avoid extra cycles for testing for the end of the loop, zero-overhead loop instructions are frequently provided in DSP processors. With such instructions, a single or a small number of instructions can be executed a fixed number of times. Processors not optimized for DSP would probably need several instructions per iteration and would therefore require a higher clock frequency, if available.

DSP-Processors

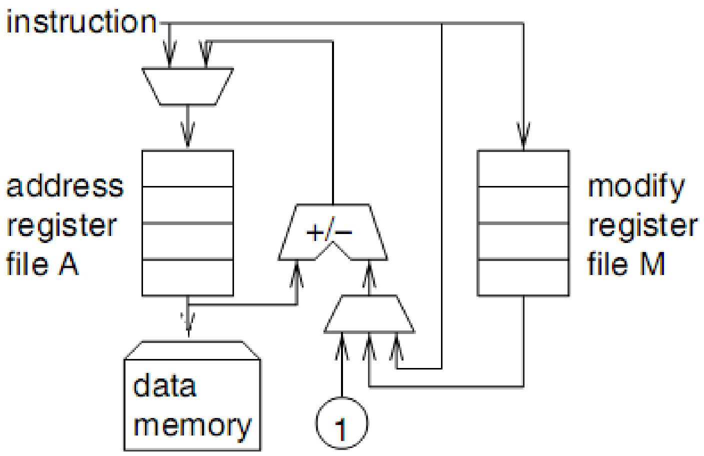
In addition to allowing single instruction realizations of loop bodies for filtering, DSP processors provide a number of other application-domain orientedfeatures:

Specialized addressing modes:

In the filter application described above, only the last n elements of x need to be available. Ring buffers can be used for that. These can be implemented easily with modulo addressing. In modulo addressing, addresses can be incremented and decremented until the first or last element of the buffer is reached. Additional increments or decrements will result in addresses pointing to the other end of the buffr.

Separate address generation units:

Address generation units (AGUs) are typically directly connected to the address input of the data memory (see fig.).

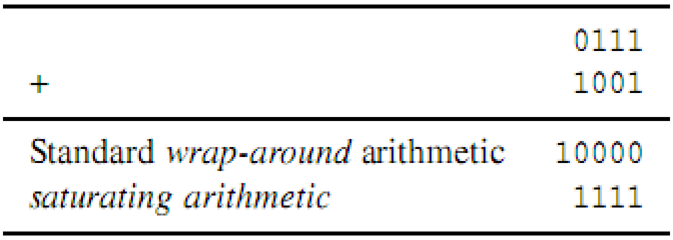


AGU using special address registers

Addresses which are available in address registers can be used in register-indirect addressing modes. This saves machine instructions, cycles and energy. In order to increase the usefulness of address registers, instruction sets typically contain auto-increment and -decrement options for mostinstructions using address registers.

Saturating arithmetic: Saturating arithmetic changes the way overflows and underflows are handled. In standard binary arithmetic, wrap-around is used for the values returned after an overflow or underflow. Fig. shows an example in which two unsigned four-bit numbers are added. A carry is generated which cannot be returned in any of the standard registers. The result register will contain a pattern of all zeros. No result could be further away from the true result than this one.

Wrap-around vs. saturating arithmetic for unsigned integers

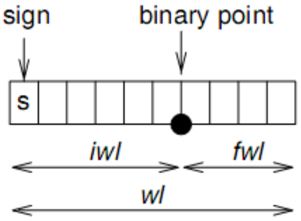


	0111
+	1001
Standard *wrap-around* arithmetic	10000
saturating arithmetic	1111

In saturating arithmetic, we try to return a result which is as close as possible to the true result. For saturating arithmetic, the largest value is returned in the case of an overflow and the smallest value is returned in the case of an underflow. This approach makes sense especially for video and audio applications: the user will hardly recognize the difference between the true result value and the largest value that can be represented. Also, it would be useless to raise exceptions if overflows occur, since it is difficult to handle exceptions in real-time. Note that we need to know whether we are dealing with signed or unsigned add instructions in order to return the right value.

Fixed-point arithmetic: Floating-point hardware increases the cost and power-consumption of processors. Consequently, it has been estimated that 80 % of the DSP processors do not include floating-point hardware. However, in addition to supporting integers, many such processors do support fixed-point numbers. Fixed-point data types can be specified by a 3-tuple (wl, iwl, sign),where wl is the total word-length, iwl is the integer word-length (the number of bits left of the binary point), and sign s ∈{s,u} denotes whether we are dealing with unsigned or signed numbers. See also fig. Furthermore, there may be different rounding modes (e.g. truncation) and overflow modes (e.g. saturating and wrap-around arithmetic).

Parameters of a fixed-point number system



For fixed-point numbers, the position of the binary point is maintained after multiplication (some low order bits are truncated or rounded). For fixedpoint processors, this operation is supported by hardware.

Real-time capability: Some of the features of modern processors used in PCs are designed to improve the average execution time of programs. In many cases, it is difficult if not impossible to formally verify that they improve the worst case execution time. In such cases, it may be better not to implement these features. For example, it is difficult (though not impossible [Absint, 2002]) to guarantee a certain speedup resulting from the use of caches. Therefore, many embedded processors do not have caches. Also, virtual addressing and demand paging are normally not found in embedded systems.

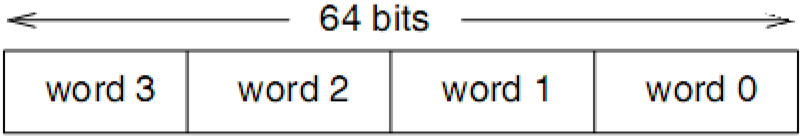
Multiple memory banks or memories: the usefulness of multiple memory banks was demonstrated in the ADSP 2100 example: the two memories D and P allow fetching both arguments at the same time. Several DSP processors come with two memory banks.

Heterogenous register files: heterogenous register files were already mentioned for the filter application.

Multiply/accumulate instructions: these instructions perform multiplications followed by additions. They were also already used in the filter application.

Multimedia processors

Registers and arithmetic units of many modern architectures are 64 bits wide. Therefore, two 32 bit data types (“double words”), four 16 bit data types (“words”) or eight 8 bit data types (“bytes”) can be packed into a single register (see fig.).



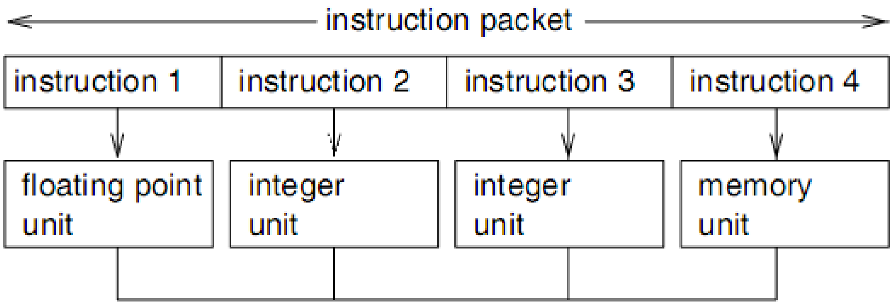
Using 64 bit registers for packed words

Arithmetic units can be designed such that they suppress carry bits at double word, word or byte boundaries. Multimedia instruction sets exploit this fact by supporting operations on packed data types. Such instructions are sometimes called single-instruction, multiple-data (SIMD) instructions, since a single instruction encodes operations on several data elements. With bytes packed into 64-bit registers, speed-ups of up to about eight over non-packed data types are possible. Data types are typically stored in packed form in memory. Unpacking and packing are avoided if arithmetic operations on packed data types are used. Furthermore, multimedia instructions can usually be combined with saturating arithmetic and therefore provide a more efficient form of overflow handling than standard instructions. Hence, the overall speed-up achieved with multimedia instructions can be significantly larger than the factor of eight enabledby operations on packed data types.

Very long instruction word (VLIW) processors

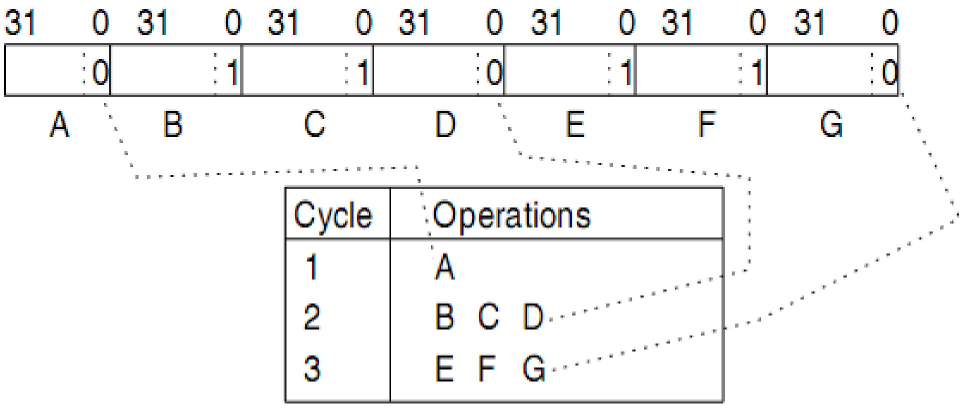
Computational demands for embedded systems are increasing, especially when multimedia applications, advanced coding techniques or cryptography are involved. Performance improvement techniques used in high-performance microprocessors are not appropriate for embedded systems: driven by the need for instruction set compatibility, processors found, for example, in PCs spend a huge amount of resources and energy on automatically finding parallelism in application programs. Still, their performance is frequently not sufficient. For embedded systems, we can exploit the fact that instruction set compatibility with PCs is not required. Therefore, we can use instructions which explicitely identify operations to be performed in parallel. This is possible with explicit parallelism instruction set computers (EPICs). With EPICs, detection of parallelism is moved from the processor to the compiler. This avoids spending silicon and energy on the detection of parallelism at runtime. As a special case, we consider very long instruction word (VLIW) processors. For VLIW processors, several operations or instructions are encoded in a long instruction word (sometimes called instruction packet) and are assumed to be executed in parallel. Each operation/instruction is encoded in a separate field of the in struction packet. Each field controls certain hardware units. Four such fields are used in fig., each one controlling one of the hardware units.

VLIW architecture (example)



For VLIWarchitectures, the compiler has to generate instruction packets. This requires that the compiler is aware of the available hardware units and to schedule their use. EPICs are sometimes also used for PCs [Transmeta, 2005, Intel, 2005]. However, legacy problems result in severe constraints for doing this.Instruction fields must be present, regardless of whether or not the corresponding functional unit is actually used in a certain instruction cycle. As a result, the code density of VLIW architectures may be low, if insufficient parallelism is detected to keep all functional units busy. The problem can be avoided if more flexibility is added. For example, the Texas Instruments TMS 320C6xx family of processors implements a variable instruction packet size of up to 256 bits. In each instruction field, one bit is reserved to indicate whether or not the operation encoded in the next field is still assumed to be executed in parallel (see fig.). No instruction bits are wasted for unused functional units.

Instruction packets for TMS 320C6xx

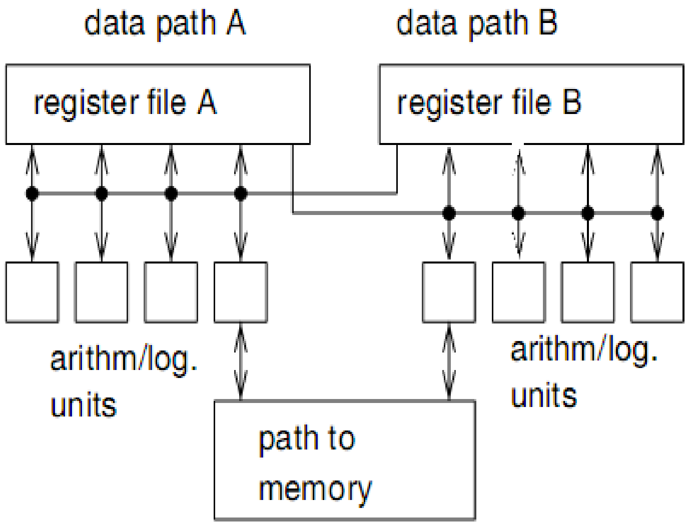


Cycle	Operations
1	A
2	B C D
3	E F G

Due to its variable length instruction packets, TMS 320C6xx processors do not quite correspond to the classical model of VLIW processors. Due to their explicit description of parallelism, they are EPIC processors, though. Partitioned Register Files. Implementing register files for VLIW and EPIC processors is far from trivial. Due to the large number of operations that can be performed in parallel, a large number of register accesses has to be provided in parallel. Therefore, a large number of ports is required. However, the delay, size and energy consumption of register files increases with their number of ports. Hence, register files with very large numbers of ports are inefficient. As a consequence, many VLIW/EPIC architectures use partitioned register files. Functional units are then only connected to a subset of the register files. As an example, fig. shows the internal structure of the TMS 320C6xx processors. These processors have two register files and each of them is connected to half of the functional units. During each clock cycle, only a single path from one register file to the functional units connected to the other register file is available.

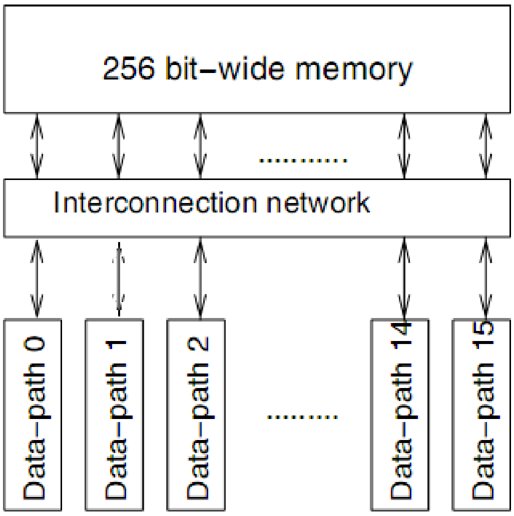
Many DSP processors are actually VLIW processors. As an example, we are considering the M3-DSP processor [Fettweis et al., 1998].

Partitioned register files for TMS 320C6xx

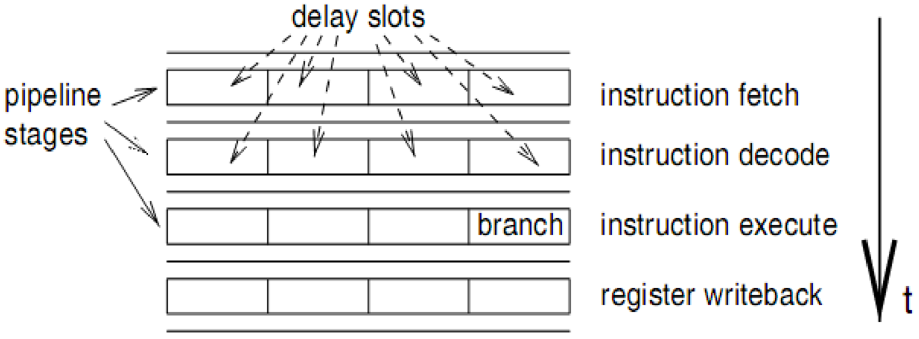


cessor is a VLIW processors containing (up to) 16 parallel data paths. These data paths are connected to a group memory, providing the necessary arguments in parallel (see fig.).

M3-DSP (simplified)



Predicated Execution. A potential problem of VLIW and EPIC architectures is their potentially large delay penalty: This delay penalty might originate from branch instructions found in some instruction packets. Instruction packets normally have to pass through pipelines. Each stage of these pipelines implements only part of the operations to be performed by the instructions executed. The fact that branch instructions exist cannot be detected in the first stage of the pipeline. When the execution of the branch instruction is finally completed, additional instructions have already entered the pipeline (see fig.).



Branch instruction and delay slots

There are essentially two ways to deal with these additional instructions:

1 They are executed as if no branch had been present. This case is called delayed branch. Instruction packet slots that are still executed after a branch are called branch delay slots. These branch delay slots can be filled with instructions which would be executed before the branch if there were no delay slots. However, it is normally difficult to fill all delay slots with useful instructions and some have to be filled with no-operation instructions (NOPs). The term branch delay penalty denotes the loss of performance resulting from these NOPs.

2 The pipeline is stalled until instructions from the branch target address have been fetched. There are no branch delay slots in this case. In this organization the branch delay penalty is caused by the stall. Branch delay penalties can be significant. For example, the TMS 320C6xx family of processors has up to 40 delay slots. Therefore, efficiency can be improved by avoiding branches, if possible. In order to avoid branches originating from if-statements, predicated instructions have been introduced. For each predicated instruction, there is a predicate. This predicate is encoded in a few bits and evaluated at run-time. If the result is true, the instruction is executed. Otherwise, it is effectively turned into a NOP. Predication can also be found in RISC machines such as the ARM processor. Example: ARM instructions, as introduced on page 104, include a four-bit field. These four bits encode various expressions involving the condition code registers. Values stored in these registers are checked at run-time. They determine whether or not a certain instruction has an effect. Predication can be used to implement small if-statements efficiently: the condition is stored in one of the condition registers and if-statement-bodys are implemented as predicated instructions which depend on this condition. This way, if-statement bodys can be evaluated in parallel with other operations and no delay penalty is incurred.

Summary:

- A processor is said to be single-purpose processor performs specific computation task, custom single-purpose processors
- Timer measures time intervals to generate, timed output events & to measure input events
- A Counter is like a timer, but counts pulses on a general input signal rather than clock
- A watchdog timer can be treated as a count down timer which executes a special code upon its expire.
- For high-performance applications and for large markets, application-specific integrated circuits (ASICs) can be designed.
- At the chip level, embedded chips include micro-controllers and microprocessors. Micro-controllers are the true workhorses of the embedded family. They are the original ’embedded chips’ and include those first employed as controllers in elevators and thermostats.

Reference Questions:

1. Write a brief notes on Application specific processors?
2. Write a brief notes on multimedia processors?
3. Write a brief notes on VLIW processors?
4. What is a sensor? Explain different types of sensors.

5. Explain the following
    a. Timers
    b. Counters
    c. Watchdog timers.

References:

- Catsoulis J.Designing embedded hardware.2005
- Embedded_Controller_Hardware_Designby ken arnold
- Embedded systems by raj kamal
- www.embedded.com

UNIT – III

7. MEMORY

Objective:

Memory chip connected in the embedded device may be external or internal. If micro processor is used as controlling device the memory connected is external and if micro controller is used memory is internal. Wherever we place memory it communicates with the CPU using data and address busses. Each controlling device has a specific addressing range. An addressing range is the number of addresses a controller can access. The addressing scheme used to access to these spaces varies from processor to processor, but the underlying hardware is similar. The different types of memories available are explained below.

RAM

Random access memory1 or RAM consists of memory addresses the CPU can both read from and write to. RAM is used for data memory and allows the CPU to create and modify data as it executes the application program. RAM is volatile; it holds its contents only as long as it has a constant power supply. If power to the chip is turned off, the contents of RAM are lost. This does not mean that RAM contents are lost during a chip reset. Vital state information or other data can be recorded in data memory and recovered after an interrupt or reset. Some chips provide an alternate RAM power supply so that memory contents can be maintained even when the rest of the chip is without power. This does not make RAM any less volatile, without a backup power source the contents would still be lost. This type of RAM is called **battery backed-up static RAM**.

ROM

ROM, read only memory, is typically used for program instructions. The ROM in a microcontroller usually holds the final application program. Maskable ROM is memory space that must be burned in by the manufacturer of the chip as it is constructed. To do this, you must provide the chip builder with the ROM contents you wish the chip to have. The manufacturer will then mask out appropriate ROM blocks and hardwire the information you have provided. Since recording chip ROM contents is part of the manufacturing process, it is a costly one-time expense. If you intend to use a small number of parts, you may be better off using chips with PROM. If you intend to use a large number of parts for your application, then the one-time expense of placing your program in ROM is more feasible.

PROM

Programmable ROM, or PROM, started as an expensive means to prototype and test application code before burning ROM. In recent years PROM has gained popularity to the point where many developers consider it a superior alternative to burning ROM. As microcontroller applications become more specialized and complex, needs for maintenance and support rise. Many developers use PROM devices to provide software updates to customers without the cost of sending out new hardware. There are many programmable ROM technologies available which all provide a similar service. A special technique is used to erase the contents of programmable ROM then a special method is used to program new instructions into the ROM. Often, the developer uses separate hardware to perform each of these steps.

EPROM

EPROM (erasable programmable ROM) is not volatile and is read only. Chips with EPROM have a quartz window on the chip. Direct exposure to ultra-violet radiation will erase the EPROM contents. EPROM devices typically ship with a shutter to cover the quartz window and prevent ambient UV from affecting the memory. Often the shutter is a sticker placed on the window. Developers use an EPROM eraser to erase memory contents efficiently. The eraser bombards the memory with high-intensity UV light. To reprogram the chip, an EPROM programmer is used, a device which writes instructions into EPROM. The default, blank state for an EPROM device has each block of memory set. When you erase an EPROM you are really setting all memory blocks to 1. Reprogramming the device resets or clears the appropriate EPROM bits to 0. Because of the way EPROM storage is erased, you cannot selectively delete portions of EPROM when you erase the memory you must clear the entire storage space.

EEPROM

EEPROM (electrically erasable programmable ROM) devices have a significant advantage over EPROM devices as they allow selective erasing of memory sections. EEPROM devices use high voltage to erase and re-program each memory block. Some devices require an external power source to provide the voltage necessary for erasing and writing and some have an onboard pump which the chip can use to build up a charge of the required voltage. Developers can reprogram EEPROM devices while the chip is operating. However, EEPROM that can be rewritten is usually restricted to data memory storage. EEPROM storage used as program memory typically requires the use of an external power source and a programmer just like EPROM storage. The most common use for EEPROM is recording and maintaining configuration data vital to the application. For example, many modems use EEPROM storage to record the current configuration settings. This makes the configuration available to the modem user after cycling the power on the modem. Often the default or factory configuration settings are stored in ROM and the user can issue a command to restore default settings by overwriting the current contents of EEPROM with the default information. Sometimes chip manufacturers build EEPROM blocks into the chip for last-minute configuration options. This saves manufacturers money as they can design and fabricate a single chip and then set the EEPROM blocks to provide special purpose versions with specific capabilities. This method is often used to produce microcontroller versions for use on an evaluation board where chip access to its own onboard ROM is turned off and replaced with external EPROM or EEPROM storage. This allows developers to test application code in cycles by downloading it to the board, programming the code into the EPROM or EEPROM, and debugging it as it executes in the target hardware.

Flash Memory

Flash memory is an economical compromise between EEPROM and EPROM technology. As with EEPROM high voltage is applied to erase and rewrite flash memory. However, unlike EEPROM, you cannot selectively erase portions of flash memory – you must erase the entire block as with EPROM devices. Many manufacturers are turning to flash memory. It has the advantages of not requiring special hardware and being inexpensive enough to use in quantity. Manufacturers often provide customers with microcontroller products whose ROM is loaded with a boot or configuration kernel where the application code is written into flash memory. When the manufacturer wants to provide the customer with added functionality or a maintenance update, the hardware can be reprogrammed on site without installing new physical parts. The hardware is placed into configuration mode which hands control to the kernel written in ROM. This kernel then handles the software steps needed to erase and re-write the contents of the flash memory. Another useful implementation of flash memory includes a device which can connect electronically to a computer owned by the manufacturer. The configuration kernel connects to the manufacturer's computer, downloads the latest version of the control application and writes this application to flash memory. Such elaborate applications are typically beyond the resources of an 8 bit microcontroller; we mention the example to show the advantage of programmable ROM technologies.

Registers

The CPU maintains a set of registers which it uses to store information. Registers are used to control program execution and maintain intermediate values needed to perform required calculations. Some microcontrollers provide access to CPU registers for temporary storage purposes. This can be *extremely* Dangerous as the CPU can at any time overwrite a register being used for its designated purpose 8 bit microcontrollers do not often provide resources for register memory outside the CPU. This means that the C register keyword is meaningless because the compiler cannot dedicate a CPU register for data storage. Some C implementations will set aside RAM for special purpose *pseudo-registers* to use when your application attempts certain operations. For example, if you attempt a 16 bit math operation, the compiler can dedicate a portion of base page RAM for 16 bit pseudo-registers which store values during math operations. You can use these special registers for temporary purposes in places where your code will not require them for their intended purpose. You must be careful, if the compiler uses a pseudo-register it will overwrite current contents.

Memory Management

In order to understand what memory management is, it's helpful to understand the motivation behind its use. There are two kinds of memory management: memory address relocation and memory performance enhancement. They are often used in conjunction, as is commonly done in personal computers. This section covers the performance enhancement aspects, while the address relocation issues will be covered in Chapter Six. The differences between different storage technologies, in terms of performance and cost, vary over many orders of magnitude. For example, semiconductor memory devices have access times that are many orders of magnitude faster (nanosecond vs. millisecond access time) than that of magnetic disks. Of course, magnetic disks also have a cost several orders of magnitude less than semiconductor memory on a cost per bit basis. This disparity in price and performance has lead to the idea of using small, fast memories to store the most frequently accessed subset of the complete collection of data present in a larger, slower memory. This technique of buffering, often referred to as *caching* memory contents in a fast memory, is essentially similar whether it is applied to the memory attached to a CPU or the magnetic or optical storage mechanisms. In fact, there may be several layers of caching in a given system, starting with the smallest, fastest memory closest to the CPU, followed by slower but larger memories.

Cache Memory

When a high speed memory is used to provide rapid access to the CPU for most frequently used portion provide rapid access to data stored on a disk, it is referred to as a *disk cache*. The objective of these approaches is to maximize the likelihood that most pieces of data will be found in the small and fast memory most of the time, thus reducing the average effective access time. The object is to succeed at finding most data in the small fast memory most of the time, minimizing the number of accesses to the big slow memory. Fast SRAM is used as a fast temporary buffer (memory cache) between main memory and the CPU. Main memory DRAM is used to buffer disk data (disk cache). Most hard disk drives also have some internal fast semiconductor RAM to cache data as it is being transferred to and from the disk.

Virtual Memory

Disk storage can be used to emulate a larger primary memory than is actually available. *Demand paged virtual memory* provides an apparently large primary memory by swapping pages of data between real primary memory and disk. This is a combination of hardware for translating logical (virtual) addresses, moving pages as needed, and operating system software to determine where and when pages should be kept and detect access attempts to pages which are not in primary memory. When address relocation mechanisms are combined with disk caching and special system software, it is possible to make the main memory appear much larger than it actually is to a program running on this type of machine. When the program attempts to access a location that is not present in the main memory, the hardware and software redirect the memory reference to a real block of memory, after the required data is loaded from disk. Thus the application program is presented with a virtual memory that is significantly larger than the actual physical main memory. This has the effect of simplifying the code, since all data can be referenced by a single address, rather than selecting a file, track, or sector on a disk.

CPU Control Lines for Memory Interfacing

Some CPUs generate signals for memory timing and synchronization with devices having various access times using a technique that generates delay cycles for slow memories, referred to as *wait states*. The 8051 processor used in this text does not use or generate wait states for simplicity. The Dallas 80C320 series of high speed microcontrollers incorporate a software-controlled mechanism for generating wait states. These extended memory cycles allow the processor to work with slower memory and peripheral chips.

DIRECT MEMORY ACCESS:

Direct memory access (DMA) is a means of having a peripheral device control a processor's memory bus directly. DMA permits the peripheral, such as a UART, to transfer data directly to or from memory without having each byte (or word) handled by the processor. Thus DMA enables more efficient use of interrupts, increases data throughput, and potentially reduces hardware costs by eliminating the need for peripheral-specific FIFO buffers. In a typical DMA transfer, some event (such as an incoming data-available signal from a UART) notifies a separate device called the *DMA controller* that data needs to be transferred to memory. The DMA controller then asserts a *DMA request* signal to the CPU, asking its permission to use the bus. The CPU completes its current bus activity, stops driving the bus, and returns a *DMA acknowledge* signal to the DMA controller. The DMA controller then reads and writes one or more memory bytes, driving the address, data, and control signals as if it were itself the CPU. (The CPU's address, data, and control outputs are restated while the DMA controller has control of the bus.) When the transfer is complete, the DMA controller stops driving the bus and desserts the DMA request signal. The CPU can then remove its DMA acknowledge signal and resume control of the bus. Each DMA cycle will typically result in at least two bus cycles: either a peripheral read followed by a memory write or a memory read followed by a peripheral write, depending on the transfer base addresses. The DMA controller itself does no processing on this data. It just transfers the bytes as instructed in its configuration registers. It's possible to do a flyby transfer that performs the read and write in a single bus cycle. However, though supported on the ISA bus and its embedded cousin PC/104, flyby transfers are not typical. Processors that support DMA provide one or more input signals that the bus requester can assert to gain control of the bus and one or more output signals that the processor asserts to indicate it has relinquished the bus. A typical output signal might be named HLDA (short for HoLD Acknowledge). When designing with DMA, address buffers must be disabled during DMA so the bus requester can drive them without bus contention. To avoid bus contention, the bus buffer used by the DMA device must not drive the address bus until after HLDA goes active to indicate that the CPU has stopped driving the bus signals, and it must stop driving the bus before the CPU drives HLDA inactive. The system design may also need pull-up resistors or terminators on control signals (such as read and write strobes) so the control signals don't float to the active state during the brief period when neither the processor nor the DMA controller is driving them. DMA controllers require initialization by software. Typical setup parameters include the base address of the source area, the base address of the destination area, the length of the block, and whether the DMA controller should generate a processor interrupt once the block transfer is complete. It's typically possible to have the DMA controller automatically increment one or both addresses after each byte (word) transfer, so that the next transfer will be from the next memory location. Transfers between peripherals and memory often require that the peripheral address not be incremented after each transfer. When the address is not incremented, each data byte will be transferred to or from the same memory location. DMA operations can be performed in either burst or single-cycle mode. Some DMA controllers support both. In burst mode, the DMA controller keeps control of the bus until all the data buffered by the requesting device has been transferred to memory (or when the output device buffer is full, if writing to a peripheral). In single-cycle mode, the DMA controller gives up the bus after each transfer. This minimizes the amount of time that the DMA controller keeps the processor off of the memory bus, but it requires that the bus request/acknowledge sequence be performed for every transfer. This overhead can result in a drop in overall system throughput if a lot of data needs to be transferred. In most designs, you would use single cycle mode if your system cannot tolerate more than a few cycles of added interrupt latency. Likewise, if the peripheral devices can buffer very large amounts of data, causing the DMA controller to tie up the bus for an excessive amount of time, single-cycle mode is preferable. Note that some DMA controllers have larger address registers than length registers. For instance, a DMA controller with a 32-

bit address register and a 16-bit length register can access a 4GB memory space, but can only transfer 64KB per block. If your application requires DMA transfers of larger amounts of data, software intervention is required after each block. The simplest way to use DMA is to select a processor with an internal DMA controller. This eliminates the need for external bus buffers and ensures that the timing is handled correctly. Also, an internal DMA controller can transfer data to on-chip memory and peripherals, which is something that an external DMA controller cannot do. Because the handshake is handled on-chip, the overhead of entering and exiting DMA mode is often much faster than when an external controller is used. If an external DMA controller or processor is used, be sure that the hardware handles the transition between transfers correctly. To avoid the problem of bus contention, ensure that bus requests are inhibited if the bus is not free. This prevents the DMA controller from requesting the bus before the processor has reacquired it after a transfer. So you see, DMA is not as mysterious as it sometimes seems. DMA transfers can provide real advantages when the system is properly designed.

Summary:

- Memory can be either permanent or temporary.
- Ranges of write ability
  - High end:processor writes to memory simply and quickly.e.g., RAM.
  - Middle range:processor writes to memory, but slower.e.g., FLASH, EEPROM.
  - Lower range: special equipment, “programmer”, must be used to write to memory.e.g., EPROM, OTP ROM.
  - Low end:bits stored only during fabrication.e.g., Mask-programmed ROM.
- Only one time programmable rom can be programmable only one time.
- Eprom can be erased and programmed number of times using ultra voilent radiation with reduced storage.
- EEprom can be erased and programmed number of times using inc ircuit system programmable.
- Ram is a random access memory used only for temporary storage purposes.

Questions:

- Write a brief notes on different types of memories.
- Explain differet types of Volatile memories.
- Explain memory management techniques.

References:

- Real-time Embedded Software Systems: An Introduction S. Agrawal & P. Bhatt http://www.embedded.com
- Michael Barr, Programming Embedded Systems in C and C++, O'Reilly Associates, August 1999.
- Analog Interfacing To Embedded Microprocessors – STUART R. BALL
- Design with 8051- FRONTLINE ELECTRONICS
- Embedded Systems Firmware Demystified - Ed Sutter.

UNIT – IV

8. INTERFACES

Objective:

In this chapter and the next, we'll look at two low-cost interfaces used to connect peripheral chips to microcontrollers, within a single embedded system. These interfaces allow you to connect devices such as real-time clocks, nonvolatile memories for parameter storage, sensor interfaces, and much more. The interfaces are easy to use and cheap to implement, making them ideal for small, embedded applications. Some microcontrollers incorporate both types of interface, whereas others may only have one or the other. The one to use really depends on what your processor has to offer and the requirements of the particular peripheral you're using.

Serial Peripheral Interface

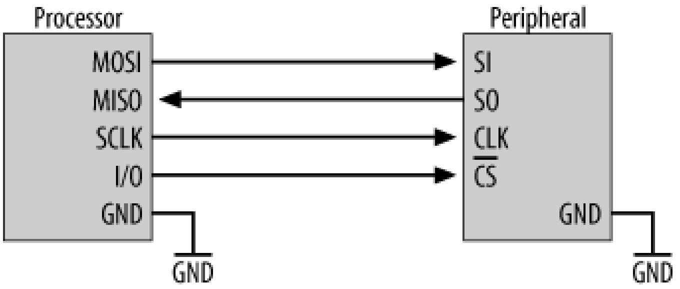
The Serial Peripheral Interface (known as SPI) was developed by Motorola to provide a low-cost and simple interface between microcontrollers and peripheral chips. (SPI is sometimes also known as a four-wire interface.) It can be used to interface to memory (for data storage), analog-digital converters, digital-analog converters, real-time clock calendars, LCD drivers, sensors, audio chips, and even other processors. The range of components that support SPI is large and growing all the time.

Unlike a standard serial port, SPI is a synchronous protocol in which all transmissions are referenced to a common clock, generated by the master (processor). The receiving peripheral (slave) uses the clock to synchronize its acquisition of the serial bit stream. Many chips may be connected to the same SPI interface of a master. A master selects a slave to receive by asserting the slave's chip select input. A peripheral that is not selected will not take part in a SPI transfer.

SPI uses four main signals: Master out Slave in (MOSI), Master in Slave out (MISO), Serial CLocK (SCLK or SCK) and Chip Select (CS) for the peripheral. Some processors have a dedicated chip select for SPI interfacing called Slave Select (SS).

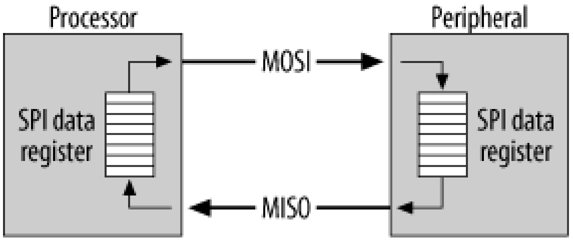
MOSI is generated by the master and is received by the slave. On some chips, MOSI is labeled simply as Serial In (SI) or Serial Data In (SDI). MISO is produced by the slave, but its generation is controlled by the master. MISO is sometimes known as Serial Out (SO) or Serial Data Out (SDO) on some chips. The chip select to the peripheral is normally generated by simply using a spare I/O pin of the master. Figure shows a microprocessor interfaced to a peripheral using SPI.

Basic SPI interface



Both masters and slaves contain a serial shift register. The master starts a transfer of a byte by writing it to its SPI shift register. As the register transmits the byte to the slave on the MOSI signal line, the slave transfers the contents of its shift register back to the master on the MISO signal line. In this way, the contents of the two shift registers are exchanged. Both a write and a read operation are performed with the slave simultaneously. SPI can therefore be a very efficient protocol.

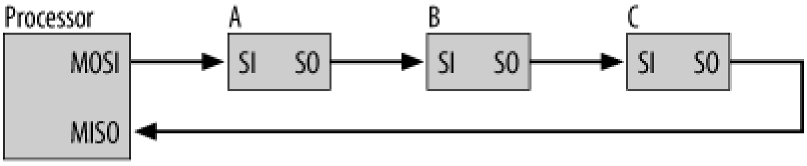
SPI transmission



If only a write operation is desired, the master just ignores the byte it receives. Conversely, if the master just wishes to read a byte from the slave, it must transfer a dummy byte in order to initiate a slave transmission.

Some peripherals can handle multiple byte transfers, where a continuous stream of data is shifted from the master. Many memory chips with SPI interfaces work this way. With this type of transfer, the chip select for the SPI slave must remain low for the entire duration of the transmission. For example, a memory chip might expect a "write" command to be followed by four address bytes (starting address), then the data bytes to be stored. A single transfer may involve the shifting of a kilobyte or more of information.

Daisy-chaining three SPI devices



Other slaves need only a single byte (for example, a command byte for an analog-digital converter), and some even support being daisy-chained together.

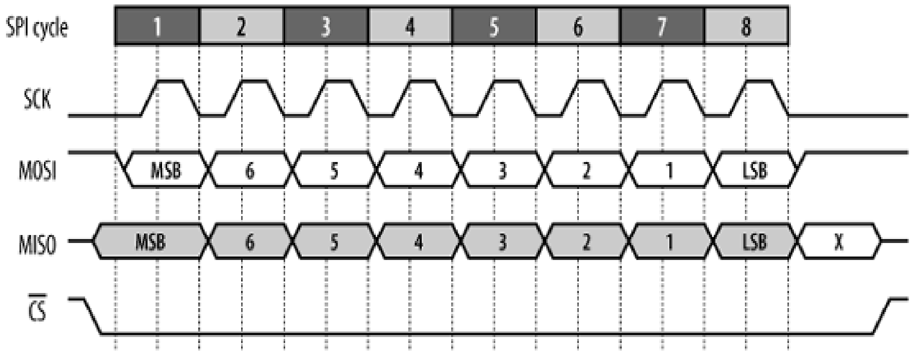
In this example, the master processor transmits three bytes out of its SPI interface. The first byte is shifted into slave A. As the second byte is transferred to slave A, the first byte is shifted out of slave A and into slave B. Similarly, as the third byte is shifted into slave A, the second byte is shifted into slave B, and the first byte is shifted into slave C. If the master wishes to read a result from slave A, it must again transfer a three-byte (dummy) sequence. This will move the byte from slave A into slave B, then into slave C, and finally into the master. In the process, the master also receives bytes from slave C and slave B in turn.

Note that daisy chaining won't necessarily work with all SPI devices, especially ones that require multi byte transfers (such as memory chips). Again, it's a case of checking the slave chips' datasheets carefully to determine what you can and can't do. If the datasheet doesn't explicitly mention daisy chaining, then it's a fair bet the device doesn't support it.

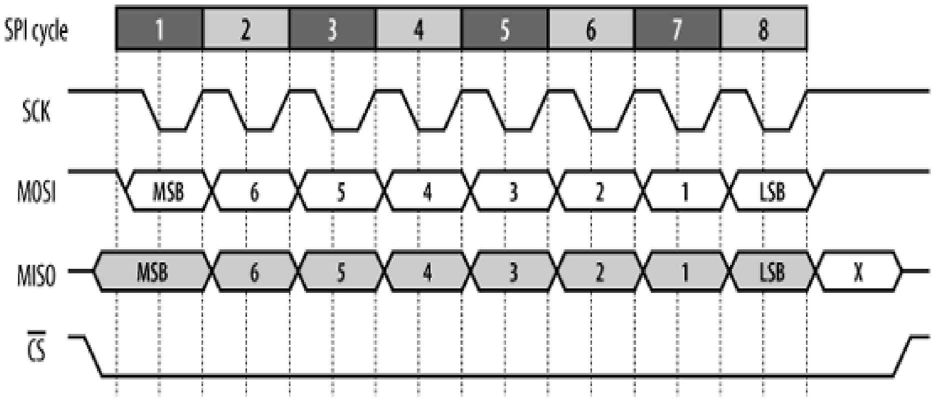
SPI has four modes of operation, depending on clock polarity and clock phase. For low clock polarity, the clock (SCK) is low when idle and toggles high during a transfer. When configured for high clock polarity, the clock is high when idle and toggles low during a transfer.

The two clock phases are known as clock phase zero and clock phase one. For clock phase zero, MOSI and MISO outputs are valid on the rising edge of the clock (SCK) if the clock polarity is low. If the clock polarity is high, these outputs are valid on the falling edge of SCK, for clock phase zero. The "X" bit output on MISO is an undefined extra bit and is a consequence of the SPI interface. You don't need to worry about it, as the SPI interfaces ignore it.

SPI timing with clock polarity low and clock phase zero

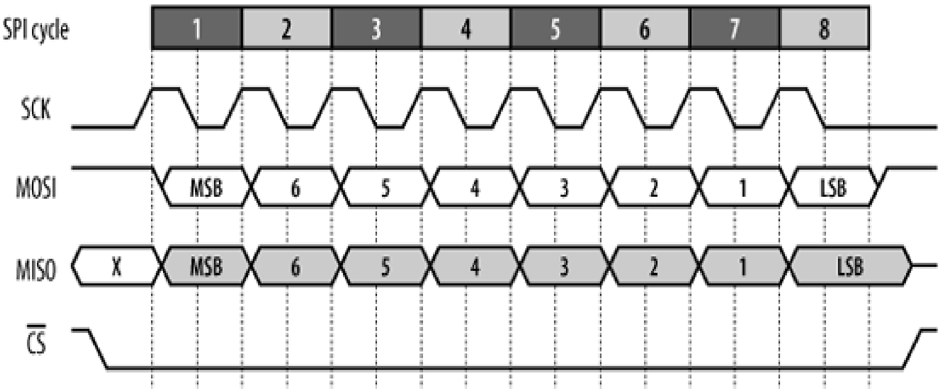


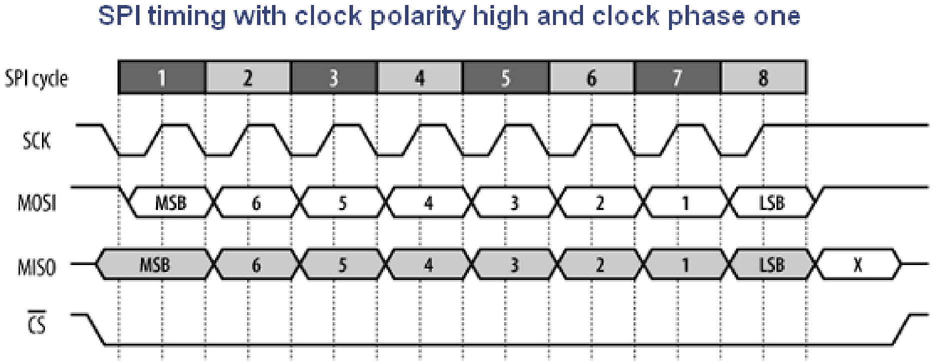
SPI timing with clock polarity high and clock phase zero



Conversely, for clock phase one, the opposite is true. MOSI and MISO are valid on the falling edge of the clock if clock polarity is low. They are valid on the rising edge of the clock if the clock polarity is high.

SPI timing with clock polarity low and clock phase one





SPI-Based Clock/Calendar

There is a wide variety of SPI devices available, and we'll be looking at several in the coming chapters. In the meantime, to see how a SPI interface is used to add a peripheral to a microcontroller, let's look at interfacing a processor to a clock/calendar chip. Such chips contain an oscillator module driven by a crystal, just like a processor. The oscillator module ticks over internal counters that track milliseconds, seconds, minutes, hours, days, months, and years. They are specifically designed to provide accurate timekeeping, and many have additional functions such as an "alarm" (whereby the processor is interrupted at a specific time) and a watchdog. Some also include voltage monitoring, such that the clock chip may act as a system monitor, alerting the processor should the power supply be wavering. There are a number of clock chips available (and not all are interfaced using SPI). For this example, we will use the Maxim DS1305.

The way in which we interface the clock chip to a processor is virtually identical for all other SPI devices. Some chips with SPI interfaces have special requirements, but most are very simple and straightforward. This makes SPI a very useful interface that makes increasing system functionality trivial.

The Maxim DS1305 Real-Time Clock (RTC) provides timekeeping services and tracks seconds, minutes, hours, day of the month, month, day of the week, and year. It knows which months have 30 days and which have 31. It even automatically adjusts for leap years, up to the year 2100. It can generate two interrupts to the microcontroller for time-of-day alarms. These alarms can be used to trigger a regular system event, such as a backup or user notification.

The DS1305 can run off two separate power sources and supports battery backup of its internal state. The chip can use a power supply in the range of 2 V to 5.5 V, allowing it to be powered from a variety of sources. It also has 96 bytes of static RAM, used for parameter storage. You could use the RAM for holding variables indicating system mode, secure password storage, or even authorization codes for your embedded software, just as desktop software does.

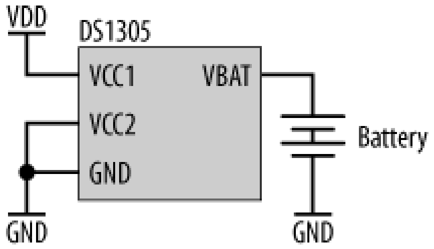
If you are producing commercial embedded systems and have problems with late-paying customers, you can use this RAM to hold a license number. When you ship the system, you design it to work for perhaps 45 days before shutting down. When your customer pays her bill, and you supply her with the right magic number, the system comes back to life again. The system stores the license number in the RAM of the RTC and from then on works as normal.

The RAM, like the timekeeping function, is battery-backed, and so its contents will be retained for the life of the battery. This can be up to 10 years, depending on the battery chosen. Thus, the contents of the internal parameter RAM will probably last for the expected operational lifespan of an embedded system.

The DS1305 is versatile in the way it can be powered. It has three power-supply inputs--VCC1, VCC2, and VBAT--from which it can choose to draw power. VCC1 is the primary supply input and is connected directly to the system's power supply. When the computer is up and running, the DS1305 draws its current from this source. VCC2 is the secondary power source, and this can be a rechargeable battery. VBAT is the third power source and is for non rechargeable batteries.

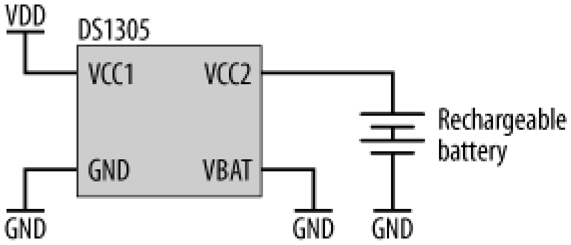
There are three, and only three, possible configurations for powering the DS1305, and it is important for correct operation that the power inputs are appropriately driven. Figure shows the DS1305 powered by a primary DC supply connected to VCC1 and a secondary, nonrechargeable battery connected to VBAT. (To keep the diagram simple, only the power pins are shown. We'll look at the data interface in a moment.) For this configuration, VCC2 is unused and must be connected to GND. When VCC1 falls below a given threshold voltage (the primary power source has failed), the internal memory and registers of the DS1305 become write-protected to prevent them from being corrupted by a failing microprocessor.

. Using the DS1305 with a nonrechargeable battery



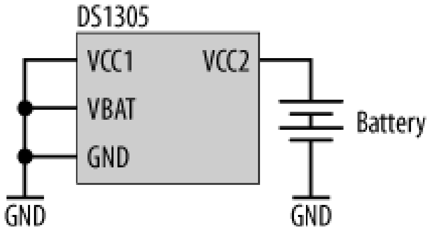
If the secondary power source is a rechargeable battery, then the DS1305 may be wired as shown in Figure. When using a rechargeable battery on VCC2, VBAT must be connected to GND. When the device is used in this mode, there is no automatic write protection for the DS1305 if VCC1 fails.

Using the DS1305 with a rechargeable battery



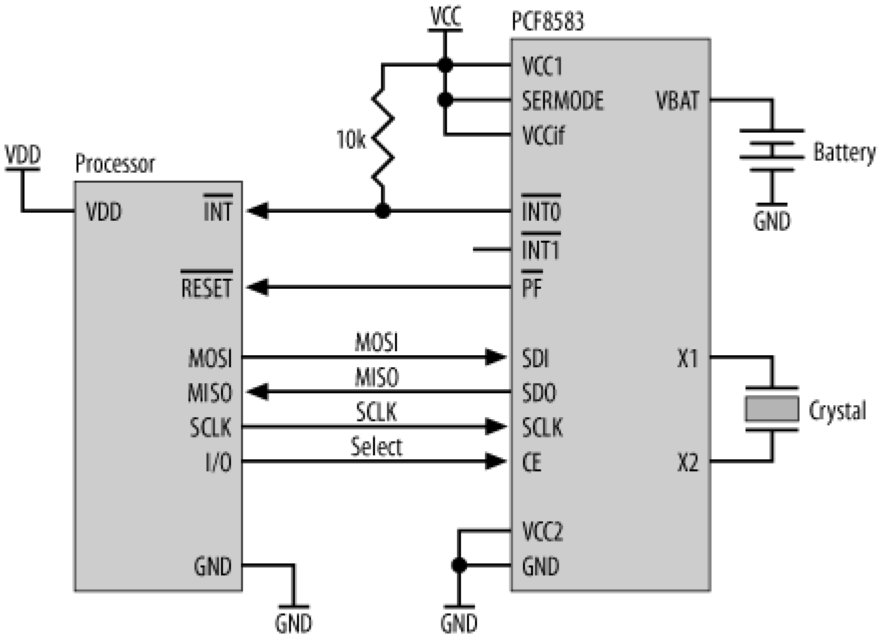
Finally, the DS1305 may be used with only a battery as its primary power source and no backup power supply. This is shown in Figure. For this configuration, both VCC1 and VBAT are connected to ground, while the battery is connected to VCC2.

Using the DS1305 with a battery as its only power source



Using the DS1305 is very simple. The schematic showing a DS1305 interfaced to a microcontroller is shown in Figure.

A DS1305 RTC interfaced to a microcontroller



The serial interface of the DS1305 can operate as either a SPI port or a three-wire port. The input SERMODE selects which serial mode to use. Connecting SERMODE to the power supply selects SPI operation. Connecting SERMODE to GND selects three-wire operation. (For three-wire operation, SDO and SDI are tied together.) The connection to a microcontroller's SPI port is straightforward, with MOSI, MISO, SCLK, and a chip select, as we've seen previously. There is one important difference, though, for the DS1305. It has an active-high CE (Chip Enable), rather than the more common active-low chip selects of other SPI devices. Therefore, the processor's I/O line driving CE must be low when the device is not selected and high when the device is selected.

[*] Developed by National Semiconductor, three-wire, also known as MicroWire, is very similar to SPI and is found is some microcontrollers and DSP processors. Unlike SPI, which has separate data lines for reading and writing, three-wire uses a common bidirectional data line.

The DS1305 has a special Power Fail output that is asserted low when the primary power source VCC1 falls below the secondary power source (VCC2 or VBAT). This can be used to alert the processor of the power fail (by using it as an interrupt) or to stop the processor (by connecting it to the processor's). This is used to prevent a failing processor from corrupting devices as the power dies. If you don't require a power-fail notification, may be left unconnected.

The input VCC if (VCC for the interface logic) selects the output voltage levels of SDO and . Since the DS1305 can be used in both 5 V and 3.3 V systems, this input allows the output levels of these pins to be set to the appropriate high voltage. VCC if is just connected to the system's power supply. Thus, for a 5 V system, VCC if is 5 V, and the outputs of the DS1305 are also 5 V. Similarly, for a 3.3 V system, VCC if is 3.3 V and so are the outputs.

Finally, the DS1305 has two interrupt outputs, These may be used to interrupt the processor when a DS1305 alarm function triggers. As the interrupt outputs are open-drain, they each require a 10k resistor to pull them high when they are inactive. If one or both of the interrupts are not required, just leave them unconnected. Only is used in our example, and so is safely ignored.

Finally, the DS1305 has two crystal inputs, X1 and X2. A 32.768 kHz watch crystal is connected across these pins, providing the timing source for the internal clock.

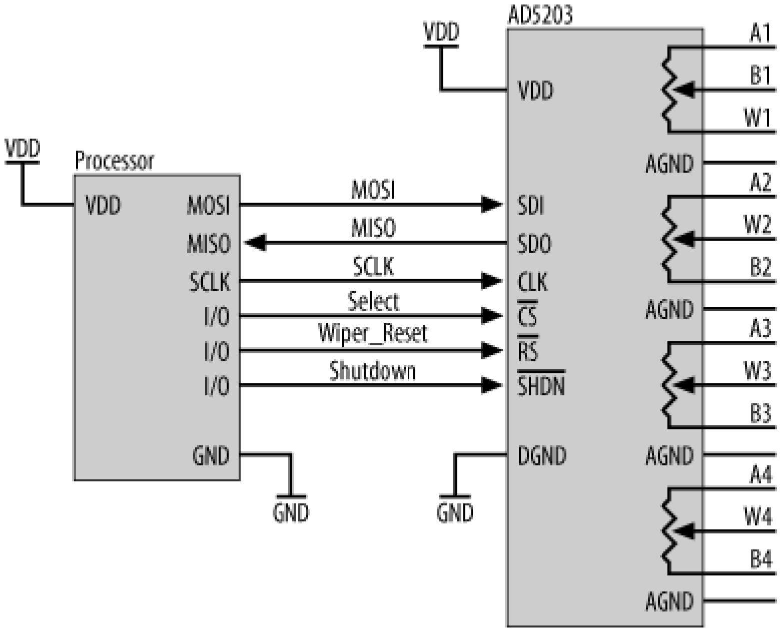
So that is the DS1305, a versatile little chip that can provide timekeeping for your embedded system. It's easy to use, and the programming information for it is contained in the device's datasheet.

SPI-Based Digital Potentiometer

Let's look at another simple SPI example. This time, we will interface a digital potentiometer to a microprocessor. Now, a standard pot is manually adjusted. It will either have a knob attached (as in a volume control or brightness adjustment), or it will have a small notch for screwdriver adjustment. Wouldn't it be great if your microprocessor could adjust the pots in your analog circuits, under software control? That way, your application software could adjust the brightness of the display or change the volume of the sound system. Well, by using a digital potentiometer, you can do just that. Televisions, computer monitors, and stereos with internal embedded controllers use digital pots to adjust settings such as volume. When you hit a volume button on a remote control, the TV or stereo adjusts the settings of digital pots, which are part of the amplifiers driving the speakers.

Figure shows an Analog Devices AD5203 digital potentiometer with a SPI interface. This chip has four potentiometers, all of which may be adjusted under software control. Each pot has 64 possible positions, and versions of the chip are available with either 10 kΩ or 100 kΩ impedances. For higher resolution, the pin-compatible AD8403 has a possible 256 settings, also configurable through a SPI interface.

Interfacing a digital potentiometer to a processor using SPI



The AD5203 has a Serial Data Input (SDI), which is connected to the processor's MOSI output. Similarly, the device's Serial Data Output (SDO) is connected to MISO. The AD5203's clock input (CLK) is positive-edge triggered midway through each SPI cycle, which means that any processor communicating with it must use high clock polarity and clock phase one on SCLK. The Chip Select (CS) of the AD5203 may be driven by a processor digital I/O line. The AD5203 has two other inputs, Shutdown (SHDN) and Reset (RS). SHDN places the device in low-power mode, and resets the potentiometer wipers to their midpoint. Both of these inputs may also be driven by a processor I/O line, or, if their functionality is not needed, they may be simply tied high using 10 kΩ pull-up resistors.

The potentiometers within the AD5203 are used as any other pots would be. The A and B terminals connect to either end of the internal resistors, and the position of the wiper (W) is adjusted under software control.

The AD5203 has several ground connections. DGND is the digital ground for the SPI interface and control logic of the chip. The AGNDs are the analog grounds of the internal potentiometers, and they should all be connected to DGND at a single point.

The datasheet for the AD5203 provides the control codes needed to configure the chip, and its use is simple and straightforward.

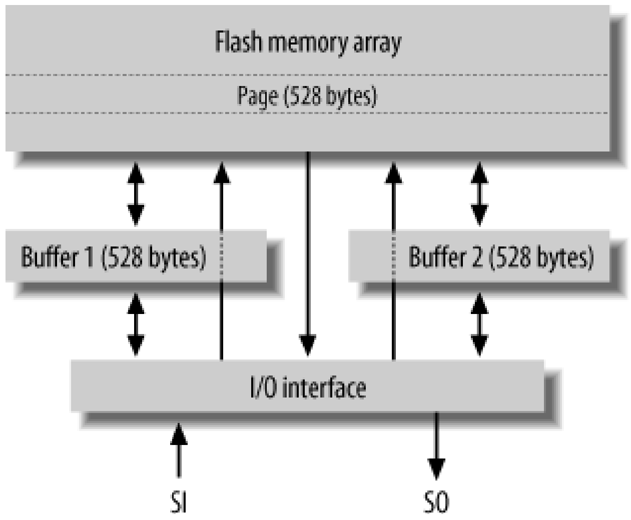
Adding Nonvolatile Data Memory with SPI

The internal memory of microcontrollers is very small, and their data storage capabilities are severely limited. We're now going to look at how you can increase the storage capacity of your embedded system by adding an Atmel AT45DB161 2M serial Data Flash using SPI. These chips are commonly used in low-cost digital cameras and answering machines. You could also use this flash chip as a virtual disk drive in your embedded system.

Most other flash chips have a bus interface, but the AT45DB161 has a serial interface, making it well suited for use with small microcontrollers. The AT45DB161 is a 2M chip, but you can get similar chips in capacities ranging from 512K to 32M. They all use the same (or similar) SPI interface, so the same design works for all. (Note, however, that their pinouts and physical packages vary, so one chip will not mount onto a circuit board design for another.)

The chip consists of an array of flash memory, organized as individual pages of 528 bytes each, and two RAM buffers, also 528 bytes each (Figure). To write data into the main flash array, the processor must first write data into one of the buffers and then issue a command to write that buffer into the array. A processor can read the contents of either of the buffers, transfer a flash page to the buffers, or read from the flash array directly. The operation of the buffers is independent, and one buffer may be accessed by the processor (via SPI) while the contents of the other buffer are being written into the flash array.

AT45DB161 internal architecture



The flash supports numerous commands for writing to and reading from the buffers, writing the buffers to the main array, and transferring an array page back to a buffer. The Atmel datasheet has full details of the software protocols and command set.

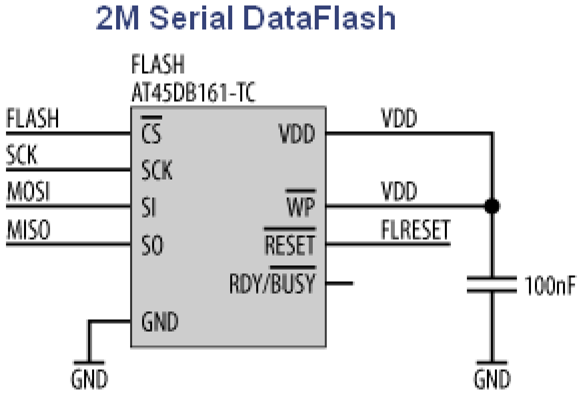
There are a few things to note about the internal architecture and the flash array. The first is that one 528-byte page of the flash array is not contiguous with the next. In other words, if you are using a pointer in your software to track the current location in the memory, you can't just increment it from the end of one page and expect it to be pointing to the next. Every 528 bytes (and it's a strange number), you have to leap forward to the next page. Think of it as pages of 528 bytes with big gaps in between.

The second catch with this memory is that it has a lifetime of only 1,000 write cycles per page. Most flash technologies (and there are several different types) support 100,000 write cycles or better, and you can normally exceed this limit and the device will keep working reliably for you. This isn't the case with the AT45DB161. Once the 1,000-write limit is exceeded, memory locations will start failing on you. The chip will read existing data back correctly, but new pages will not write successfully. Depending on the application, this limit may not be a problem. I've used this particular chip in my design for long-duration data loggers. These machines are deployed for yearlong deployments, collecting (and compressing) data and storing it away in the flash chip. The logger gradually builds a page image in one of the buffers before storing it to the array in a single write. Since during a deployment, a page will be written only once (and then the logger will move on to the next page), the 1,000-write limitation isn't a problem. It would take 1,000 deployments before the chip would fail. However, if you're using the chip for variable storage and are modifying the flash pages on a byte-by-byte basis, you're in trouble. Individually changing 528 bytes within a page

Note

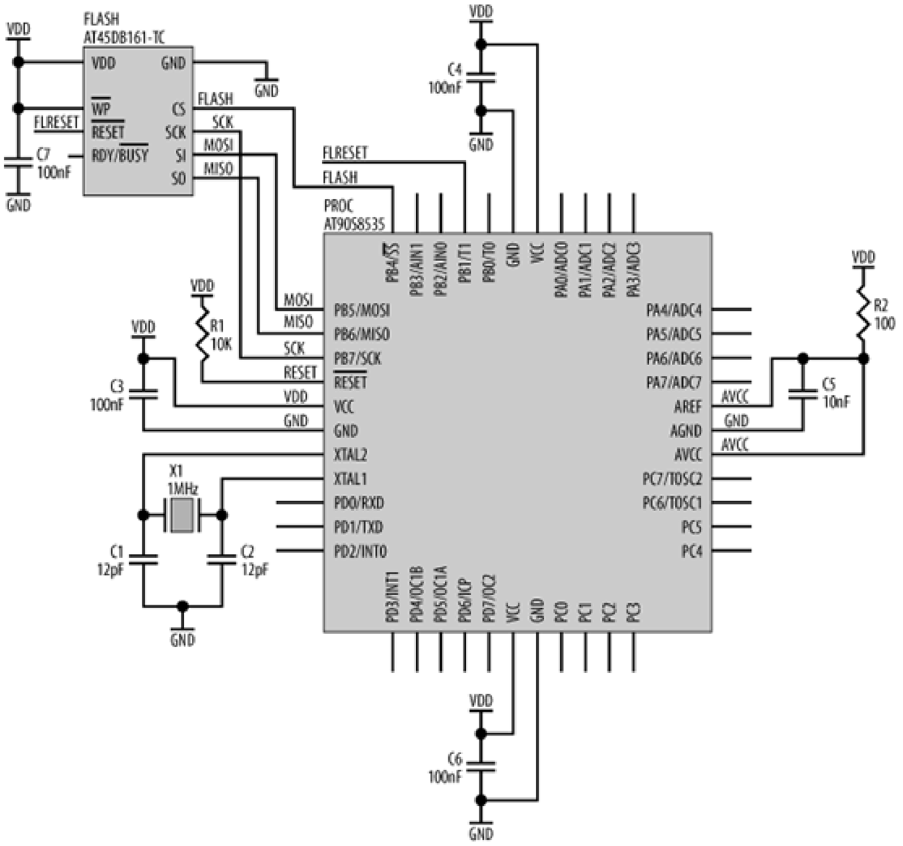
counts as 528 writes. So do that twice to a page, and suddenly you're over the limit. Therefore, this flash is well suited to some applications and not others.

The basic design for using an AT45DB161 is shown in Figure.



On the left of the chip are the SPI interface connections, MOSI, MISO, SCK, and a chip select (FLASH). The chip will support SPI transfers at up to 20 MHz, so the SPI interface can be run very fast indeed. On the right of the chip is the power supply, VDD, which is decoupled to ground using a 100 nF capacitor. The AT45DB161 requires a power supply in the range 2.5 V to 3.6 V. However, its logic inputs are 5 V tolerant, meaning this chip can be used in systems with mixed power supplies. In other words, while this chip requires a 3 V power supply, it can be directly interfaced to a processor with a 5 V supply (and 5 V logic levels). The AT45DB161 has a write-protect pin (WP), which, when driven low, prevents the contents of the flash from being modified. If you don't require write protection, simply tie this input high, as shown in the schematic. The flash also has a RESET input so that the chip can be manually reset under software control. The flash incorporates an inbuilt power-on reset that will put the device into a known state, and therefore a "manual" reset at power up should be unnecessary. However, I've found that the internal power-on reset generator is somewhat finicky and doesn't always kick in as it should. Under such circumstances, the flash fails to enter a known state and is unusable in the system. Therefore, I have found it good practice to give the processor control of the flash's reset. As part of the processor's initialization routines executed in its reset firmware, I get the processor to reset the flash, nudging it into reality. It's a simple thing, but it makes all the difference for a reliable system. Pin 1 is a status output (RDY/BUSY) indicating whether the device is ready or if it is still completing an internal operation. The connections for interfacing this memory chip to an Atmel 90S4434 AVR processor are shown in Figure. The AVR portion of the schematic is no different from the examples we have seen previously. That's the nice thing about simple interfaces such as SPI. They form little subsystem modules that "bolt together" like building blocks. Start with the basic core design and just add peripherals as you need them. The schematic also shows decoupling capacitors for the power supplies, the crystal oscillator for the processor, and a pull-up resistor for PB1. Pin 41 (PB1) is used as a "manual" (processor-controlled) reset input to the flash.

A 2M DataFlash interfaced to an AT90S4434

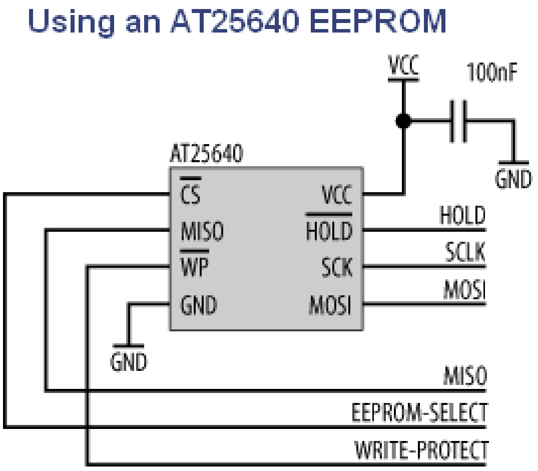


Adding a Parameter Memory Using SPI

We saw in the previous section how to add a large-capacity serial flash for data storage. It is often useful to use nonvolatile memory to hold system parameters, a way of preserving important variables during periods of no power. But the AT45DB161 Data Flash is just not the device for that task. It is better suited to data recording, and its large capacity is overkill for parameter storage. So, now we're going to look at how you can use SPI to add a small parameter memory (in the form on an EEPROM) to your embedded system. The EEPROM I've chosen is the Atmel AT25640. This device will hold data for at least 100 years without power, and will endure more than one million write cycles (significantly more than an AT45DB161!). As such, your software can happily alter parameter variables without fear of limiting the lifespan of the chip. The AT25640 has only 8K of memory, which might not sound like much. But don't forget, that's 8192 `char` variables, which is more than enough storage space for most parameters. If 8K is too much, there are also versions of the chip with 1K (AT25080), 2K (AT25160), and 4K (AT25320) bytes of memory.

The architecture and use of the AT25640 is much simpler than that of the AT45DB161. Full details of the required software protocol are in the Atmel datasheet for this chip.

The schematic for an AT25640 circuit is shown in Figure.



The interface is standard SPI, and the chip also has a write-protect input and a hold input. Asserting HOLD allows the processor to temporarily stall a serial transfer (while it performs other tasks) without terminating the access to the AT25640. And as you might expect, write-protect, when asserted, turns the chip into a read-only device. These control inputs may be driven by programmable I/O lines of the processor. The only other requirement is power (which is decoupled to ground using a 100 nF capacitor) and ground. The chip is available in two types. One will operate from a supply voltage of between 2.7 V and 5.5 V, while the other needs a supply voltage of between 1.8 V and 3.6 V.

Adding Peripherals Using I^{2}C

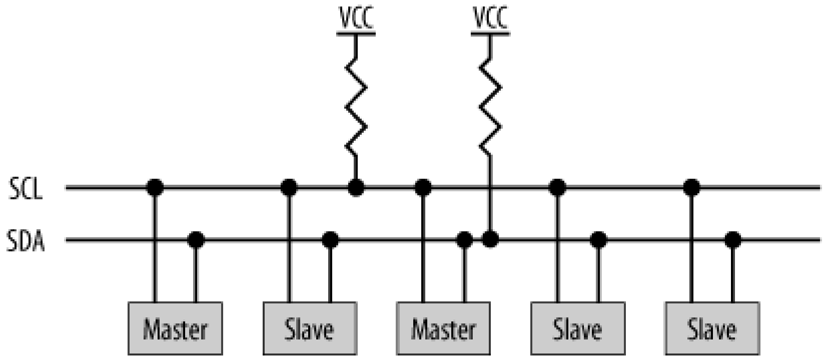
In the last chapter, we looked at the low-cost SPI interface used to connect peripheral chips to microcontrollers. In this chapter, we'll examine the alternate serial interface for connecting peripherals, I^{2}C.

Overview of I^{2}C

I^{2}C (Inter-Integrated Circuit) bus is a very cheap yet effective network used to interconnect peripheral devices within small-scale embedded systems. It is sometimes also known as IIC and has been in existence for more than 20 years. It is the equivalent of SPI, but its operation is somewhat different.

I^{2}C uses two wires to connect multiple devices in a multi-drop bus. The bus is bidirectional, low-speed, and synchronous to a common clock. Devices may be attached or detached from the I^{2}C bus without affecting other devices. Several manufacturers, such as Microchip, Philips, Intel, and others produce small microcontrollers with I^{2}C built in. The data rate of I^{2}C is somewhat slower than SPI, at 100 kbps in standard mode, and 400 kbps in fast mode.

The two wires used to interconnect with I^{2}C are SDA (serial data) and SCL (serial clock). Both lines are open-drain. They are connected to a positive supply via a pull-up resistor and therefore remain high when not in use. A device using the I^{2}C bus to communicate drives the lines low or leaves them pulled high as appropriate. Each device connected to the I^{2}C bus has a unique address and can operate as either a transmitter (a bus master), a receiver (a bus slave), or both (Figure). I^{2}C is a multi-master bus, meaning that more than one device may assume the role of bus master.

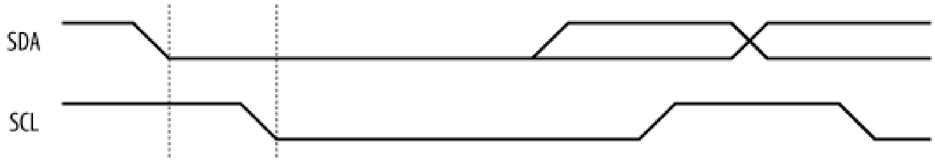


I2C network

An open-drain or open-collector pin has output drivers that can only pull the signal line to ground. They cannot drive it high. This has the advantage that more than one device connected to a signal line may pull it low. If this were not the case, one device attempting to pull the line low while another tried to pull it high would result in a short circuit, with disastrous results. Interrupt lines are typically open-collector. All open-collector signals need a pull-up resistor and are active low. The idle state (when no device is asserting) is to be pulled high by the resistor.

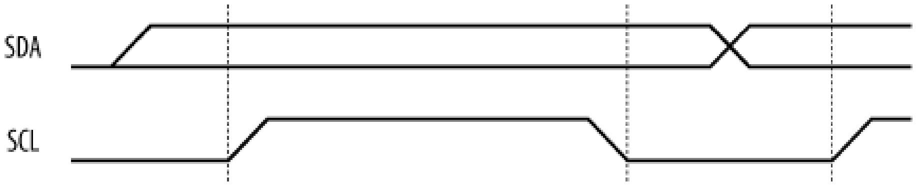
Both SDA and SCL are bidirectional. Unlike SPI, which has separate data lines for each direction of communication, I^{2}C shares the same signal line for master transmission and slave response. Also unlike SPI, I^{2}C does not have several modes of operation. The timing relationship between the clock, SCL, and the data line, SDA, is simple and straightforward. When idle, both SDA and SCL are high. An I^{2}C transaction begins with SDA going low, followed by SCL (Figure). This indicates to all receivers on the bus that a packet transmission is commencing. While SCL is low, SDA transitions (high or low) for the first valid data bit. This is known as a "START condition."

Figure: Start of packet



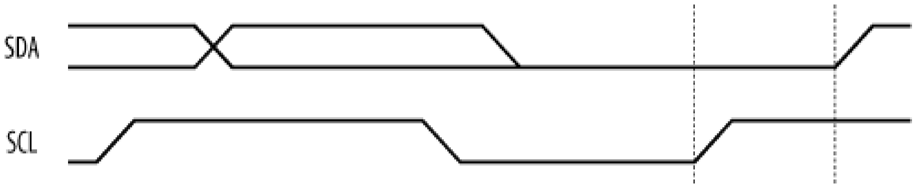
For each bit that is transmitted, the bit must become valid on SDA while SCL is low. The bit is sampled on the rising edge of SCL and must remain valid until SCL goes low once more. Then SDA transitions to the next bit before SCL goes high once more (Figure).

Timing relationship between SDA and SCL



Finally, the transaction completes by SCL returning high (inactive) followed by SDA (Figure). This is known as a "STOP condition."

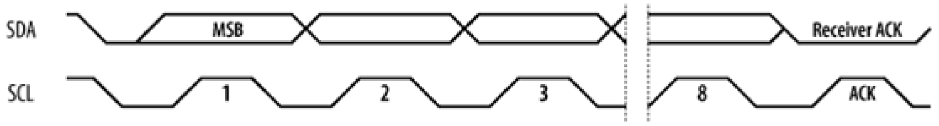
End of packet



Any number of bytes may be transmitted in an I^{2}C packet. As with SPI, the most significant bit of the packet is transmitted first. If the receiver is unable to accept any more bytes, it can abort the transmission by holding SCL low. This forces the transmitter to wait until SCL is released again.

Each byte transmitted must be acknowledged by the receiver. Upon the transmission of the eighth data bit, the master releases the data line SDA. The master then generates an additional clock pulse on SCL. This triggers the receiver to acknowledge the byte by pulling SDA low (Figure). If the receiver fails to pull SDA low, the master aborts the transfer and takes appropriate error-handling measures.

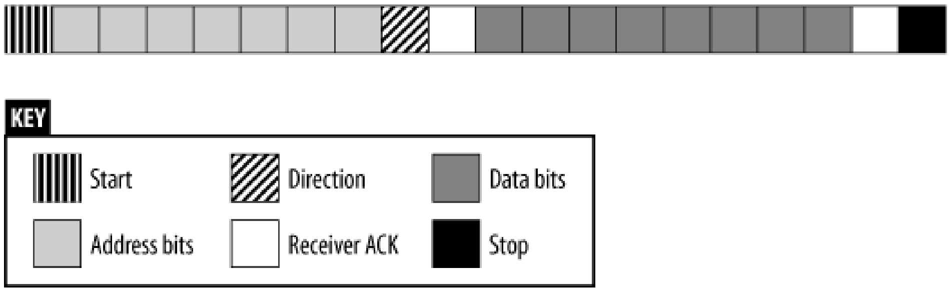
I^{2}C packet with receiver acknowledge



Now, I^{2}C is a multi-master bus. So, more than one master may attempt to start transmission at the same time. Since the bus's default state is high, a master transmitting a 0 bit will pull SDA low but will leave the bus in its default state if the bit is to be a 1. Thus, if two masters begin simultaneous transmission, a master leaving the bus in its default state for a 1 bit, but detecting the bus pulled low by another master (for a 0 bit), will register an error condition and abort the transmission.

SPI uses a separate chip select to enable a receiving slave. Each SPI slave has a separate chip select that is generated by the master. I^{2}C does not have such a selection mechanism. Instead, each device on the bus has a unique address, and the packet transmission begins with address bits, followed by the data. An address byte consists of seven address bits, followed by a direction bit. If the direction bit is a 0, the transmission is a write cycle and the selected slave will accept the data as input. If the direction bit is a 1, then the request is for the slave to transfer data back to the master. A sample packet, transferring one byte of data, is shown in Figure.

Figure: An I^{2}C packet



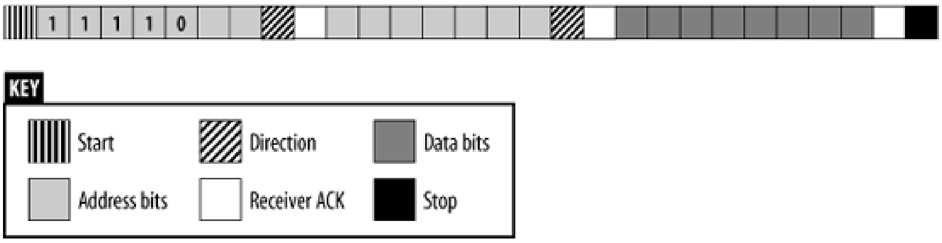
There is a special address, known as the general call address, which broadcasts to all I^{2}C devices. This address is `%0000000` with a direction bit of `0`. The general call is the mechanism by which the master determines what slaves are available, and there are several types of general call. The second byte of a general call indicates the purpose of the general call to the slaves. Upon receiving the second byte, individual slaves will determine whether the command is applicable to them, and, if so, they will acknowledge. If the command is not applicable to a given slave, then the slave simply ignores the general call and does not acknowledge. If the second byte is `0x06` (`%00000110`), then this indicates that appropriate slaves should reset and respond with their addresses. If the second byte is `0x04` (`%00000100`), slaves respond with their addresses but do not reset. Any other second byte of a general call, where the least significant bit is a 0, should be ignored.

If the least significant bit of the second byte is a 1, then the general call is by a master device identifying itself to other masters in the system by transmitting its own address. The other bits of the second byte contain the master's address.

There is another special address byte, known as the START byte. This byte is `%00000001` (`0x01`). It is used to indicate to other masters that a long data transfer is beginning. This is particularly important for masters that do not have dedicated I^{2}C hardware and must monitor the bus by software polling. When a master detects a START byte generated by another master, it can reduce its polling rate, allowing it more time for other software tasks.

I^{2}C also supports an extended 10-bit addressing mode, allowing up to 1,024 peripherals. Devices that use 7-bit addressing may be mixed with 10-bit addressing devices in a single system. In 10-bit addressing, two bytes are used to hold the address. If the (first) address byte begins with `%11110XX`, then a 10-bit address is being generated. The two least significant bits of the first byte, combined with the eight bits of the second byte, form the 10-bit address (Figure). 7-bit devices will ignore the transaction.

Figure: An I^{2}C packet with 10-bit addressing

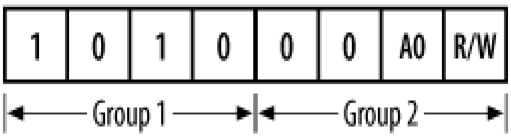


Adding a Real-Time Clock with I^{2}C

We saw in the previous chapter how to interface a Real-Time Clock (RTC) to a microprocessor using a SPI interface. Now let's look at how we'd do the same using the I^{2}C interface. For this example, we'll use the tiny Philips PCF8583. It also has 240 bytes of RAM, which, like the DS1305, may be used for parameter storage. Unlike the DS1305, it does not have an integrated battery-backup system. So, you would need to provide an external battery-backup circuit. There are many other I^{2}C RTCs available, and some do incorporate battery-fail protection. I've chosen to look at this one because it makes for a very simple example of an I^{2}C interface.

The PCF8583 has two pins (OSCI and OSCO) for connecting a 32.768 kHz watch crystal. This crystal pulses an internal circuit that performs the timekeeping functions. The address pin, A0, determines the address of the device on the I^{2}C bus. Most I^{2}C chips provide several address pins, allowing a range of possible addresses to be wired. The PCF8583 has only one, to reduce the pin count of the chip. Six of its address bits are hardwired internally. Only the least significant, A0, is available to the system designer. The address configuration of the PCF8583 is shown in Figure. (Note how the transfer direction [read or write] is incorporated into the address field.)

Figure: PCF8583 addresses

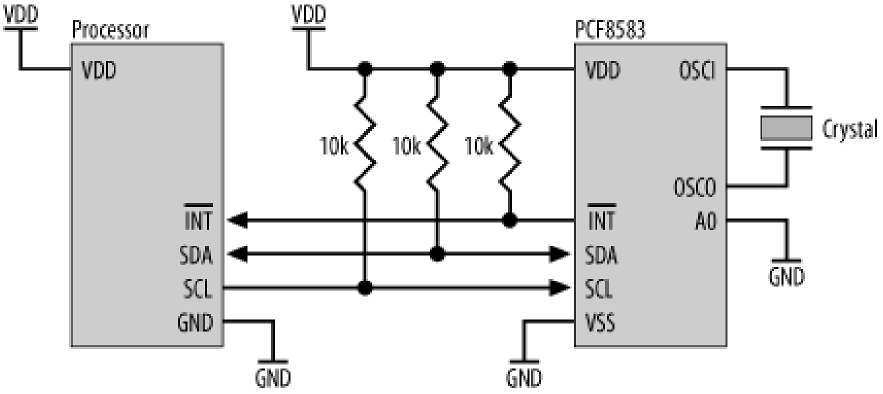


1	0	1	0	0	0	A0	R/W

Connecting A0 directly to ground sets that address bit to `0` and therefore maps the PCF8583 to I^{2}C address `0x50`. Alternatively, if A0 is tied to VDD, then the address of the device is `0x51`.

The schematic for interfacing the PCF8583 to a microcontroller is shown in Figure.

Figure: Interfacing a PCF8583 to a microcontroller

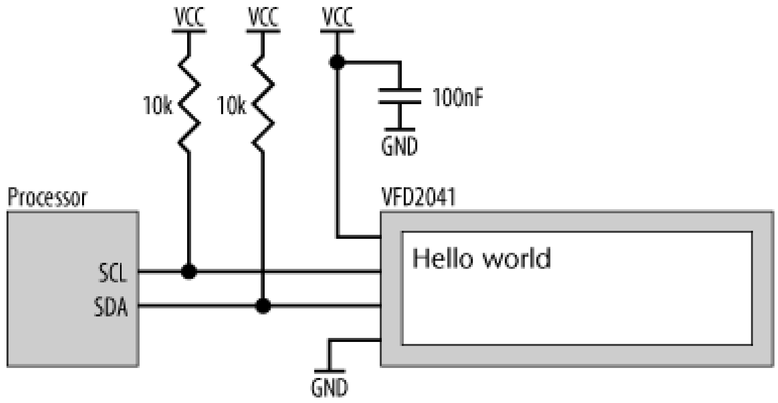


SDA and SCL both require pull-up resistors to VDD. The PCF8583 also has an internal alarm function and asserts an output (   ) for interrupting the processor. Since this output is open-drain, a pull-up resistor is also required.

Adding a Small Display with I^{2}C

You can use I^{2}C to add simple LCDs (and other equivalent display technologies) to your embedded computer. These LCDs are usually just a few lines of text high, but are useful for simple message display functions. Matrix Orbital (http://www.matrixorbital.com) produces a number of display modules that are easy to interface, such as the VFD2041. This display module is 80 characters wide by 4 lines deep. The interface circuit is shown in Figure, and, as you can see, there's almost nothing to it. The types of LCDs found in laptops are considerably more complicated, and interfacing them to small processors is just not an option. But for simple message displays (such as on the front panel of an appliance), a circuit like this is ideal.

Interfacing a VFD2041 display using I^{2}C



Many Matrix Orbital displays also come with RS-232C interfaces, so if you're embedded processor doesn't support I^{2}C; it's still easy to add a small display.

Serial Ports

In this chapter, we'll look at connecting your embedded systems to the outside world through the ubiquitous serial port. We'll see how you implement the classic serial port, RS-232C, and even take a look at how you can power your embedded system through an RS-232C port. From there, we'll take a look at the more robust RS-422, designed for faster data rates over longer distances. Finally, we'll look at RS-485, an extension of RS-422 designed for low-cost networking of embedded computers.

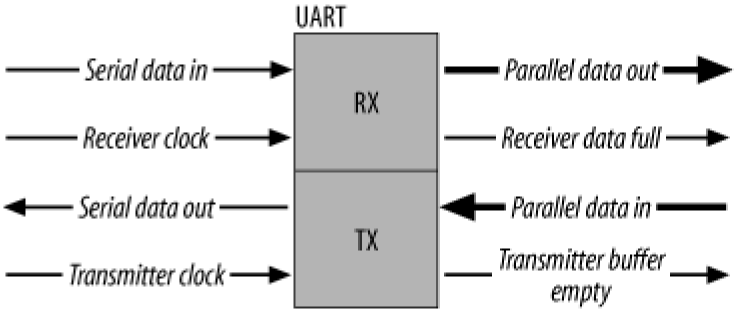
UARTs

Serial I/O involves the transfer of data over a single wire for each direction. All serial interfaces convert parallel data to a serial bit stream, and vice versa. Serial communication is employed when it is not practical, either in physical or cost terms, to move data in parallel between systems. Such serial communication may be between a computer and a terminal or printer, the infrared beaming of a Palm computer or remote control, or, in more advanced forms, high-speed network communication such as Ethernet. For embedded computers, a simple serial interface is the easiest and cheapest way to connect to a host computer, either as part of the application or merely for debugging purposes.

The simplest form of serial interface is that of the Universal Asynchronous Receiver Transmitter (UART). UARTs are also sometimes called Asynchronous Communication Interface Adapters (ACIAs). They are termed asynchronous because no clock is transmitted with the serial data. The receiver must lock onto the data and detect individual bits without the luxury of a clock for synchronization.

Figure shows a functional diagram of a UART. It consists of two sections: a receiver (Rx) that converts a serial bit stream to parallel data for the microprocessor and a transmitter (Tx) that converts parallel data from a microprocessor into serial form for transmission. The UART also provides status information, such as whether the receiver is full (data has arrived) or that the transmitter is empty (a pending transmission has completed). Many microcontrollers incorporate UARTs on-chip, but for larger systems, the UART is often a separate device.

Figure: Functional diagram of a UART

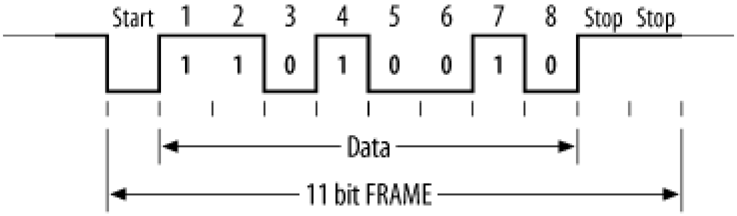


Serial devices send data one bit at a time, so normal "parallel" data must first be converted to serial form before transfer. Serial transmission consists of breaking down bytes of data into single bits and shifting them out of the device one at a time. A UART's transmitter is essentially just a parallel-to-serial converter with extra features. The essence of the UART transmitter is a shift register that is loaded in parallel, and then each bit is sequentially shifted out of the device on each pulse of the serial clock. Conversely, the receiver accepts a serial bit stream into a shift register, and then this is read out in parallel by the processor.

One of the problems associated with serial transmission is reconstructing the data at the receiving end. Difficulties arise in detecting boundaries between bits. For instance, if the serial line is low for a given length of time, the device receiving the data must be able to identify if the stream represented "00" or "000." It has to know where one bit stops and the next starts. The transmitting and receiving devices can accomplish this by sharing a common clock. Hence, in a synchronous serial system, the serial data stream is synchronized with a clock that is transmitted along with the data stream. This simplifies the recovery of data but requires an extra signal line to carry the serial clock. Asynchronous serial devices, such as UARTs, do not share a common clock; rather, each device has its own, local clock. The devices must operate at exactly the same frequency, and additional logic is required to detect the phase of the transmitted data and phase lock the receiver's clock to this.

Asynchronous transmission is used in systems where one character is sent at a time, and the interval of time between each byte transmission may vary. The transmission format uses one start bit at the beginning and one or two stop bits at the end of each character (Figure). The receiver synchronizes its clock upon receiving the start bit and then samples the data bits (either seven or eight, depending on the system configuration). Upon receiving the stop bit(s) in the correct sequence, the receiver assumes that the transfer was successful and that it has a valid character. If it did not receive an appropriate stop sequence, the receiver assumes that its clock drifted out of phase, and a framing error or bit-misalignment error is declared. It's up to the application software to check for such errors and take appropriate action.

Figure: Asynchronous serial data



The conversion from parallel to serial format is usually accomplished by dedicated UART hardware, but in systems where only parallel I/O is available, the conversion may be performed by software, which toggles a single bit of a parallel I/O port acting as the serial line.

Error Detection

In any transfer of data over a potentially noisy medium (such as a serial cable), the possibility of errors exists. To detect such errors, many serial systems implement parity as a simple check for the validity of the data. The parity bit of a byte to be transmitted is calculated by the sending UART and included with the byte as part of the transmission. The receiving UART also calculates the parity bit for the byte and compares this against the parity bit received. If they match, the receiver assumes that everything is fine. If they do not, the receiver then knows that something went amiss and that an error exists.

There are several types of parity, the main two being even parity and odd parity. In any byte of data, there is either an even number of "1" bits or an odd number of "1" bits. An extra bit (the parity bit) is added to the byte to make the number of "1" bits even (even parity) or odd (odd parity). For successful transmission, both the receiver and transmitter must be set for the same type of parity generation. There is no protocol for establishing common parity settings between UARTs; it must be done manually at either end.

So for the binary sequence `%01000000`, the parity bit would be "1" for even parity and "0" for odd parity. Similarly, for `%11111111`, the parity bit would be "0" if we were using even parity and "1" if we had odd parity.

The generation and detection of parity is done automatically by dedicated hardware within the UART. It's not something you explicitly have to calculate. You do have to make sure your UART is set to the correct type of parity generation; otherwise, it will not know how to process the parity information accordingly.

The parity bit is checked at the receiving end against the data to check whether any of the bits were corrupted during transmission. Say we sent `%01000000`. If our UART was set to even parity, the calculated parity bit from `%01000000` would be 1. Now, let's say this transmission was corrupted along the way, such that what was actually received was `%01000001`. The receiver would calculate the even parity of the byte to be 0. In comparing this to the received parity bit of 1, a parity error would be detected, and the receiver would take appropriate action (such as requesting that the byte be sent again). Note that how parity errors are handled is the responsibility of the programmer. The UART itself takes no action beyond flagging the error. It is up to the software to implement appropriate error handling.

Now, what if the medium was particularly noisy and two bits were corrupted? Again, if we sent `%01000000` with even parity (computed parity bit = 1), and this was corrupted along the way to be `%01001001`, the receiver would calculate the even parity of the byte to be 1. The transmission was corrupted, but no parity error would be detected! As you can see, the usefulness of this form of error detection is extremely limited, and, for this reason, more complicated error detection (and correction) schemes are often implemented. A good example of this is the Cyclic Redundancy Check (CRC) algorithm. If you need to implement CRC, there's plenty of source code available on the Web—just use your favorite search engine.

That covers the basics of how bits are transmitted serially. Now, it's time to look at how you physically implement a serial interface. We'll start with the old standard for serially connecting two computers (or just about anything else digital) together.

Old Faithful: RS-232C

RS-232C is a serial communication interface standard that has been in use, in one form or another, since the 1960s. RS-232C is used for interfacing serial devices over cable lengths of up to 25 meters and at data rates of up to 38.4 kbps. You can use it to connect to other computers, modems, and even old terminals (useful tools for monitoring status messages during debugging). In days of old, printers, plotters, and a host of other devices came with RS-232C interfaces. With the need to transfer large amounts of data rapidly, RS-232C is being supplanted as a connection standard by high-speed networks, such as Ethernet. However, it can still be a useful and (importantly) simple connection tool for your embedded system.

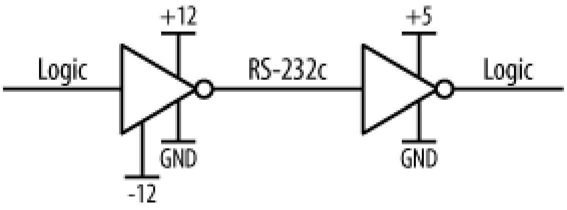
RS-232C is unbalanced, meaning that the voltage level of a data bit being transmitted is referenced to local ground. A logic high for RS-

232C is a signal voltage in the range of -5 to -15 V (typically -12 V), and a logic low is between +5 and +15 V (typically +12 V). So, just to make that clear, an RS-232C high is a negative voltage, and a low is a positive voltage, unlike the rest of your computer's logic.

The terminology used in RS-232C also dates back to the 1960s. In those days of mainframes, a high (1) was called a "space," and a low (0) was called a "mark." You'll still find these terms kicking around in RS-232C, where you'll hear phrases like "mark parity" and "space parity." It's also not unheard of to see RS-232C systems still using 7-bit data frames (another leftover from the '60s), rather than the more common 8-bit. In fact, this is one of the reasons why you'll still see email being sent on the Internet limited to a 7-bit character set, just in case the packets happen to be routed via a serial connection that supports only 7-bit transmissions. It's nice how pieces of history still linger around to haunt us! More commonly, RS-232C data transmissions use 8-bit characters, and any serial port you implement should do so, too.

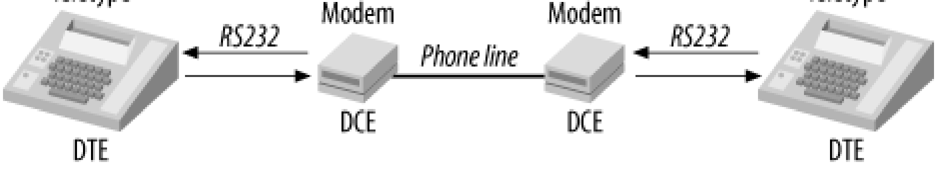
An RS-232C link consists of a driver and a comparator, as shown in Figure.

Figure . RS-232C



RS-232C also defines connectors and pin assignments, although there is a lot a room for variation (and thus a lot of incompatibilities exist). RS-232C was originally intended for connecting Data Terminal Equipment (DTE) to Data Communication Equipment (DCE) (Figure). The standard therefore assumes that at one end of an RS-232C link is a DTE device, and at the other end, there is a DCE. Before the advent of computers, a DTE was a terminal or teletype, and a DCE was a modem. The modem (Modulator/ Demodulator) provided an interface to the phone line, and thereby a connection to a remote modem and terminal.

Figure: Original use of RS232: connecting teletypes to modems



This worked simply and clearly in the days before desktop computers. The "problem" arises when you wish to connect either a terminal or a modem to the serial interface of a computer. Do you treat the computer as a DTE or a DCE? The RS-232C standard implies that if a terminal is at one end of the link, then the other end should be a DCE. So, if you were connecting a terminal to a Unix workstation, the RS-232C standard would like the workstation to be a DCE (Figure1). Conversely, if you were connecting a modem to a computer, the computer should be a DTE (Figure2). It's all a bit schizophrenic.

Figure 1. DTE device connected to a computer

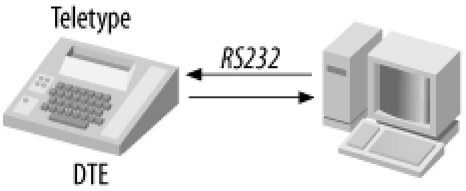
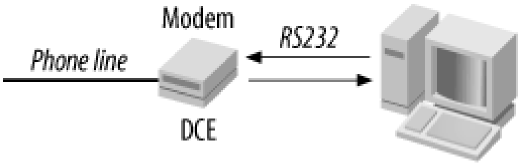
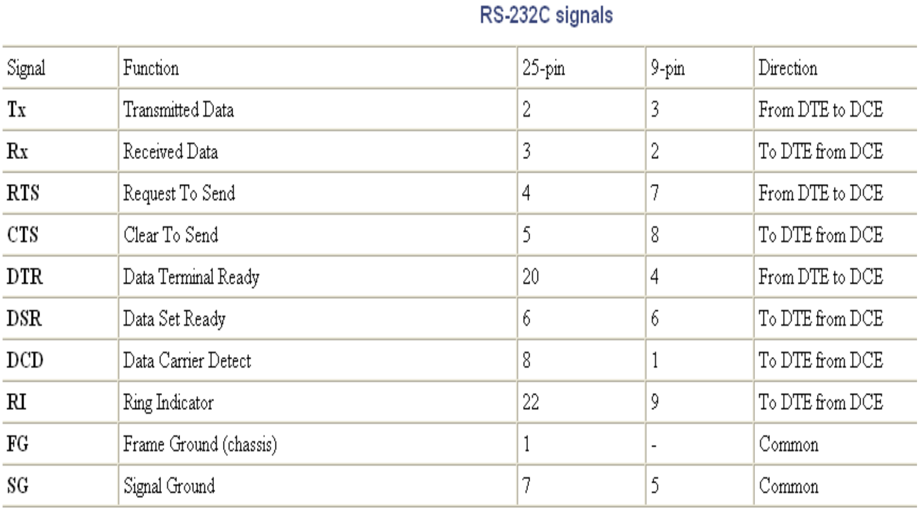


Figure 2. DCE device connected to a computer



Manufacturers, when faced with this problem, arbitrarily chose one or the other. The IBM PC has a DTE-type connector, whereas the makers of Unix workstations (such as Sun Microsystems) often choose to make their machines with DCE connectors, since they are more likely to be connected to terminals. To connect a PC to a modem, you need a DTE-DCE cable. To connect a PC to a terminal, you need a DTE-DTE cable. To connect a Sun workstation to a terminal, you need a DCE-DTE cable. To connect a Sun to a modem you need a DCE-DCE cable. To connect a Sun to another Sun, you need a DCE-DCE null modem cable (where Rx and Tx cross over), and to connect a Sun to a PC, you need a DCE-DTE null modem cable. If, however, you need to connect two PCs together, you need a DTE-DTE null modem cable. So, for just two types of device (DTE and DCE), you need six types of cable to cope with the permutations! Variety, as they say, is the spice of life, but it's the bane of RS-232C!

Table shows the "standard" connections for RS-232C, for both 25-pin and 9-pin connectors. The signal names are DTE-relative. For example, Tx refers to data being transmitted from the DTE but received by a DCE.



RS-232C signals

Signal	Function	25-pin	9-pin	Direction
Tx	Transmitted Data	2	3	From DTE to DCE
Rx	Received Data	3	2	To DTE from DCE
RTS	Request To Send	4	7	From DTE to DCE
CTS	Clear To Send	5	8	To DTE from DCE
DTR	Data Terminal Ready	20	4	From DTE to DCE
DSR	Data Set Ready	6	6	To DTE from DCE
DCD	Data Carrier Detect	8	1	To DTE from DCE
RI	Ring Indicator	22	9	To DTE from DCE
FG	Frame Ground (chassis)	1	-	Common
SG	Signal Ground	7	5	Common

Many of these signals are intended for modem control. To form a very simple link between a computer and a terminal, the only signals required are Tx, Rx, and SG. Many systems tie FG and SG together.

Shake Hands

When two remote systems are communicating serially, there needs to be some way to prevent the transmitter from sending new data before the receiver has had a chance to process the old data. This process is known as handshaking, or flow control. The way it works is simple. After transmitting a byte (or data packet), the transmitter will not send again until it has been given confirmation that the receiver is ready. There are three forms of handshaking: hardware, software, and none.

The no-handshaking option is obviously the simplest and is used in situations where the transmitting system is much slower in preparing and sending data than the receiver is in processing. For example, if you had a small, embedded computer running at a pokey 1 MHz that was feeding data into a high-speed computer system running at 4 GHz, it would not be unreasonable to assume that the faster machine would be able to keep up. However, if the faster machine is running a certain popular operating system (renowned for poor responsiveness to real-time events), it may very well be the case that it may not be able to keep up. In this case, handshaking would be required, and it's probably good practice to incorporate it anyway. If you're using the serial port to provide a human interface to your computer, then you can safely assume that no human will type faster than your computer can handle. So, for serial ports used solely for user access or debugging purposes, you can skip the handshaking.

Hardware handshaking in RS-232C uses two signals, RTS (Request To Send) and CTS (Clear To Send). When the transmitter wishes to send, it asserts RTS, indicating to the receiver that there is pending data. The receiver asserts CTS when it is ready, indicating to the transmitter that it may send. In this way, the flow of data is limited to the rate at which it may be processed.

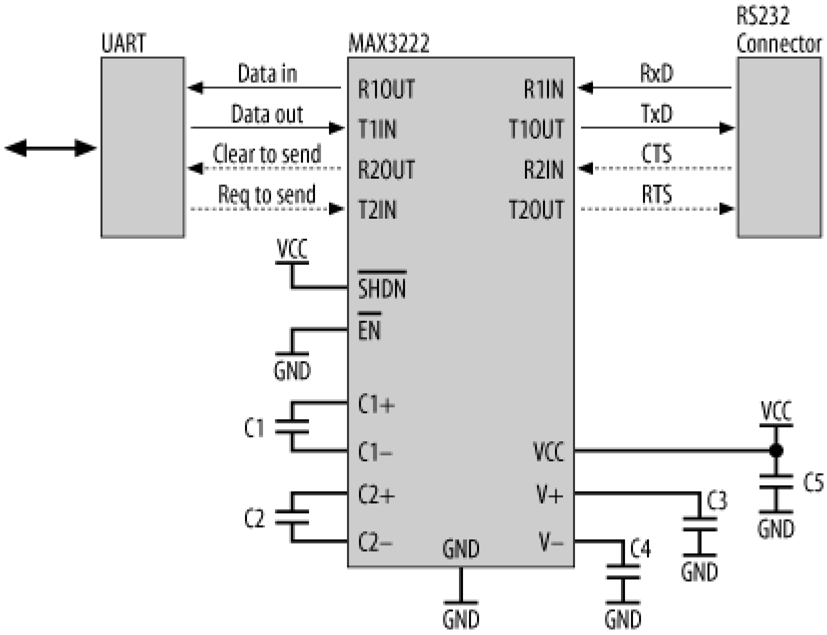
Software handshaking, also known as XON/XOFF, is used where it is not possible to have hardware handshaking between the transmitter and receiver, such as when the transmission occurs over a phone line. Software handshaking chooses two characters to represent a request to "suspend transmission," and a "clear to resume." These are normally the characters Ctrl-S (`0x13`) and Ctrl-Q (`0x11`). The caveat is that you then can't have these characters as part of the transmitted file, because they would be interpreted as flow control by the receiver and not as received data. If you're only sending ASCII text, this is not a problem, but it can be a real headache if you're sending binary data. The common solution is to preprocess the binary data prior to transmission and convert it to ASCII representation. For example, the byte `0x2F` becomes the ASCII characters "2" (`0x32`) and "F" (`0x46`). Software on the receiving end converts the ASCII characters back into binary data again. Examples of software that will do this are `uuencode` under Unix and `BinHex` under Mac OS.

Implementing an RS-232C Interface

Adding an RS-232C interface to a system is easy. Most microcontrollers (except the very tiny) incorporate a UART within the chip, so all that is required is an external level shifter to convert the serial transmissions to and from RS-232C levels. Maxim makes a huge range of RS-232C interface chips (level shifters) that greatly simplify your design. No matter what your specific conversion requirements, doubtless there's a Maxim part to meet your need. A good generic choice is the MAX3222 transceiver. Since nearly all RS-232C transceivers are used in the same way, looking at a design with a MAX3222 provides a good example of what to do for any transceiver. Unlike many other level shifters, the Maxim parts can operate from a low supply voltage, in the range of 3.0 V to 5.5 V. Many other manufacturers' devices need supplies of +12 V and -12 V, and therefore require additional voltage regulators. The MAX3222 consumes minimal power (1 mA in normal operation and as low as 1 uA in shutdown mode), making it ideal for portable and battery-powered applications. If the ability to shut down the serial port into low-power operation is not required, the MAX3232 can be substituted. It is functionally the same, except that it lacks shutdown capability.

Using the MAX3222 is trivial, as there is almost no design work involved at all. The only external support components required are capacitors for the chip's internal charge pumps. These pumps generate the +12 V and -12 V voltages required for RS-232C transmission, and they do so without requiring (additional) external voltage regulators. Figure shows the schematic.

Figure :RS-232C interface using a MAX3222



The capacitor C1 must be a minimum of 0.1 uF. If you are operating the chip at less than 3.6 V, C2, C3, and C4 can also be 0.1 uF. If the supply voltage is to be as high as 5.5 V, then C2, C3, and C4 must be a minimum of 0.47 uF. Since these are minimum values, larger capacitors may be used. However, if C1 is increased, then the remaining capacitors must also be increased accordingly. C5, the decoupling capacitor for VCC, is nominally 0.1 uF. All capacitors should be as close to the appropriate pins of the chip as possible.

The only remaining connections are the serial data lines from the UART and the signals to the RS-232C connector. If you are implementing a minimal serial interface, only Rx, Tx, and ground are required. RTS and CTS are optional. The RS-232C connector may be either a 25-pin or a 9-pin DB connector (its shape looks like the letter "D"). However, the connector could also be just a row of pins, a parallel header, or even just wires soldered directly onto the PCB.

The MAX3222 has two control inputs, SHDN(shutdown) and EN(enable). SHDN places the RS-232C transmitters in high impedance, thereby disabling them. This reduces the chip's current consumption to less than 1 uA. When in shutdown mode, the receivers are still active. Thus, the UART is still able to receive data even if the MAX3222 is in low-power mode. If SHDN is not required, just connect it directly to VCC.

Similarly, EN is used to control the receiver outputs. Placing high puts the receiver outputs into high impedance, while the transmitter outputs are unaffected. To enable the receivers, EN is asserted (pulled low). If disabling the receivers is not required, then tie EN to ground to permanently activate them.

If needed, SHDN and EN may be controlled by a microcontroller's I/O lines, or by simple digital outputs using a latch.

The MAX3222 is sufficient to implement a minimal RS-232C interface, using just Rx, Tx, and ground. It also has additional drivers to support RTS and CTS, allowing for basic flow control. Should you require a full RS-232C interface, the MAX3241 is a good choice. Its operation is similar to the MAX3222, but it has additional transceivers allowing the inclusion of DTR, DSR, DCD, and RI for modem control. The MAX3421 may also be used to interface to a serial mouse, since it is able to meet the appropriate voltage and current requirements.

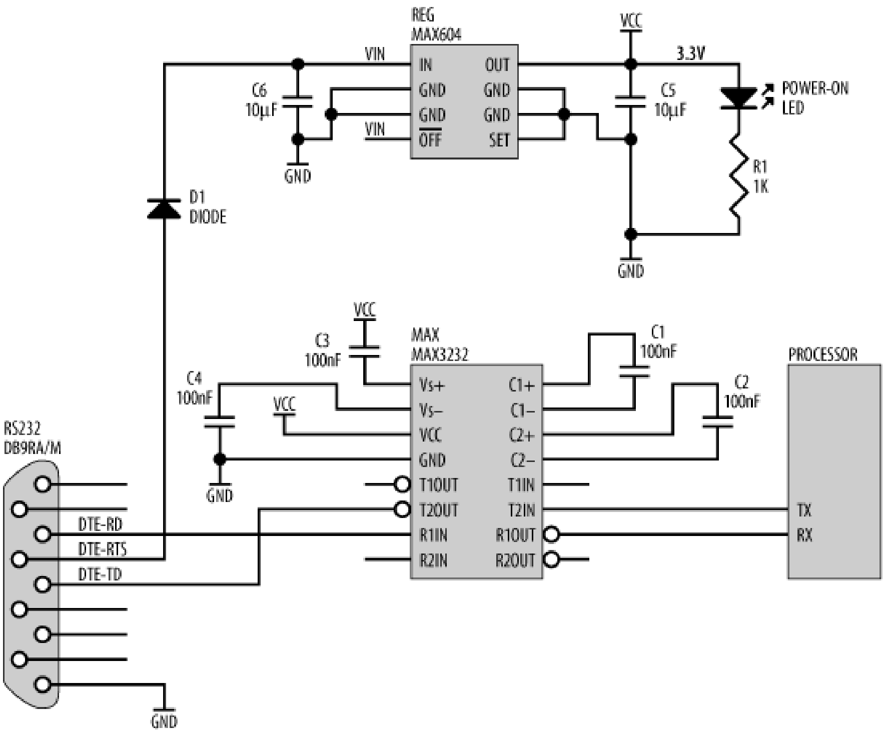
Using a Serial Port as a Power Supply

If an embedded system is to be permanently connected to a host computer via an RS-232C serial interface, it is possible to parasitically power the embedded system from the serial interface. Many RS-232C signals go unused and can supply a moderate amount of current, nominally 50 mA. However it can vary (considerably) from device to device, and, as always, you should check the specific system to which you are interfacing. If your embedded system requires less than this for its total current draw, you can use an RS-232C control signal for power.

For instance, the RTS (Ready To Send) or DTR (Data Terminal Ready) signals may not be used in many RS-232C applications. Either can be used as the power input to a voltage regulator, and thereby provide the system with power. The host computer therefore uses RTS of its serial port as the power control for the embedded system. Under software, the host sends RTS high, and the embedded system is powered up. If the host sends RTS low, the embedded system is powered down. The caveat to all this is to ensure that your embedded system's current draw is low enough so that it can be powered by RTS. The advantage of this technique is that you require no external power supply for your embedded system. It works, as if by magic, whenever it is plugged into a serial port. The catch is that you can't then use that RS-232C control signal for its original purpose. It must turn on and stay on to provide your embedded computer with power.

A sample schematic of this is shown in Figure, which also includes an RS-232C interface for a microcontroller, using a MAX3232. Note the diode, D1. Since RTS will be a negative voltage (as low as -15 V) when low, some protection is required for the voltage regulator, since it is not designed to have its input taken below zero volts. The diode can be any garden-variety power diode, such as a 1N4004, and will conduct only when RTS is positive. The voltage regulator (MAX604) converts the voltage from RTS to a supply of 3.3 V for the embedded system. If we required a supply of 5 V, we'd simply use a MAX603 instead. The circuit would otherwise be the same. The output of the regulator is smoothed by the capacitor C5, and a power-on LED is provided to show us when we have power. The MAX3232 sits between the RS-232C port and the processor, level-shifting the serial transmissions from the processor's logic levels to RS-232C, and vice versa.

Figure: Using RTS as a power source in a low-powered embedded system



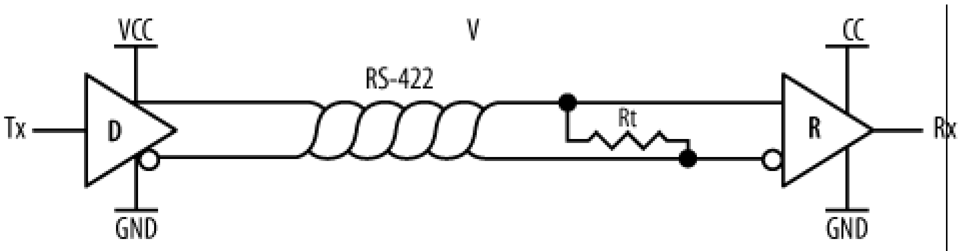
There we have the basics of RS-232C. It's a very common interface that is easy to use, but it does have its limitations and quirks. It was originally intended for connecting dumb terminals and teletypes to modems, not for interconnecting computers and peripherals. A better choice is RS-422, designed for more robust and versatile serial connections.

RS-422

Unlike RS-232C, which is referenced to local ground, RS-422 uses the difference between two lines, known as a twisted pair or a differential pair, to represent the logic level. Thus, RS-422 is a balanced transmission, or, in other words, it is not referenced to local ground. Any noise or interference will affect both wires of the twisted pair, but the difference between them will be less affected. This is known as common-mode rejection. RS-422 can therefore carry data over longer distances and at higher rates with greater noise immunity than RS-232C. RS-422 can support data transmission over cable lengths of up to 1,200 meters (approximately 4,000 feet).

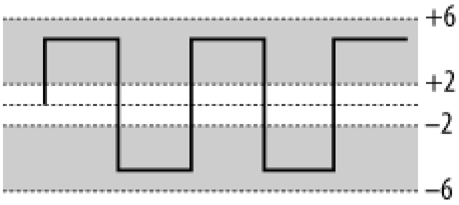
Figure shows a basic RS-422 link, where a driver (D) of one embedded system is connected to a receiver (R) of another embedded system via a twisted pair. The resistor, Rt, at receiving end of the twisted pair is a termination resistor. It acts to remove signal reflections that may occur during transmission over long distances, and it is required. Rt is nominally 100-120 Ω.

Figure :RS-422



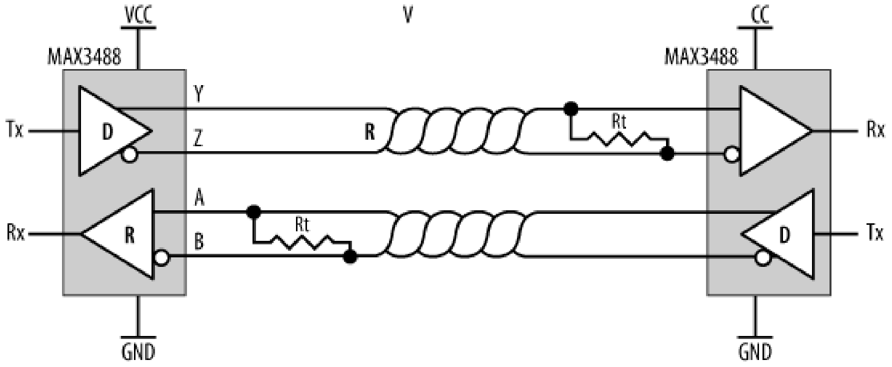
The voltage difference between an RS-422 twisted pair is between ±4 V and ±12 V between the transmission lines (Figure). RS-422 is, to a degree, compatible with RS-232C. By connecting the negative side of the twisted pair to ground, RS-422 effectively becomes an unbalanced transmission. It may then be mated with RS-232C. Since the voltage levels of RS-422 fall within the acceptable ranges for RS-232C, the two standards may be interconnected. RS-422 was the serial interface found on early Apple Macintosh computers, quietly dropped with the coming of the iMacs.

Figure: RS-422 voltage levels



There is a wide variety of RS-422 interface chips available. Figure shows a simple RS-422 bidirectional interface implemented using two Maxim MAX3488s. The Tx and Rx pairs of each MAX3488 are connected to UARTs within each embedded system, just as we did with RS-232C.

Figure : Bidirectional RS-422 interface



It's important to note that RS-422 specifies only the voltages for the standard, not the physical implementation (pinouts or connectors). That is covered by RS-449. Now, no one seems to bother with RS-449, mainly because it is unnecessarily complex for most uses. People using RS-422 just seem to do their own thing, picking whatever cable and connectors (and pinouts!) they feel are appropriate for their application. Self-expression and RS-422 seem to go hand in hand.

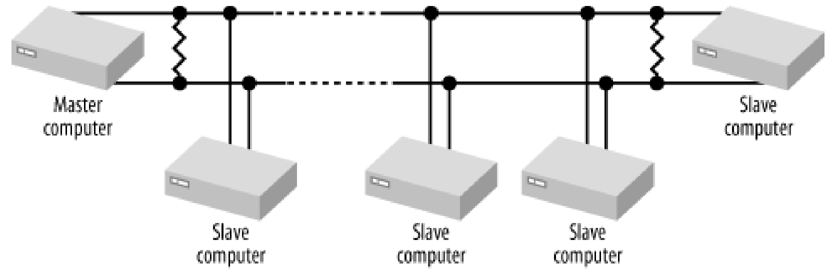
Some RS-422 interface chips have an optional enable input. When enabled, the chip outputs and drives a transmission onto the twisted pair. When disabled, the chip's output is high-impedance, and the chip appears "invisible." Because of the ability of the interface chip to "disappear" from the connection, it is possible to have multiple interface chips (and therefore more than two embedded systems) connected to the twisted pair. In this way, it is possible to extend RS-422 into a low-cost, robust, simple network. When implemented in this fashion, it becomes RS-485.

RS-485

RS-485 is a variation on RS-422 that is commonly used for low-cost networking and in many industrial applications. It is one of the simplest and easiest networks to implement. It allows multiple systems (nodes) to exchange data over a single twisted pair (Figure).

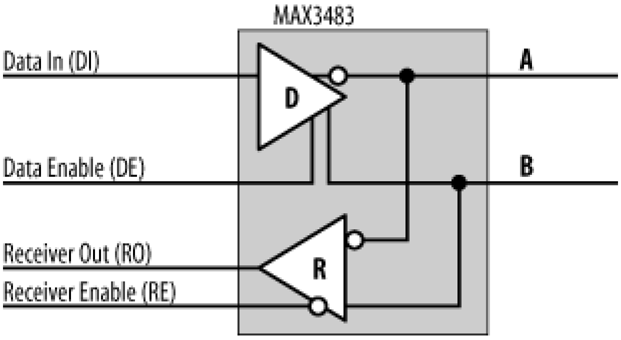
RS-485 is based on a master-slave architecture. All transactions are initiated by the master, and a slave will transmit only when specifically instructed to do so. There are many different protocols that run over RS-485, and often people will do their own thing and create a protocol specific to the application at hand.

Figure:RS-485 network



The interface to the RS-485 network is provided by a transceiver, such as a Maxim MAX3483 (Figure).

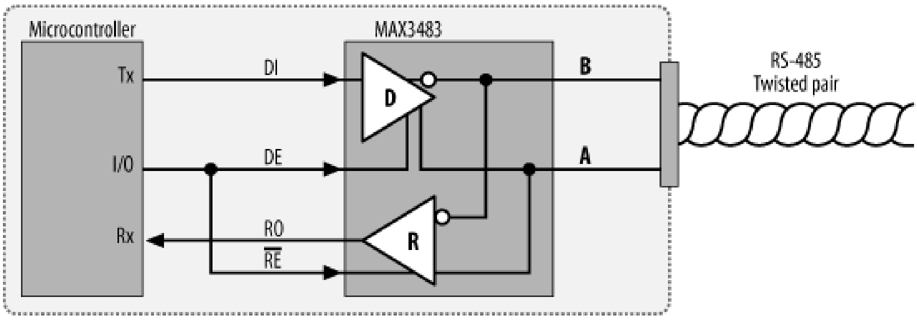
Figure: RS-485 transceiver



The MAX3483 is just an RS-422 transceiver with enable inputs, and using it in a design is straightforward. On the network side, the MAX3483 has two signal lines, A and B. This is the twisted pair (network cable) attachment point. The MAX3483 also has Data In (DI) and Receiver Out (RO). These are connected to the Tx and Rx signals of the UART (or microcontroller), respectively.

Since it is connected to a common network on which it must both listen and transmit, it has two control inputs, Data Enable (DE) and Receiver Enable (DE). A high input to DE allows the DI input to be transmitted on the network. A low input to DE disables the output of the transmitter. Similarly, a low input to enables the receiver, and network traffic is passed through to RO. DE and are normally controlled by an I/O line of the processor. Now, you'll notice that DE is active high, and is active low. This is not by chance. A node on the network won't be receiving traffic if it's transmitting and, conversely, won't be transmitting if it is receiving. Therefore, only one of the two—the transmitter or the receiver—should be active at any one time. If the transmitter is on, the receiver should be off, and vice versa. The control for the transmitter is therefore the logical opposite of the control for the receiver. By having DE active high and active low, a single control line may be used for both. Figure shows a MAX3483 interfaced to a microcontroller in this way. The microcontroller normally has DE/ low so that it is listening to network traffic. When it wishes to transmit, it sends DE/ high. Upon completion of transmission, it returns DE/ low and resumes listening.

Figure: Connecting a MAX3483 to a microcontroller



RS-485 may be implemented as half duplex, where a single twisted pair is used for both transmission and reception (Figure1), or full duplex, where separate twisted pairs are used for each direction (Figure2). Full-duplex RS-485 is sometimes known as four-wire mode. Note that for full-duplex operation, the MAX3483s are replaced with MAX3491s that have dual network interfaces.Figure1. Half-duplex RS-485

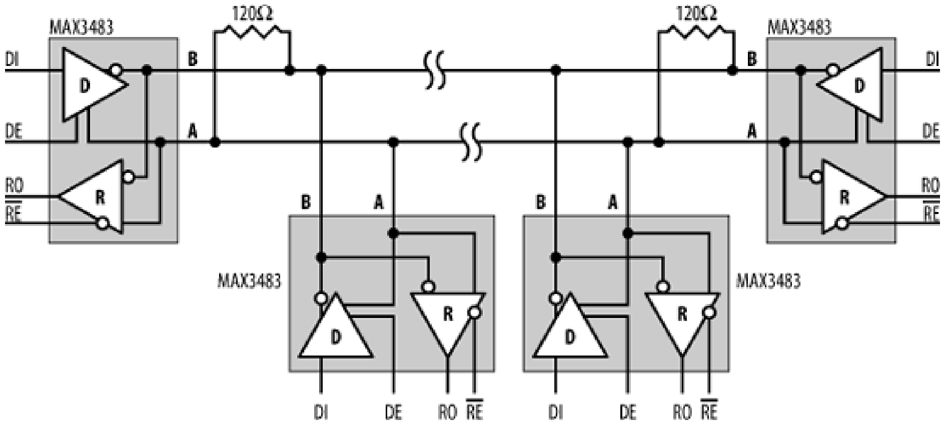
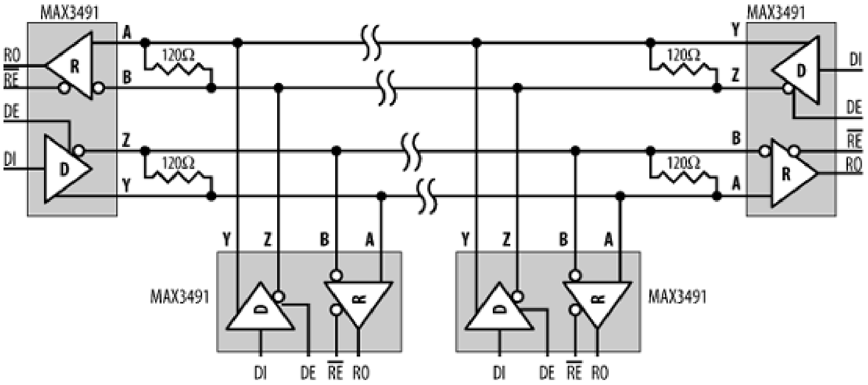


Figure: Full-duplex RS-485



These examples show four computers (nodes) connected to an RS-485 network. Each RS-485 interface chip (MAX3483 or MAX3491) exists in a separate embedded computer. The UART transmitter output, Tx, in each embedded system is connected to the respective DI of each of the RS-485 interface chips. Similarly, RO connects to the Rx input of each UART. The driver of each RS-485 interface chip is enabled by asserting DE, and, similarly, reception is enabled by asserting .

Normally, all systems connected to the RS-485 network have their receivers enabled and listen to the traffic. Only when a system wishes to transmit does it enable its driver. There are a number of formal protocols that use RS-485 as a transmission medium, and twice as many homespun protocols as well. The main problem you need to avoid is the possibility of two nodes of the network transmitting at the same time. The simplest technique is to designate one node as a master node and the others as slaves. Only the master may initiate a transmission on the network, and a slave may only respond directly to the master, once that master has finished.

The number of nodes possible on the network is limited by the driving capability of the interface chips. Normally, this limit is 32 nodes per network, but some chips can support up to 512 nodes.

USB

In previous chapter, we looked at RS-232C, that old standard of communication that's not so standard after all. RS-232C has lots of problems and lots of limitations. Getting any two RS-232C devices to talk is not as simple as it could or should be. You need the right cable with the right sort of connectors, and then you need to manually co-ordinate the communication parameters such as data rate, parity, and handshaking. At best it is a nuisance, at worst a headache. For hardware manufacturers, it presents a dilemma. Your goal in developing your product should be to make that product as easy to use as possible. You don't want users stumbling around with incorrect cables, manually configuring settings, and failing to seamlessly integrate your product with the rest of their system. This doesn't make for a happy user.

Universal Serial Bus (USB) is the solution. It allows peripherals and computers to interconnect in a standard way with a standard protocol and opens up the possibility of "plug and play" for peripherals. USB is rapidly dominating the desktop computer market, making RS-232C an endangered species. Apple Macintoshes no longer have RS-232C/RS-422 ports, and soon all PCs will go the same way. Therefore, an understanding of USB (and how to build a USB port) is critical if you wish to interface your embedded computer to the desktop machines of the near future. USB supports the connection of printers, modems, mice, keyboards, joysticks, scanners, cameras, and much more.

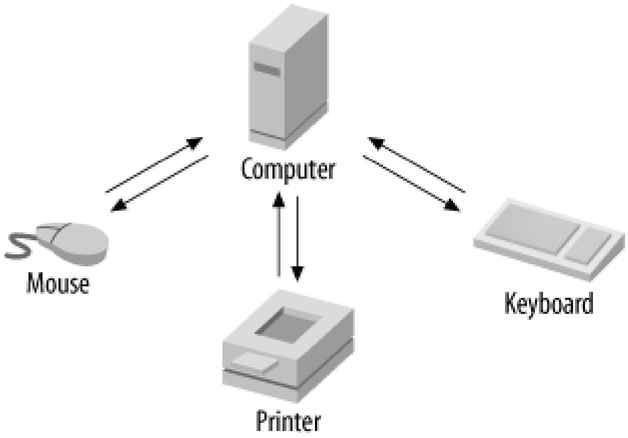
USB opens a wealth of possibilities, but developing with it is more complex than with RS-232C. USB has the advantage (for the user) that devices interact with the host computer's OS. No manual setup is required. However, it does add an extra layer of complexity to your software, since your embedded code must interact with the host in the appropriate way. USB can even provide power to peripherals through the same cable as data. No external power supply (or power cable) is required. So for the user, a USB peripheral is simplicity itself.

In this chapter, you'll get an overview of USB and then go on to see how you can incorporate a USB interface into your embedded system. The protocols and specifications for USB are long and complex, and well beyond the scope of this book. Fortunately, to design USB-based hardware, the task is much simpler. We'll simply take an overview and then look at a physical USB implementation.

Introduction to USB

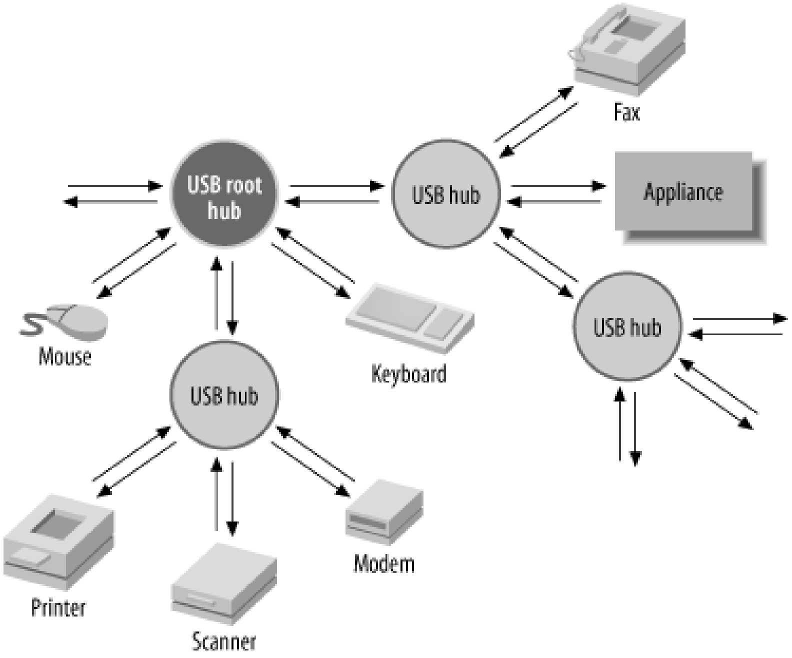
There are two specifications for USB: USB 1.1 and USB 2.0. USB 2.0 is fully compatible with USB 1.1. USB supports data rates of 12 Mbps and 1.5 Mbps (for slower peripherals) for USB 1.1, and data rates of 480 Mbps for USB 2.0. Data transfers can be either synchronous or asynchronous. USB is a high-speed bus that allows up to 127 devices to be connected (Figure). No longer are having only one or two ports on your computer a limitation. Further, one standard for cables and connectors eliminates the confusion that existed with RS-232C. Devices are able to self-identify to a host computer, and they can be hot-swapped, meaning that the systems do not need to be powered down before connection or disconnection.

Figure :USB allows a host to connect with a variety of peripherals



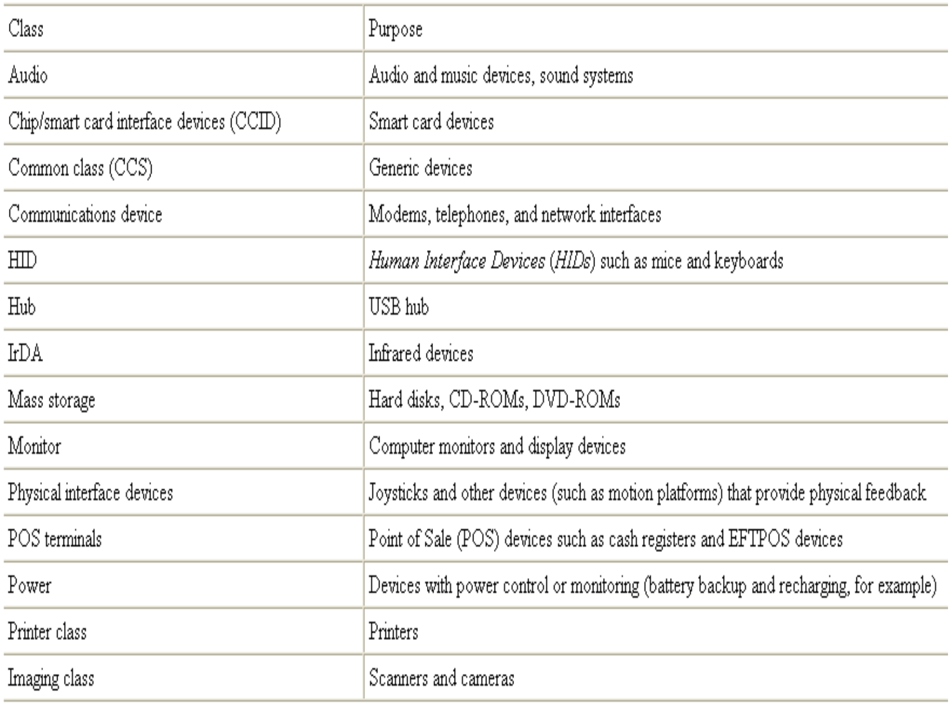
The basic structure of a USB network is a tiered star. A USB system consists of one or more USB devices (peripherals), one or more hubs, and a host (controlling computer). The host computer is sometimes known as the host controller. Only one host may exist in a USB network. The host controller incorporates a root hub, which provides the initial attachment points to the host. The hubs form nodes to which devices or other hubs connect, and they are (largely) invisible to USB communication. In other words, traffic between a device and a host is not affected by the presence of hubs.

Hubs are used to expand a USB network. For example, a given host computer may have five USB ports. By connecting hubs, each with additional ports, to the host, the physical connectivity of the system is increased (Figure). Many USB devices, such as keyboards, incorporate inbuilt hubs allowing them to provide additional expansion as well as their primary function. Figure: USB is expandable using hubs



The host will regularly poll hubs for their status. When a new device is plugged into a hub, the hub advises the host of its change in state. The host issues a command to enable and reset that port. The device attached to that port responds, and the host retrieves information about the device. Based on that information, the host operating system determines what software driver to use for that device. The device is then assigned a unique address, and its internal configuration is requested by the host. When a device is unplugged, the hub advises the host of the change in state when polled, and the host removes the device from its list of available resources. The detection and identification of USB devices by a host is known as bus enumeration.

USB "knows" about and supports different classes of devices. Each class represents the functionality that the device can provide to the host. Some sample classes (and sample devices) are listed in Table. A single, physical USB peripheral can encompass several classes.



Class	Purpose
Audio	Audio and music devices, sound systems
Chip/smart card interface devices (CCID)	Smart card devices
Common class (CCS)	Generic devices
Communications device	Modems, telephones, and network interfaces
HID	*Human Interface Devices* (*HIDs*) such as mice and keyboards
Hub	USB hub
IrDA	Infrared devices
Mass storage	Hard disks, CD-ROMs, DVD-ROMs
Monitor	Computer monitors and display devices
Physical interface devices	Joysticks and other devices (such as motion platforms) that provide physical feedback
POS terminals	Point of Sale (POS) devices such as cash registers and EFTPOS devices
Power	Devices with power control or monitoring (battery backup and recharging, for example)
Printer class	Printers
Imaging class	Scanners and cameras

USB Packets

There are four types of transfers that can take place over USB. A control transfer is used to configure the bus and devices on the bus, and to return status information. A bulk transfer moves data asynchronously over USB. An isochronous transfer is used for moving time-critical data, such as audio data destined for an output device. Unlike a bulk transfer, which can be bidirectional, an isochronous transfer is uni-directional and does not include a cyclic-redundancy-check (CRC) field. An interrupt transfer is used to retrieve data at regular intervals, ranging from 1 to 255 milliseconds.

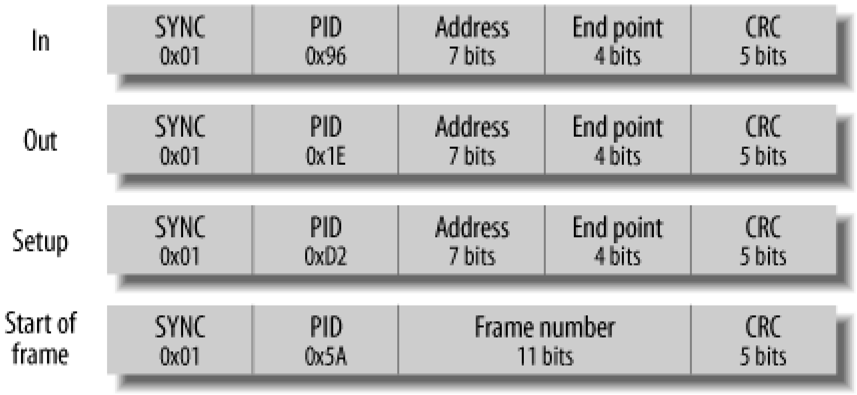
Data is transferred between USB devices using packets, and a transfer can comprise one or more packets. A packet consists of a SYNC (synchronization) byte, a PID (Packet ID), content (data, address, etc.), and a CRC.

The SYNC byte phase locks the receiver's clock. This is equivalent to the start bit of an RS-232C frame. The PID indicates the function of the packet, such as whether it is a data packet or a setup packet. The upper four bits of the packet ID are the inverse of the lower four bits, for additional error checking. For example, the packet ID for a data packet is `0x3C`. In binary, this is `%0011 1100`.

USB packets can be one of four types: token, data, handshaking, or preamble.

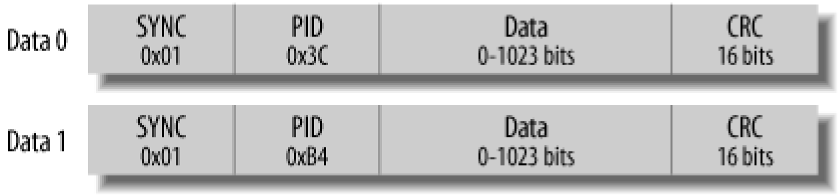
Tokens are 24-bit packets that determine the type of transfer that is to take place over the bus. There are four types of token packet (Figure). A token packet consists of a SYNC byte, a packet ID (indicating packet type), the address of the device being accessed by the host, the end-point address, and a 5-bit CRC field. The end-point address is the internal destination of the data within the device.

Figure: USB token packets



In	SYNC 0x01	PID 0x96	Address 7 bits	End point 4 bits	CRC 5 bits
Out	SYNC 0x01	PID 0x1E	Address 7 bits	End point 4 bits	CRC 5 bits
Setup	SYNC 0x01	PID 0xD2	Address 7 bits	End point 4 bits	CRC 5 bits
Start of frame	SYNC 0x01	PID 0x5A	Frame number 11 bits		CRC 5 bits

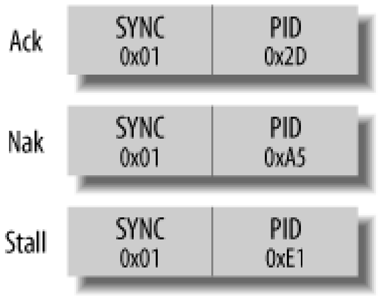
There are two types of data packet, known as DATA0 and DATA1 (Figure). The transmission of data packets alternates between the two types. A single data packet can transfer between 0 and 1,023 bytes, and the data packet's CRC is 16 bits due to the larger packet size. Figure: USB data packets



Data 0	SYNC 0x01	PID 0x3C	Data 0-1023 bits	CRC 16 bits
Data 1	SYNC 0x01	PID 0xB4	Data 0-1023 bits	CRC 16 bits

There are three types of handshaking packets (Figure). A successful data reception is acknowledged with an Ack packet. The receiver notifies the host of a failed transmission by sending a Nak (No Acknowledge) packet. A Stall is used to pause a transfer.

Figure: USB handshaking packets



A descriptor is a data packet used to inform the host of the capabilities of the device. It contains an identifier for the device's manufacturer, a product identifier, class type, and the device's internal configuration, such as its power needs and end points. Each manufacturer has a unique ID, and each product in turn will also have a unique ID. Software on the host computer uses information obtained from a descriptor to determine what services a device can perform and how the host can interact with that device.

Physical Interface

USB uses a shielded, four-wire cable to interconnect devices on the network (Table 11-2). Data transmission is accomplished over a differential twisted pair (much like RS-422/485) labeled D+ and D-. The other two wires are V_{BUS}, which carries power to USB devices, and GND. Devices that use USB power are known as bus-powered devices, while those with their own external power supply are known as self-powered devices. To avoid confusion, the wires within a USB cable are color-coded.

Connector pin	Signal	Purpose	Wire color
1	VBUS	USB device power (+5V)	Red
3	D+	Differential data line	Green
2	D-	Differential data line	White
4	GND	Power and signal ground	Black

Some USB chips refer to D+ and D- as DP and DM, respectively.

The connection from a device back to a host is known as an upstream connection. Similarly, connections from the host out to devices are known as downstream connections. Different connectors are used for upstream and downstream ports, with the specific intention of preventing loopback. The only way to connect a USB network is a tiered star. USB uses two types of plugs (jacks) and two types of receptacles (sockets) for cables and equipment. The first type is Series A, shown in Figure. Series A connectors are for upstream connections. In other words, a series A receptacle is found on a host or hub, and a series A plug is at the end of the cable that attaches to the host or hub.

Figure: Series A plug and receptacle

Series B connectors are shown in Figure. A series B receptacle is found on a USB device, and a series B plug is at the end of the cable coming downstream from a host or hub.

Figure: Series B plug and receptacle

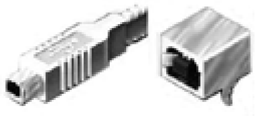
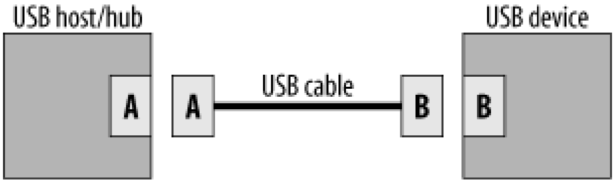


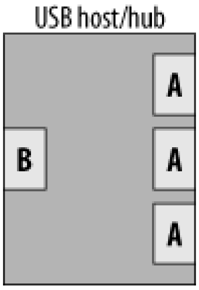
Figure shows how this works in practice. This ensures that USB devices, hosts/hubs, and USB cables are always connected in the right way. It should not be possible to have a cable plugged in the wrong way or to directly connect two USB peripherals.

Figure: USB connectors and cable



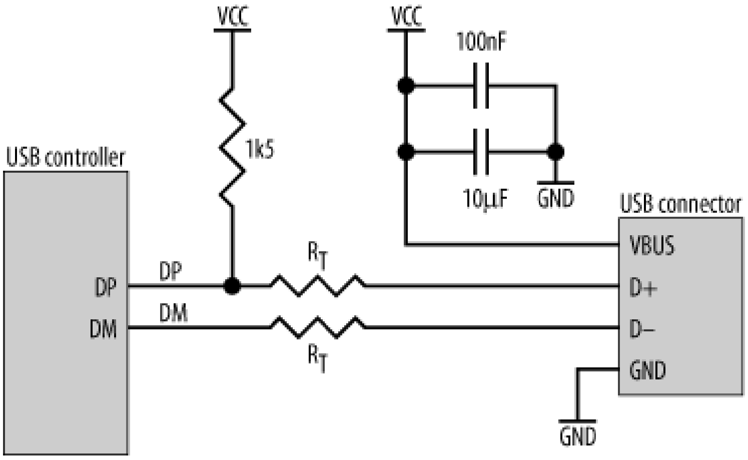
Since a hub will be connected to USB devices downstream and to a USB host or hub upstream, it will have both types of receptacle (Figure).

Figure: Receptacles on a USB hub



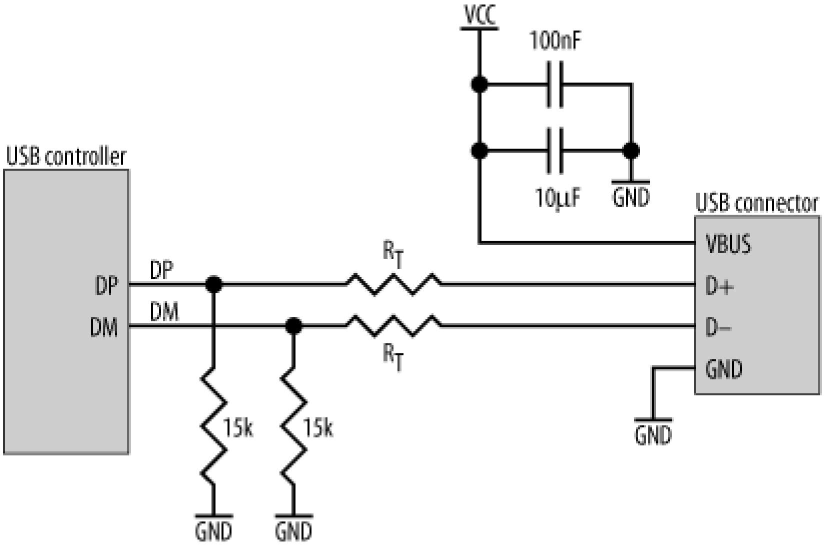
Chips that implement a USB interface require very few external components for the USB port. The schematic for an upstream port is shown in Figure.

Figure: Upstream USB port



In this example, the embedded system is powered from the USB port. If the embedded computer has its own power source, then no connection is made between VCC and pin 1 (V_{BUS}) of the USB connector. The pull-up resistor connected to DP is required only on upstream ports. If you are implementing downstream ports on a hub, the pull-up is not required. However, downstream ports require pull-down resistors on both DP and DM (Figure).

Figure. Downstream USB port



In both figures, DP and DM have series resistors (R_T) that terminate the USB connection. The total resistance of the termination should be 45Ω. However, the pins of the USB controller will have inherent impedance that will need to be taken into account. If the pin impedance is 21Ω (say), then the series resistors should be 24Ω. The catch here is that not all chip manufacturers are thorough enough to specify the pin impedance in their technical data. In such cases, you can either hound the manufacturer for the data, or take a punt. Ballpark values for the series resistors should be between 20Ω and 30Ω. Many microcontrollers, such as the Microchip PIC16C745 and PIC16C765, include USB modules as part of their suite of I/O. Implementing USB with such processors is easy. You simply need to add the physical interface to the DP and DM pins of the processor. However, if the chip you have chosen to use as the primary embedded processor does not include USB, you have to provide USB functionality with an external device.

Implementing a USB Interface

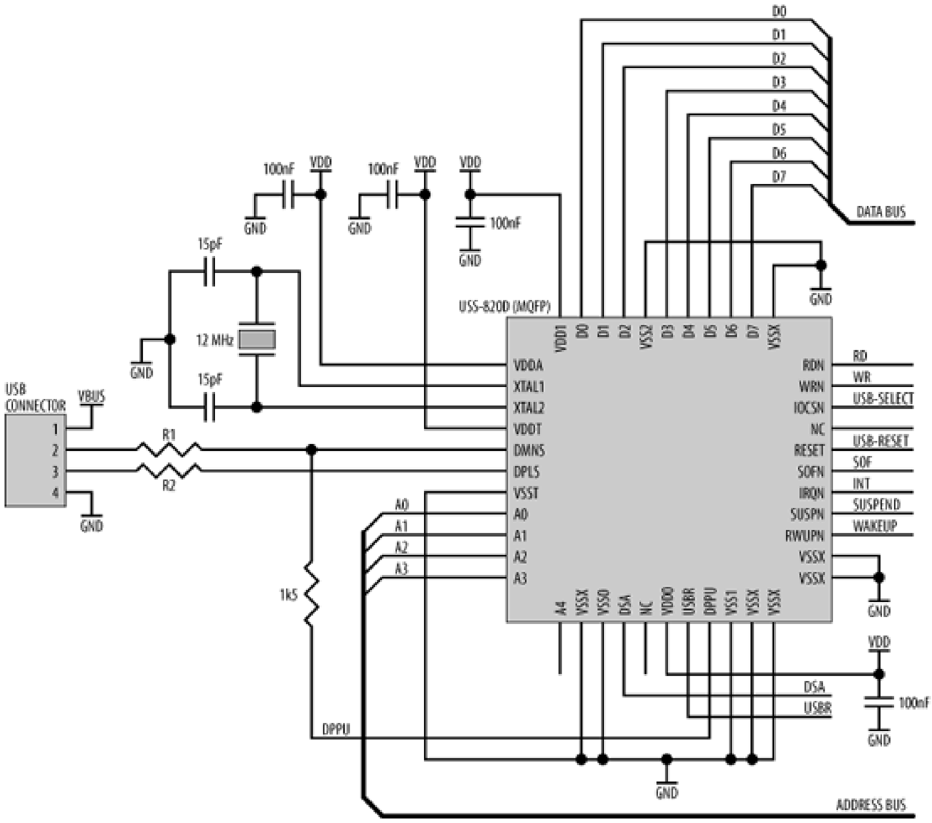
One possible solution to implementing USB in your embedded system is to use a USB-to-SPI bridge, such as the Atmel AT76C711. This chip is an AVR processor with a USB subsystem, designed to act as a slave USB controller to a host processor. It has 2K of data RAM, 2K of dual-port RAM for packet processing, 16K of program RAM (organized as 8K x 16 words), an inbuilt DMA controller, an upstream USB port (with one control and five data end points), a separate IrDA-compatible UART, and SPI. The processor may be run at up to 24 MHz and operates off a 3.3 V supply. At reset, the AT76C711 automatically loads its software from an external AT45DBxxx data flash to the program RAM. Since the AT76C711's program space is small, one of the smaller AT45DBxxx data flashes will be sufficient. Alternatively, a host processor may load the AT76C711's code directly into its program RAM while it is held in reset.

The AT76C711 may act as a standalone processor, performing USB bridging functions to RS-232C, RS-422/RS-485, IrDA, or other protocols. Alternatively, it may be incorporated as a slave processor in a larger embedded system. The host processor may communicate with the AT76C711 either via SPI or by a standard serial interface through one of the AT76C711's UARTs. The AT76C711 also has general-purpose I/O lines and a UART module that supports RZ encoding for IrDA.

If the processor you are using has a bus interface, then you can add USB using a chip such as the USS-820D by Agere Systems. It supports transfers of up to 12 Mbps and is specifically designed for use in USB devices, unlike a lot of USB chips that are intended for use in hubs. It can support up to eight endpoints, each with receive and transmit buffers of up 1,120 bytes.

The schematic of an upstream USB interface, using the USS-820D, is shown in Figure. This chip is available in two footprints; the

MQFP is shown in this circuit. For both footprints, the signals are the same. The only difference is the pin numbering.



The USS-820D has several power-supply inputs (V_{DDA}, V_{DDT}, V_{DD0}, V_{DD1}), all of which operate from a 3.3 V supply (VDD in the schematic). Each power pin is decoupled to ground using a 100 nF capacitor. The 5 V power (V_{BUS}) available from the USB connector cannot be used to drive the USS-820D directly. However, a MAX604 voltage regulator circuit will convert V_{BUS} to the required 3.3 V supply. The USS-820D also has numerous ground pins (V_{SST}, V_{SSX}, V_{SS0}, V_{SS1}, V_{SS2}), all of which are connected to ground. Even though this chip uses a 3.3 V supply, its digital (non-USB) inputs are compatible with 5 V logic, and so it may be interfaced directly to a processor operating on a 5 V supply.

XTAL1 and XTAL2 are the connections for a 12 MHz crystal, providing timing for the USB controller.

The connections to a microprocessor are straightforward. The USS-820D's data pins, D0 through D7, connect directly to the processor's data bus. Similarly, the low-order address pins, A0 through A4, connect to the corresponding signals from the processor. These address bits are used to select internal registers within the USS-820D. The processor's read (RD) and write (WR) signals connect directly to USS-820D's read (RDN) and write (WRN) pins. (Agere places an "N" after pin names that are active low.) The USS-820D is selected when IOCSN is asserted low. Therefore, this pin is driven from an address decoder output (which I've labelled USB-SELECT in the schematic).

The USS-820D is reset when its RESET pin is sent high (not low like most other devices). So, for normal operation this pin should be held low. To allow the USS-820D to be reset under software control, this pin could be driven by a processor digital output line.

The USS-820D has a number of outputs that may be used to notify a host processor of the current USB status. DSA (Data Set Available), USBR (USB Reset detected), SUSPN (Suspend), and SOFN (Start Of Frame) may either be read as digital inputs by the host microcontroller, or, for processors that have several interrupt inputs, these signals may be used to generate an interrupt. If the host processor has only a limited interrupt capability, all of these events will trigger the USS-820D interrupt pin (IRQN). This pin can therefore serve as the sole interrupt input to the processor. A word of caution, however: this pin can be configured under software control to be either active high or active low. Getting this wrong can put your embedded system in a permanent state of interrupt. The default state for this pin is active low, which suits most processors. For processors that have active-high interrupts, such as Intel processors, the firmware should configure USS-820D for the correct interrupt polarity before enabling the processor's interrupt-handling capability.

The RWUPN pin is an input that signals a Remote Wake-Up condition. In other words, this embedded system has been asleep (in suspend mode) and has awoken. This pin notifies the USS-820D of the change in state so that it can alert other USB systems. RWUPN is simply driven by a processor digital output line.

The USB differential data signals are pins DPLS (Data Plus, D+) and DMNS (Data Minus, D-). These are connected to the USB connector through series-termination resistors. Agere Systems suggests a nominal value of 24Ω. For an upstream connection, DPLS (D+) requires a pull-up resistor of 1.5 kΩ. Normally, this resistor is connected to +5 V. However, the USS-820D provides a special pin (DPPU) specifically for this purpose. Thus, under software control, the USS-820D can simulate a USB-device disconnect. It will appear to an upstream hub that the system containing the USS-820D has been physically disconnected, even though it is still attached. This can be useful during development and testing. It also allows the USB device to decide whether or not a host knows it is connected. DPPU may be decoupled to ground using a 10 nF capacitor. Chips such as the USS-820D make adding USB functionality to your embedded hardware simple and easy. Through USB, you can develop peripherals based on embedded processors for desktop computer systems. Alternatively, you can use USB to connect existing peripherals to your embedded computer to further increase its functionality.

Summary:

- The Serial Peripheral Interface (known as SPI) was developed by Motorola to provide a low-cost and simple interface between microcontrollers and peripheral chips.

- SPI uses four main signals: Master out Slave in (MOSI), Master in Slave out (MISO), Serial CLOCK (SCLK or SCK) and Chip Select (CS) for the peripheral.

- The Maxim DS1305 Real-Time Clock (RTC) provides timekeeping services and tracks seconds, minutes, hours, day of the month, month, day of the week, and year.

- The AD5203 has a Serial Data Input (SDI), which is connected to the processor's MOSI output.

- The internal memory of microcontrollers is very small, and their data storage capabilities are severely limited.

- The EEPROM is the Atmel AT25640. This device will hold data for at least 100 years without power, and will endure more than one million write cycles.

- I^2C (Inter-Integrated Circuit) bus is a very cheap yet effective network used to interconnect peripheral devices within small-scale embedded systems.

- The simplest form of serial interface is that of the Universal Asynchronous Receiver Transmitter (UART). UARTs are also sometimes called Asynchronous Communication Interface Adapters (ACIAs).

- RS-232C is used for interfacing serial devices over cable lengths of up to 25 meters and at data rates of up to 38.4 kbps.

- When two remote systems are communicating serially, there needs to be some way to prevent the transmitter from sending new data before the receiver has had a chance to process the old data. This process is known as handshaking, or flow control.

- Hardware handshaking in RS-232C uses two signals, RTS (Request To Send) and CTS (Clear To Send).

- Figure shows a basic RS-422 link, where a driver (D) of one embedded system is connected to a receiver (R) of another embedded system via a twisted pair.

- There are two specifications for USB: USB 1.1 and USB 2.0. USB 2.0 is fully compatible with USB 1.1. USB supports data rates of 12 Mbps and 1.5 Mbps (for slower peripherals) for USB 1.1, and data rates of 480 Mbps for USB 2.0.

Question:

- Write short notes on serial peripheral Interface?
- How ADS1305 is interfaced with a micro controller?
- How a digital potentiometer is interfaced through ISP?
- Write a short notes on I^{2}C communication?
- What is UART? How error is detected in UART?
- Write a short notes on USB?
- Write notes on following interfaces

  a) RS232 b)RS432 c)RS485

References:

- An_Embedded_Software_Primer by David E simon
- Mathai Joseph, Real-time Systems: Specification, Verification and Analysis, Prentice Hall International, London, 1996
- Comp.realtime FAQ, newsgroup comp.realtime
- Real-time Embedded Software Systems: An Introduction S. Agrawal & P. Bhatt http://www.embedded.com
- Michael Barr, Programming Embedded Systems in C and C++, O'Reilly Associates, August 1999.

9. ANALOG

Objective:

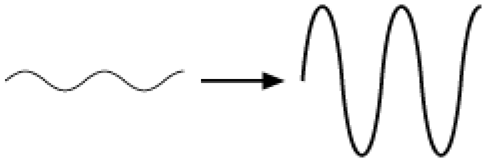
In this chapter, we'll look at how you sample external voltages and convert these into digital values for processing by your embedded system. Such voltages may be generated by sensors and may represent light levels, temperature, or vibration. Or perhaps the voltages are the output of a microphone or audio system and need to be converted into digital data. Later, we'll take a look at how you turn digital data into an analog output voltage. We'll conclude the chapter with hardware to control electric motors.

First, though, let's take a quick look at amplifiers and sampling theory. Note that this is a very complex field. Since the background theory is well beyond the scope of this book, we'll just take an overview, giving enough background to allow you to interface your embedded system to simple analog circuitry. This discussion is by no means exhaustive, and it is deliberately simplified.

Amplifiers

Amplifiers are used to interface one analog circuit to another. An amplifier is a circuit that increases (or decreases) a given input voltage to produce an output voltage. For example, say you had a sensor that produced a maximum output that was 5 mVpp, and this was to be interfaced to a sampling system that required an input signal of 5 Vpp. You would use an amplifier between the sensor and the sampling system to increase the sensor's output accordingly (Figure).

Figure . Amplifying a waveform



The waveform of the amplifier's output signal should be identical to the input signal; only its amplitude will have changed. The amount of increase or decrease in the signal is known as the gain of the amplifier. Gain is calculated easily by dividing the output voltage by the input voltage:

$$\text{Gain} = V_{OUT} / V_{IN}$$

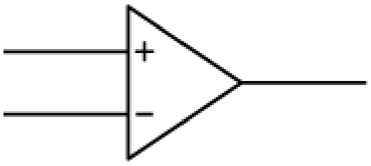
Thus, an amplifier that doubles the input signal will have a gain of 2. The ability of a circuit to respond to a changing signal is typically limited to a given range of frequencies. This is known as the frequency response of a circuit. For example, the amplifier in your home stereo may have a frequency response of 20-20 kHz. This means that it will amplify audio signals that have a frequency between 20 Hz (low bass) and 20 kHz (high treble). Try to pump a 100 MHz signal into the audio amp and it simply will not be able to amplify the signal. The signal is said to be outside its frequency range.

Ideally, the frequency response of a circuit, such as the audio amplifier, should be flat over its frequency range. This means that its response to an input signal will be the same, no matter the frequency (within the appropriate range). So, in the case of the audio amp, the gain will be constant for any frequency of signal in the appropriate range. Thus, the volume will not vary with frequency (ignoring any differences due to the original music). At either end of the frequency range, the ability of the amplifier to perform ideally degrades. At these extremes of frequency, the amplifier's gain diminishes. This is known as roll off. Some small degree of roll off is considered acceptable (and unavoidable). The frequency response is normally defined as the frequency range where the gain is within a certain limit of the ideal.

The limitation of an amplifier to replicate the input signal at its output is the distortion of the amplifier. For audio amplifiers, you'll sometimes see the term Total Harmonic Distortion(THD) listed in the specifications. The smaller this number is, the better the amplifier.

In days of old, amplifiers were constructed using discrete transistors[*] or vacuum tubes (also known as valves). These days, amplifiers are available packaged in integrated circuits. These amplifiers are known as operational amplifiers, or op amps for short. They make the designer's life much easier. They are cheap, reliable, and so very easy to use. Throughout this chapter, whenever we need to amplify a circuit, we'll use an appropriate op amp for the job. The schematic symbol for an op amp is shown in Figure.

Figure . Schematic symbol for an op amp

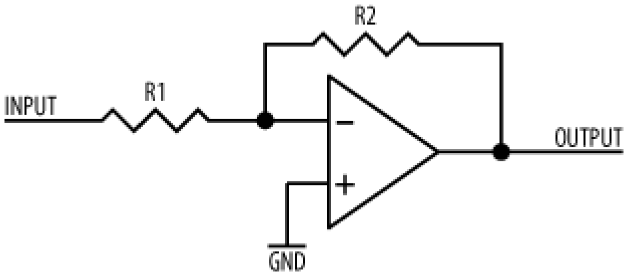


The input marked with "+" is known as the noninverting input, and the input marked with "-" is the inverting input. If the voltage present at the noninverting input is greater than that present at the inverting input, the output of the op amp is positive. Conversely, if the noninverting input is less than the inverting input, the output is negative. Typically, an op amp's output will not go as low as its negative power supply, nor as high as its positive power supply, due to the limitations of the internal circuitry. An op amp whose output voltage range does span the difference between its positive and negative power supplies is said to have rail-to-rail operation.

In order to function correctly, an op amp requires feedback. Feedback involves coupling the output of an amplifier back to its input. Negative feedback uses the output to reduce the gain of the amplifier and, in doing so, improves the amplifier's other characteristics, such as the flatness of the frequency response and immunity to distortion. Negative feedback is achieved simply by connecting a resistor between the output and the inverting input, as we will shortly see. (A circuit with no feedback is said to be open-loop.) Op amps are designed in such a way as to make the output change to cancel the difference between the inputs via a feedback resistor. Thus, the output waveform follows the difference between the input waveforms. The magnitude of the output is proportional to the feedback resistor. The larger the resistor, the more the feedback of the output is attenuated. Thus, the op amp makes the output larger to compensate. In this way, the output is an amplified version of the input.

An op amp may either be used as an inverting amplifier (Figure) or a noninverting amplifier (Figure). An inverting amplifier "flips" the signal in addition to amplifying it.

Figure. Inverting amplifier



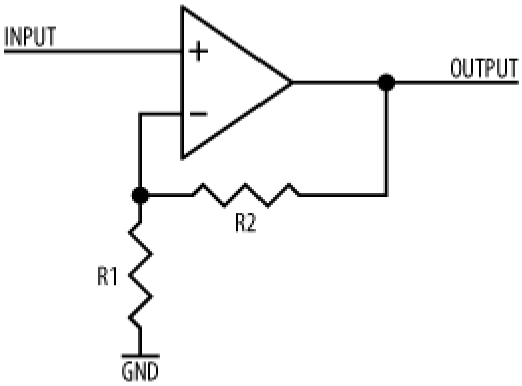
The gain of an inverting amplifier is given by:

Gain = - R2 / R1

Note the minus sign. That's because this amplifier inverts the signal.

You are more likely to use a noninverting amplifier (Figure), which doesn't flip the signal. These are commonly used in audio and sensor applications.

Figure. Noninverting amplifier



The gain of a noninverting amplifier is given by:

Gain = 1 + R2 / R1

The gain of the amplifier may be set under software control by using a digital potentiometer for R2.

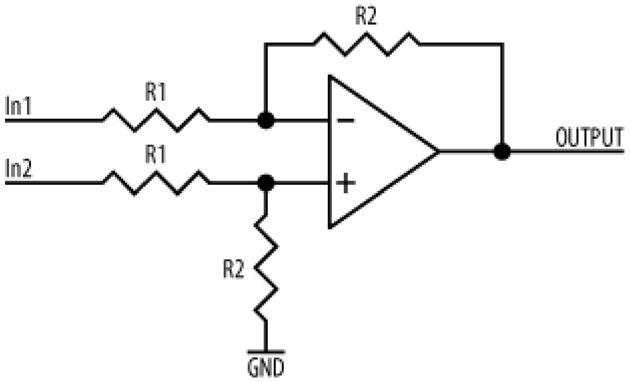
A differential amplifier (Figure) multiplies the difference between two input signals and is used to amplify small signals that may be subject to noise. By amplifying the difference between the signal of interest and a reference, any noise present is reduced (since the noise will affect both the signal and the reference equally). When both inputs to a differential amplifier change in the same way, this is known as a common-mode change. Ideally, a differential amplifier should be immune to common-mode changes, since its purpose is to amplify the signal difference. Its immunity to common-mode changes is known as its Common-Mode Rejection Ratio (CMRR). The higher the CMRR, the better. To achieve a high CMRR, it is important to match the values (and tolerances) of the resistors as closely as possible.

The output voltage of this differential amplifier is given by:

V_{OUT} = (In2 - In1) * (R2 / R1) 13.2. Analog to Digital Conversion

A device that converts an analog input voltage to a digital number is known as an Analog to Digital Converter, or simply and more commonly as an ADC. You may have also heard the term codec (COder DECoder) before. A codec is an ADC

Figure. Differential amplifier



combined with a Digital to Analog Converter (DAC), providing both analog input and analog output in one chip. We'll look at DACs in more detail later in this chapter.

ADCs are found in cell phones and digital phones, converting your voice to digital data for transmission. They are also used in your computer to digitize the input from a microphone for speech recognition. Professional recording studios use ADCs to convert audio to digital data in preparation for CD mastering. Similarly, video is sampled using ADCs prior to DVD mastering. Your scanner, web cam, and digital camcorder all have ADCs in them. At the other end of the application spectrum, ADCs are used to sample inputs from sensors. These applications can range from automated weather stations to the system monitoring the processor temperature in your PC.

There are several different types of ADC. Integrating ADCs use an internal voltage-controlled oscillator to produce a clock signal whose frequency is proportional to the voltage being sampled. The clock signal is used to drive a counter, which provides the digital value for the sample. The higher the sampled voltage, the higher the clock frequency, and therefore the higher the number reached by the counter. The counter is reset prior to each conversion. Because of this conversion technique, integrating ADCs are not known for their speed of conversion.

A successive approximation ADC uses a DAC to provide an analog reference voltage that is compared to the input voltage. By incrementing the digital code driving the DAC, the reference voltage is increased until a match is found. Once this happens, the code used to drive the DAC is used as the digital output of the ADC.

Flash ADCs (also known as parallel ADCs) use a bank of comparators to compare the input voltage to a range of reference voltages. The conversion of the input analog voltage to a digital value is therefore very fast. The catch is that flash ADCs tend to be more expensive than other types of ADC and, due to their complexity, normally have a lower resolution than other forms of ADC.

The process of converting an analog signal to digital is known as sampling or quantization. ADCs have two principle characteristics:

sample rate and resolution. Sample rate is expressed as samples per second (SPS) and refers to how frequently an analog input signal is converted into a digital code. The faster an ADC's sample rate, the more expensive that chip will be. Resolution determines the accuracy of each sample. For example, an "8-bit ADC" will return an 8-bit code representing the sampled input signal. This means that the input has been quantized into one of 256 discrete values. An "11-bit ADC" will quantize the signal into one of 4,096 values, yielding a more accurate result. However, the higher the resolution, the more expensive the ADC. Further, high resolution is not always required. If, for example, you're sampling a temperature sensor that has a range of 0ºC to 100ºC, with an accuracy of ± 0.5º C, then that sensor has only 200 meaningful voltage levels. For this sensor, an 8-bit ADC is fine. While you could use an 11-bit ADC to sample this sensor, the additional resolution is overkill.

An ADC will convert the analog signal into a number that represents the ratio of the input signal to a given reference voltage. For example, if the ADC's reference voltage is 5 V, and the input signal is 3 V, then the ratio of input to reference is 60%. So for an 8-bit ADC, where 255 represents full scale, the sampled input will be returned as `153` (`0x99`). From your point of view, you receive the value `153` from the ADC, and must work back from this to calculate the original analog voltage:

Signal = (sample / max_value) * reference_voltage

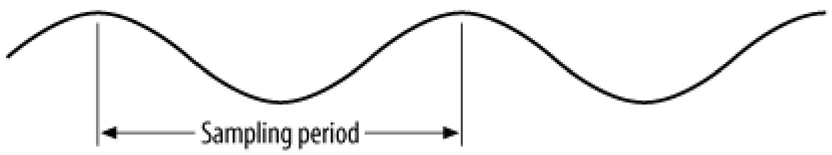
= (153 / 255) * 5

= 3 Volts

Sample Rates

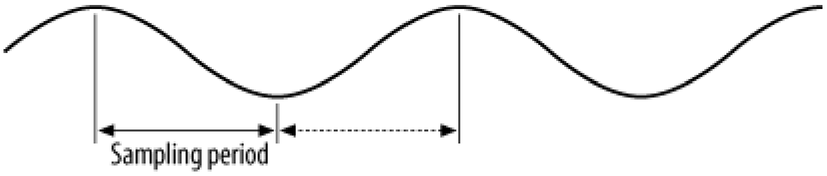
The rate at which a signal is sampled can have a dramatic effect on the quantized result and therefore can also affect the way in which software interprets that result. Figure shows a sinusoidal signal that is sampled at a rate equal to its period. In this example, the sample happens to coincide with a peak in the signal. The signal changes in between samples, but our choice of sample rate means that we get the same value each time. We get a completely false picture of what is really happening to that signal. To our sampling software, each value returned is the same, and so the signal appears to us as though it were a flat line!

Figure . Poorly chosen sample rate gives inaccurate signal reading



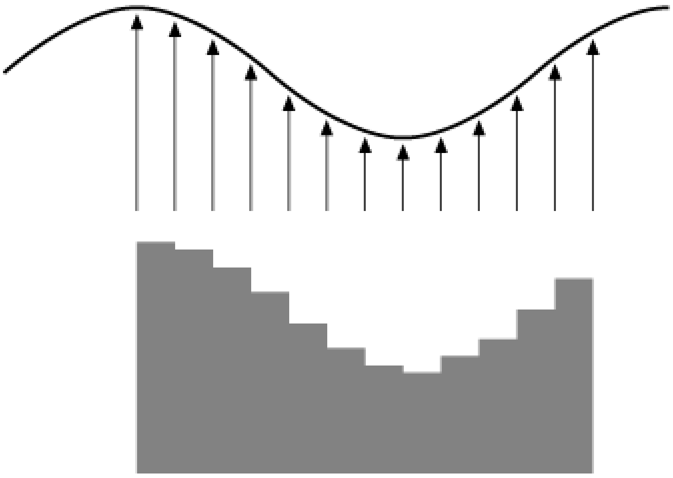
If we choose a sample rate that is double (or more) than the signal's highest frequency component, we can see the signal in more detail (Figure). This sampling frequency is known as the Nyquist frequency and is the lower limit of what will produce usable results. If the sample rate is slower than the Nyquist frequency, false artifacts (such as our sine wave appearing as a straight line, as we saw previously) may appear in the sampled result. These phantoms are known as aliasing.

Figure. Shorter sampling period



The faster the sample rate, the more accurate your sampled results will be. Since your sampling is quantizing the signal both in terms of amplitude (ADC resolution) and time (sample rate), a quantization error will always result (Figure).

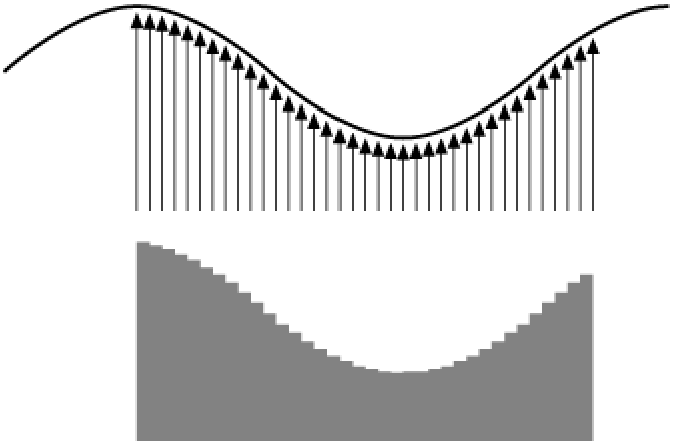
Figure. Sampling period and corresponding quantization



As you can see, the smooth sine wave of the original signal has become a jagged representation. Now, if you were monitoring temperature, this might be sufficient. You might not care how the temperature signal changed. Instead, you might be interested in the temperature only at specific intervals, and with only a limited accuracy. In such a case, this effect is not really a problem.

However, if you were sampling audio, this quantization effect could be a real problem. By increasing the sample rate, a more accurate representation of the original signal is obtained (Figure).

Figure . Fast sample period results in less quantization



A voice-mail system may use a sample rate of only 8 kHz and a resolution of 12 bits, and the resultant sound quality is limited. However, CD audio uses a sample rate of 44.1 kHz with 16-bit data and achieves a significant improvement in quality as a result. DVD audio uses a sample rate of 48 kHz with 24-bit data for even greater audio fidelity. To further improve sound quality, both CD and DVD players have special output filters to smooth the transitions between each sample when the data is converted back into analog form.

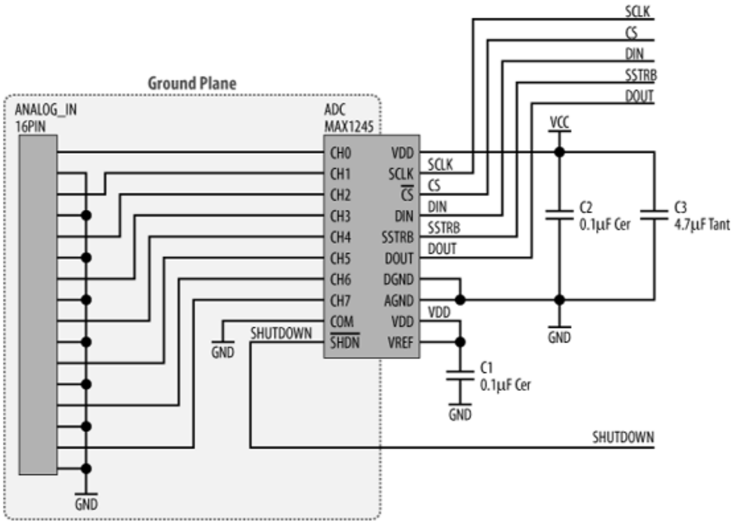
The take-home message is that you should choose your ADC resolution and sample rate carefully, keeping in mind exactly what you're sampling and what you intend to use it for

Interfacing an External ADC

There is a very wide range of ADCs available, for every considerable purpose. Choose from very low-cost, low-speed ADCs for simple voltage conversion to very high-speed, precise (and expensive) ADCs for sampling video streams. Many microcontrollers have inbuilt ADC subsystems, making analog interfacing simple. However, if the processor doesn't incorporate an ADC, or its ADC is not suited to your application, it becomes necessary to add an external device.

A good general-purpose ADC for sensor applications is the Maxim MAX1245. It has eight channels of analog input and can sample at 100,000 samples per second, with a resolution of 12 bits. (There are similar devices with resolutions ranging from 8 bits to 16 bits, and with interfaces such as SPI, I^{2}C, and processor bus.) The MAX1245 has an internal track and hold, preventing a changing signal from corrupting the result during a conversion. The MAX1245 is interfaced to a host processor via an interface that is compatible with SPI, Microwire, and the serial interfaces found in Texas Instruments TMS320-series DSP processors (Figure). As you can see, the MAX1245 is very easy to use. In this schematic, the analog input comes in via an IDC header, the 16-pin connector on the left of the figure. Note that every second pin on the connector is tied to ground. This means that every second wire in the connected cable will be grounded, providing a degree of noise immunity to our analog signals.

Figure. MAX1245 interface



The DOUT, DI, and SCLK signals correspond to a processor's SPI MISO, MOSI, and SCLK signals, respectively. is simply generated using a processor I/O line.

A conversion commences by sending a start command to the ADC via the SPI interface. The start command is simply a byte that specifies the channel and other ADC settings for that particular conversion. (Refer to the MAX1245 datasheet for more information on the software interface.) The MAX1245 may use an internal clock source to drive the conversion process, or it may have an external clock. The SPI SCLK also doubles as the conversion clock, when the ADC is used in external-clock mode. When used in internal-clock mode, the output, SSTRB (Serial Strobe), goes low at the beginning of a conversion and returns high once the conversion is complete. When an external clock is used, SSTRB pulses high in the clock period prior to the most significant bit being processed. SSTRB may be used to flag the completion of a conversion to a host processor by acting as an interrupt input. Alternatively, when used in external clock mode, the conversion result is ready once the start command has been sent.

The MAX1245 has the ability to enter low-power mode. This can be done either through hardware or software control. The MAX1245 has an input pin, SHDN, which, when asserted low, places the ADC into low-power operation. Now, interestingly, SHDN is also used to specify the clock frequency of the ADC's internal sampling. Sending this input high sets the clock to 1.5 MHz, whereas leaving the input to float (no connection) sets the clock to 225 kHz. If SHDN is driven by a microcontroller's I/O pin, changing that pin's configuration from an output to an input will effectively float SHDN. In this way, you can still use the "no connection" option even when the pin is connected. The MAX1245 can also be placed into low-power mode by software. If the two least significant bits of the start command are both 0, then the MAX1245 is placed into shutdown. The advantage of software power-down is that you can request a conversion and place the device into shutdown with a single command. The ADC will complete the conversion before shutting down, and its interface will remain active so that the result may be clocked out to the microcontroller.

Power for the MAX1245 (VDD) can be in the range of 2.7 V to 3.3 V. The MAX1245 has three ground pins: COM, DGND, and AGND. COM is the ground reference for the analog inputs, DGND is the ground connection for the digital section of the ADC, and AGND is the ground connection for the analog section of the ADC. These three grounds need to be connected together, but only at a single point, close to AGND. This is known as a star ground point. The two power inputs (VDD) need two decoupling capacitors to remove noise from the supply voltage. A 0.1 μF capacitor and a 4.7 μF capacitor should be used to decouple VDD and should be placed as close to the star ground point as possible. For particularly noisy power supplies, a 10 resistor should be placed in series between the power source and VDD. The analog inputs should be shielded from all nearby digital signals to prevent interference, and a ground shield (a fill) should be placed under the MAX1245 to further isolate the device from noise.

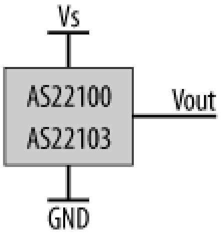
Now that we have seen how to add an ADC to a microcontroller, let's give it something to sample. We'll now take a look at some sensors and see how to interface them to an ADC. There are lots of different sensors available, from many manufacturers. Many are hard to use, awkward to interface, and require much more effort than seems necessary. But not all sensors are created equal. I have sought out and selected a range of sensors that are trivial to use and require little or no design effort. Electronics can be hard, but it doesn't always have to be so, as you will see.

Temperature Sensor

We'll start with something simple: a temperature sensor. This little sensor has a wide range of applications. The most obvious is as an environment monitor or weather station, but you could also use it to sense temperatures inside rooms and to control the appropriate heating or cooling systems. Combine it with a datalogger design, and you have a temperature recorder. Such devices are used in the shipment of fruits, vegetables, frozen foods, and flowers to ensure that they get to market in their best condition. It can also be used in the shipment of blood products and pathology samples, making sure that these critical substances are not exposed to adverse temperatures.

The AD22100 and AD22103 temperature sensors, by Analog Devices, are very easy to use. They are 3-pin devices, requiring only power (V_S) and ground to give you a voltage output that is proportional to temperature (Figure). The AD22100 requires a 5 V supply, and the AD22103 requires a 3.3 V supply.

Figure. AD22100/AD22103



What could be easier than that?

The output voltage corresponds to 22.5 mV/ºC over the temperature range -50ºC to +150ºC for the AD22100 and 28 mV/ºC over the temperature range 0ºC to 100ºC for the AD22103. The transfer functions (how the output relates to the input) for the two devices are given by:

$V_{OUT} = (V_S / 5) \times [1.375 + (0.0225 \times T_A)]$ AD22100

$V_{OUT} = (V_S / 3.3) \times [0.25 + (0.028 \times T_A)]$ AD22103

where V_{OUT} is the output voltage, V_S is the power supply, and T_A is the ambient temperature.

So, turning the equations around, the relationship between temperature and output voltage is:

$T_A = (((V_{OUT} \times 5) / V_S) - 1.375) / 0.0225$ AD22100

$T_A = (((V_{OUT} \times 3.3) / V_S) - 0.25) / 0.028$ AD22103

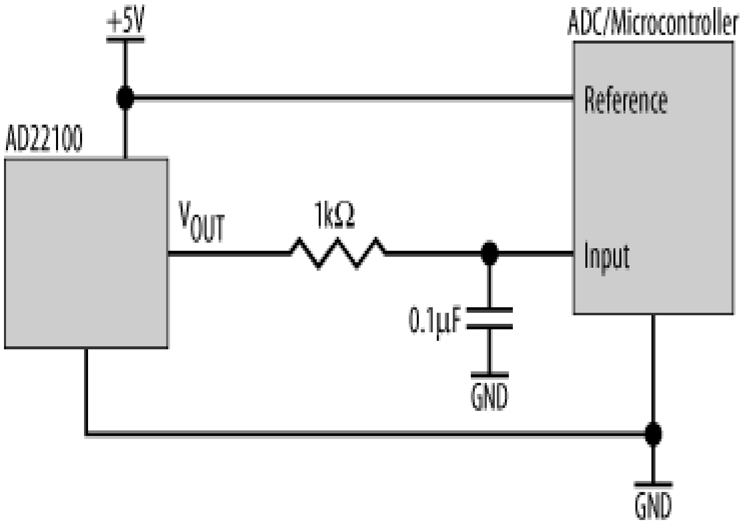
For example, if we were using an AD22100 temperature sensor with a supply voltage of 5 V (V_S = 5 V), then our function becomes simply:

$T_A = (V_{OUT} - 1.375) / 0.0225$

Thus, if we measured an output voltage of 1.94 V, the corresponding temperature would be 25.1ºC.

Interfacing the temperature sensor to an ADC is simple. The output may be directly connected to an input of the ADC. Alternatively, since temperature changes relatively slowly, we can add an RC filter between the sensor and the ADC to remove any noise that may be present in the output (Figure).

Figure. ADD22100/AD22103 with an RC filter

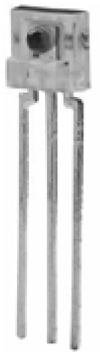


Light Sensor

Now we'll take a look at a light sensor. The obvious use of a light sensor is to monitor natural light levels, and perhaps use the results to control artificial-lighting systems. But combine this sensor with a directional light source (such as a bright LED enclosed in a baffle), and you have a security detector. As long as the sensor can "see" the LED, everything's fine. But when the light is interrupted, you know that someone or something has passed between.

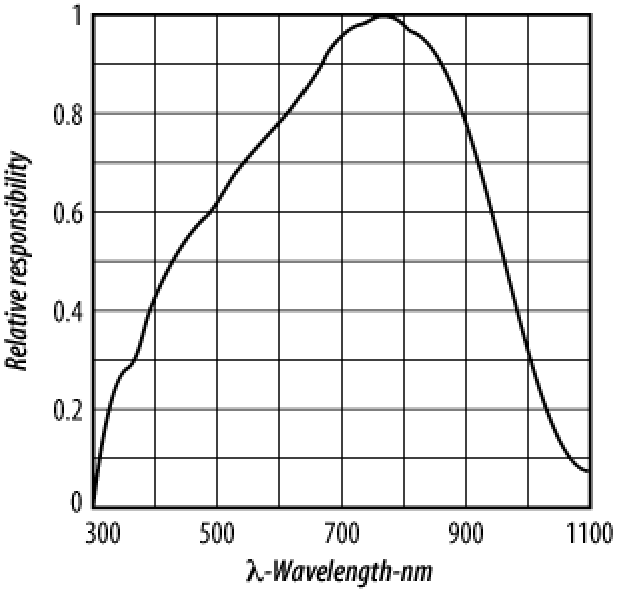
There are lots of commercial light sensors available. We're going to take a look at the TAOS TSL250R sensor. The TSL250R (Figure) consists of a photodiode (a semiconductor that is responsive to light) and an integrated amplifier. Like the temperature sensor we've just seen, the TSL250R just needs power and ground, and it will give you an analog voltage output that is proportional to incident light.

Figure . TAOS TSL250R light sensor



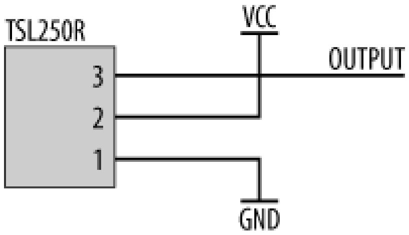
The spectral response for the TSL250R, shown in Figure, ranges from ultraviolet (left) to infrared (right) and peaks in the visible part of the spectrum.

Figure. Spectral response of a TAOS TSL250R



The TSL250R can operate from a supply voltage of between 2.7 V and 5.5 V and typically consumes only 1.1 mA of current. The basic circuit for the TSL250R is very simple (Figure).

Figure. Using the TAOS TSL250R



The maximum output voltage (under full irradiance) for this sensor is just under 4 V, when the part is powered from a 5 V supply. So, if we choose, we can interface this sensor directly to a (5 V referenced) ADC without any additional amplification. Now, because the output does not span the full scale of the ADC's range, we lose a small amount of resolution. For an 8-bit ADC, a 4 V input corresponds to `0xCC`, and so our range of values for this sensor go from `0x00` to `0xCC`. Depending on your application, this may not be a problem. For example, if you are interested only in detecting the difference between light and darkness, or when a given low-light threshold is crossed, this will work fine.

Amplifying the Light Sensor

If you do want to sample the full range of the sensor, you need to amplify the sensor's output. Since the sensor's maximum output is 4 V and the reference of the ADC is 5 V, the gain of the amplifier must be 1.25.

A good general-purpose op amp is the AD623 by Analog Devices. It has rail-to-rail operation, can run from a single supply voltage, requires very little current, and is exceptionally easy to use. Analog Devices has done a lot of the hard work already, and the AD623 requires only a single external resistor to set the gain. The value of the resistor is calculated using the relation:

R_G = 100 kΩ / (Gain - 1)

So, for our required gain of 1.25, we need a resistance of:

R_G = 100 kΩ / (1.25 - 1)

= 100 kΩ / 0.25

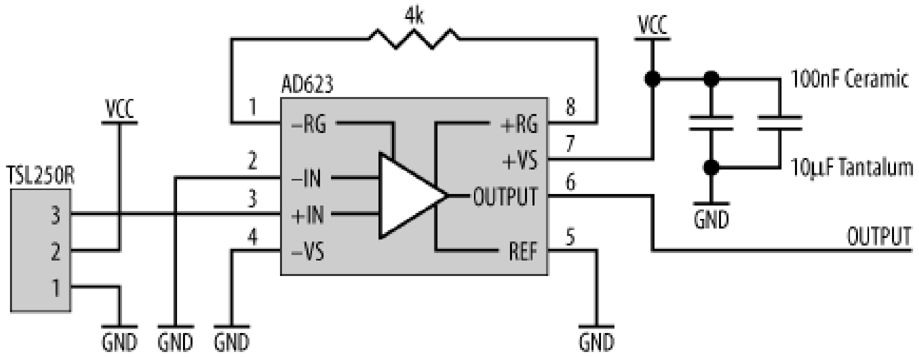
= 4 kΩ

The resistor should have a tolerance (accuracy) of 1% or better. Standard off-the-shelf resistors are normally 5% and just aren't accurate enough.

The circuit with the TSL250R interfaced to the AD623 is shown in Figure.

The output of the TSL250R sensor (pin 3) is connected to the noninverting input of the AD623 op amp (pin 3), while the inverting input is tied to ground. The gain resistor is connected between pins 1 and 8. The negative power supply, -VS, is connected to ground for single-supply operation. The positive power supply, +VS, is

Figure. Amplifying the output of the TSL250R light sensor



connected to VCC and is decoupled to ground using two capacitors. The op amp's reference input (REF) is also tied to ground. The output of the op amp at pin 6 is then connected directly to the analog input of an ADC.

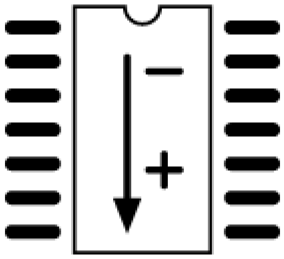
Accelerometer

Now we're going to take a look at an interesting sensor. Analog Devices makes some really nice accelerometers, and we'll learn how to interface an ADXL150 to an embedded system. You can use an accelerometer for a number of applications, not just for measuring linear acceleration of vehicles. The ADXL150 is a single-axis (one-dimensional) accelerometer with a resolution of 10 m g and a full-scale range of ±50 g. For dual-axis (two-dimensional) sensing, choose the ADXL250.

Such a fine resolution means you can use this sensor to measure gentle vibrations and shifts. You could use it in a seismometer for geophysical applications or to measure vibrations or ground shift in mines, tunnels, or at building sites. You could use it to monitor motion and, by placing three accelerometers orthogonally, get an accurate 3-D motion recorder. The same setup could also be used as a digital carpenter's spirit level by sensing the direction of the Earth's gravitational field. Perhaps you might use it to monitor violent physical shock, such as crash-test measurements. Ever moved to a new house only to discover that Granny's fine crystal glassware was smashed by the movers? Place one of these (along with an appropriate small datalogger) into the packing boxes, and you'll be able to prove just how rough the gorillas from the moving company were. As you can see, this chip has lots of applications.

The axis of sensitivity for the ADXL150 runs along the chip's length from top to bottom (Figure). It is important when using this device that it be securely mounted to the circuit board. Rather than just relying on solder, also use strong glue under the chip to bind it to the circuit board.

Figure. Axis of sensitivity

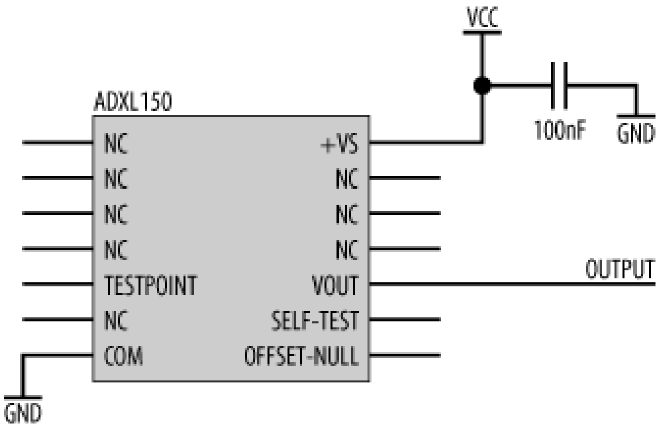


The ADXL150 requires no external components (save for power-supply decoupling) and is a completely self-contained unit, incorporating not only the sensor, but also signal conditioning and amplification. Its output can be interfaced directly to an ADC. The schematic for using the ADXL150 is shown in Figure. Most of the pins are No Connection (NC) and can be ignored, as can the TESTPOINT and SELF-TEST pins. The TESTPOINT pin is used during manufacture only and should be left alone.

The ADXL150 operates off a power supply in the range of 4 V to 6 V. However, for ideal operation, the supply should be exactly 5.0 V. The closer to 5 V the supply is, the more accurate your measurements of acceleration will be. The output voltage is proportional to both acceleration and power supply (V_S) and is given by the relation:

$V_{OUT} = V_S/2$ - (sensitivity * $V_S/5$ * acceleration)

Figure. Using the ADXL150



The sensitivity value varies from device to device and is in the range 33.0 to 43.0, with a nominal value of 38.0. The standard sensitivity value gives a range of ± 50 g; however, the sensitivity may be doubled (giving a range of ± 25 g) by connecting the output to the OFFSET-NULL pin.

The SELF-TEST pin is used for verifying the correct operation of both the internal mechanics of the sensor, as well as its signal conditioning and amplification electronics. Applying logic 1 to this input pin artificially imposes a force on the sensor, and thus the sensor can be shown to be operating correctly.

Pressure Sensors

Now let's take a look at pressure sensors. The most obvious use of these sensors is in measuring air pressure for weather monitoring and prediction. But pressure sensors are also used in cars to measure manifold pressure, in washing machines to measure water levels, and in biomedical applications such as measuring blood pressure. Another application of pressure sensors is to measure altitude, since air pressure changes with height above sea level. Ocean depth can similarly be measured.

When using pressure sensors, the substance you are measuring can adversely affect the device. Remember that these are sensitive electronic components, and fluids or corrosive gases can destroy them. So unless you're measuring clean, dry air, you'll need to provide some degree of environmental protection for your sensor. Just how you do that really depends on what the application is, what environmental conditions you must protect against, and how far your budget stretches.

Pressure sensors work by measuring the deflection of a diaphragm separating two chambers. One chamber is exposed to the pressure that is being measured, while the other chamber holds a reference pressure. The pressure difference between the two chambers causes the diaphragm to deflect, and this deflection is converted into a voltage that is proportional to the pressure difference. Pressure sensors come in three types: absolute, differential, and gauge.

In an absolute pressure sensor, the reference chamber is sealed. Pressure readings are referenced to an absolute pressure, hence the name. Absolute sensors normally have the reference chamber pressure at vacuum, or at 1 atmosphere.

In a differential sensor, the reference chamber is not sealed, and an external pressure reference may be applied. Differential sensors are used to measure the relative pressures between two gases or two liquids. A differential sensor may be treated as an absolute sensor by providing it with a sealed and stable reference pressure.

A gauge sensor is a variation of the differential pressure sensor, where the reference pressure chamber is open to the atmosphere. Thus, the measured pressure is referenced to atmospheric pressure, and variations of atmospheric pressure (such as those caused by weather conditions or altitude) are taken into account. One interesting use of a gauge pressure sensor is to measure airspeed. If the measuring chamber is exposed to the oncoming airflow (caused by the aircraft's motion), and the reference chamber is exposed to the air but sheltered from the effects of the airflow, then the difference in pressure can be used to calculate the airspeed of the aircraft.

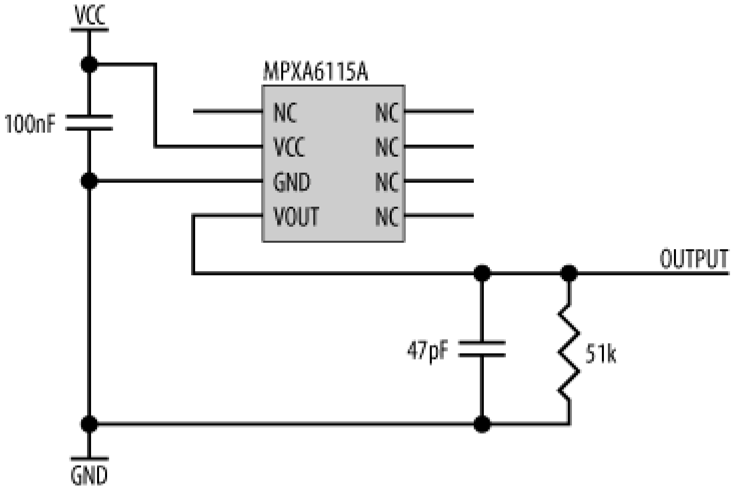
So, with all that in mind, let's take a look at some pressure sensors. The first sensor is a Freescale (formerly Motorola) MPXA6115A absolute pressure sensor (Figure). It operates from a 5 V supply and will give an output voltage of between 0.2 V and 4.8 V, proportional to pressures of 15 kPa to 115 kPa. (Pa is short for Pascals, which is a unit of pressure.) Unlike most other pressure sensors, which require external signal conditioning, temperature compensation, and signal amplification, the MPXA6115A integrates it all in one neat little package. It comes in an 8-pin chip package, with or without snorkel!

The NC pins are no-connection and should be left unwired. The only additional components required are a decoupling capacitor on the power supply and a resistor and capacitor in parallel at the output. The output may be directly connected to an ADC's input.

The second pressure sensor we will look at is also an absolute pressure sensor. But, unusually, rather than producing an analog output, it incorporates an inbuilt ADC. It is interfaced to a microcontroller using SPI and, being digital, it is much less susceptible to noise and interference. The sensor is the KP100, made by Infineon Technologies (http://www.infineon.com) in Munich, Germany.

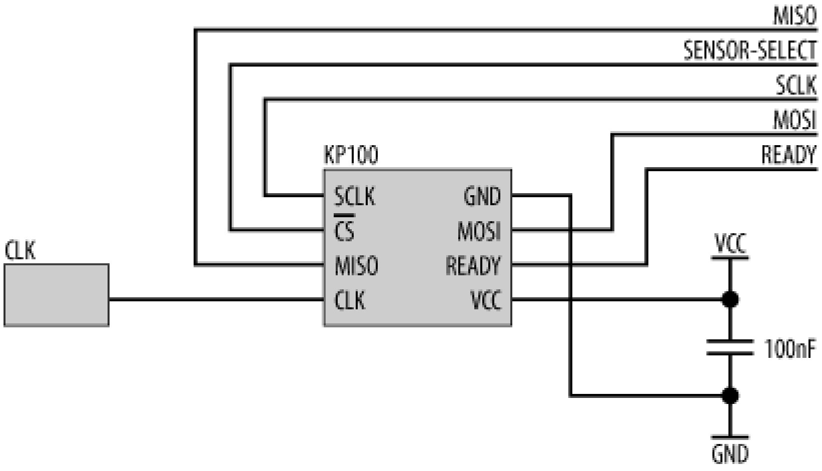
The schematic for a circuit based on the KP100 is shown in Figure. The sensor operates off a 5 V supply, and this is decoupled to ground using a 100 nF capacitor to reduce noise. The sensor has a standard SPI-style interface and is connected to a microcontroller, as with any SPI device. The sensor also provides a READY output, which may be used to interrupt the host processor, or may simply be connected to a spare I/O and read as a digital status flag. The KP100 also requires a separate clock (CLK) input.

Figure. Interfacing the Free scale MPXA6115A pressure sensor



This clock can be either 4 MHz or 8 MHz. If the processor is running at one of these speeds, then the sensor can share the same clock input as the processor. However, if the processor is operating at a different clock frequency,

Figure. KP100 pressure sensor circuit

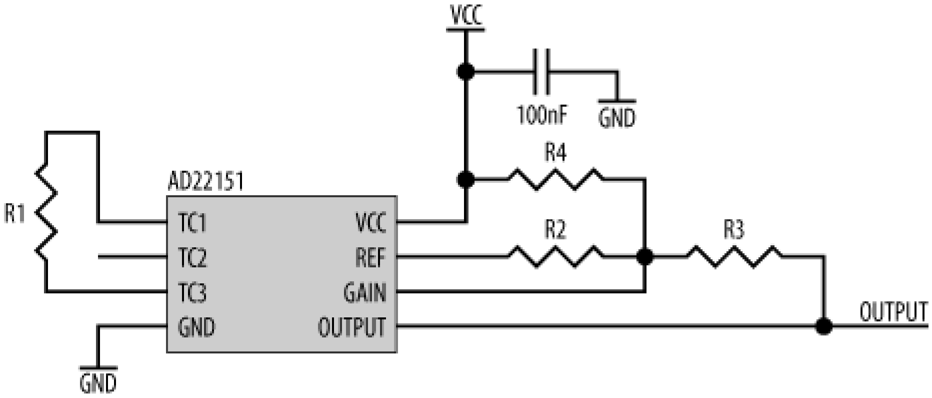


the KP100's clock may be easily generated using a clock module. These 4-pin devices are available in a variety of standard frequencies and require only power and ground to generate a clock output.

Magnetic-Field Sensor

The final sensor we'll look at is the AD22151 magnetic-field sensor by Analog Devices. Its primary use is for position and proximity sensing. A magnetic source is used as a reference point, and the sensor's distance from that source may be easily determined by the measured field strength. The sensor has inbuilt temperature compensation and amplification. The circuit for this sensor is shown in Figure. It's a little bit more complicated than the other sensors we've looked at so far.

Figure. AD22151 magnetic-field sensor circuit



The sensor operates off a 5 V supply, decoupled to ground using a 100 nF capacitor. There are four resistors required for correct operation. R1 is the temperature compensation resistor, which should be connected between pins 1 and 3, or pins 2 and 3, depending on the applied magnetic field. For large external fields, R1 connects pins 1 and 3, as shown in Figure. For smaller fields, connect R1 between pins 2 and 3. The AD22151 datasheet has plots of values for R1 versus required compensation levels. Check with the manufacturer of your magnetic source as to the required compensation value, and use this in conjunction with the datasheet to determine R1. R2 and R3 set the signal gain of the internal amplifier, and R4 provides a voltage offset. The datasheet for the sensor contains equations and technical data for computing values of these resistors, based on your specific needs. The output of the sensor circuit may be connected directly to an ADC input for sampling.

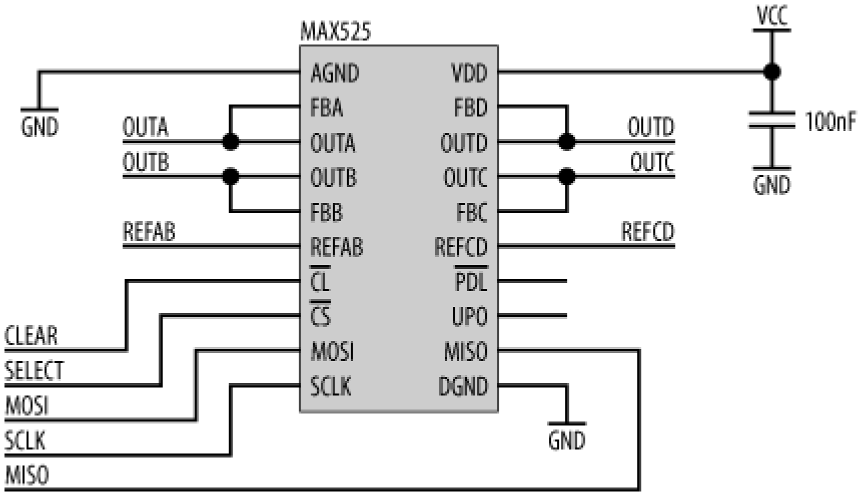
Digital to Analog Conversion

So far, we have looked at how you can sense real-world effects and convert these into digital data. Now let's see how to do the reverse: take digital data and convert it into an analog signal by using a chip known as a Digital-Analog Converter (DAC). We'll also look at how you can produce an analog output using nothing more than a single digital I/O line.

All DACs have a digital input (a microprocessor bus, SPI, or I^{2}C) and will provide you with one or more channels of analog output.

The Maxim MAX525 is an 11-bit DAC that interfaces to a host processor using SPI. It has four channels of analog output and incorporates output amplifiers on-chip. The inverting input of each amplifier is accessible so that you can alter their respective gains. A sample circuit for a MAX525 is shown in Figure.

Figure. MAX525 circuit



The four analog output channels are OUTA, OUTB, OUTC, and OUTD. These are tied directly to their respective feedback inputs (FBA, FBB, FBC, and FBD) for standard unipolar operation. There are two voltage reference inputs, REFAB (for channel A and channel B) and REFCD (for channels C and D). These two reference inputs must be at least 1.4 V or more below VCC at all times. The output voltage for each channel is given by the relation:

V_{OUT} = (V_{REF} * code / 4096) * gain

where code is the digital value written to that channel. In our sample circuit, the gain is 1. If our reference voltage is set to 3.6 V, the digital value 4095 (0xFFF) generates an output voltage of:

V_{OUT} = (V_{REF} * 4095 / 4096) * gain

= 3.6 * 0.9997 * 1 = 3.59 V

Similarly, the digital value 2048 (0x800) generates an output voltage of:

V_{OUT} = (V_{REF} * 2048 / 4096) * gain

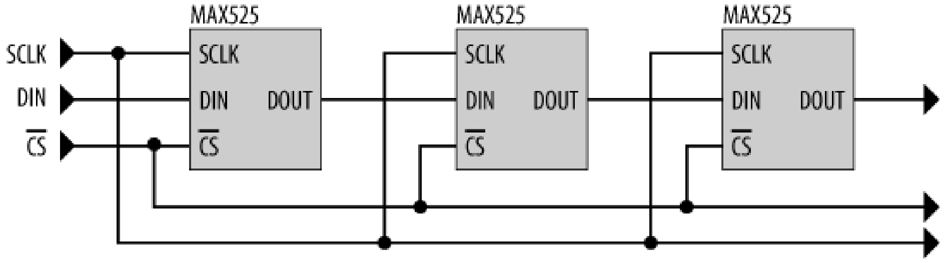
= 3.6 * 0.5 * 1

= 1.8 V

Note the separate analog and digital grounds in the schematic. These should be connected together, but only at a single point close to the DAC.

The MAX525 has a standard SPI connection to a microprocessor. Multiple MAX525s may be daisy-chained together for efficiency (Figure).

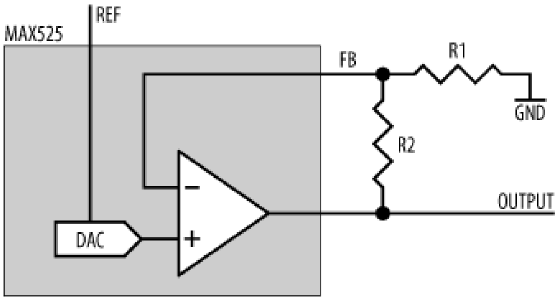
Figure. Daisy-chained MAX525s



The MAX525 also has a input, which, when driven low by an I/O line, sends all outputs to their lowest value. The MAX525 can be put into low-power mode under software control. The input is Power-Down Lockout, and when driven low, it prevents the MAX525 from being shut down. This is important if the outputs are being used to drive a critical circuit or system. You don't want the controlling voltages disappearing by accident. Finally, the good people at Maxim have provided a signal called UPO (User Programmable Output). This is a general-purpose output that can be driven high or low under software control. Use it for whatever purpose you require.

Now, if you wanted a gain other than 1 (non-unity gain), external resistors are required for the output amplifier. The schematic for this (for a single output channel) is shown in Figure.

Figure. Feedback resistors for non-unity gain

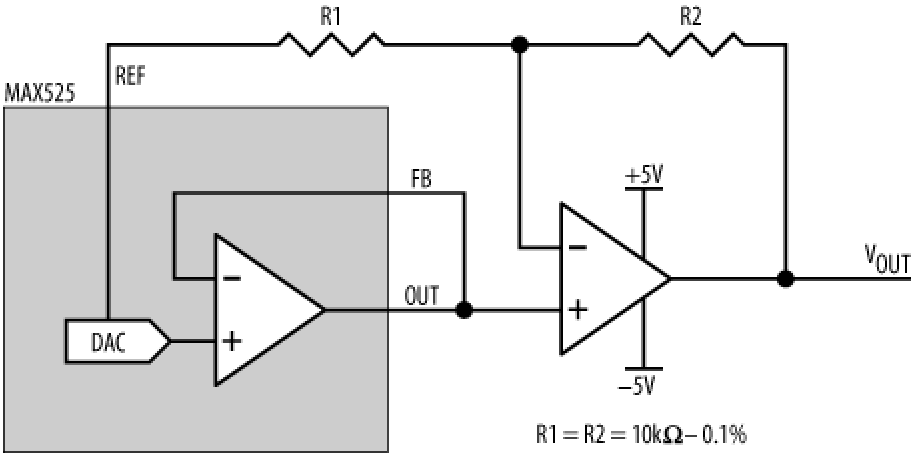


From before, we know that the gain of a noninverting amplifier is given by:

Gain = 1 + R2 / R1

For bipolar output on a given channel, an external amplifier (with bipolar supplies) does the job (Figure).

Figure. Bipolar output

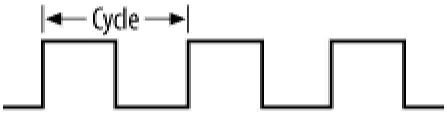


PWM

Using a DAC may seem the obvious way to generate an analog output voltage, but there is another way that uses nothing more than a digital I/O line configured as an output. This technique is known as Pulse Width Modulation (PWM).

Consider the average, garden-variety, square wave shown in Figure.

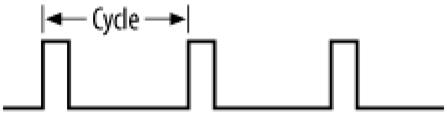
Figure. A ubiquitous square wave



The width of the high is equal to the width of the low, so this wave is said to have a 50% duty cycle. In other words, it is high for exactly half the cycle. Now, if the amplitude of this square wave is 5 V, for example, the average voltage over the cycle is 2.5 V. It is as though we had a constant voltage of 2.5 V.

Now consider the square wave in below Figure .

Figure. 10% duty cycle



This wave has a 10% duty cycle, which means that the average voltage over the cycle is 0.5 V.

A low-pass (averaging) filter on the PWM output will convert the pulses to an analog voltage, proportional to the duty cycle of the PWM signal.

By varying the duty cycle, we can vary the analog voltage. Hey, presto! We have digital-to-analog conversion without a DAC. That's the basic idea behind PWM.

Motor Control

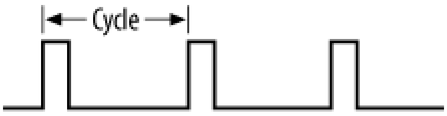
One of the fun things you can do with an embedded computer is get it to actually move something, whether it be an external system or the embedded computer itself. Motion implies motor, and this section will look at how you interface an embedded computer to an electric motor. The possible applications could range from controlling locomotives on your model railroad layout to experiments in robotics, and anything in between. A note of caution, though: if your hardware and software are responsible for moving a physical object, then a bug can easily cause physical damage too. So be careful.

Let's say that we have an electric motor than operates from a 12 V supply. Applying 12 V across the motor will cause it to turn at full speed. Similarly, by applying 6 V, we can get the motor spinning at half speed. By varying the applied voltage, we can vary the speed at which the motor turns.

There are several ways to generate this voltage to drive the electric motor. The most obvious may seem to be to use a DAC to generate an analog output voltage and then use an amplifier to boost the signal to the voltage and current required to turn the motor. The speed of the motor is proportional to the output voltage. However, this technique has a major drawback. For very low-speed operation, the required output voltage may be too low to actually cause the motor to turn.

A better way is to use PWM. Consider the PWM signal in Figure, with an amplitude of 12 V.

Figure . PWM signal with a 10% duty cycle



With a 10% duty cycle, the effective analog output voltage of this PWM signal is 1.2 V. Now, by itself, 1.2 V may not be enough to turn a motor. But we're not using 1.2 V; we're actually pulsing the motor with 12 V, its maximum drive voltage. The duration of the pulses gives the equivalent speed of a motor voltage of 1.2 V. However, by using a full 12 V amplitude, we're ensuring that the motor will turn. This is the advantage of PWM. To control speed, we vary the width of the pulse and not the amplitude.

Using PWM, you can get very slow motor speeds and very fine control. The pulses can cause a jerkiness to the motor if the overall frequency is low, but by choosing a high frequency, the jerkiness is averaged out.

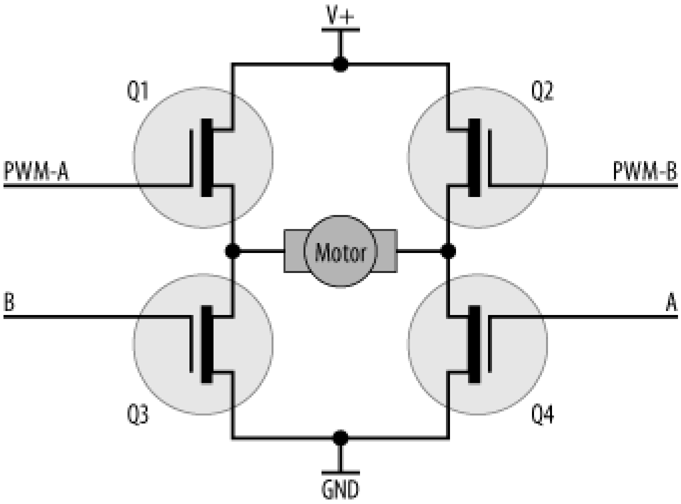
Many microcontrollers have internal, software-programmable PWM modules that make generating PWM signals easy. Even if a processor does not have a PWM module, you can still generate PWM under software control simply by using a digital output line.

Let's now take a look at how you would interface a processor to an electric motor using PWM. Due to the voltages and currents required by motors, you cannot simply hang a motor off the pins of a processor and expect it to work. You need an interface circuit that will take your logic-level, PWM output and use this to switch much higher voltages and currents.

Figure shows a conceptual model (in a crude and simplified form) of such an interface circuit for driving a small electric motor. This type of circuit is known as an H-bridge.

It's not as confusing as it first looks. Don't be too worried about the transistors (Q1-Q4) in the circuit. They simply act as switches. Our motor operates from a supply voltage, V+. Apply V+ with one polarity, and the motor turns in the forward direction. Reverse the polarity, and the motor reverses too. To drive the circuit, we use four outputs from the processor: two PWMs (which I've called PWM-A and PWM-B) and two general I/O lines (which I've called A and B). Initially, all outputs are low, everything is turned off, and the motor is stationary.

Figure . Motor drive circuit using an H-bridge



If we send A high, the transistor Q4 turns on and connects the right "side" of the motor to ground. If we then send PWM-A high, the transistor Q1 turns on. Thus, the left "side" of the motor is connected to V+, and the motor spins. By generating a PWM signal on PWM-A, we can control the speed of the motor in that direction.

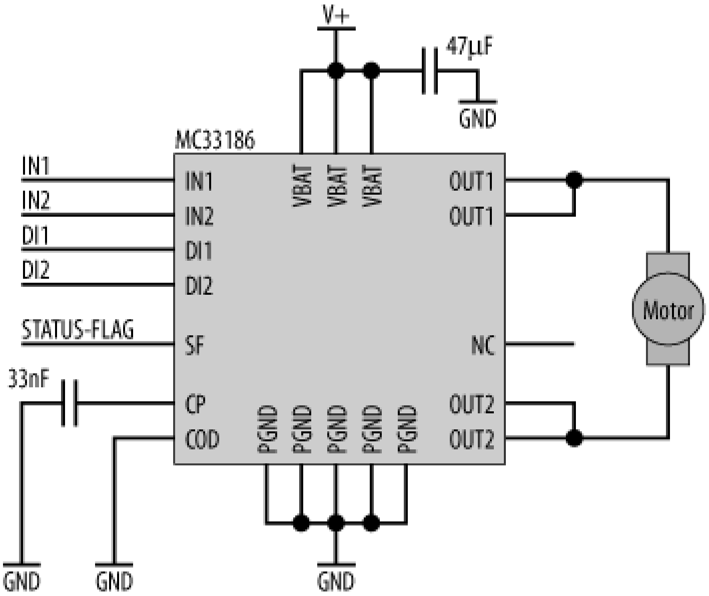
Conversely, by leaving A and PWM-A low and setting B and PWM-B high, transistors Q2 and Q3 turn on, and the motor spins in the reverse direction. By generating a PWM signal on PWM-B, we can control the speed in the reverse direction.

Care must be taken in your software. If both Q1 and Q3 are turned on, or both Q2 and Q4 are turned on, then you effectively connect V+ to ground, with very little resistance in between! The results would be spectacular and short-lived! A proper H-bridge circuit normally contains protection to prevent such a state from occurring.

The actual implementation of an H-bridge is a little more complicated and requires additional components such as protection diodes and so forth. Now, while you could design such an H-bridge circuit using discrete components, there is an easier way

Let's look at a sample H-bridge, the Freescale MC33186. This chip is more sophisticated than the simple H-bridge used to explain the concept. It provides more functionality, yet is easier to control. This chip can operate from a supply voltage (V+) of between 5 V and 28 V and can switch continuous currents as high as 5 A, yet it has logic inputs that are compatible with TTL levels. It has inbuilt short-circuit and over-current protection. Figure shows an MC33186 circuit.

Figure. MC33186 motor drive circuit



The chip has three power-supply inputs, V_{BAT}, all of which must be connected to the supply voltage, V+. The power-supply input needs to be decoupled using a 47 µF capacitor. The internal charge pump also needs a decoupling capacitor. The pin, CP, provides access to the charge pump and is connected to a 33 nF capacitor. The chip also has five ground pins, which, similarly, must all be connected to ground.

OUT1 and OUT2 are the pins that directly drive the motor. There are two of each, so that the high output currents are not traveling through a single pin.

IN1 and IN2 control both the motor's speed and direction. DI1 and DI2 serve to disable the MC33186. These four control signals may be driven by a microcontroller's I/O lines. For normal operation, DI1 is low and DI2 is high. Sending either DI1 high or DI2 low will disable the MC33186 and stop the motor. Table shows how IN1, IN2, DI1, and DI2 affect the motor's operation.

MC33186 states of operation

DI1	DI2	IN1	IN2	OUT1	OUT2	Motor
Low	High	High	Low	V+	Ground	Forward
Low	High	Low	High	Ground	V+	Reverse
Low	High	Low	Low	Ground	Ground	Free-wheeling
Low	High	High	High	V+	V+	Free-wheeling
High	Don't care	Don't care	Don't care	High impedance	High impedance	Disabled
Don't care	Low	Don't care	Don't care	High impedance	High impedance	Disabled

If we want the motor to run forward, we generate a PWM signal on IN1 and leave IN2 low. If we want to run the motor backward, we leave IN1 low and place a PWM signal on IN2. The duty cycle of the PWM signal determines the motor's speed.

If IN1 and IN2 are in the same state, then there's no voltage difference applied across the motor's terminals, and so the motor is not driven.

Pin 2 of the MC33186, SF, is an output status flag. If the MC33186 is operating correctly, SF is high. If there is a fault, SF is driven low. SF may therefore be used as an interrupt to alert the host processor of a problem.

The input COD determines how the chip functions during a fault. If COD is left unconnected or is connected to ground, a change on either input DI1 or DI2 will reset the fault condition. If COD is connected to VCC (that's +5 V, not necessarily V+), then DI1 and DI2 are disabled. The fault condition can be reset only by a change on IN1 or IN2.

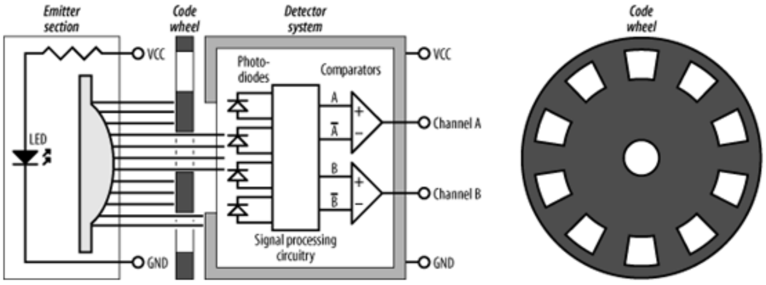
Using an integrated H-bridge circuit, such as the MC33186, greatly simplifies interfacing your embedded system to motors.

Sensing Motor Speed

In a control application, it is very useful to be able to sense a motor's speed. The physical system (load) that the motor is driving will affect the motor's rotation. If the motor must move a heavy load, then its actual speed of rotation may be less than the intended speed. In such situations, it is useful to measure the actual speed so that the embedded control system can compensate.

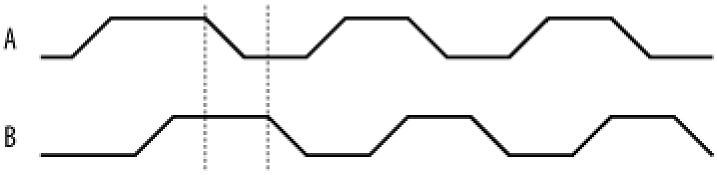
The easiest way to measure a motor's rotational speed is to use an optical encoder module, such as the Agilent HEDS-9000 or a similar device. The encoder consists of a light source (LED) and an array of photo-detectors, separated from each other by a slotted disc known as a code wheel(Figure). The disc is mounted on the rotating motor shaft. Each time a slot passes between the LED and a detector, the detector receives a flash of light and generates an electrical pulse. The rate at which the pulses are generated corresponds directly to the rotational speed of the motor. The resolution of the code wheel is known as its counts per revolution (CPR) value. The HEDS series of encoders are available with CPRs ranging from 96 all the way up to 2,048.

Figure. Block diagram of a HEDS-9000 optical encoder and a code wheel



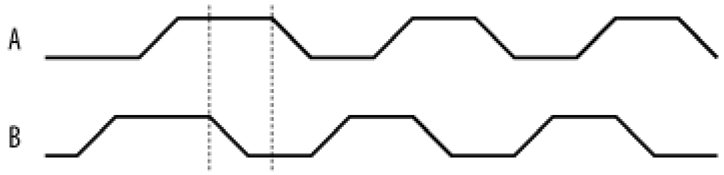
The HEDS-9000 optical encoder operates from a 5 V supply and has two outputs, A and B. These outputs are derived from two adjacent optical sensors. If the code wheel is rotating in one direction, output A will trigger before output B (Figure).

Figure . Output waveforms for the optical encoder



If the wheel is rotating in the opposite direction, then B will trigger before A (Figure).

Figure. Output waveforms for the optical encoder, with rotation in the opposite direction



The rate at which the pulses arrive gives the motor's speed, and the order in which they arrive shows the direction. This is known as quadrature encoding.

Most microcontrollers have timer/counter inputs that can measure external trigger events such as these. Under software control, you can use the timers to monitor these quadrature signals. However, Agilent makes a series of devices known as quadrature counters: the 11-bit HCTL-2000, the 16-bit HCTL-2016, and the 16-bit, cascadable HCTL-2020. These chips provide a bus-based interface to a processor and convert quadrature signals into a binary number representing motor position. A 16-bit position counter is capable of measuring 32,767 increments in either direction, which corresponds to approximately 15 turns of a 2,048 CPR encoder. To determine the present motor speed or position, the processor simply reads from the quadrature counter as though it were just another memory location. Quadrature counters also have noise filters on their inputs and so provide a more reliable and accurate way of determining motor position.

The schematics showing an optical encoder and quadrature counter are shown in Figures. The optical encoder is placed on a separate, small PCB so that it may be easily mounted next to the motor's shaft. The quadrature counter is located on the embedded computer's PCB. IDC headers (J1 and J2) and a ribbon cable connect the two circuit boards.

Figure. Optical encoder circuit

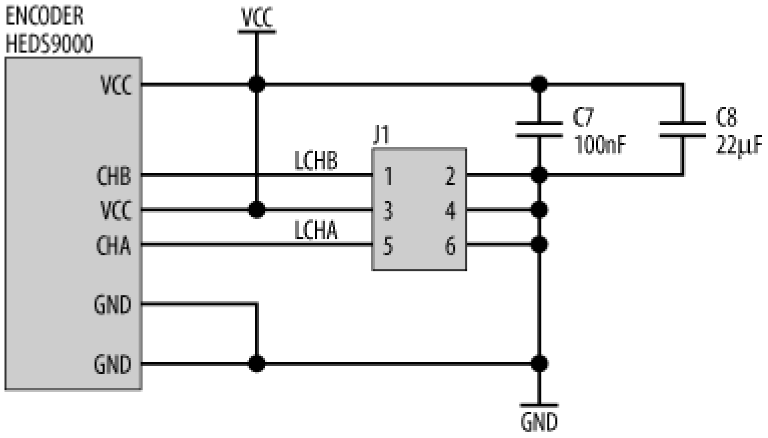
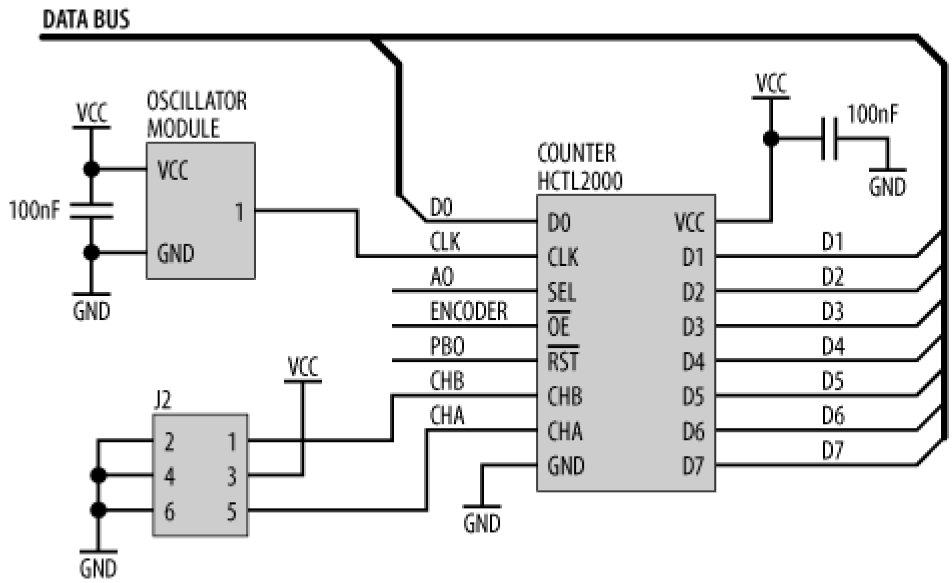


Figure. Quadrature counter circuit



The quadrature counter requires a 14 MHz clock. This is easily provided by an oscillator module. CHA and CHB are the quadrature inputs from the encoder. The counter has a reset input, , which clears the counter. Asserting zeros the quadrature counter and indicates that the motor is in the "home" position. This input is driven by a digital output of the microcontroller so that the counter can be reset under software control.

D0 to D7 are the data buses through which the processor reads the current position. Since the counters are either 12 bits or 16 bits, two reads are necessary to retrieve the value through the 8-bit bus. The counter therefore occupies two locations in memory, and the SEL input is used to select which byte is being read. If SEL is low, then the higher-order bits are read. If SEL is high, then the lower-order bits are read. To make these two bytes appear in adjacent memory locations, the processor's address line, A0, is used to drive SEL. Thus, the least significant address of the two selects the upper eight bits, while the next address selects the lower eight bits.

Now, the counter does not have a chip select as such. Since it is a read-only device, the counter's output enable, , functions as a combined chip select and output enable. Therefore, this input is driven by the output of the address decoder that corresponds to the region of the address space to which the counter is mapped. When the processor reads from that address range, is asserted and the counter responds with data. Note that if the processor attempted to write to the counter, the counter would be selected and would respond with data. Therefore, both the processor and the counter would be attempting to drive data onto the data bus. This could potentially damage both chips. Now, with careful coding this would not be a problem. However, a crashing program may inadvertently cause this situation to arise. To prevent this, a better solution is to include the processor's read strobe as part of the address decode for this particular device. In other words, the counter is selected if (and only if) both the address is correct and the processor is performing a read. If the processor is performing a write to the counter's address, the counter is not selected and the access is ignored.

Switching Big Loads

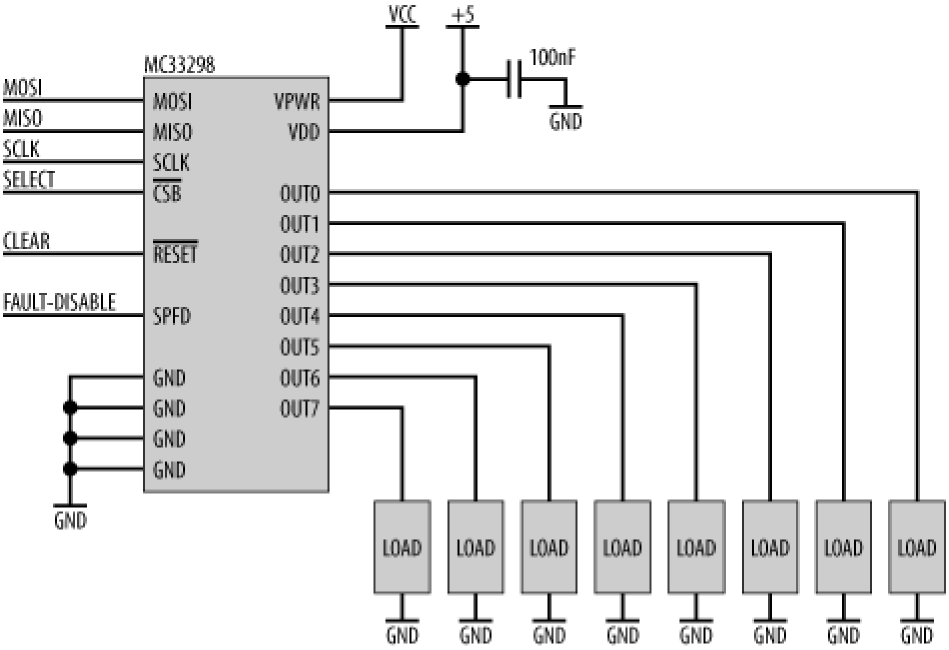
We've already seen how to use an H-bridge chip to switch relatively large voltages (and the corresponding big currents) needed to drive electric motors. There are many other cases where you want to turn large voltages on or off, and, in this section, you'll learn an easy way of doing just that.

The Freescale MC33298 is a chip that is controlled by a microprocessor using SPI that can switch eight power sources on or off. This chip can handle voltages between 5 V and 26.5 V, with currents as large as 6 Amps. If you need to turn electrical systems on or off, this chip is for you. Its primary use is for industrial and automotive applications, controlling power to subsystems such as heaters, small air-conditioning units, moderate-voltage light bulbs, small pumps, and so on. Obviously, it won't handle the high AC voltages that come out of your wall socket, so don't use it for switching power to your home appliances!

The basic schematic for the circuit is shown in Figure.

The MC33298 has two power-supply pins. VDD is a 5 V supply and powers the chip's internal digital logic. It's decoupled to ground using a 100 nF capacitor. V_{PWR} is the supply voltage for the external subsystems (represented in the figure by each "LOAD" rectangle) and can range from 5 V to 26.5 V. There are eight switch outputs, labeled OUT0 through OUT7. When a given switch is activated, the corresponding output is connected to the V_{PWR} supply, thereby turning on that subsystem. The MC33298 has short-circuit detection and shutdown (with automatic retry), over-voltage detection and shutdown, current limiting on the outputs, output clamping during inductive switching, and thermal shutdown if the device is dissipating too much power. Higher currents may be switched by tying two or more outputs together so that the current is shared by more than one pin. By tying all outputs together, currents as high as 48 A may be switched, limited only by the total power dissipation and corresponding thermal shutdown limit.

Figure. MC33298 circuit



The chip has a standard SPI port, allowing it to be interfaced to, and therefore controlled by, most microprocessors. The SPI signals MOSI, MISO, and SCLK are connected directly to a processor's SPI pins. The chip's select input, CSB, is controlled by a digital output of the processor and is used to select the device during a SPI transfer. The device may be reset and all outputs turned off by asserting its RESET input. Again, this too can be driven by a digital output of the processor so that the chip may be turned off under software control. The MC33298 supports SPI daisy chaining, so multiple devices may be coupled together.

The SPFD pin is Short Fault Protect Disable. Sending this pin high allows the internal over-current detection circuitry to be disabled. When switching some loads, such as light bulbs, there is a very high current for a short period of time. This would normally cause the MC33298 to register an over-current fault and shut off that output. The SPFD pin allows this protection to be overridden so that such loads may be controlled. Even though the over-current protection is bypassed, the MC33298 is still protected. If the high current lasts long enough, the chip's thermal shutdown circuit will kick in, thereby preventing damage. SPFD may be driven by a processor digital output, and should be used with caution! For normal operation (with over-current protection on), this pin should be low.

Now we've finished looking at I/O options for our embedded computers. In the next chapter, we'll look at some processors and see how to design complete embedded systems.

Summary

- An amplifier is a circuit that increases (or decreases) a given input voltage to produce an output voltage.
- A differential amplifier multiplies the difference between two input signals and is used to amplify small signals that may be subject to noise.
- A device that converts an analog input voltage to a digital number is known as an Analog to Digital Converter, or simply and more commonly as an ADC.
- Flash ADCs (also known as parallel ADCs) use a bank of comparators to compare the input voltage to a range of reference voltages.
- The rate at which a signal is sampled can have a dramatic effect on the quantized result and therefore can also affect the way in which software interprets that result.
- A good general-purpose ADC for sensor applications is the Maxim MAX1245. It has eight channels of analog input and can sample at 100,000 samples per second, with a resolution of 12 bits.
- The AD22100 and AD22103 temperature sensors, by Analog Devices, are very easy to use. They are 3-pin devices, requiring only power (V_S) and ground to give you a voltage output that is proportional to temperature.
- The Maxim MAX525 is an 11-bit DAC that interfaces to a host processor using SPI.

Questions

- What is amplifier?
- Differences between amplifier and differential amplifier?
- Write a brief notes on ADC?
- How a stepper motor is controlled through PWM?
- Write brief notes on following sensors
  - a) Temparature Sensors.
  - b) Pressure Sensors.
  - c) Magnetic Field Sensors.

d) Accelerometer.

e) Light sensors

References

- C Programming for Embedded Systems – KIRK ZURELL
- Design with 8051- FRONTLINE ELECTRONICS
- Embedded Controller Hardware Design - Ken Arnold
- Embedded Software The Works – colin walls
- Embedded Systems Firmware Demystified - Ed Sutter
- Embedded_Controller_Hardware_Design – KEN ARNOLD
- Programming Embedded Systems in C and C++ - Michael Barr
- The Art of Designing Embedded Systems - Jack G. Ganssle

10. Networks

Objective:

In this chapter, we'll look at connecting your embedded computer to the real world by adding a Local Area Network (LAN) interface. There is a wide variety of networks employed—some very common, some not so common. We'll take a look at CAN and Ethernet, the two most common networks. CAN is a network for industrial applications, where a conventional network just won't do. CAN is suited to electrically noisy and harsh conditions and is the network of choice in electrically severe environments. Ethernet is the intranet network that connects the world's desktop computers, as well as a host of other devices such as routers, gateways, printers, and other peripherals.

Controller Area Network (CAN)

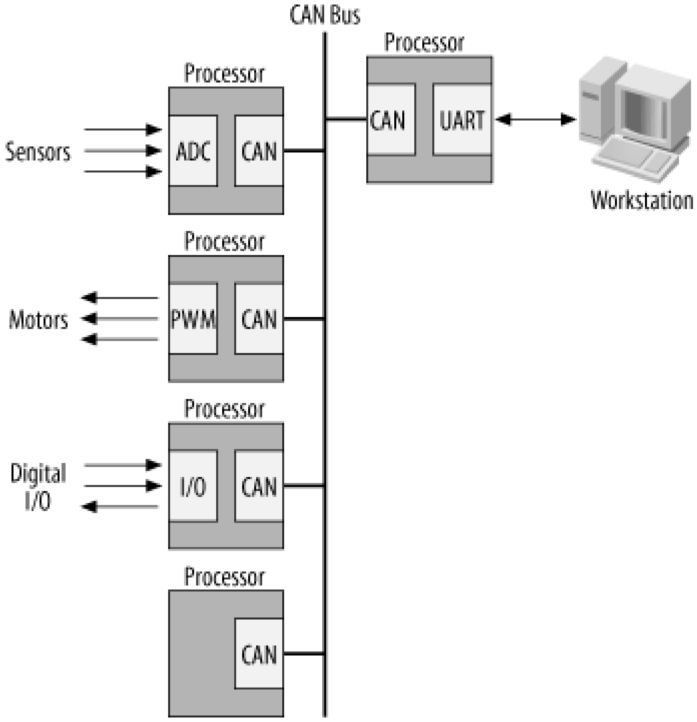
Through the late 70s and 80s, the complexity of automotive electronics grew considerably, with engine-management systems, ABS braking, active suspension, electronic transmissions, automated lighting, air-conditioning, security, and central locking. Each of these systems does not exist in isolation but is part of an integrated whole. A considerable amount of information exchange is required, and, therefore, some means of system interconnection must be provided. The conventional method was point-to-point wiring, which provided discrete interconnection between each subsystem. This methodology was a natural evolution from the simple electrics of earlier cars, but as automotive complexity grew, such a scheme proved vastly inadequate. Each car could have several kilometers worth of wiring and dozens of connectors. Such complex wiring systems added greatly to the cost of producing a car, added unnecessary weight, reduced reliability, and made servicing a nightmare.

The obvious solution was to replace complexity with simplicity and implement intersystem communication using a low-cost digital network. The automotive electrical environment is very noisy. With electric motors, ignition systems, RF emissions, and so on, the 12 V supply to automotive electronics can have ± 400 V transients. The required communication network must therefore be able to cope with this noise and work reliably. The network must provide high-noise immunity and error detection and handling, with retransmission of failed packets. Thus was born the Controller Area Network, more commonly known as CAN, implementing real-time communication at up to 1 Mbps, over a 2-wire serial network. CAN specifies only the physical and data-link layers of the ISO-OSI model, with higher layers left to the specific implementation.

Bosch developed CAN in Europe in the late 1980s, originally for use in cars. Because of its robustness, CAN has expanded beyond its automotive origins and can now be found in industrial automation, trains, ship navigation and control systems, medical systems, photocopiers, agricultural machinery, household appliances, office automation, and elevators. CAN is now an international standard under ISO11898 and ISO11519-2.

CAN supports multiple masters on the network, with each master responsible for local sensing and control within the distributed system (Figure).

Figure. CAN distributed system



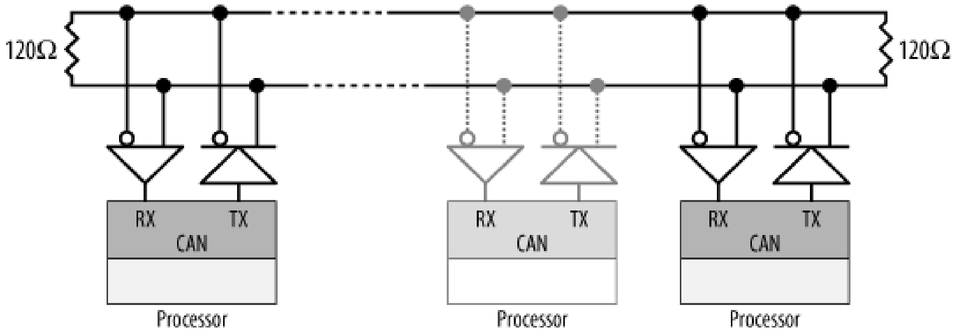
Each CAN packet contains address information and priority as part of the header, and the nodes may connect to the network, or disconnect from the network, without affecting network traffic between other nodes.

The CAN network uses wired-AND logic, with a maximum bus length of 1,000 meters (3,300 feet), and a bus length of 40 meters (133 feet) at maximum data rate over twisted-pair wiring. Each end of the bus requires termination resistors to prevent transmission reflections (Figure).

Many processors intended for use in harsh or electrically noisy industrial applications include a CAN module. A number of Philips microcontrollers include CAN, as do a few PICs. For processors that do not include CAN, CAN interface modules are available. The Microchip MCP2510 provides a CAN module and interfaces to a host processor via SPI. Adding CAN to any embedded system is therefore a simple task.

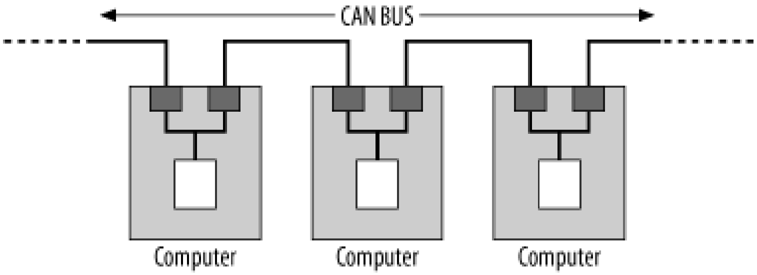
Typically, a microprocessor that supports CAN will include a CAN interface module, which provides most of the functionality. The only additional support required is a CAN interface driver. Philips Semiconductor produces a CAN driver, the PCA82C250T, which makes interfacing to the CAN bus very easy.

Figure. CAN bus



Your embedded computer must also have some way of physically attaching to the bus. The simplest method is simply to bring the bus into the computer system on one connector, tap off it, and then route it out through another connector (Figure).

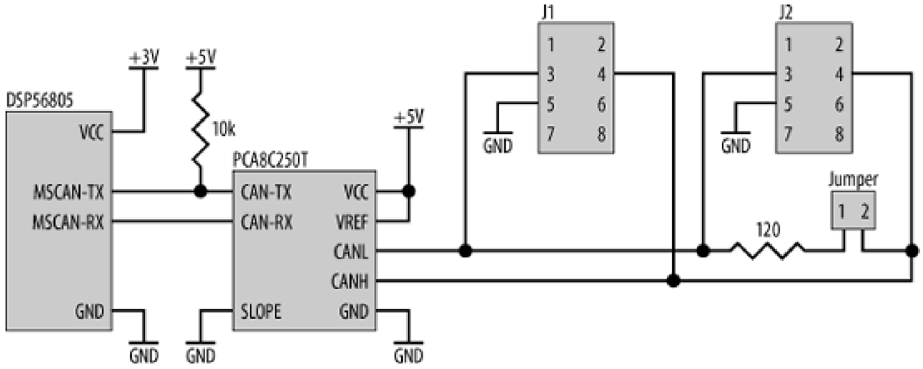
Figure. Tapping into a CAN bus by using two connectors on a PCB



To see how we can use CAN, let's look at the DSP56805 processor. This processor has a CAN network module as part of its suite of onboard peripherals. The schematic for interfacing a processor's CAN module to a CAN bus is shown in Figure.

The DSP56805 has two CAN interface signals, MSCAN-TX and MSCAN-RX, which are the CAN transmitter and receiver, respectively. These are connected to the PCA82C250T, which provides the interface to the CAN bus. Note that the DSP56805 requires a 3.3 V supply, while the PCA82C250T requires a 5 V supply. A pull-up resistor brings the MSCAN-TX output of the processor to the required logic-high level for the PCA82C250T. While CAN requires only two signal lines and ground, the actual connectors have eight pins. Since the CAN bus requires a Termination resistor at each end, we provide a 120 resistor should our computer be placed at the bus end. A jumper allows it to be brought in-circuit or disabled as needed. So, if our computer is at the end of the CAN bus, the jumper is closed and the bus is terminated. If our computer is not an endpoint machine, the jumper is left open and the resistor plays no part. Note that having a termination resistor active (jumper closed) when this computer is not at an endpoint is a good way to ensure an unreliable CAN bus! Resistors should be active at bus ends only.

Figure. CAN interface for a DSP56805 processor



Many implementations of CAN just use standard IDC-type headers for the connectors. However, the actual CAN standard specifies that the connector should be a 9-pin Sub-D connector. The pinouts for this connector are listed in Table.

Pin	Signal/use
1	Reserved
2	CAN_L
3	Ground
4	Reserved
5	Reserved
6	Ground
7	CAN_H

Pin	Signal/use
8	Reserved
9	V+ (optional power source)

Although this is the same type of connector used in some RS-232C implementations (such as the serial ports on PCs), do not connect a CAN bus and RS-232C together. They are not even remotely compatible!

Ethernet

Anyone even remotely involved with computers has heard of Ethernet. Developed at Xerox PARC in the early 1970s, this local-area networking standard has found its way into every possible application and has evolved over time to encompass a number of standards ranging from wireless networks (802.11) to gigabit Ethernet.

In this section, we'll look at how you add a simple Ethernet interface to your embedded computer. We will develop a 10 Mbps interface only, as higher-speed interfaces require special attention to PCB design and EMC issues. So, for the sake of ease and reliability, we'll keep it simple and low-speed.

By adding Ethernet to your embedded system, you gain access to a network and all the possibilities that it brings. You can send data to a host computer at high speed, as well as access printers, file servers, databases, and even the Internet. You can also monitor and control your embedded system from afar, or even have it send you email when it needs attention. Take an AT90S8515 AVR and add an Ethernet interface and some high-capacity flash memory, and you have yourself a simple web server. Add an ADC and some sensors, and your web server becomes a weather station showing current or past conditions to anyone on the Internet. Use a higher-speed processor, several Ethernet ports, and the appropriate software, and you have yourself a simple gateway or firewall. You could even build an Ethernet-to-Ethernet (or serial, parallel-port, or USB) bridge. The possibilities are limited only by your imagination.

There was a time when developing an Ethernet interface was a major exercise. These were complicated circuits, using lots of chips and hundreds of support components. An Ethernet interface could fill a moderate PCB all on its own. Not anymore. In these days of large-scale integration, adding Ethernet to your design is easy, as we will see.

Adding an Ethernet Interface

Crystal Semiconductor, now part of Cirrus Logic (http://www.cirrus.com), produces a single-chip Ethernet controller known as the CS8900A. This chip allows you to add a simple (and low-cost) 10 Mbps Ethernet interface to your embedded system. Full documentation on this chip is available from the Cirrus Logic web site. As the CS8900A is a commonly used Ethernet controller, there is plenty of source code available on the Internet. Just use your favorite search engine to hunt it down. When you design a system based on the CS8900A, you can actually email your design to the engineers at Cirrus Logic, and they will check it out for you, offering advice and pointing out mistakes. The email address for this service is ethernet@crystal.cirrus.com.

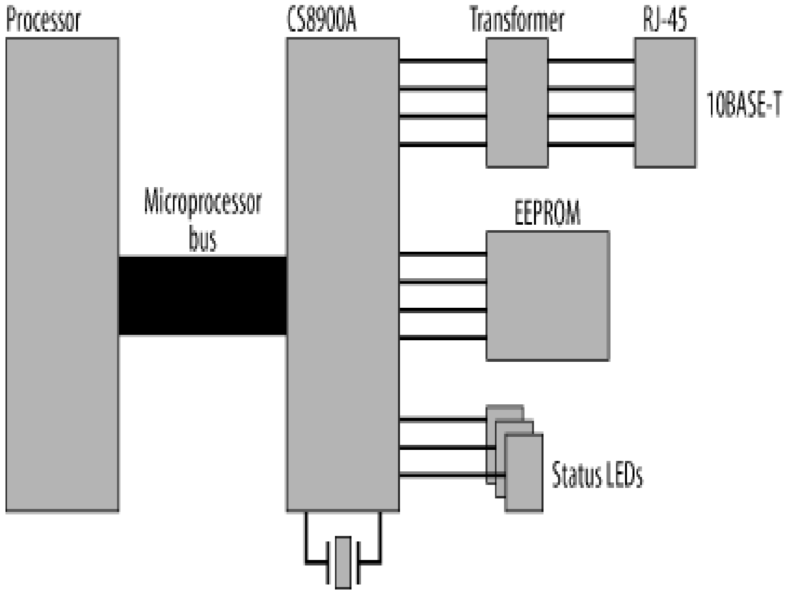
The CS8900A supports 10BASE-2, 10BASE-T, and AUI (Attachment Unit Interface) Ethernet ports. 10BASE-T and 100BASE-T are by far the most common types of Ethernet interface, supporting data rates of 10 Mbps and 100 Mbps, respectively. Your desktop computer's Ethernet interface is most likely a 10/100BASE-T port with an 8-pin RJ-45 connector. (RJ-45 connectors look like, but are not the same as, standard telephone jacks.) The cabling used is UTP (Unshielded Twisted Pair) Category 5 cable, more commonly known simply as CAT5. Just like RS-422, RS-485, USB, and CAN, 10/100BASE-T Ethernet transmits using balanced differential signals. Four wires are used: two for the transmitter pair and two for the receiver pair. One wire of the pair carries a signal voltage of 0 to +2.5 V, while the other wire carries a voltage of 0 to -2.5 V, giving a signal difference of 5 Vpp.

Table shows the pin connections for an RJ-45 connector. The wires within the CAT5 cable are color-coded for easy identification.

Pin	Signal name	Purpose	Wire color
1	TD+	Transmitted data	White/orange
2	TD-	Transmitted data	Orange
3	RD+	Received data	White/green
4	NC	No connection	Blue
5	NC	No connection	White/blue
6	RD-	Received data	Green
7	NC	No connection	White/brown
8	NC	No connection	Brown

A block diagram of a CS8900A implementation is shown in Figure.

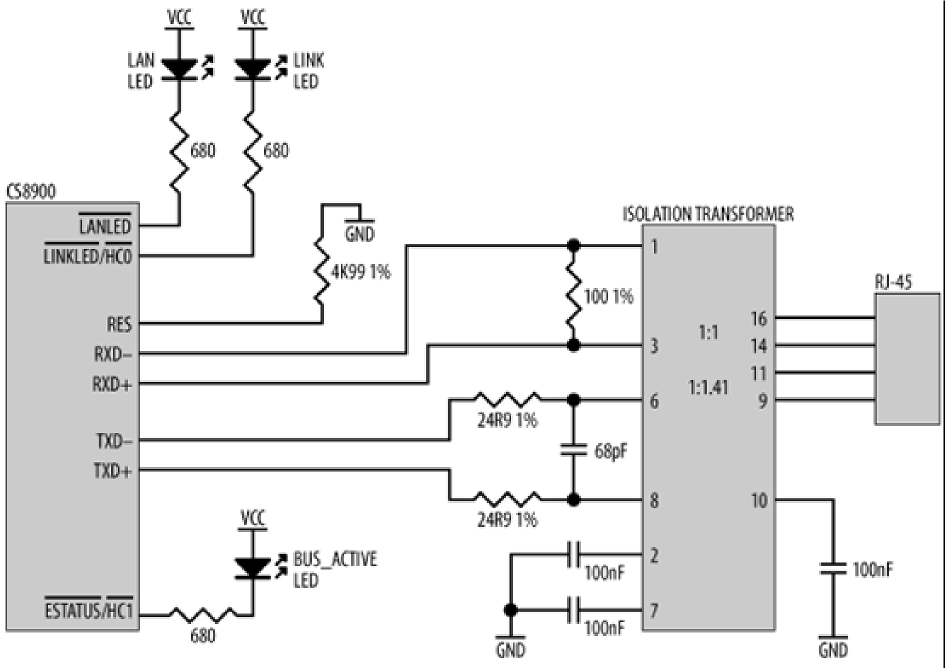
Figure. Block diagram showing a CS8900A implementation



As the CS8900A has 100 pins and several different modes of operation, we won't cover an entire schematic in one hit. Instead, we'll work through each stage of a CS8900A's design, and learn its functionality and use as we go. This discussion will be targeted at small, embedded application. Some of the more complicated aspects of the CS8900A, which are applicable to desktop PCs, will be left alone.

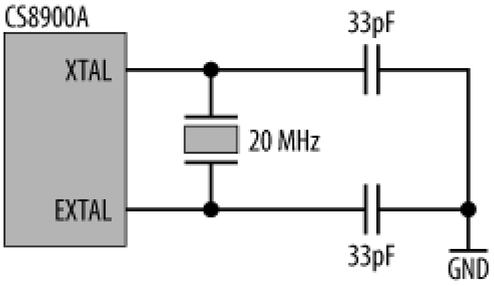
The CS8900A is connected to its 10BASE-T port through an isolation transformer. This transformer must have a winding ratio of 1:1 for the receiver, and a winding ratio n of 1:1.41 for the transmitter, if the CS8900A is used with a 5 V supply. If used with a 3.3 V supply, the transformer's winding ratio for the transmitter must be 1:2.5. There are a number of manufacturers that make isolation transformers (packaged as chips) with these winding ratios, such as Valor, PCA, YCL, and Bel. The transmitter requires series-termination resistors of 24.9, ± 1%. The transmitter differential pair must be decoupled with each other using a 68 pF capacitor. A 100 resistor (± 1%) is required in parallel between the receiver's differential pair. The CS8900A can also directly drive LEDs, indicating Ethernet link status and bus and network activity. The CS8900A has an additional pin (RES) that requires a 4.99 k (± 1%) pull-down resistor. Figure shows the CS8900A connected to a 10BASE-T port.

Figure. 10BASE-T interface



An external 20 MHz crystal provides timing for the CS8900A. The crystal is connected across the XTAL1 and XTAL2 pins, and each pin is bypassed to ground using 33 pF capacitors (Figure).

Figure. Crystal connections for the CS8900A



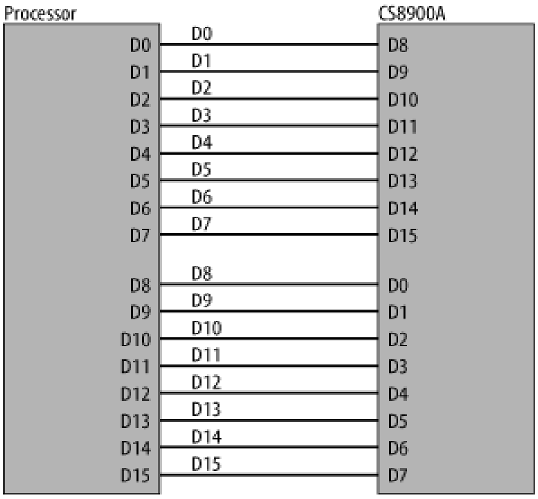
This Ethernet chip supports the 16-bit ISA bus architecture, the expansion bus found in older-model PCs. However, ISA can easily be adapted to work with a range of non-ISA processors. The CS8900A may therefore be implemented in a variety of computer systems without difficulty. The CS8900A also supports operation in 8-bit mode and thus can also be interfaced to microcontrollers with an 8-bit data bus, such as the AT90S8515 AVR. The CS8900A's input SBHE is used to place the chip in 16-bit mode operation after reset. Any activity on SBHE will place the CS8900A in 16-bit mode. The easiest way to ensure that there is activity on this input is simply to connect SBHE to the processor's address line, A0. As soon as the processor begins to use its bus, the activity will place the CS8900A in 16-bit mode. For 8-bit operation, SBHE is tied to ground. When used in 8-bit mode, interrupts are disabled and the CS8900A's status must be polled by software.

Before we look at the processor interface of the CS8900A, there are some important characteristics we need to note. On the CS8900A, RESET is active high. This can catch an unwary designer used to active-low resets. The reason that RESET is active high derives from the fact that this chip was designed principally for use in PCs, as Intel processors also have an active-high reset. The CS8900A's reset may be driven by a digital output of a microcontroller so that it can be reset under software control. Alternatively, in systems where the CS8900A is to have a hardware-generated reset at the same time as the processor, the processor's active-low reset signal must be inverted for the CS8900A. The CS8900A's interrupt outputs (INTRQ0, INTRQ1, INTRQ2, INTRQ3) are also active high, and each must be inverted before connecting to an active-low interrupt input of a microprocessor.

Another consequence of its design for use in Intel-based systems is that the CS8900A is little endian in operation. When used in 16-bit mode with big-endian processors such as the MC68000 or the DSP56805, this endian difference is important. There are two possible solutions. The first is to simply byte-swap in software. Your code then changes the 16-bit word to little-endian format before writing to the CS8900A. And when reading from the CS8900A, the processor must byte-swap the retrieved 16-bit word prior to processing.

However, there is an old saying that you should never fix in software what you can correct in hardware. The second solution is simply to byte-swap the data bus between the processor and the CS8900A. D0:D7 of the processor is connected to D8:D15 of the CS8900A, and D8:D15 of the processor similarly go to D0:D7 of the CS8900A. In this way, the endian-ness is reversed by the actual circuit board, and the software never needs to know the difference (Figure).

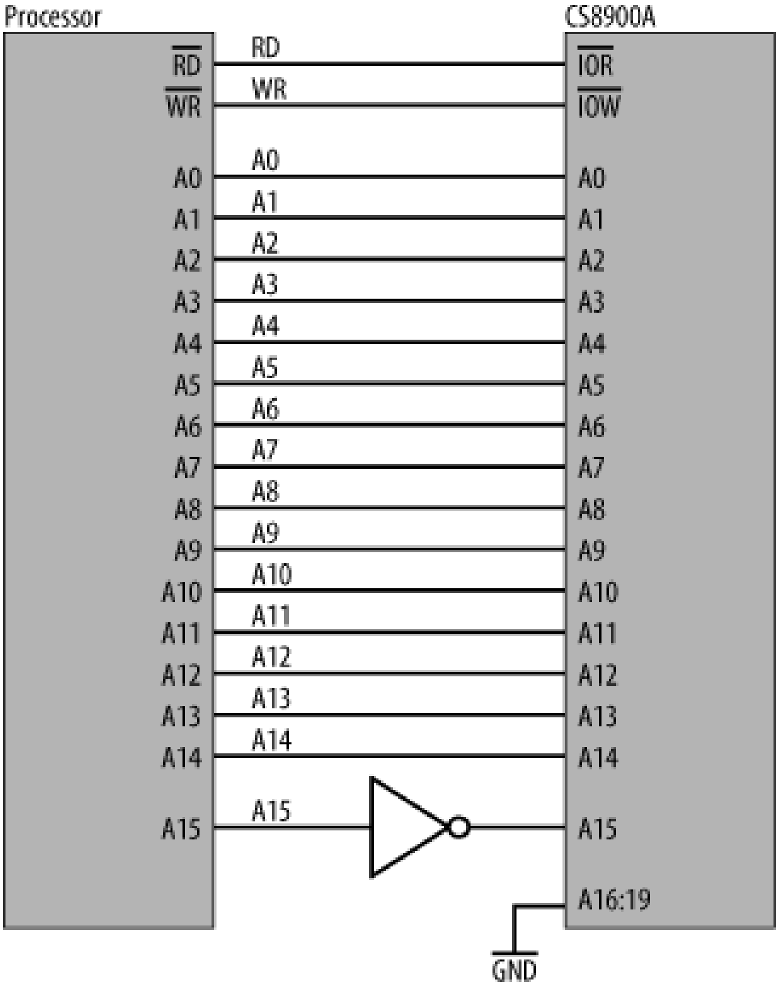
Figure. Endian swapping in hardware



The CS8900A has 20 address inputs. This may seem like a lot of address inputs for a peripheral, and it is. However, there is a reason. The CS8900A is principally an ISA-bus device, and the ISA bus supports separate memory and I/O memory spaces. Hence, the CS8900 has two separate processor interfaces. In one, it appears as part of the memory space of a processor and is accessed as though it were a memory device. A chip-select input, CHIPSEL , enables the CS8900A when it is used as a memory-mapped device. When it is used as a device within an I/O space, there is no externally generated chip select. Instead, devices mapped into the I/O space of an ISA bus are expected to do their own address decoding, and that is why the CS8900A has 20 address lines. Inside the CS8900A is an address decoder specifically for this chip. When the CS8900A is reset, it defaults to I/O address 0x00300. This address can be remapped under software control by writing to the appropriate register of the CS8900A. When used as an I/O-mapped device, CHIPSEL is ignored and the CS8900A will respond to the appropriate address on its address inputs in conjunction with IOR (I/O read) and IOW (I/O write). You can use the CS8900A in I/O mode within a memory-mapped I/O system. The system address decoder includes the address allocation for the CS8900A but simply does not select it. What the system address decoder must do is ensure that no other device is selected when the address(es) corresponding to the CS8900A is being accessed.

The default setting for the CS8900A is I/O mode operation. To use the CS8900A in memory-mapped mode, and therefore to have it recognize CHIPSEL and its memory read (MEMR) and memory write (MEMW) inputs, the CS8900A must first be accessed as an I/O-mapped device and reconfigured in software. Therefore, to use the memory-mapped option, you still have to support the I/O-mapped addressing scheme to get to it! Therefore, it is much simpler to stick with the I/O-mapped mode and map this within your memory space as just described. If you're using the CS8900A with a processor that has only a 16-bit address bus, simply tie the additional address inputs of the CS8900A to ground. The CS8900A's default address of 0x00300 may be inconvenient for use with some processors that already have internal I/O systems mapped within that region. An access to that address will be intercepted by the internal I/O and never reach the CS8900A. In such cases, it will be impossible to remap the CS8900A's address through software. You will simply never reach the appropriate register. But there is a solution, and it lies within hardware. If you invert some of the address bits from the processor before they reach the CS8900A, you can perform the remapping automatically. The CS8900A still thinks it lies at address 0x00300, but to the processor it is accessed at a completely different address. Figure shows an example of this for a processor with a 16-bit address bus.

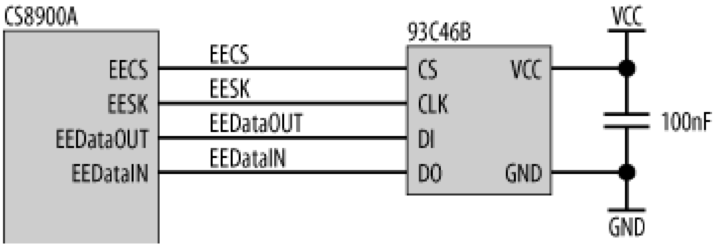
Figure. Address remapping in hardware



In this example, address bit A15 is inverted. So, when the processor accesses address 0x8300 (%1000 0011 0000 0000), this is converted to address 0x0300 (%0000 0011 0000 0000), which is recognized by the CS8900A.

The CS8900A also has support for a serial EEPROM. This can be used to store CS8900A configuration information and the system's unique Ethernet address. Note that this EEPROM is optional, as the host processor can store this data elsewhere in the system. Figure shows the CS8900A interfaced to a configuration EEPROM. The interface is standard SPI, and the appropriate pins of the CS8900A are directly connected to the corresponding EEPROM pins. The only other component required is a decoupling capacitor for the EEPROM's power-supply pin. The EEPROM interface is disabled in 8-bit mode, so the host processor must supply all configuration information.

Figure . CS8900A interfaced to a configuration EEPROM



Finally, any used inputs, such as the DMA signals ($\overline{\text{DMACK0}}$.,$\overline{\text{DMACK1}}$ and$\overline{\text{DMACK2}}$), $\overline{\text{TEST}}$, $\overline{\text{SLEEP}}$, $\overline{\text{MEMW}}$, $\overline{\text{MEMR}}$, AEN, and $\overline{\text{REFRESH}}$should be tied inactive. These signals are not used in a typical embedded system.

Summary

- CAN is a real-time communication at up to 1 Mbps, over a 2-wire serial network. CAN specifies only the physical and data-link layers of the ISO-OSI model, with higher layers left to the specific implementation.

- Crystal Semiconductor, now part of Cirrus Logic (http://www.cirrus.com), produces a single-chip Ethernet controller known as the CS8900A. This chip allows you to add a simple (and low-cost) 10 Mbps Ethernet interface to your embedded system.

- Four wires are used: two for the transmitter pair and two for the receiver pair. One wire of the pair carries a signal voltage of 0 to +2.5 V, while the other wire carries a voltage of 0 to -2.5 V, giving a signal difference of 5 Vpp.

Questions

- What is CAN?

- Briefly explain Ethernet?

- Explain address remapping in controllers?

- What is Endian swapping in hardware?

- What is 10base-T interface?

References

- C Programming for Embedded Systems – KIRK ZURELL
- Design with 8051- FRONTLINE ELECTRONICS
- Embedded Controller Hardware Design - Ken Arnold
- Embedded Software The Works – colin walls
- Embedded Systems Firmware Demystified - Ed Sutter
- Embedded_Controller_Hardware_Design – KEN ARNOLD
- Programming Embedded Systems in C and C++ - Michael Barr
- The Art of Designing Embedded Systems - Jack G. Ganssle.

UNIT – V

11. The PIC Microcontrollers

Objective:

This chapter introduces you to the Microchip PIC. To start our discussion of microprocessor hardware, we'll look at the basics of creating computer hardware by designing a small computer based on a simple 8-pin PIC processor. The same design principles apply to the AVR and many other microcontrollers. This PIC processor is so simple that building a computer based on one of them is trivial, as you will see. From there, we'll look at a mid-range PIC processor and see just what you need to do to design an embedded computer based on one. First, though, let's take a quick tour of the PIC architectures before getting into designing some computers.

A Tale of Two Processors

In the late 1970s, General Instruments had a 16-bit processor known as the CP1600. It has since passed into extinction and is all but forgotten, long ago losing out to the Intel 8086 and the Motorola 68000. One major failing of the CP1600 was that it had limited I/O capability, and so General Instruments designed a tiny companion processor to act as an I/O controller. The idea was that this controller could provide not only the I/O for the CP1600, but being a processor in its own right, it could provide some degree of intelligent control. This processor was called the Peripheral Interface Controller, or PIC. The CP1600 died a quiet death, passing gently into oblivion, but its little companion lives on. In the mid-80s, the microelectronics division of General Instruments was spun off into Microchip, and the PIC processor was its core product. Today, PICs are widely used. They live in the hand controllers of Sony PlayStations, children's toys, consumer appliances, and industrial systems.

The original PIC architecture has only one accumulator (known as the working register, or w register) and 25 to 368 bytes of RAM in the original processors. The program counter's least significant byte, the status register, and various control registers are mapped into the lowest part of the RAM space and may be accessed by standard memory move operations. The upper part of the RAM space is for data. Microchip refers to the RAM space as "registers," although they have limited functionality as true registers. They are primarily for data storage.

The processor has a stack that is fixed to a depth of between two and eight entries (depending on the particular processor) and is used solely for holding return addresses for subroutine calls and interrupts. There is a single register, known as the FSR (File Select Register), which can act as an index register into the RAM space. Limited indexed addressing is available using the FSR, and it can be used to implement a pseudostack for user data.

Apart from a few exceptions, the PIC has no external buses and is a self-contained computer within a single chip. Only limited expansion is possible using the processor's peripheral interfaces (SPI and I_2C) or digital I/O ports. The PIC excels in applications for which size and power consumption are critical. Being able to drop a tiny computer system into a design is a great bonus, and it is ideal for battery-powered applications, since it can (almost) run off the field of a stray electron.

The PIC is also very robust. It takes a lot to kill a PIC. I had one client that inadvertently switched power and ground on his PIC-based computer and left it that way for a week. At the end of it, the little processor was still operational (once powered the right way). Another time, one of my PIC-based data loggers was tested for its long endurance by attaching it to the Indian Pacific express. This is a long-haul passenger train that goes between Sydney and Perth, crossing the deserts of central Australia. Unfortunately, during the trial the Indian Pacific was involved in a serious rail accident. A signaling fault caused a commuter train to impact the rear of the express, completely demolishing the end carriages. The data logger had been attached (externally) to the rear of the train. It absorbed the full impact of the collision, and, when recovered from the wreckage, the data logger was still operating normally. PICs are tough little processors!

The PIC is very RISC-like in many respects. The architecture is Harvard, with separate data and code spaces. The data space is 8 bits wide, while the code spaces are between 12 and 16 bits wide, depending on the particular PIC family. The data space is mapped into multiple banks, including most control registers. With only one accumulator, banked memory, and limited addressing modes, a reasonable percentage of a given program can be spent simply shuffling data around, much more so than many other processors. The PIC excels in small-scale, simple applications. However, the lure of its ultra-low power consumption sometimes means that it is pressed into service running some quite involved algorithms. Writing complicated software for the PIC sometimes feels as impossible as trying to solve a Tower of Hanoi puzzle that has only a single peg. It can be a challenge! Many a PIC programmer has wished for just a bit more memory, and just a few more accumulators. The new dsPIC architecture, which is a significant advance over the standard PIC, has been received with chortles of joy by PIC developers around the world, as is a much more advanced processor.

A number of commercial C compilers are also available for the PIC, but there is no port of the gnu C compiler for it. (At the time of writing, there are rumors that the gnu compiler will be ported to the new dsPIC architecture.)

For many simple digital applications, a small microprocessor is a better choice than discrete logic, because it is able to execute software. It is therefore able to perform certain tasks with much less hardware complexity. So, let's see just how easy it is to produce a small, embedded computer.

Starting Simple

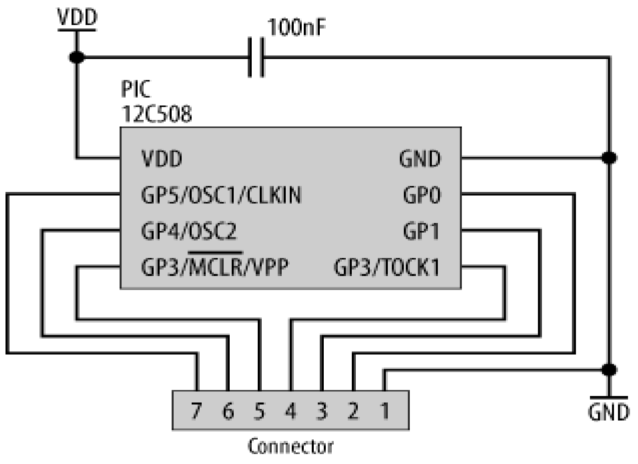
The PIC12C508 processor is a tiny 8-pin computer with just 512 words of internal program memory and just 25 bytes of internal RAM. It is intended for the simplest of control functions. It can be used in any small application for which you need to monitor digital inputs or turn something on or off. Its I/O pins can be used to synthesize a SPI or I_2C interface, or to control a motor using PWM.

Figure shows the schematic for a small computer based on the PIC12C508. The digital I/O signals of the PIC are brought out through a 7-pin connector. If the design were implemented using surface-mount components wherever possible, the connector would be the largest component on the PCB!

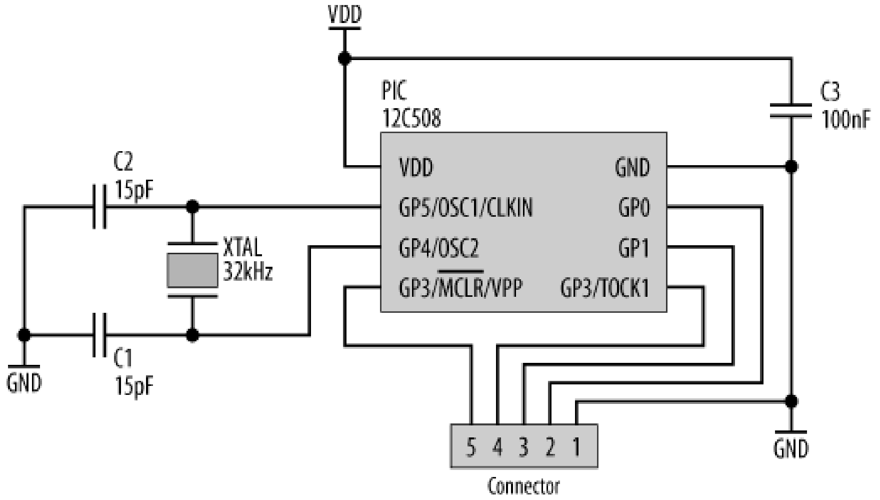
This particular PIC processor includes an internal RC oscillator that runs at 4 MHz, so we can use it without any external oscillator circuit. The design in Figure shows the same PIC-based design, but this time using an external 32 kHz watch crystal for its oscillator. By running off a (slower) 32 kHz crystal, we have the advantage of greatly reducing the processor's power consumption. This is important for battery-powered applications.

Two 15 pF capacitors remove unwanted higher-order harmonics from the crystal's oscillation. The values for the capacitors vary depending on what speed and type of crystal you are using. The processor datasheet has tables showing recommended capacitor values for various crystal frequencies.

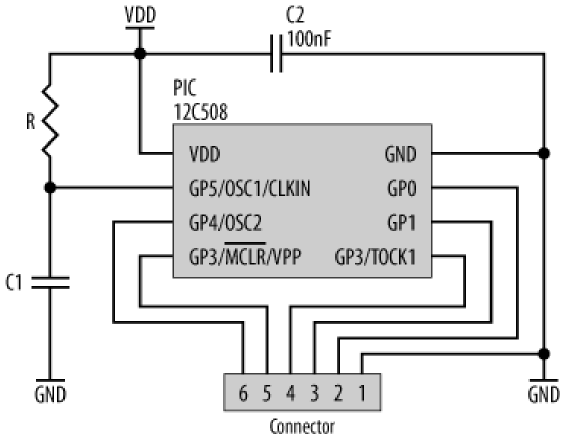
Minimal PIC12C805 computer



A basic PIC12C508 computer; just add power The alternative is to use an external RC circuit as the clock source (Figure). While not the most precise timing option, it is by far the cheapest. The actual frequency of oscillation depends on a combination of the values of the resistor, the capacitor, the supply voltage, the variation in tolerances for the components, and the current operating temperature. To be clear, only an approximate operating frequency can be determined for an RC oscillator.



The alternative is to use an external RC circuit as the clock source (Figure). While not the most precise timing option, it is by far the cheapest. The actual frequency of oscillation depends on a combination of the values of the resistor, the capacitor, the supply voltage, the variation in tolerances for the components, and the current operating temperature. To be clear, only an approximate operating frequency can be determined for an RC oscillator. For stable operation, Microchip recommends that the resistor should be between 3 k and 100 k, and the capacitor greater than 20 pF. If you wish to use an external RC oscillator, refer to the processor's datasheet, as Microchip has detailed information on RC component selection, taking into account voltage and temperature effects. External RC oscillator

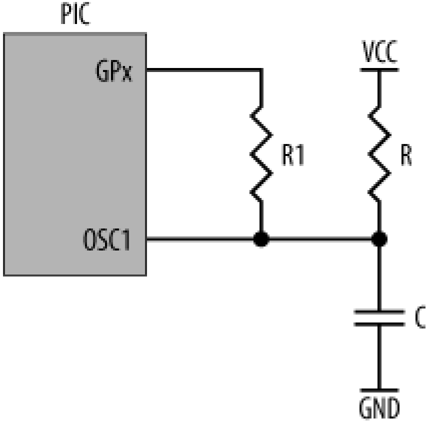


Variable-Speed Oscillator

One of the neat tricks you can do if using an external RC oscillator is have a variable-speed computer. This is accomplished by adding a pull-up resistor (R1) between the oscillator input and an I/O pin (Figure). For normal operation, the I/O pin is configured as an input. By configuring the I/O pin as an output and placing it high, the resistor R1 is effectively placed in parallel with the resistor R. The overall resistance is increased by the relationship RTOTAL = 1 / (1/R + 1/R1), and the oscillator slows accordingly. This is a useful technique to reduce power consumption under software control.

When using an external RC circuit to drive the internal oscillator, an extra PIC I/O line (GP4) becomes available for use.

Variable-speed RC oscillator



Power-on Reset

No external reset is needed for this PIC. Instead, the design relies on the internal power-up reset circuit of the processor. Further, not even an external resistor is required on the reset input, RESET, since the processor incorporates a weak pull-up resistor for this purpose. When not used as a reset input. The power supply (VDD) for the PIC12C805 can range from 2.5 V to 5.5 V.

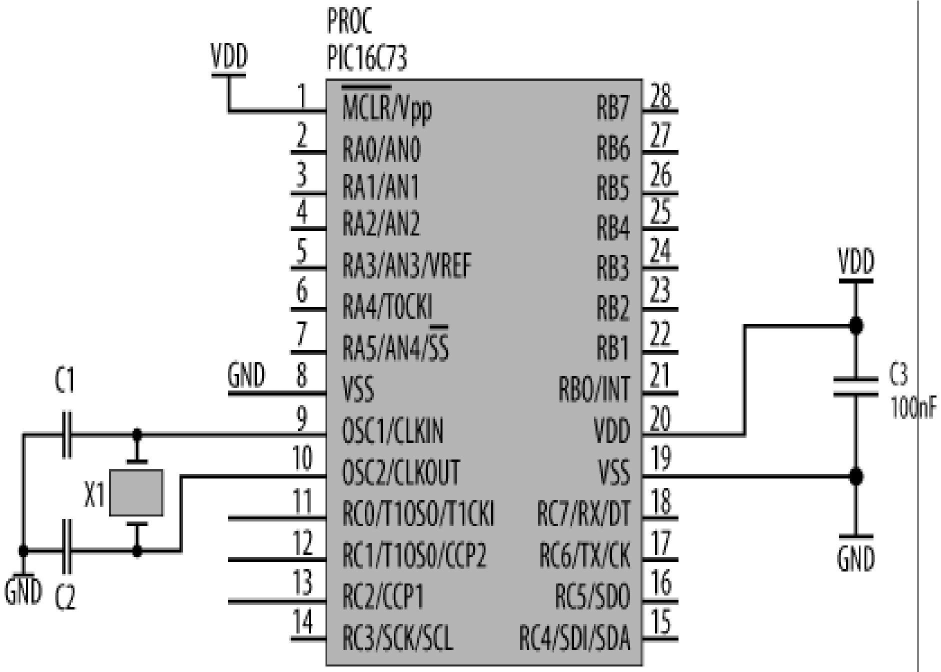
That covers the basics of a PIC12C805 system, and it's not that much different from the corresponding AVR computer, which we'll look at in the next chapter. The real differences lie in their internal architectures (and instruction sets) and in the subtleties of their operating voltages and interfacing capabilities. As you can see, there's not a lot of hard work involved in putting one of these little machines into your embedded system.

A Bigger PIC

In this section, we'll look at the PIC16C73 processor. For a mid-range PIC, the design is not dissimilar to the simpler PIC we've already seen. The only real difference is that the processor has more pins, more I/O, and more functionality. Designing for PIC17 and PIC18 processors is not dissimilar to creating machines based on the PIC16 family. What you learn here is applicable to many other PICs.

The schematic for this processor is shown in Figure. This processor has 4K words of program memory, 192 bytes of RAM, and a variety of I/O subsystems, such as three timer modules, SPI, I_2C, a UART, five channels of analog input, and up to 22 digital I/O pins.

PIC16C73 processor and support components



This processor has one power pin (VDD) and two ground pins (VSS). As always, power is decoupled to ground with a small capacitor (C3). The only other requirements are some form of clock generation—in this case provided by a crystal, X1--and two decoupling capacitors, C1 and C2. The clock could just as easily have been provided using an RC circuit, as we saw with the 12C508 PIC. The reset input, RESET, is tied directly to the power supply, such that it is permanently inactive. In this case, we are relying on the processor's internal power-on reset circuitry and don't need to provide an external reset. It is common practice to use a pull-up resistor to tie an unused input, such as RESET , inactive. However, in this case I have found that a pull-up resistor can affect the activation of the internal power-on reset to the point where it fails to kick in. (The internal capacitance of the pin combines with the resistor to form an RC circuit, which delays RESET from reaching the appropriate level.) Thus, the resistor can actually cause the processor to never start properly. So in this case, it's better to leave it out.

Port A (RA0 . . . RA5) functions as an analog input port or a general-purpose digital port. Port B (RB0 . . . RB7) is a general-purpose digital port with weak internal pull-up resistors. Port C (RC0 . . . RC7) can act as a digital port or provide timers, PWM, a serial port, a SPI port, or an I_2C interface. Depending on your application, you may use some or all of these pins in your design, connecting to other subsystems as appropriate.

This basic design, in combination with the appropriate datasheet, can be adapted to most other PIC processors that you will come across.

PIC-Based Environmental Data logger

Now let's look at a complete system based on a PIC processor. The design presented here is a simplified version of my DL4 data logger product. This data logger is designed for extended recording of data (for at least a year), using a minimum of power. It has 1M of nonvolatile memory, capable of retaining data without power for as much as 20 years. The sensors fitted are light and temperature, but you could easily adapt this design to record any analog sensor you like, from acceleration to magnetic field. It's also small. The entire data logger fits onto a circuit board smaller than your smallest finger.

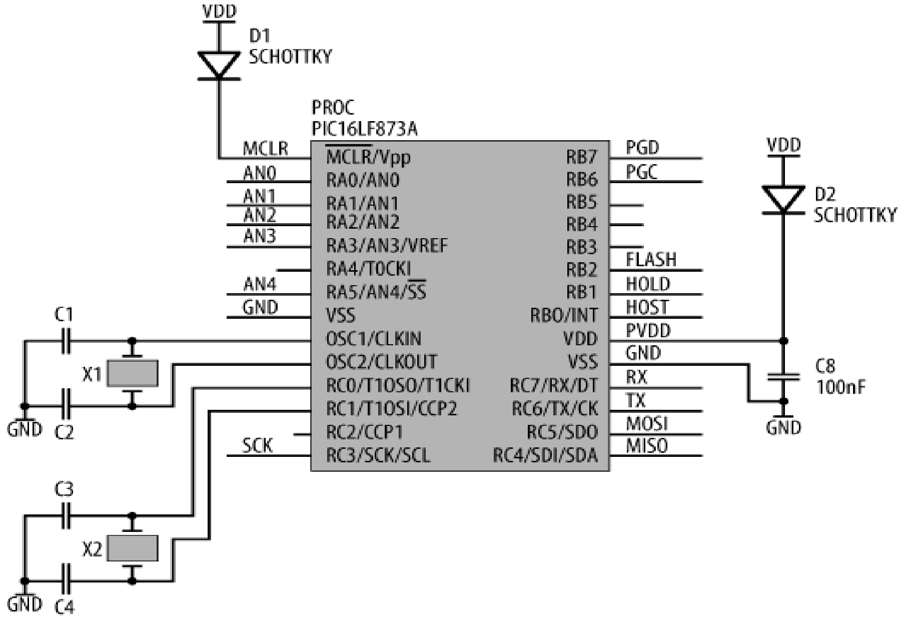
The processor is a PIC16LF873A, a variant of the PIC16C73. The "L" means that it is low-power, and the "F" means that it is a flash-based part (rather than EPROM or OTP) that can be reprogrammed in-circuit, making debugging (and life) so much easier. The "8" indicates that the processor includes EEPROM for nonvolatile parameter storage, useful for holding user preferences and machine state. Finally, the "A" tell us that it is a second revision (version) of the silicon. The basic circuit for the processor and its support components is shown in Figure. Note that the processor's power pin is connected to a net labeled PVDD rather than the system's VDD. Since the processor can be reprogrammed in-circuit, we must consider this in our design. During programming, the burner provides its own supply voltage (+5 V) for the processor. Now the flash chip used in the data logger requires a nominal supply voltage of 3.3 V, and the 5 V supplied by the burner could potentially damage the chip. Hence, we use a Scotty diode, D2, to isolate the system supply voltage from the processor when it is being reprogrammed. When not being programmed, D2 conducts and supplies the processor with power. During programming, D2 doesn't conduct, and the rest of the data logger remains unpowered. In the same way, we use a Scotty diode (D1) to isolate the processor's reset pin from the supply voltage. When not being programmed, D1 conducts and pulls RESET to VDD. However, during programming, D1 isolates RESET and allows the burner to pull this reset input to a higher voltage as required.

The processor has two crystals, X1 and X2. X1 provides the timing for the processor that drives its internal operation. Depending on our application, X1 could be anything from 32 kHz to 20 MHz. The choice of crystal for X1 affects the power consumption of the data logger. The faster the crystal, the more juice the machine uses. For ultra-low-power operation, a 32 kHz crystal is the best choice. However, this does have a drawback. Since the internal oscillator is used to drive the UART's transmitter and receiver clocks, a slow crystal limits the baud rate that we can achieve. Using a 32 kHz crystal gives a maximum baud rate of only 300 baud. Downloading a megabyte of data from the data logger at that speed takes a whopping 7 hours and 42 minutes! (And you thought your Internet connection was slow.) Hence, you need to choose a value of X1 that best suits your needs. If you can live with a 7-hour download and want the maximum possible operating life from your battery, use a 32 kHz crystal. If battery life is not critical, use a faster crystal.

X2 is used to drive the processor's internal TIMER1 subsystem, which we use for timekeeping functions and for scheduling the sampling process. While it is possible to use the processor's main oscillator to drive the internal timers, this oscillator is shut down during the execution of the SLEEP instruction. (The internal watchdog circuit is used to reawaken the processor.) Hence, it is better to use a second crystal on TIMER1 for your timekeeping, as TIMER1 continues to operate even during sleep.

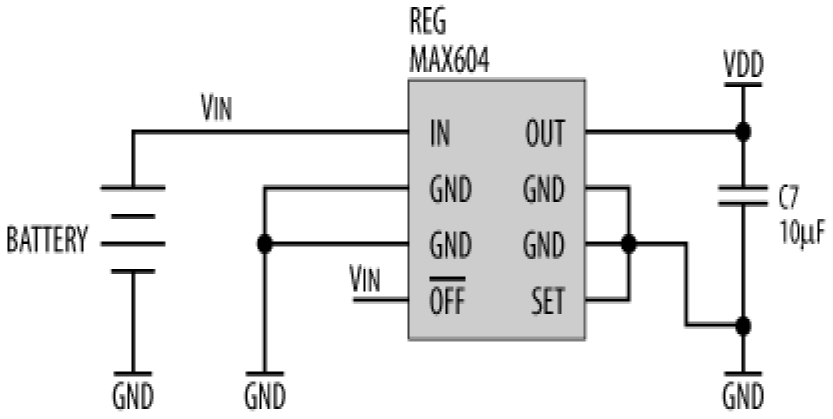
The voltage regulator circuit is shown in Figure. It's a standard MAX604 circuit, providing a 3.3 V supply voltage on VDD. Since we are using a battery to supply power, we can live without a capacitor on the input. There's a huge choice of batteries available. I like the Energizer EL123 battery. It's relatively small (two-thirds the length of a AA and slightly fatter) and can run the data logger for well over a year.

Datalogger processor and support components



The regulator is important in the data logger for two reasons. First, it ensures that the supply voltage within the data logger is constant. This is critical, as the supply voltage

Datalogger power supply



is used as a reference for the analog-to-digital converter. While it is perfectly possible to run the processor directly off a battery, as the battery's voltage begins to drop with use, the readings from the sensors will become increasingly meaningless. We'd be recording data quite successfully, but its relevance would be nil.

The second reason for using the regulator is that it is able to operate off a lower voltage than other components in the system and still provide a stable 3.3 V. This means that even as our battery is draining, we can still get the maximum operating life possible.

Figure 14-8 shows the nonvolatile flash (made by ST Electronics), which is used for holding data. The flash uses a simple SPI interface to connect to the host processor. It is selected by the FLASH chip select, which is controlled from a processor I/O line (RB2). The HOLD input to flash acts as a "pause" function during accesses. This allows the processor, until software control, to temporarily suspend its access to the flash and perform SPI transfers with other peripherals. This feature currently isn't used in the data logger but is included to allow for future functionality, such as the inclusion of digital, SPI-based sensors. Finally, the chip select (FLASH) is pulled high to ensure that the flash is not inadvertently selected as the data logger is powering up.

Data logger nonvolatile memory

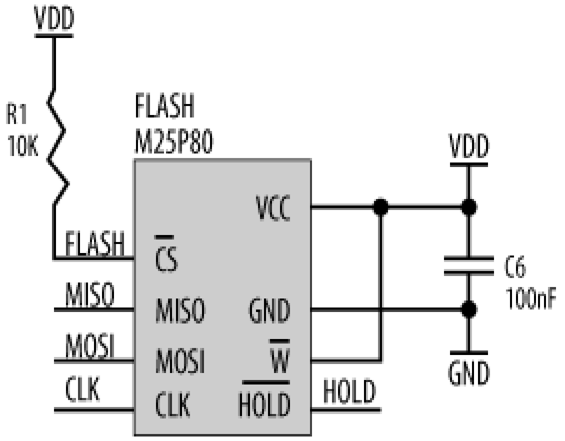
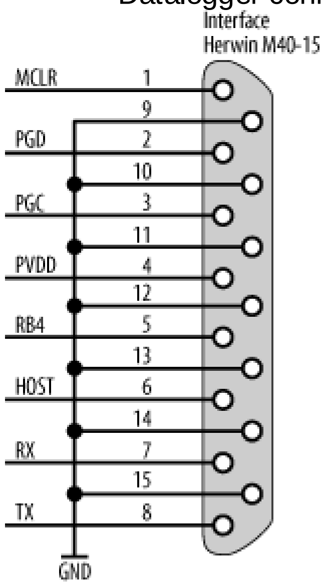
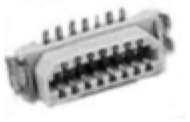


Figure shows the data logger's interface to the outside world. The DL4 data logger uses a small Harwin connector, also shown in Figure, but you could use whatever suits your application, even an IDC header.

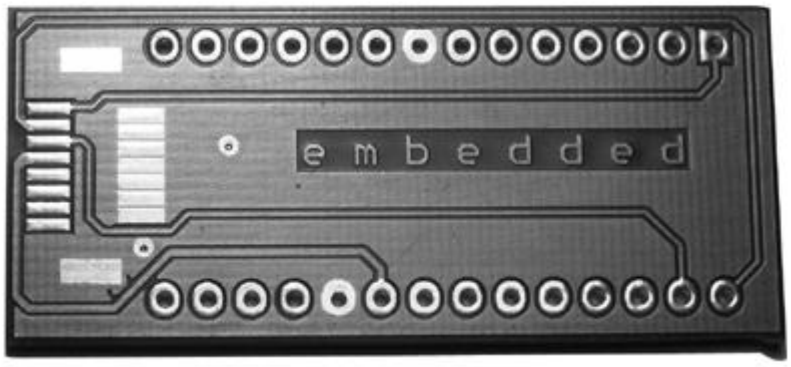
The connector is used to mate with two separate devices. The first is shown in Figure. This in-house adaptor allows the datalogger to be plugged into a Microchip PICSTART Plus programmer for burning new code. It simply maps the signals required during programming (PGD, PGC, power, and ground) to the equivalent pins for a DIP-based PIC. In essence, it converts the datalogger's connector into the pinout of a DIP-based PIC. The adaptor board has pins underneath that insert into the programmer

Datalogger connector





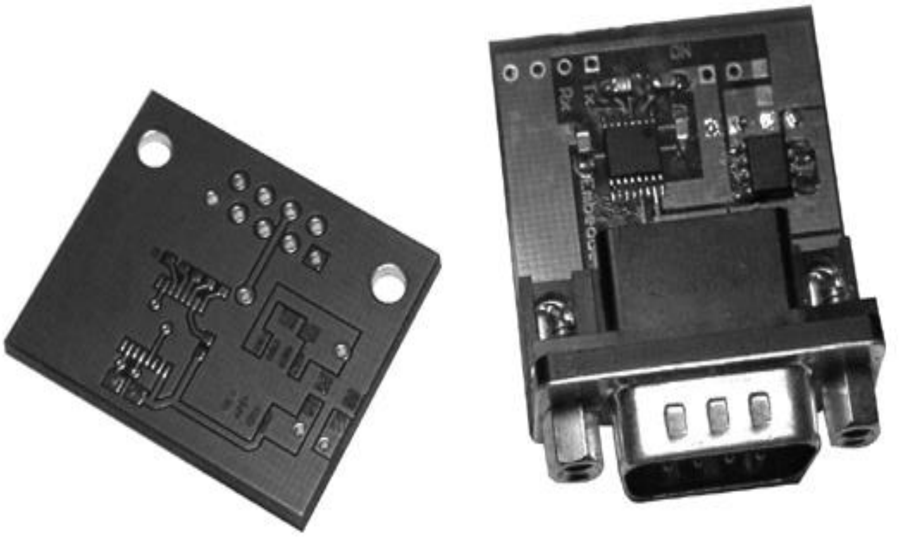
Programming adaptor for the DL4 datalogger



The second device into which the data logger plugs is an RS-232C adaptor module, shown in Figure.

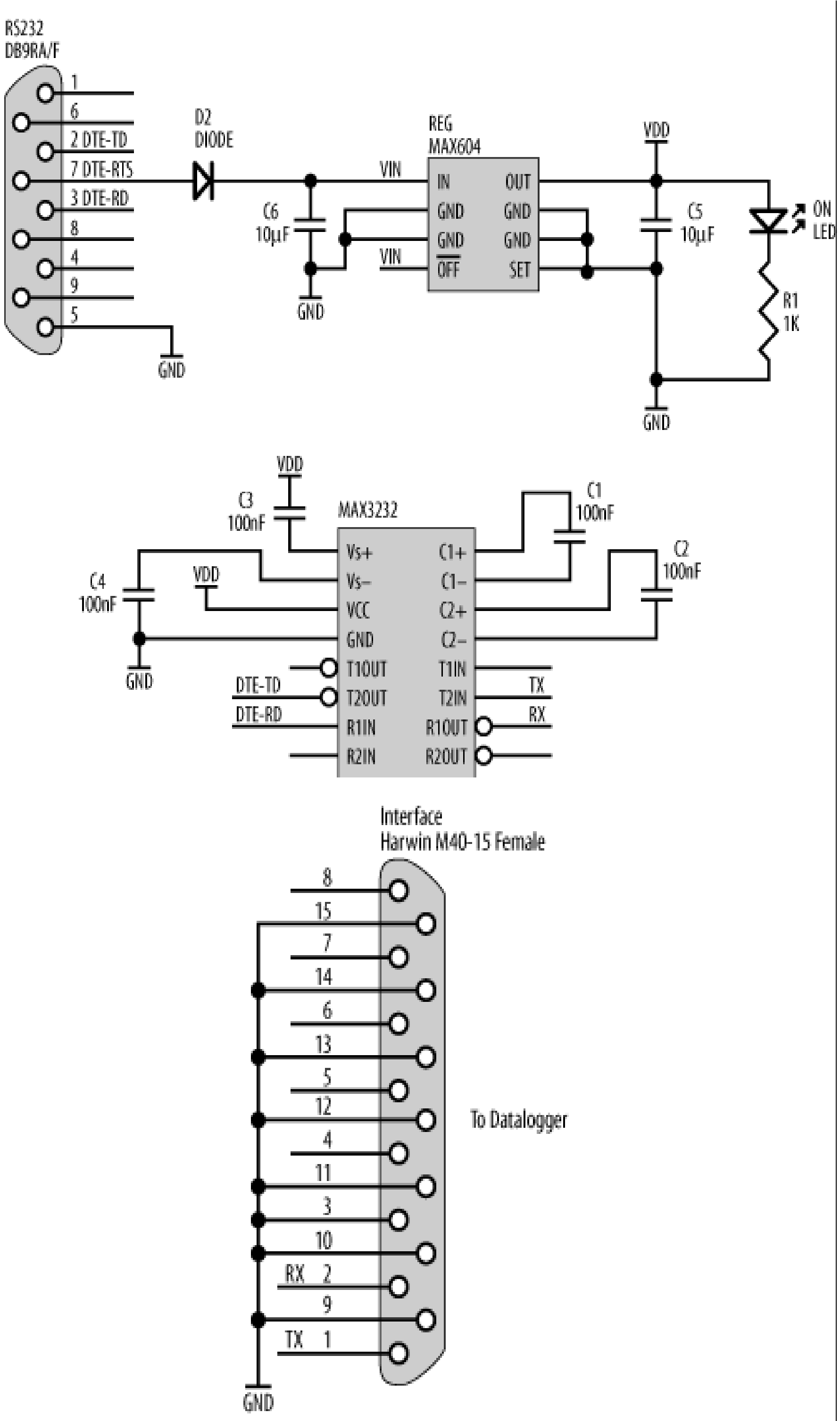
Serial adaptor

This module allows the data logger to connect to a host computer for both configuration setup and data recovery. The adaptor has a Maxim RS-232C level-shifter and a voltage regulator (both fitted to the top side of the circuit board), and a DB-9 connector. It draws its power from the RTS signal of the host computer's serial port and, as such, requires no external power supply. The schematic for this circuit is shown in Figure. Note that this is an independent circuit from that of the data logger. The diode D2 is needed since RTS can have negative voltages as well as positive voltages. D2 prevents damage to the regulator when RTS is negative. During normal operation, RTS is set at +12 V by the host computer under software control, turning on the serial adaptor. The regulator turns this +12 V into +3.3 V, which powers the MAX3232 and illuminates the LED.



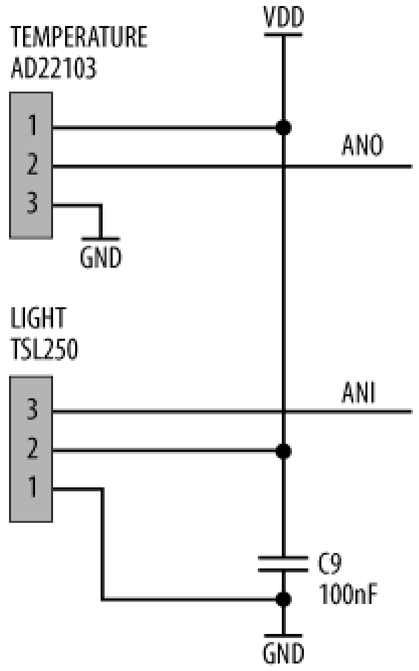
Note that pin 6 of the Harwin connector on the data logger is connected to a net called HOST. This pin corresponds to pin 3 of the adaptor's Harwin connector and is tied to ground. When the data logger is plugged into the adaptor module, HOST is pulled low, but at all other times it is high due to the internal pull-up resistors inside the PIC. In this simple way, software running on the data logger can tell whether a host computer is present. This is useful as the data logger's UART may be disabled to save power until it is required. Further, it can act as a simple switch for the firmware, toggling between "talk to host" mode and "data logging" mode. It's interesting to note that the Harwin connector is much bigger on the schematic than the DB-9 due to the number of pins. Yet, when placed next to a DB-9, the Harwin connector is physically tiny.

Serial adaptor schematic



Finally, Figure shows the sensors of the data logger. You could just as easily use other sensors or provide a connector allowing for interfacing to interchangeable sensor modules or external analog subsystems.

Data logger sensors



Motor Control with a PIC

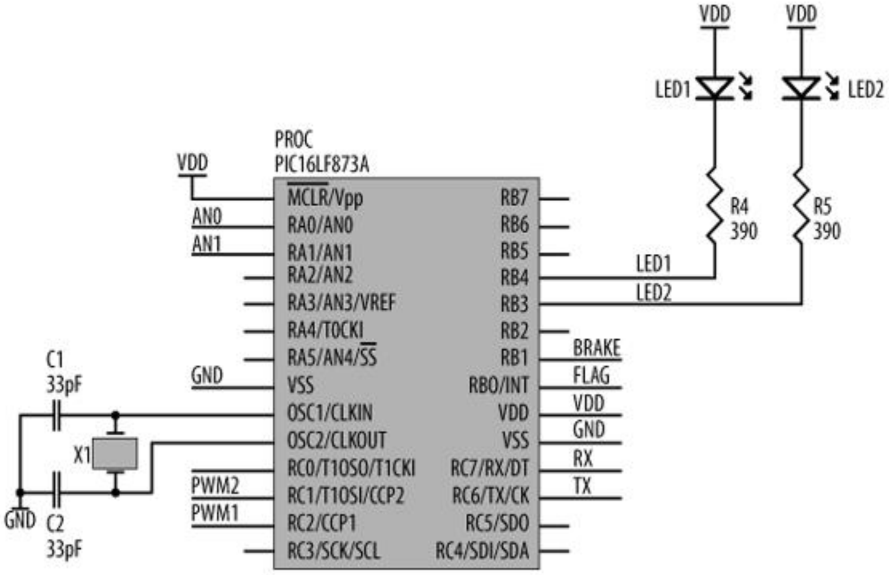
Now let's look at using a PIC in a completely different sort of application: motor control via user input. While the design presented in this section is targeted at a specific application, it is just as easily adapted to any task where small DC motors need to be controlled, from tools to robotics.

My young nephews are into model trains, and the standard controller that came with the railroad was a fairly simple device. The speed of the locomotives is controlled by simply varying the voltage on the track. Turn up the voltage, and the trains go faster; turn down the voltage, and the trains stall on dirty rails. Fine control and realistic operation just wasn't possible. I decided to solve this problem by throwing a little high-tech at it and designed for them a microprocessor-based controller using a PIC. It's this design that I will share with you.

The hardware design takes the basic concept one step further by adding momentum control and braking. The momentum control allows you to specify the mass of the train under control so that when you open up the throttle, the train gradually builds up speed, and rolls to a halt when the throttle is closed. Braking speeds up the stopping. All this sounds complicated, but the hardware to do it is trivially simple when you use a microcontroller. All the real "work" is done in the software, and you can keep it as simple or as fancy as you want.

The processor schematic is shown in Figure. It's not that different from the one used in the datalogger.

Figure. Processor



You'll notice that there's only one crystal, since there's no need for timekeeping. Two status LEDs are provided for user feedback by software. As there are several spare pins on Port B (RB2, RB5..7), you could add extra LEDs if you feel so inclined. BRAKE and FLAG are inputs to the processor. BRAKE comes from a push-button switch, and FLAG is an output from the H-bridge chip that indicates an over-current fault.

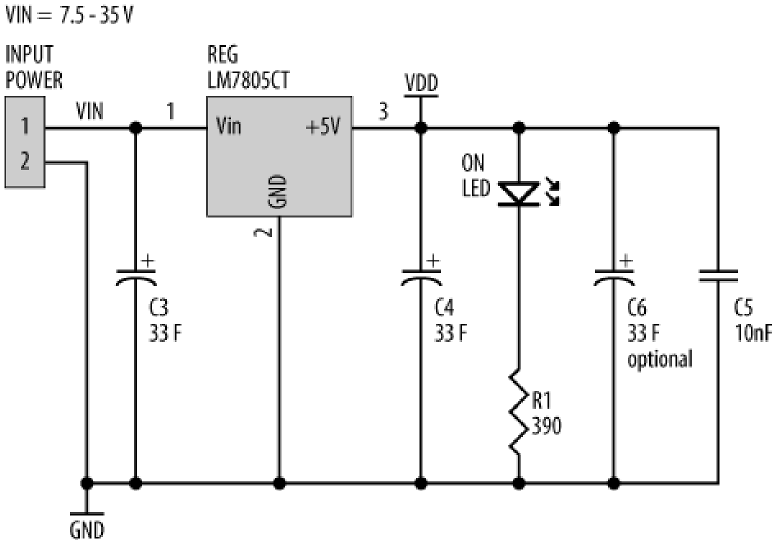
You'll also notice that there's no provision for in-circuit programming as was provided in the data logger design. For this particular project, I used a DIP-based PIC processor that I had lying around, and to reprogram the part it was a simple matter to remove it from its socket on the circuit board and drop it into the burner. If you want to be able to reprogram the chip in-circuit, simply use the same connections as in the data logger design.

The choice of crystal frequency is up to you, but the choice you make does have an interesting consequence. The frequency of the crystal relates directly to the PWM frequency the processor is able to generate. The clock signal from the crystal circuit is divided by four before being fed into the processor's PWM modules. Registers within the PWM modules can then scale this back further to produce a slower PWM frequency. Commercial train controllers use a PWM frequency of 16,125 Hz. Any slower than this and you will hear noise from the motors; any faster and there is a loss of torque. So to achieve a PWM frequency of 16 kHz, you'll need a crystal with a frequency greater than 64 kHz. As you can't easily obtain crystals of that exact frequency, the best choice would be a 1 MHz crystal, using the PWM modules' registers to scale it back appropriately.

The voltage regulator circuit is shown in Figure. The regulator chosen is a standard (and cheap) LM7805 that provides a constant +5 V output from an input voltage of between +7 V and +35 V. Since the motors of model locomotives run on a nominal maximum of +12 V, this is the actual supply voltage for the system. Also included in the regulator schematic is a LED to indicate when power is on and extra decoupling capacitors to help eliminate digital noise in the system.

Figure shows the controls used to drive the train. The throttle and momentum controls are simply 50 k potentiometers, which are used as voltage dividers. The wipers of these pots provide a voltage of between 0 V and +5 V to the analog inputs of the PIC. Thus, position of the control can easily be read by software. The direction and brake controls are simple push buttons. The direction control connects directly to an input on the H-bridge chip (discussed shortly), and the brake control is used as a digital input to the processor. The direction control has a 100 k pull-up resistor, and the brake control relies on the internal pull-ups of Port B of the processor.

Figure. Voltage regulator



Controls

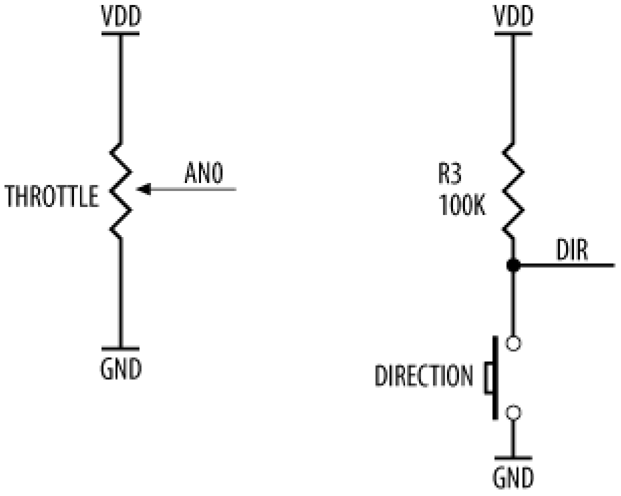
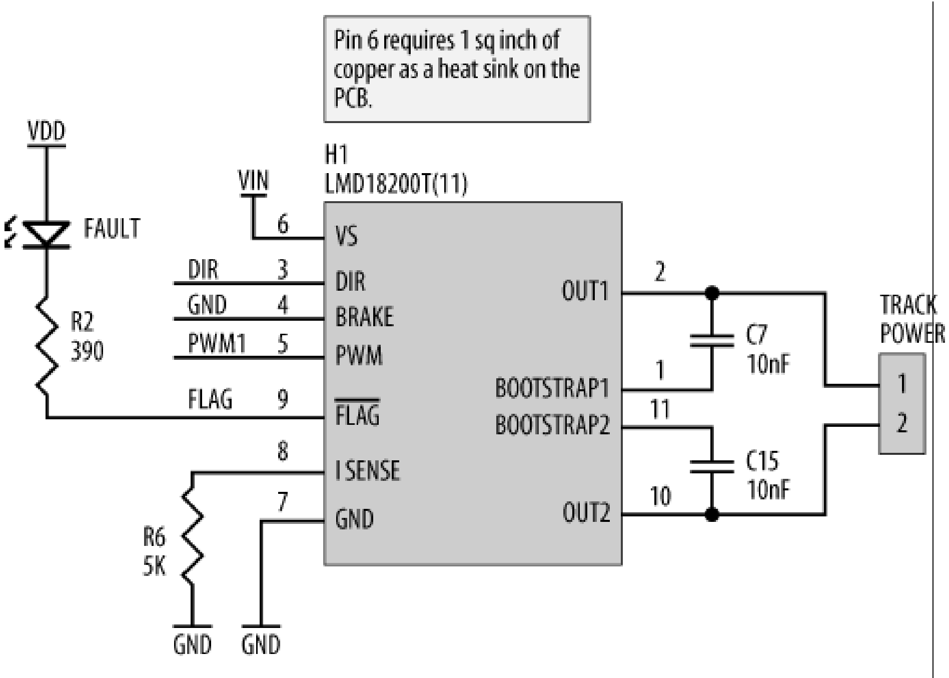


Figure shows the H-bridge chip. This converts the PWM output of the processor to voltage levels appropriate for driving the small DC motors found in model locomotives or, for that matter, any sort of small DC motor.

Figure. H-bridge



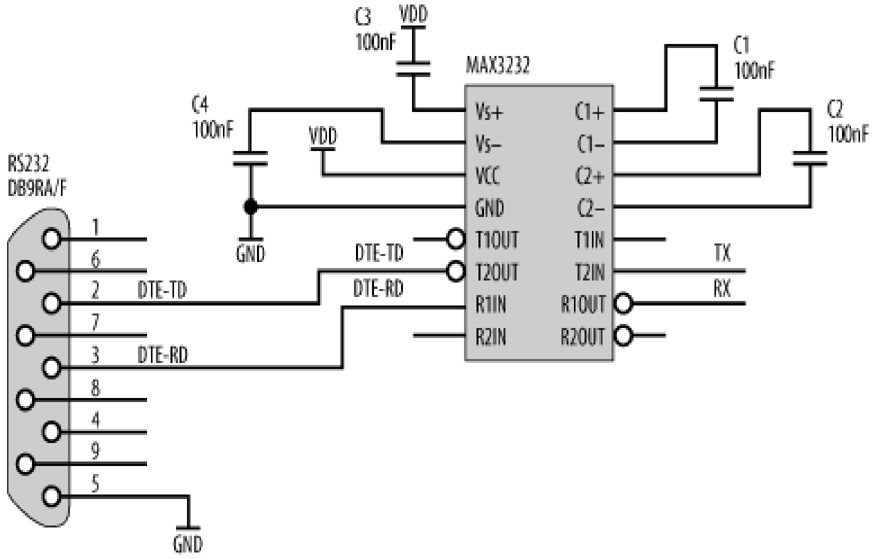
The H-bridge uses VIN (+12 V) as its power source because it must supply that voltage to the motors. As mentioned, the DIR (direction) input comes from a simple push button and determines the polarity of the output. Alternatively, the direction switch could be connected to a spare digital input of the processor, and a digital output could drive DIR. Since it would be a simple mapping of input to output by the software, it's easier just to connect it straight through and bypass the PIC.

The output of the PIC's PWM1 module is connected directly to the PWM input of the H-bridge. The H-bridge converts the 5 V amplitude, PWM signal from the PIC to a 12 V amplitude, PWM output whose polarity is determined by DIR.

The H-bridge has internal over-current sensing and will shut itself off if its temperature rises too high (as would happen if the outputs were shorted together). The FAULT output indicates when this occurs. This active-low output is used to control a LED and is also connected to an input on the processor so that the software can be made aware of the fault condition.

Finally, we can add an optional serial port to the controller, shown in Figure. This is a standard RS-232C level-shifter circuit and is connected to the RX and TX pins of the processor's UART. The serial port can be used for software debugging to display status messages to a host computer or terminal. If you wanted to get very fancy, you could use the serial port to allow a host computer to send commands to the train controller. If you're adapting this design for robotics, a serial port would be a very useful addition. However, if you are providing control from an external host, don't forget to connect the DIR input of the H-bridge to the PIC and not directly to a push button.

Figure. Serial port



In the next chapter, we'll take a look at the AVR processor family. These processors are comparable to PICs in terms of I/O and functionality but have a higher throughput and a more versatile architecture.

Summary:

- The original PIC architecture has only one accumulator (known as the working register, or w register) and 25 to 368 bytes of RAM in the original processors.
- The processor has a stack that is fixed to a depth of between two and eight entries (depending on the particular processor) and is used solely for holding return addresses for subroutine calls and interrupts.
- The architecture is Harvard, with separate data and code spaces. The data space is 8 bits wide, while the code spaces are between 12 and 16 bits wide, depending on the particular PIC family.
- The PIC12C508 processor is a tiny 8-pin computer with just 512 words of internal program memory and just 25 bytes of internal RAM.
- PIC processor includes an internal RC oscillator that runs at 4 MHz, so we can use it without any external oscillator circuit.
- 16F73 processor has 4K words of program memory, 192 bytes of RAM, and a variety of I/O subsystems, such as three timer modules, SPI, I_2C, a UART, five channels of analog input, and up to 22 digital I/O pins.
- RS-232C level-shifter circuit and is connected to the RX and TX pins of the processor's UART.

Question:

- Explain the features of PIC controller briefly?
- What is power on RESET? Explain?
- Explain pin diagram of PIC16C63.
- How to design a data logger using PIC, Explain with diagrams?
- Write notes on RS232 in PIC?

References:

- A Beginner's Guide to Using the PIC Microcontroller, by David Benson
- PIC Microcontroller Applications Guide by David Benson
- PIC Microcontroller Applications Guide by David Benson
- PIC Microcontroller Serial Communications, by Roger Stevens
- C What Happens, Using PIC® Microcontrollers and the CCS C Compiler, by David Benson

12. The AVR Microcontrollers

Objective

In this chapter, we'll look at the Atmel AVR processor. Like the PIC, this processor family is a range of completely self-contained computers on chips. They are ideally suited to any sort of small control or monitoring application. They include a range of inbuilt peripherals and also have the capability of being expanded off-chip for additional functionality.

Like the PIC, the AVR is a RISC processor. Of the two architectures, the AVR is the faster in operation and arguably the easier for which to write code, in my personal experience. The PIC and AVR both approach single-cycle instruction execution. However, I find that the AVR has a more versatile internal architecture, and therefore you actually get more throughputs with the AVR.

In this chapter, we'll look at the basics of creating computer hardware by designing a small computer based on the AVR ATtiny15. We'll also see how you can download code into an AVR-based computer and how it can be reprogrammed in-circuit. From there, we'll go on to look at some larger AVR processors, with a range of capabilities.

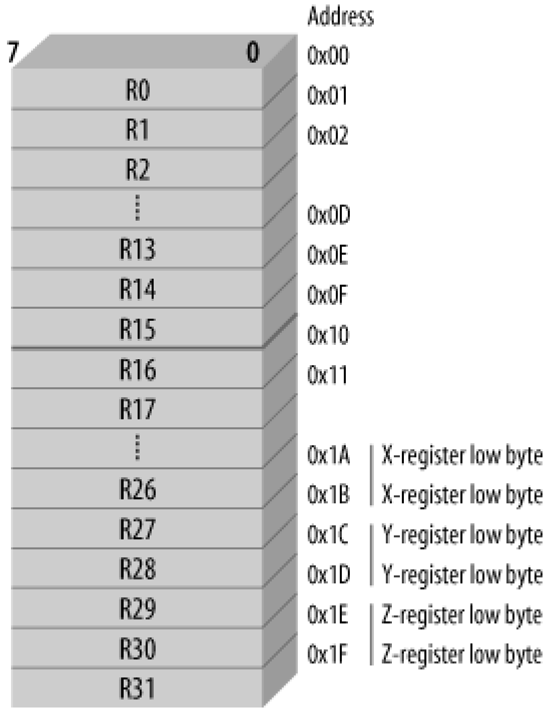
Later in the chapter, we're going to look at interfacing memory (and peripherals) to a processor using its address, data, and control buses. For most processors, this is the primary method of interfacing, and, therefore, the range of memory devices and peripherals available is enormous. You name it, it's available with a bus interface. So, knowing how to interface bus-based devices opens up a vast range of possibilities for your embedded computer. You can add RAM, ROM (or flash), serial controllers, parallel ports, disk controllers, audio chips, network interfaces, and a host of other devices.

Most small microcontrollers are completely self-contained and do not "bring out" the bus to the external world. In this chapter, we'll take a look at the Atmel AT90S8515 processor. It is the only processor of the AVR family that allows you access to the CPU's buses. But first, let's take a look at the AVR architecture in general.

The AVR Architecture

The AVR was developed in Norway and is produced by the Atmel Corporation. It is a Harvard-architecture RISC processor designed for fast execution and low-power consumption. It has 32 general-purpose 8-bit registers (r0 to r31), 6 of which can also act as 3 16-bit index registers (X, Y, and Z) (Figure). With 118 instructions, it has a versatile programming environment.

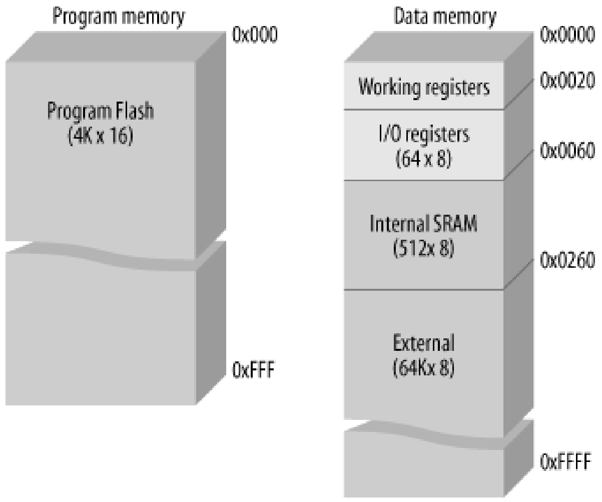
AVR registers



In most AVRs, the stack exists in the general memory space. It may therefore be manipulated by instructions and is not limited in size, as is the PIC's stack.

The AVR has separate program and data spaces and supports an address space of up to 8M. As an example, the memory map for an AT90S8515 AVR processor is shown in Figure.

Atmel AT90S8515 memory map



Atmel provides the following sample C code, which it compiled and ran on several different processors:

```
int max(int *array)

{
  char a;
  int maximum = -32768;
  for (a = 0; a < 16; a++)
    if (array[a] > maximum)
      maximum = array[a];
  return (maximum);
}
```

The results are interesting (Table).

Atmel's comparison of processor speed and efficiency

Processor	**Compiled code size**	**Execution time (cycles)**
AVR	46	335
8051	112	9,384
PIC16C74	87	2,492
68HC11	57	5,244

This indicates that, when running at the same clock speed, an AVR is 7 times faster than a PIC16, 15 times faster than a 68HC11, and a whopping 28 times faster than an 8051. Alternatively, you'd have to have an 8051 running at 224 MHz to match the speed of an 8 MHz AVR. Now, Atmel doesn't give specifics of which compiler(s) it used for the tests, and it is certainly possible to tweak results one way or the other with appropriately chosen source code. However, my personal experience is that with the AVR, you certainly do get significantly denser code and much faster execution.

There are three basic families within the AVR architecture. The original family is the AT90xxxx. For complex applications, there is the ATmega family, and for small-scale use, there's the ATtiny family. Atmel also produces large FPGAs (Field-programmable Gate Arrays), which incorporate an AVR core along with many tens of thousands of gates of programmable logic.

For software development, a port of gcc is available for the AVR, and Atmel provides an assembler, simulator, and software to download programs into the processors. The Atmel software is freely available on its web site. The low-cost Atmel development system is a good way of getting started with the AVR. It provides you with the software and tools you need to begin AVR development.

The AVR processors at which we'll be looking are the small ATtiny15, the AT90S8535/AT90S4434, and the AT90S8515.

The ATtiny15 Processor

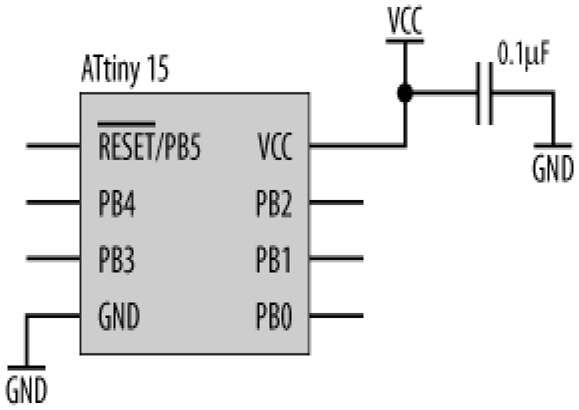
For many simple digital applications, a small microprocessor is a better choice than discrete logic, because it is able to execute software. It is therefore able to perform certain tasks with much less hardware complexity. You'll see just how easy it is to produce a small, embedded computer for integration into a larger system using an Atmel ATtiny15 AVR processor. This processor has 512 words of flash for program storage and no RAM! (Think of that the next time you have to install some 100 MB application on your desktop computer!) This tiny processor, unlike its bigger AVR siblings, relies solely on its 32 registers for working variable storage.

Since there is no RAM in which to allocate stack space, the ATtiny15 instead uses a dedicated hardware stack that is a mere three entries deep, and this is shared by subroutine calls and interrupts. (That fourth nested function call is a killer!) The program counter is 9 bits wide (addressing 512 words of program space), and therefore the stack is also 9 bits wide. Also, unlike the bigger AVRs, only two of the registers (R30 and R31) may be coupled as a 16-bit index register (called "Z").

The processor also has 64 bytes of EEPROM (for holding system parameters), up to five general-purpose I/O pins, eight internal and external interrupt sources, two 8-bit timer/counters, a four-channel 10-bit analog-to-digital converter, an analog comparator, and the ability to be reprogrammed in-circuit. It comes in a tiny 8-pin package, out of which you can get up to 8 MIPS performance. We're not going to worry about most of its features for the time being. That will all be covered in later chapters when we take a look at the I/O features of some larger AVRs. Instead, we're just going to concentrate on how you use one for simple digital control.

Using a small microcontroller such as the ATtiny15 is very easy. The basic processor needs very little external support for its own operation. Figure shows just how simple it is.

A simple AVR computer



Let's take a quick run-through of the design (what there is of it). VCC is the power supply. It can be as low as 2.7 V or as high as 5.5 V. VCC is decoupled to ground using a 0.1 uF capacitor. The five pins, PB0 through PB4, can act as digital inputs or outputs. They can be used to read the state of switches, turn external devices on or off, generate waveforms to control small motors, or even synthesize an interface to simple peripheral chips. The digital I/O lines, PB0 through PB4, get connected to whatever you're using the processor to monitor or control. We'll look at some examples of that later in the chapter.

Finally, one input, RESET, is left unconnected. On just about any other processor, this would be fatal. Many processors require an external power-on reset (POR) circuit to bring them to a known state and to commence the execution of software. Some processors have an internal power-on reset circuit and require no external support. Such processors still have a reset input, allowing them to be manually reset by a user or external system. Normally, the reset input still requires a pull-up resistor to hold it inactive. But the ATtiny15 processor doesn't require this. It has an internal power-on reset and an internal pull-up resistor. So, unlike most (all) other processors, RESET on the ATtiny15 may be left unconnected. In fact, on this particular processor, the RESET pin may be utilized as a general-purpose input (PB5) when an external reset circuit is not required. One important point: the normal input protection against higher-than-normal voltage inputs is not present on RESET /PB5, since it may be raised to +12 V during software download by the program burner. Therefore, if you are using PB5, you must take great care to ensure that the input never exceeds VCC by more than 1 V. Failing to do so may place the processor into software-download mode and thereby effectively crash your embedded computer.

The AVR processors (and PICs too) include an internal circuit known as a brownout detector (BOD). This detects minor fluctuations on the processor's power supply that may corrupt its operation, and if such a fluctuation is detected, it generates a reset and restarts the processor. There is also an additional reset generator, known as a watchdog, used to restart the computer in case of a software crash. It is a small timer whose purpose is to automatically reset the processor once it times out. Under normal operation, the software regularly restarts the watchdog. It's a case of "I'll reset you before you reset me." If the software crashes, the watchdog isn't cleared and thus times out, resetting the computer. Processors that incorporate watchdogs normally give software the ability to distinguish between a power-on reset and a watchdog reset. With a watchdog reset, it may be possible to recover the system's state from memory and resume operation without complete re-initialization.

Now the other curious aspect of the above design is that there is no clock circuit. The ATtiny15 can have an external crystal circuit. (On the ATtiny15, PB3 and PB4 function as the crystal inputs, XTAL1 and XTAL2). But our design doesn't have a crystal, or even need one. The reason is that this little processor includes a complete internal oscillator (in this case, an RC oscillator) running at a frequency of 1.6 MHz and so requires no external components for its clock. The catch is that RC oscillators are not that stable and have the tendency to vary their frequency as the temperature changes. (The ATtiny15's oscillator can vary between 800 kHz and 1.6 MHz.) Generally, an RC oscillator is not really suitable for timing-critical applications (in which case, you'd use an external crystal instead). But if your ATtiny15 is just doing simple control functions, timing may not be an issue. You can therefore get by with using the internal RC oscillator and save on complexity. Atmel provides an 8-bit calibration register (OSCCAL) in the ATtiny15 that enables you to tune the internal oscillator, thus making it more accurate.

There we have the basic design for an ATtiny15 machine. In essence, it's a very cheap, very small, versatile computer that requires no work for the core design. The only design effort needed is to ensure that the computer will work correctly with the I/O devices to which it is interfaced. If you're going to power the system off a battery, then the capacitor is optional as well! The only component that must be there is the processor itself. (And you thought designing computer hardware was going to be hard!)

So, that's the basic AVR computer hardware with minimal components. We'll look at how you download code to it shortly.

That covers the basics of a ATtiny15 system, and it's not that much different from the corresponding PIC12C805 computer. The real differences lie in their internal architectures (and instruction sets) and in the subtleties of their operating voltages and interfacing capabilities. As you can see, there's not a lot of hard work involved in putting one of these little machines into your embedded system.

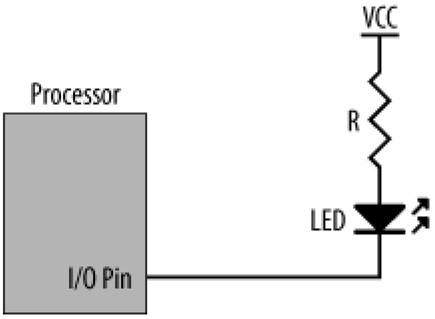
So far, our computer isn't interfaced to anything. Let's start with something simple by adding a LED to the AVR. The basic technique applies to all microcontrollers with programmable I/O lines as well.

Adding a Status LED

LEDs produce light when current flows through them. Being a diode, they conduct only if the current is flowing in the right direction, from anode (positive) to cathode (negative). The cathode end of a LED is denoted on a schematic by a horizontal bar. The anode is a triangle.

The circuit for a status LED is shown in Figure. It uses an I/O line of the microcontroller to switch the LED on or off. Sending it low will turn on the LED. Sending it high will turn off the LED, as we'll soon see. The resistor (R) is used to limit the current sinking into the I/O line, as we shall also see shortly.

Status LED



When conducting (and thereby producing light), LEDs have a forward voltage drop, meaning that the voltage present at the cathode will be less than that at the anode. The magnitude of this voltage drop varies between different LED types, so check the datasheet for the particular device you are using.

The output low voltage of an ATtiny15 I/O pin is 0.6 V when the processor is operating on a 5 V supply and 0.5 V when operating on a 3 V supply. Let's assume (for the sake of this example) that we are using a power supply (VCC) of 5 V, and the LED has a forward voltage drop of 1.6 V. Now, sending the output low places the LED's cathode at 0.6 V. This means the voltage difference between VCC (5 V) and the cathode is 4.4 V. If the LED has a voltage drop of 1.6 V, this means the voltage drop across the resistor is 2.8 V (5 V - 1.6 V - 0.6 V = 2.8 V).

Now, from the datasheet, the digital I/O pins of an AVR can sink up to 20 mA if the processor is running on a 5 V supply. We therefore have to limit the current flow to this amount, and this is the purpose of the resistor. If the resistor has a voltage difference across it of 2.8 V (as we calculated) and a current flow of 20 mA, then from Ohm's Law we can calculate what value resistor we need to use:

R = V / I
= 2.8 V / 20 mA
= 140 □

The closest available resistor value to this is 150 □, so that's what we'll use. (This will give us an actual current of 18.6 mA, which is fine.)

The next question is, how much power will the resistor have to dissipate? In other words, how much energy will it use in dropping the voltage by 2.9 V? This is important because if we try to pump too much current through the resistor, we'll burn it out. We thus need to choose a resistor with a power rating greater than that required. Power is calculated by multiplying voltage by current:

P = V * I
= 2.8 V * 20 mA
= 0.056 Watts = 56 mW

That's negligible, so the resistor value we need for R is 150 □ and 0.0625 W. (0.0625 W is the lowest power rating commonly available in resistors.)

So, what happens when the I/O line is driven high? The AVR I/O pins output a minimum of 4.3 V when high (and using a 5 V supply). With the output high, the voltage at the LED's cathode will be at least 4.3 V, so the voltage difference between the cathode and VCC will be only 0.7 V (or less). But the forward voltage drop of the LED is 1.6 V. Thus, there is not enough voltage across the LED to turn it on.

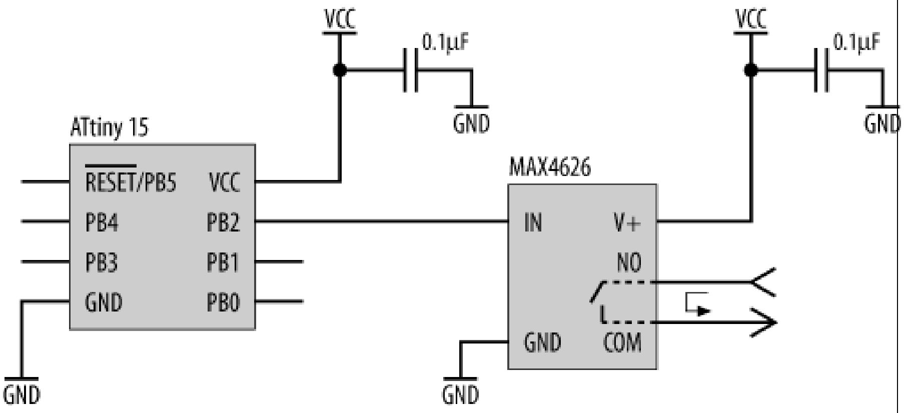
In this way, we can turn the LED on or off using a simple digital output of the processor. We have also seen how to calculate voltages and currents. It is very important to do this with every aspect of a design. Ignoring it can result in a nonfunctioning machine or, worse, charred components and that wafting smell of burning silicon.

We've just seen how to use the digital outputs of the AVR to control a LED. This will work with any device that uses less than 20 mA. In fact, for low-power components, such as some sensors, it is possible to use the AVR's output to provide direct power control, just as we provided direct power control for the LED. In battery-powered applications, this can be a useful technique for reducing the system's overall power consumption.

Switching Analog Signals

We can also use the digital I/O lines of the processor to control the flow of analog signals within our system. For example, perhaps our embedded computer is integrated into an audio system and is used to switch between several audio sources. To do this, we use an analog switch such as the MAX4626, one for each signal path. This tiny component (about the size of a grain of rice in the surface-mount version) operates from a single supply voltage (as low as 1.8 V and as high as 5.5 V). It also incorporates inbuilt overload protection to prevent device damage during short circuits. The schematic showing a MAX4626 interfaced to an ATtiny15 AVR is shown in Figure. Driving the AVR's output (PB2) high turns on the MAX4626 and makes a connection between NO and COM. Sending PB2 low breaks the connection. In this way, the MAX4626 can be used to connect an output to an input, under software control.

Switching an analog signal



The question is: will it work with an AVR? When operating on a 5 V supply, the input to the MAX4626 (pin 4, IN) requires a logic-low input of less than 0.8 V and a logic-high input of at least 2.4 V. The AVR's logic-low output is 0.6 V or less, and its logic-high output is a minimum of 4.3 V. So, the AVR's digital output voltages match the requirements of the MAX4626. As for current, the MAX4626 needs to sink or source only a miniscule 1 A. For an AVR, this is not a problem.

If the MAX4626 doesn't meet your needs, MAXIM and other manufacturers produce a range of similar devices with varying characteristics. There's bound to be something that meets your needs.

The schematic in Figure includes a push-button connected to PB3, where PB3 is acting as a digital input. Now, there are a couple of interesting things to note about this simple input circuit. The first is that there is no external pull-up resistor attached to PB3. Normally, for such a circuit, an external pull-up resistor is required to place the input into a known state when the button is open (not being pressed). The pull-up resistor takes the input high, except when the button is closed and the input is connected directly to ground. The reason we can get away with not having an external pull-up resistor is that the AVR incorporates internal pull-up resistors, which may be enabled or disabled under software control.

The second interesting thing to note is that there is no debounce circuitry between the button and the input. Any sort of mechanical switch (and that includes a keyboard key) acts as a little inductor when pressed. The result is a rapid ringing oscillation on the signal line that quickly decays away (Figure).

So, instead of a single change of state, the resulting effect is as if the user had been rapidly hammering away on the button. Software written to respond to changes in this input will register the multiple pulses, rather than the single press the user intended. Removing these transients from the signal is therefore important and is known as debouncing. Now, there are several different circuits you could include that will cleanly remove the ringing. But here's the thing: you don't always need to!

Figure. Push-button input

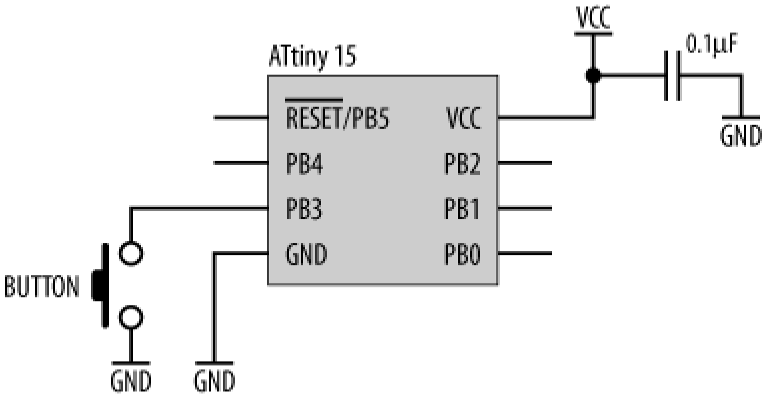
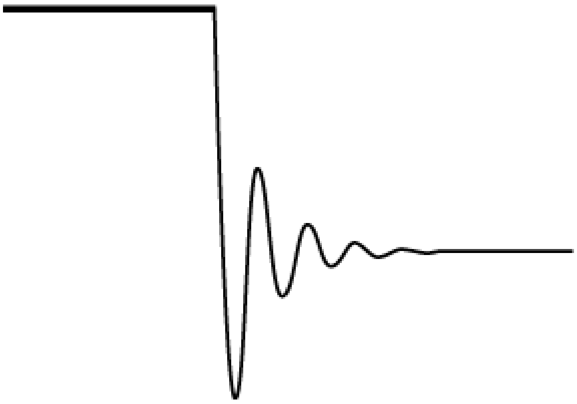


Figure . Signal bounce



When a user presses a button, he will usually hold that button closed for at least half a second, maybe more, by which time the ringing has died away. The problem can therefore be solved in software. The software, when it first registers a low on the input, waits for a few hundred milliseconds and then samples the input again (perhaps more than once). If it is still low, then it is a valid button press, and the software responds. The software then "re-arms" the input, awaiting the next press. Debouncing hardware does become important, however, if the button is connected to an interrupt line or reset.

So far, we have seen how to use the AVR to control digital outputs and read simple digital inputs. The astute among you may ask, "When looking at the previous two circuits, why do we need the processor?" After all, it is certainly possible to connect the button directly to the input of the MAX4626. Of what use can the processor be? Well, we've already seen one use. The processor can replace debounce circuitry on the input. Since it has internal memory and the ability to execute software, the processor can also keep track of system state (and mode), monitor various inputs in relation to each other, and provide complicated control sequencing on the outputs. In short, the inclusion of a microprocessor can reduce hardware complexity while increasing system functionality. They can be very useful tools. With more advanced processors, and with more diverse I/O, the functionality and usefulness of an embedded computer can be significant.

Downloading Code

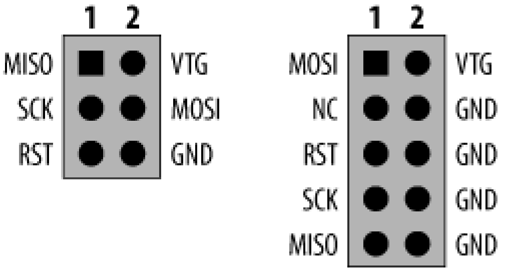
The AVR processors use internal flash memory for program storage, and this may be programmed in-circuit or, in the case of socketed components, out of circuit as well. The AVR processors are reprogrammed via a SPI port on the chip. Even AVR processors such as the ATtiny15, which do not have a SPI interface for their own use, still incorporate a SPI port for reprogramming. The pins PB0, PB1, and PB2 take on SPI functions (MOSI, MISO, and SCK) during programming.

VCC can be supplied by the external programmer downloading the code. For programming, VCC must be 5 V. If the embedded system's local supply will provide 5 V, then the connection to the programmer's VCC may be left unmade. However, if the embedded system's supply voltage is something other than 5 V, the programmer's VCC must be used, and any local power source within the embedded system should be disabled. RESET plays an important role in downloading code. Programming begins with RESET being asserted (driven low). This disables the CPU within the processor and thus allows access to the internal memory. It also changes the functionality of PB0, PB1, and PB2 to a SPI interface. The development software then sends, via the SPI interface, a sequence of codes to "unlock" the program memory and enable software to be downloaded. Once programming is enabled, sequences of write commands are performed, and the software (and other settings) are downloaded byte by byte. The Atmel software takes care of this, so normally you don't need to worry about the specifics. If you need to do it "manually," perhaps from some other type of host computer, the Atmel datasheets give full details of the protocol.

The Atmel development system comes with a special adaptor cable that plugs into the company's development board and allows you to reprogram microprocessors via a PC's parallel port. By including the right connector (with the appropriate connections) in your circuit, it's possible to use the same programming cable on your own embedded system. Depending on the particular development board, there is one of two possible connectors for in-circuit programming. The pinouts for these are shown in Figure. (VTG is the voltage supply for the target system. If the target has its own power source, of the appropriate voltage level for programming (+5 V), then VTG may be left unconnected.) Pin 3 is labeled as a non connect on some Atmel application notes; however, some development systems use this to drive a LED indicating that a programming cycle is underway.

The schematic showing how to make your computer support this is shown in Figure. Note that MOSI on the connector goes to MISO on the processor, and, similarly, MISO goes to MOSI on the processor. This is because during programming, the processor is a slave and not a master.

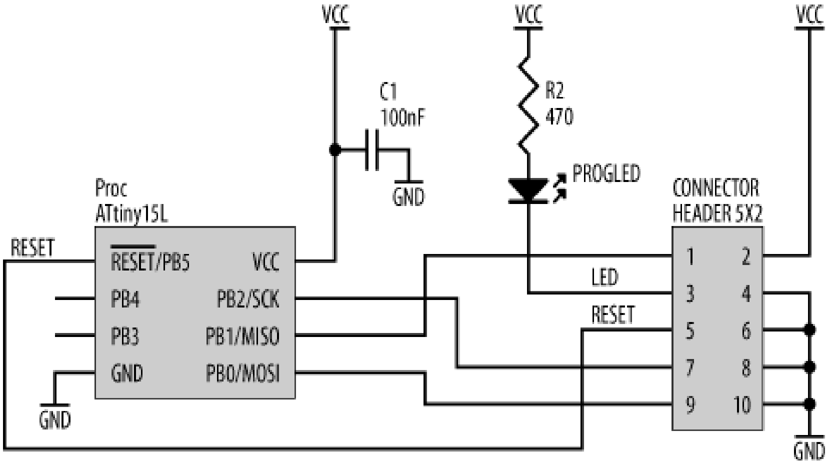
Figure. In-circuit programming connectors



The connector type is an IDC header, and the cable provides all the signals necessary for programming, including one to drive a programming indicator LED. When not being used for programming, the connector may also double as a simple I/O connector for the embedded computer, allowing access to the digital signals. Thus, one piece of hardware can assume dual roles.

There is something important to note, however. If you use PB0, PB1, or PB2 to interface to other components within your computer, care must be taken that the activity of programming does not adversely affect them. For example, our circuit with the MAX4626 used PB2 as the control input. During programming, PB2 acts as SCK, a clock signal. Therefore, the MAX4626 would be rapidly turning on and off as code was downloaded to the processor. If the MAX4626 was controlling something, that device would also rapidly turn on and off, with potentially disastrous effects. Conversely, if there are other components in your system, these must not attempt to drive a signal onto PB0, PB1, or PB2 during the programming sequence. To do so would, at the very least, result in a failed download and, at worst, damage both the embedded system and the programmer. It's therefore vitally important to consider the implications of in-circuit programming on other components within the system.

In-circuit programming



There is something important to note, however. If you use PB0, PB1, or PB2 to interface to other components within your computer, care must be taken that the activity of programming does not adversely affect them. For example, our circuit with the MAX4626 used PB2 as the control input. During programming, PB2 acts as SCK, a clock signal. Therefore, the MAX4626 would be rapidly turning on and off as code was downloaded to the processor. If the MAX4626 was controlling something, that device would also rapidly turn on and off, with potentially disastrous effects. Conversely, if there are other components in your system, these must not attempt to drive a signal onto PB0, PB1, or PB2 during the programming sequence. To do so would, at the very least, result in a failed download and, at worst, damage both the embedded system and the programmer. It's therefore vitally important to consider the implications of in-circuit programming on other components within the system.

So, what's the answer? Well, we could use PB3 to control the MAX4626 instead, since it doesn't take part in the programming process. Alternatively, if we needed to use PB2, we could provide a buffer between the processor and the MAX4626, perhaps controlled by . When is low (during programming), the buffer is disabled and the MAX4626 is isolated. Another solution may simply be to use a DIP version of the processor, mounted via a socket, and physically remove it for reprogramming. If you're using a surface-mount version of the processor, perhaps the processor could be mounted on a small PCB that plugs into the embedded computer (much like a memory SIMM on a desktop computer) and may be removed for programming. There are plenty of alternatives, and the best one really depends on your application.

Some AVRs (not the ATtiny15) have the capability of modifying their own program memory with the SPM (Store Program Memory) instruction. With such processors, it is possible for your software to download new code via the processor's serial port and write this into the program memory. To do so, you need to have your processor preprogrammed with a boot loader program. Normally, you would load all your processors with the boot loader (and Version 1.0 of your application software) during construction. The self-programming can then be used to update the application software when the systems are out in the field. To facilitate this, the program memory is divided into two separate sections: a boot section and an application section. The memory space is divided into pages of either 128 or 256 bytes (depending on the particular processor). Memory must be erased and reprogrammed one page at a time. During programming, the Z register is used as a pointer for the page address, and the r1 and r0 registers together hold the data word to be programmed. The Atmel application note (AVR109: Self-programming), available on its web site, gives sample source code for the boot loader and explains the process in detail.

No matter what processor you are using, the technical data from the chip manufacturer will tell you how to go about putting your code into the processor.

A Bigger AVR

So far, we have looked at a small AVR with very limited capabilities. In later chapters, we will look at various forms of input and output commonly found in embedded systems. For this, we will need processors with more functionality. We have exhausted the ATtiny15, so now we need to move on to processors with a bit more "grunt." Before getting into the details of I/O in the later chapters, you'll be introduced to these processors and learn what you need to do to include them in your design.

The first processor is the Atmel AT90S8535. This is a mid-range AVR with lots of inbuilt I/O, such as a UART, SPI, timers, eight channels of analog input, an analog comparator, and internal EEPROM for parameter storage. The processor has 512 bytes of internal RAM and 8K of flash memory for program storage. Its smaller sibling, the AT90S4434, is identical in every other way except that it has smaller memory spaces of 4K for program storage and 256 bytes of RAM. But from a hardware point of view, the AT90S8535 and the AT90S4434 are the same.

The basic schematic for an AT90S8535-based computer, without any extras, is shown in Figure. It is not that different from the ATtiny15, save that it has a lot more pins as an external pull-up 10k resistor. The processor has an external crystal (X1), and this requires two small decoupling capacitors, C1 and C2. There are four power pins for this processor, and each is decoupled with a 100 nF ceramic capacitor. One of the power inputs (AVCC) is the power supply for the analog section of the chip, and this is isolated from the digital power supply by a 100 □ resistor, R2. This is to provide a small barrier between the analog section and any switching noise that may be present from the digital circuits. The remaining pins are general-purpose digital I/O, as with the ATtiny15. However, unlike the ATtiny15, these pins have dual functionality. They may be configured, under software control, for alternative I/O functions. The processor's datasheet gives full details for configuring the functionality of the processor under software control.

This basic AVR design is applicable to most AVRs that you will find. The pin outs may be different, but the basic support required is the same. As with everything, grab the appropriate datasheet, and it will tell you the specifics for the particular processor that you are using.

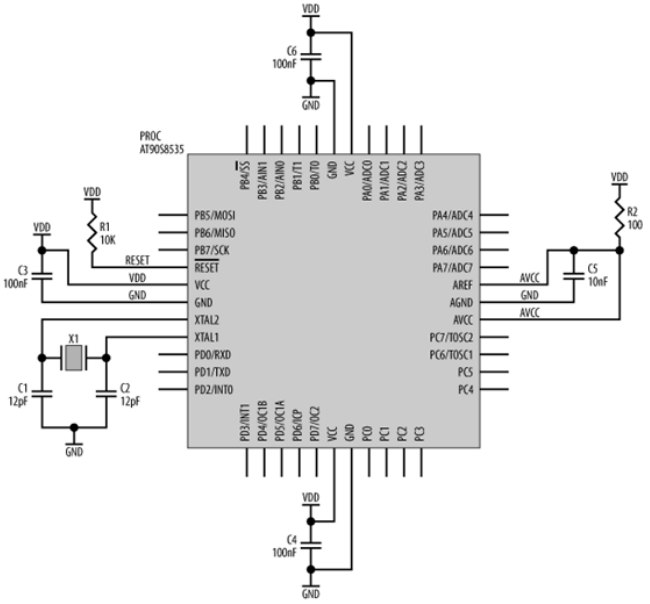
AVR-Based Data logger

In the previous chapter, we saw how to design a data logger based on a PIC16F873A. A data logger based on an AVR is not too dissimilar. Figure shows the basic schematic.

The connections for interfacing a serial data flash memory chip to an Atmel 90S4434 AVR processor are simply SPI, as with the PIC processor. The AVR portion of the schematic is no different from the examples we have seen previously. That's the nice thing about simple interfaces such as SPI. They form little subsystem modules that "bolt together" like building blocks. Start with the basic core design and just add peripherals as you need them. The schematic also shows decoupling capacitors for the power supplies, the crystal oscillator for the processor, and a pull-up resistor for RESET. Pin 41 (PB1) is used as a "manual" (processor-controlled) reset input to the flash.

The analog inputs, ADC0:ADC7 can be connected to an IDC header allowing for external sampling, or they can be interfaced directly to sensors, as we saw with the PIC data logger. The serial port signals, RXD and TXD, connect to a MAX3233 in the same way as we saw in the PIC design.

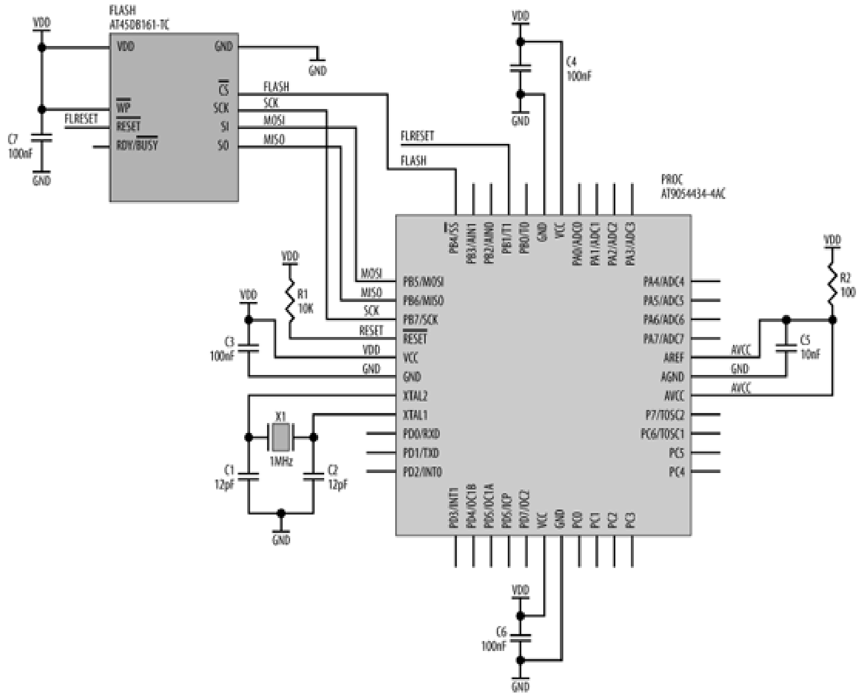
AT90S8535 processor and support components



Bus Interfacing

In this section, I'll show you how to expand the capabilities of your processor by interfacing it to bus-based memories and peripherals. Different processor architectures have different signals and different timing, but once you understand one, the basic principles can be applied to all. Since most small microcontrollers don't have external buses, the choice is very limited. We'll look at the one, and only, AVR with an external bus—the AT90S8515. In the PIC world, the PIC17C44 is capable of bus-based interfacing.

A 2M Data Flash interfaced to an AT90S4434



AT90S8515 Memory Cycle

A memory cycle (also known as a machine cycle or processor cycle) is defined as the period of time it takes for a processor to initiate an access to memory (or peripheral), perform the transfer, and terminate the access. The memory cycle generated by a processor is usually of a fixed period of time (or multiples of a given period) and may take several (processor) clock cycles to complete.

Memory cycles usually fall into two categories: the read cycle and the write cycle. The memory or device that is being accessed requires that the data is held valid for a given period after it has been selected and after a read or write cycle has been identified. This places constraints on the system designer. There is a limited amount of time in which any glue logic (interface logic between the processor and other devices) must perform its function, such as selecting which external device is being accessed. The setup times must be met. If they are not, the computer will not function. The glue logic that monitors the address from the processor and uniquely selects a device is known as an address decoder. We'll take a closer look at address decoders shortly.

Timing is probably the most critical aspect of computer design. For example, if a given processor has a 150 ns cycle time and a memory device requires 120 ns from when it is selected until when it has completed the transfer, this leaves only 30 ns at the start of the cycle in which the glue logic can manipulate the processor signals. A 74LS series TTL gate has a typical propagation delay of 10 ns. So, in this example, an address decoder implemented using any more than two 74LS gates (in sequence) is cutting it very fine.

A synchronous processor has memory cycles of a fixed duration, and all processor timing is directly related to the clock. It is assumed that all devices in the system are capable of being accessed and responding within the set time of the memory cycle. If a device in the system is slower than that allowed by the memory cycle time, logic is required to pause the processor's access, thus giving the slow device time to respond. Each clock cycle within this pause is known as a wait state. Once sufficient time has elapsed (and the device is ready), the processor is released by the logic and continues with the memory cycle. Pausing the processor for slower devices is known as inserting wait states. The circuitry that causes a processor to hold is known as a wait-state generator. A wait-state generator is easily achieved using a series of flip-flops acting as a simple counter. The generator is enabled by a processor output indicating that a memory cycle is beginning and is normally reset at the end of the memory cycle to return it to a known state. (Some processors come with internal, programmable wait-state generators.)

An asynchronous processor (such as a 68000) does not terminate its memory cycle within a given number of clock cycles. Instead, it waits for a transfer acknowledge assertion from the device or support logic to indicate that the device being accessed has had sufficient time to complete its part in the memory cycle. In other words, the processor automatically inserts wait states in its memory cycle until the device being accessed is ready. If the processor does not receive an acknowledge, it will wait indefinitely. Many computer systems using asynchronous processors have additional logic to cause the processor to restart if it waits too long for a memory cycle to terminate. An asynchronous processor can be made into a synchronous processor by tying the acknowledge line to its active state. It then assumes that all devices are capable of keeping up with it. This is known as running with no wait states.

Most microcontrollers are synchronous, whereas most larger processors are asynchronous. The AT90S8515 is a synchronous processor, and it has an internal wait-state generator capable of inserting a single wait state.

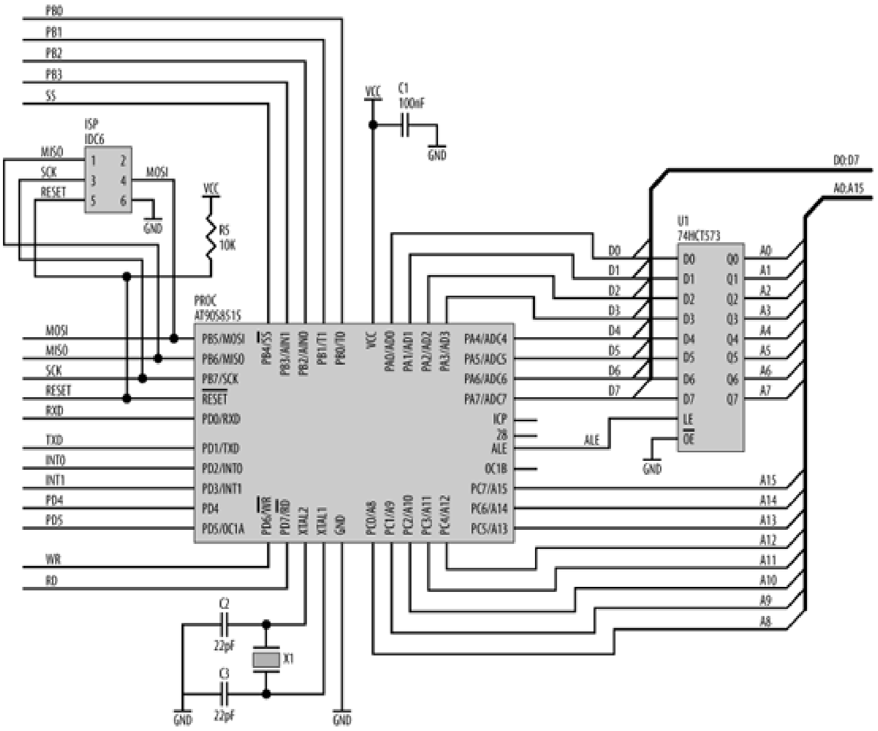
Bus Signals

Figure shows an AT90S8515 processor with minimal support components. The AT90S8515 has an address bus, a data bus, and a control bus that it brings to the outside world for interfacing. Since this processor has a limited number of pins, these buses share pins with the digital I/O ports ("port A" and "port C") of the processor. A bit in a control register determines whether these pins are I/O or bus pins. Now, a 16-bit address bus and an 8-bit data bus add up to 24 bits, but ports A and B have only 16 bits between them. So how does the processor fit 24 bits into 16? It multiplexes the lower half of the address bus with the data bus. At the start of a memory access, port A outputs address bits A0:A7. The processor provides a control line, ALE (Address Latch Enable), which is used to control a latch, such as a 74HCT573 (shown on the right in Figure). As ALE falls, the latch grabs and holds the lower address bits. At the same time, port B outputs the upper address bits, A8:A15. These are valid for the entire duration of the memory access. Once the latch has acquired the lower address bits, port A then becomes the data bus for information transfer between the processor and an external device. Also shown in Figure are the crystal circuit, the In-System Programming port, decoupling capacitors for the processor's power supply, and net labels for other important signals.

The timing diagrams for an AT90S8515 are shown in Figure. The cycle "T3" exists only when the processor's wait-state generator is enabled.

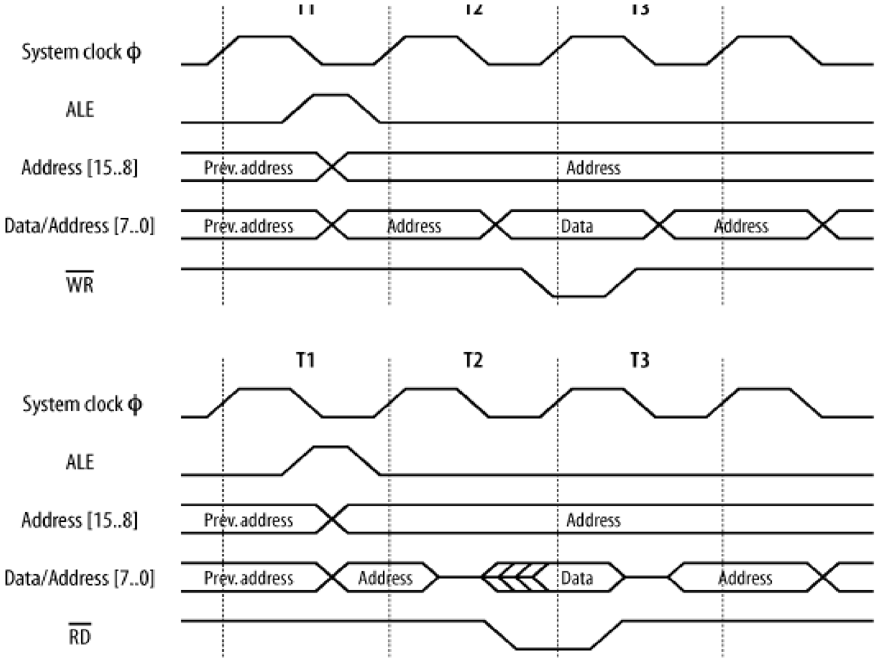
Now, let's look at these signals in more detail. (We'll see later how you actually work with this information. For the moment, we're just going to "take a tour" of the timing diagrams.) The numbers for the timing information can be found in the datasheet, available from Atmel's web site. Figure shows the timing information as presented in the Atmel datasheet, complete with timing references.

Address bus demultiplexing

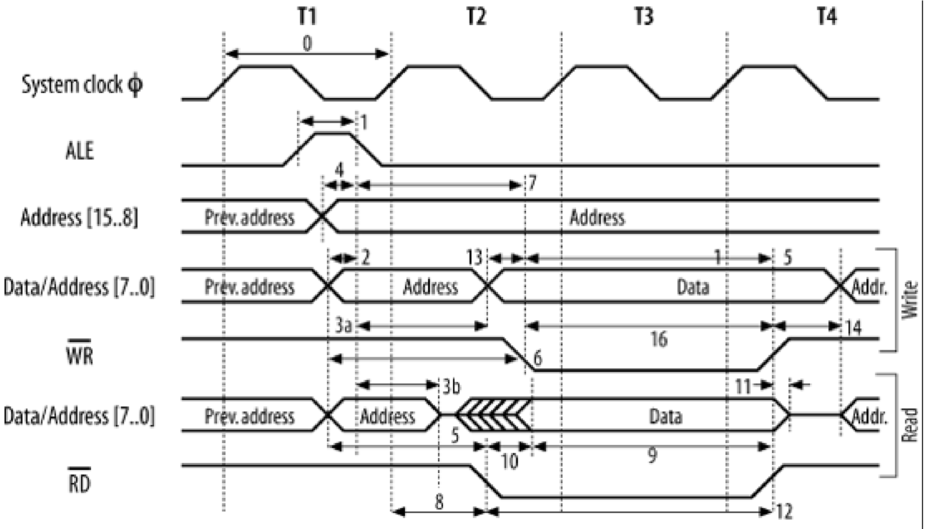


A

T90S8515 memory cycles



AT90S8515 memory cycles with timing parameters



The references are looked up in the appropriate table in the processor's datasheet (Table)

8 MHz oscillator		Variable oscillator					
Ref. #	**Symbol**	**Parameter**	**Min**	**Max**	**Min**	**Max**	**Unit**
0	$^1/t_{CLCL}$	Oscillator Frequency			0.0	8.0	MHz
1	t_{LHLL}	ALE Pulse Width	32.5		0.5 t_{CLCL}-30.0		ns
2	t_{AVLL}	Address Valid A to ALE Low	22.5		0.5 t_{CLCL}-40.0		ns
3a	$t_{LLAX...ST}$	Address Hold after ALE Low, ST/STD/STS Instructions	67.5		0.5 t_{CLCL}+5.0		ns
3b	$t_{LLAX...LD}$	Address Hold after ALE Low, LD/LDD/LDS Instructions	15.0		15.0		ns
4	t_{AVLLC}	Address Valid C to ALE Low	22.5		0.5 t_{CLCL}-40.0		ns
5	t_{AVRL}	Address Valid to RD Low	95.0		1.0 t_{CLCL}-30.0		ns
6	t_{AVWL}	Address Valid to WR Low	157.5		1.5 t_{CLCL}-30.0		ns
7	t_{LLWL}	ALE Low to WR Low	105.0	145.0	1.0 t_{CLCL}-20.0	1.0 t_{CLCL}+20.0	ns

8 MHz oscillator		Variable oscillator						
Ref. #	**Symbol**	**Parameter**	**Min**	**Max**	**Min**	**Max**	**Unit**	
8	t_{LLRL}	ALE Low to RD Low	42.5	82.5	0.5 t_{CLCL}-20.0	0.5 t_{CLCL}+20.0	ns	
9	t_{DVRH}	Data Setup to RD High	60.0		60.0		ns	
10	t_{RLDV}	Read Low to Data Valid		70.0		1.0 t_{CLCL}-55.0	ns	
11	t_{RHDX}	Data Hold after RD High	0.0		0.0		ns	
12	t_{RLRH}	RD Pulse Width	105.0		1.0 t_{CLCL}-20.0		ns	
13	t_{DVWL}	Data Setup to WR Low	27.5		0.5 t_{CLCL}-35.0		ns	
14	t_{WHDX}	Data Hold after WR High	0.0		0.0		ns	
15	t_{DVWH}	Data Valid to WR High	95.0		1.0 t_{CLCL}-30.0		ns	
16	t_{WLWH}	WR Pulse Width	42.5		0.5 t_{CLCL}-20.0		ns	

The system clock is shown at the top of both diagrams for reference, since all processor activity relates to this clock. The period of the clock is designated in the Atmel datasheet as "t_{CLCL}" and is equal to 1/frequency. For an 8 MHz clock, this is 125 ns. The width of T1, T2, and T3 are each t_{CLCL}. Datasheet nomenclature can often be very cryptic. The "CL" comes from "clock." Since Atmel is using four-character subscripts for its timing references, it pads by putting "CL" twice. You don't really need to know what the subscripts actually mean; you just need to know the signals they refer to and the actual numbers involved.

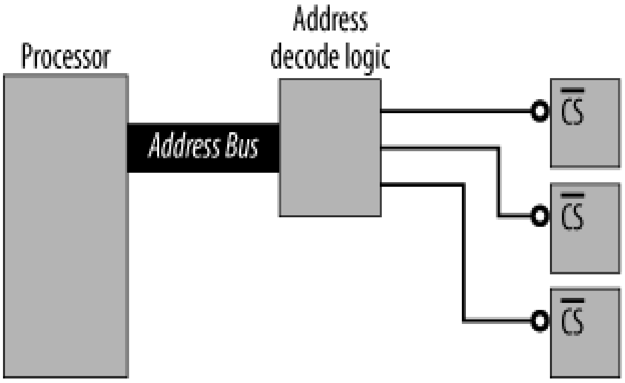
No processor cycle exists in isolation. There is always a preceding cycle and following cycle. We can see this in the timing diagrams. At the start of the cycles, the address from the previous access is still present on the address bus. On the falling edge of the clock, in cycle T1, the address bus changes to become the valid address required for this cycle. Port A presents address bits A0:A7, and port B presents A8:A15. At the same time, ALE goes high, releasing the external address latch in preparation for acquiring the new address from port A. ALE stays high for 0.5 x tCLCL - 30 ns. So, for example, with an AT90S8515 running at 8 MHz, ALE stays high for 32.5 ns. ALE falls, causing the external latch to acquire and hold the lower address bits. Prior to ALE falling, the address bits will have been valid for 0.5 x tCLCL - 40 ns or, in other words, 40 ns before the system clock rises at the end of the T1 period. After ALE falls, the lower address bits will be held on port A for 0.5 x tCLCL + 5 ns for a write cycle before changing to data bits. For a read cycle, they are held for a minimum of 15 ns only. The reason this is so much shorter for a read cycle is that the processor wishes to free those signal pins as soon as possible. Since this is a read cycle, an external device is about to respond, which means the processor needs to "get out of the way" as soon as it can. For a write cycle, tCLCL - 20 ns after ALE goes low, the write strobe,goes low. This indicates to external devices that the processor has output valid data on the data bus. This time allows the external device to prepare to read in (latch) the data.

So, that is how an AT90S8515 expects to access any external device attached to its buses, whether those devices are memory chips or peripherals. But how does it work in practice? Let's look at designing a computer based on an AT90S8515 with some external devices. For this example, we will interface the processor to a static RAM and some simple latches that we could use to drive banks of LEDs. Since we've covered oscillators and in-circuit programming previously, I'll ignore those in this discussion. That doesn't mean you should leave them out of your design!

Memory Maps and Address Decoding

To the processor, its address space is one big, linear region. Although there may be numerous devices within that space, both internal to the processor and external, it makes no distinction between devices. The processor simply performs memory accesses in the address space. It is up to the system designer (that's you) to allocate regions of memory to each device and then provide address-decode logic. The address decoder takes the address provided by the processor during an external access and uniquely selects the appropriate device (Figure). For example, if we have a RAM occupying a region of memory, any address from the processor corresponding to within that region should select the RAM and not select any other device. Similarly, any address outside that region should leave the RAM unselected.

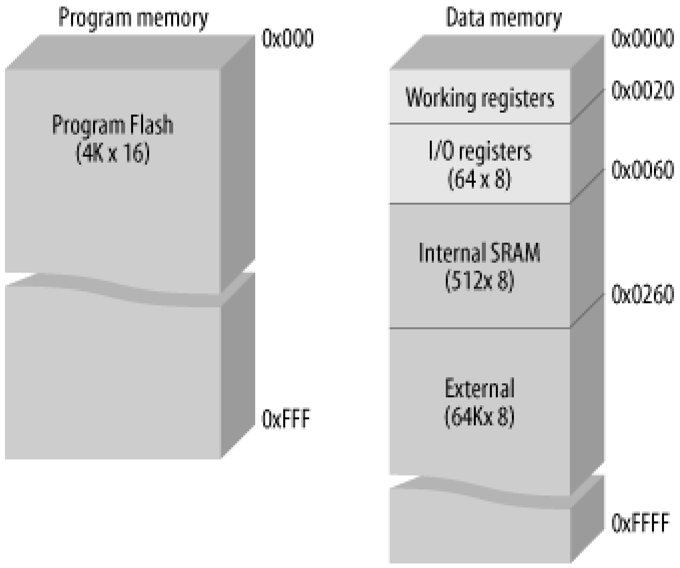
An address decoder uses the address to select one of several devices



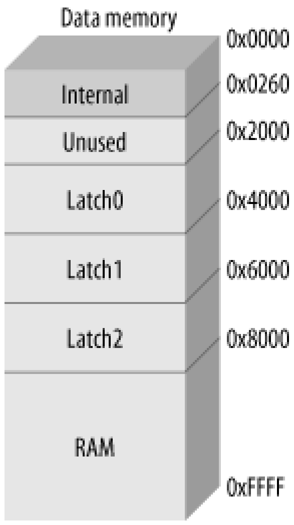
The allocation of devices within an address space is known as a memory map or address map. The address spaces for an AT90S8515 processor are shown in Figure. Any device we interface to the processor must be within the data memory space. Thus, we can ignore the processor's internal program memory. As the processor has Harvard architecture, the program space is a completely separate address space. Within the 64K data space lie the processor's internal resources: the working registers, the I/O registers, and the internal 512 bytes of SRAM. These occupy the lowest addresses within the space. Any address above 0x0260 is ours to play with. (Not all processors have resources that are memory-mapped, and, in those cases, the entire memory space is usable by external devices.)

Now, our first task is to allocate the remaining space to the external devices. Since the RAM is 32K in size, it makes sense to place it within the upper half of the address space (0x8000-0xFFFF). Address decoding becomes much easier if devices are placed on neat boundaries. Placing the RAM between addresses 0x8000 and 0xFFFF leaves the lower half of the address space to be allocated to the latches and the processor's internal resources. Now a latch need only occupy a single byte of memory within the address space. So, if we have three latches, we need only three bytes of the address space to be allocated. This is known as explicit address decoding. However, there's a good reason not to be so efficient with our address allocation. Decoding the address down to three bytes would require an address decoder to use 14 bits of the address. That's a lot of (unnecessary) logic to select just three devices. A better way is simply to divide the remaining address space into four, allocating three regions for the latches and leaving the fourth unused (for the processor's internal resources). This is known as partial address decoding and is much more efficient. The trick is to use the minimal amount of address information to decode for your devices. Our address map allocated to our static RAM and three latches is shown in Figure. Note that the lowest region leaves the addresses in the range 0x0260 to 0x1FFF unused.

Atmel AT90S8515 memory map



Allocated memory map



Any address within the region 0x2000 to 0x3FFF will select Latch0, even though that latch needs only one byte of space. Thus, the device is said to be mirrored within that space. For simplicity in programming, you normally just choose an address (0x2000 say) and use that within your code. But you could just as easily use address 0x290F, and that would work too.

We now have our memory map, and we need to design an address decoder. We start by tabling the devices along with their addresses (Table

15-3). We need to look for which address bits are different between the devices, and which address bits are common within a given device's region.

Device	Address range	A15 .. A0
Unused	0x0000-0x1FFF	0000 0000 0000 0000 0000 0001 1111 1111 1111 1111
Latch0	0x2000-0x3FFF	0010 0000 0000 0000 0000 0011 1111 1111 1111 1111
Latch1	0x4000-0x5FFF	0100 0000 0000 0000 0000 0101 1111 1111 1111 1111
Latch2	0x6000-0x7FFF	0110 0000 0000 0000 0000 0111 1111 1111 1111 1111
RAM	0x8000-0xFFFF	1000 0000 0000 0000 0000 1111 1111 1111 1111 1111

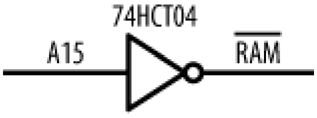
So, what constitutes a unique address combination for each device? Looking at the table, we can see that for the RAM, address bit (and address signal) A15 is high, while for every other device it is low. We can therefore use A15 as the trigger to select the RAM. For the latches, address bits A15, A14, and A13 are critical. So we can redraw our table to make it clearer. This is the more common way of doing an address table, as shown in Table. An "x" means a "don't care" bit.

Device	Address range	A15 .. A0
Unused	0x0000-0x1FFF	000x xxxx xxxx xxxx xxxx
Latch0	0x2000-0x3FFF	001x xxxx xxxx xxxx xxxx
Latch1	0x4000-0x5FFF	010x xxxx xxxx xxxx xxxx
Latch2	0x6000-0x7FFF	011x xxxx xxxx xxxx xxxx
RAM	0x8000-0xFFFF	1xxx xxxx xxxx xxxx xxxx

Therefore, to decode the address for the RAM, we simply need to use A15. If A15 is high, the RAM is selected. If A15 is low, then one of the other devices is selected and the RAM is not. Now, the RAM has a chip select (CS) that is active low. So when A15 is high, CS should go low. So, our address decoder for the RAM is simply to invert A15 using an inverter chip such as a

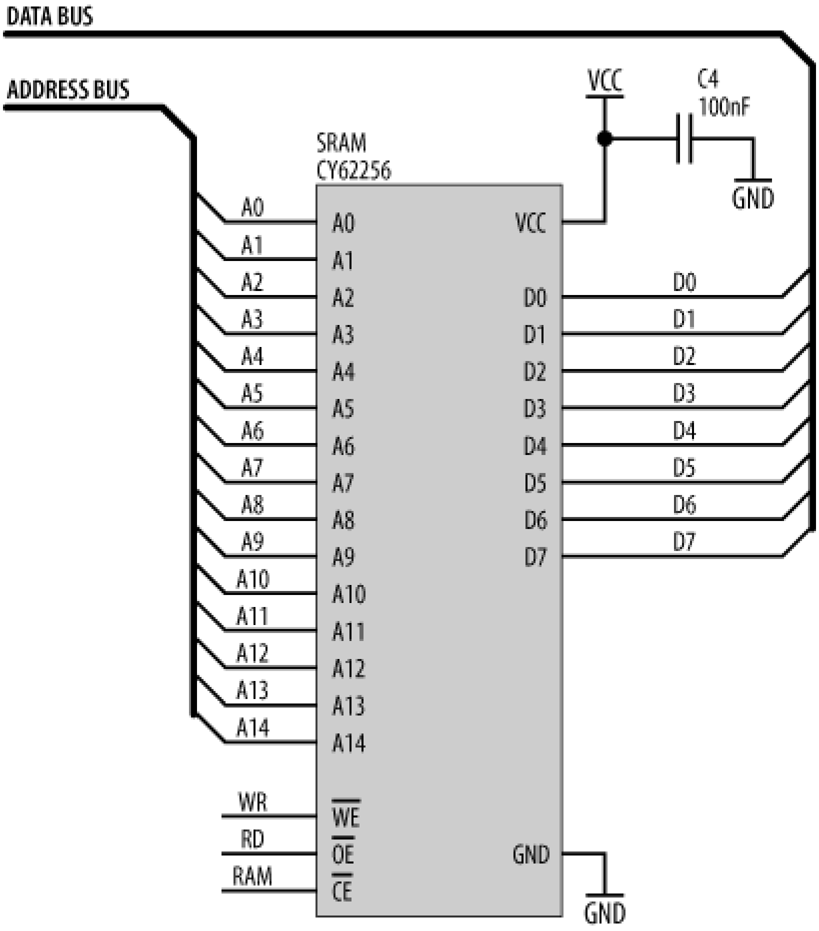
74HCT04 (Figure). It is common practice to label the chip-select signal after the device it is selecting. Hence, our chip select to the RAM is labeled CS.

Address decode for the RAM



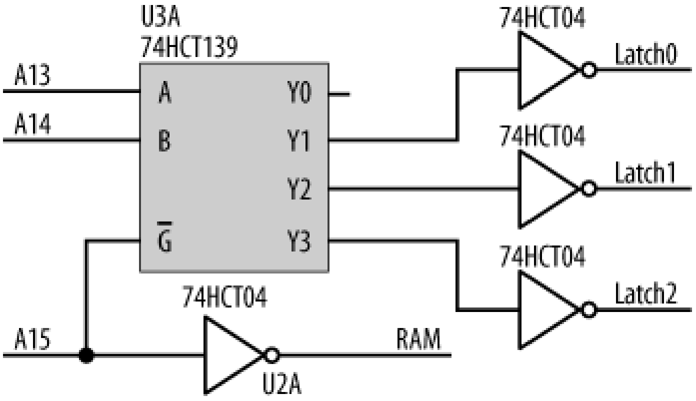
Note that for the RAM to respond, it needs both a chip select and either a read or write strobe from the processor. All other address lines from the processor are connected directly to the corresponding address inputs of the RAM (Figure).

Connections to the SRAM



Now for the other four regions, A15 must be low, and A14 and A13 are sufficient to distinguish between the devices. Having our address decoder use discrete logic would require several gates and would be "messy." There's a simpler way. We can use a 74HCT139 decoder, which takes two address inputs (A and B) and gives us four unique, active-low, chip-select outputs (labeled Y0:Y3). Since the latches require active-high enables, we use inverters on the outputs of the 7HCT139. So our complete address decoder for the computer is shown in Figure.

Complete address decoder



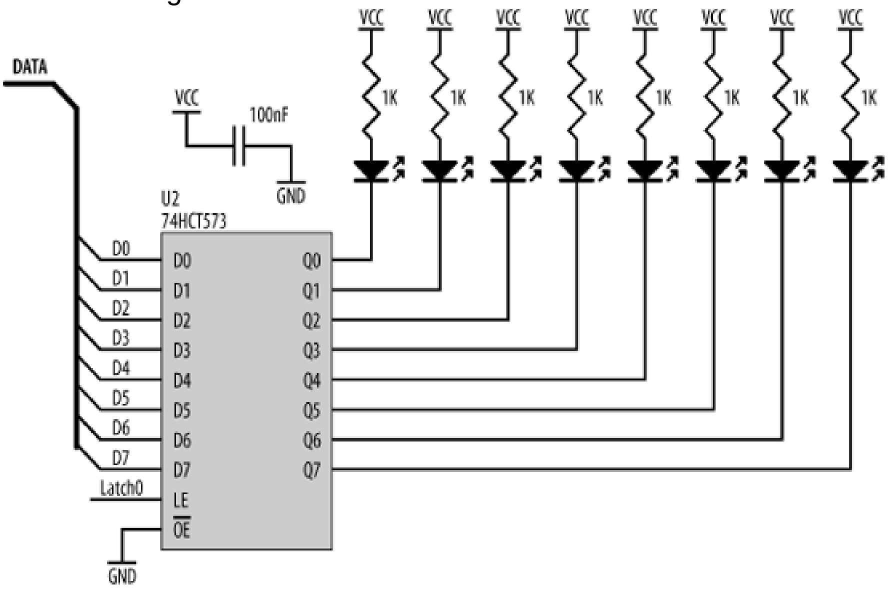
The 74HCT139 uses A15 (low) as an enable (input), and, in this way, A15 is included as part of the address decode. If we needed to decode for eight regions instead of four, we could have used a 74HCT138 decoder, which takes three address inputs and gives us eight chip selects. The interface between the processor and an output latch is simple. We can use the same type of latch (a 74HCT573) that we used to demultiplex the address. Such an output latch could be used in any situation in which we need some extra digital outputs. In the sample circuit shown in Figure, I'm using the latch to control a bank of eight LEDs.

The output from our 74HCT139 address decoder is used to drive the LE (Latch Enable) input of the 74HCT573. Whenever the processor accesses the region of memory space allocated to this device, the address decoder triggers the latch to acquire whatever is on the database. And so, the processor simply writes a byte to any address in this latch's address region, and that byte is acquired and output to the LEDs. (Writing a "0" to a given bit location will turn on a LED; writing a "1" will turn it off.)

Note that the latch's output enable (OE) is permanently tied to ground. This means that the latch is always displaying the byte that was last written to it. This is important, as we always want the LEDs to display, and not just transitorily blink on, while the processor is accessing them.

Using the 74HCT139 in preference to discrete logic gates makes our design much simpler, but there's an even better way to implement system glue.

Using a 74HCT573 latch to control a bank of LEDs



PALs

It is now rare to see support logic implemented using individual gates. It is more common to use programmable logic (PALs, LCAs, or PLDs)[*] to implement the miscellaneous "glue" functions that a computer system requires. Such devices are fast, take up relatively little space, have low power consumption, and, as they are reprogrammable, make system design much easier and more versatile.

There is a wide range of devices available, from simple chips that can be used to implement glue logic (just as we are about to do) to massive devices with hundreds of thousands of gates. These big chips are sophisticated enough to contain entire computer systems. Soft cores are processor designs implemented in gates, suitable for incorporating into these logic devices. You can also get serial interfaces, disk controllers, network interfaces, and a range of other peripherals, all for integration into one of these massive devices. Of course, it's also fun to experiment and design your own processor from the ground up.

Each chip family requires its own suite of development tools. These allow you to create your design (either using schematics or some programming language such as VHDL), simulate the system, and finally download your creation into the chip. You can even get C compilers for these chips that will take an algorithm and convert it, not into machine code, but into gates. What was software now runs not on hardware, but as hardware. Sounds cool, but the tools required to play with this stuff can be expensive. If you just want to throw together a small, embedded system, they are probably out of your price range. For what we need to do for our glue logic, such chips are overkill. Since our required logic is simple, we will use a simple (and cheap) PAL that can be programmed using freely available, public-domain software.

PALs are configured using equations to represent the internal logic. "+" represents OR, "*" represents AND, and "/" represents NOT. (These symbols are the original operator symbols that were used in Boolean logic. If you come from a programming background, these symbols may seem strange to you. You will be used to seeing "|", "&", and "!".) The equations are compiled using software such as PALASM, ABEL, or CUPL to produce a JED file. This is used by a device known as a PAL burner to configure the PAL. In many cases, standard EPROM burners will also program PALs.

PALs have pins for input, pins for output, and pins that can be configured as either input or output. Most of the PAL's pins are available for your use. In your PAL source code file (PDS file), you declare which pins you are using and label them. This is not unlike declaring variables in program source code, except that instead of allocating bytes of RAM, you're allocating physical pins of a chip. You then use those pin labels within equations to specify the internal logic. Our address decoder, implemented in a PAL, would have the following equations to specify the decode logic:

```
RAM = /A15
LATCH0 = (/A15 * /A14 * A13)
LATCH1 = (/A15 * A14 * /A13)
LATCH2 = (/A15 * A14 * A13)
```

I have (deliberately) written the above equations in a form that makes it easier to compare them with the address tables listed previously. You could simplify these equations, but there is no need. Just as an optimizing C compiler will simplify (and speed up) your program code, so too will PALASM rework your equations to optimize them for a PAL.

A PDS file to program a 22V10 PAL for the above address decode might look something like:

```
TITLE decoder.pds         ; name of this file

PATTERN

REVISION 1.0

AUTHOR John Catsoulis
DATE January 2005
CHIP decoder PAL22V10     ; specify which PAL device you
                          ; are using and give it a name ("decoder")
PIN  2  A15               ; pin declarations and allocations
PIN  3  A14
PIN  12  LATCH0
PIN  13  LATCH1
PIN  14  LATCH2
PIN   15  RAM
EQUATIONS                ; equations start here
RAM = /A15
LATCH0 = (/A15 * /A14 * A13)
LATCH1 = (/A15 * A14 * /A13)
LATCH2 = (/A15 * A14 * A13)
```

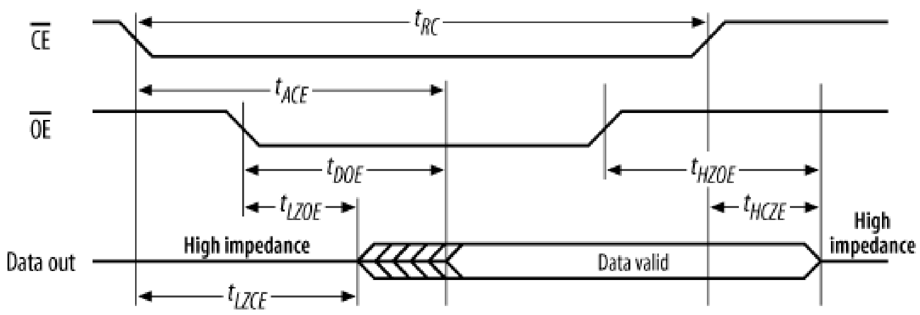
The advantages of using a PAL for system logic are twofold. The PAL equations may be changed to correct for bugs or design changes. The propagation delays through the PAL are of a fixed and small duration (no matter what the equations), which makes analyzing the overall system's timing far simpler. For very simple designs, it probably doesn't make a lot of difference whether you use PALs or individual chips. However, for more complicated designs, programmable logic is the only option. If you can use programmable logic devices instead of discrete logic chips, please do so. They make life much easier.

Timing Analysis

Now that we have finished our logic design, the question is: will it actually work? It's time (pardon the pun) to work through the numbers and analyze the timing. This is the least fun, and most important, part of designing a computer.

We start with the signals (and timing) of the processor, add in the effects of our glue logic, and finally see if this falls within the requirements of the device to which we are interfacing. We'll work through the example for the SRAM. For the other devices, the analysis follows the same method. The timing diagram for a read cycle for the SRAM is shown in Figure.The RAM I have chosen is a CY62256-70 (32K) SRAM made by Cypress Semiconductor. Most 32K SRAMs follow the JEDEC standard, which means their pinouts and signals are all compatible. So, what works for one 32K SRAM should work for them all. But the emphasis is on should, and, as always, check the datasheet for the individual device you are using.

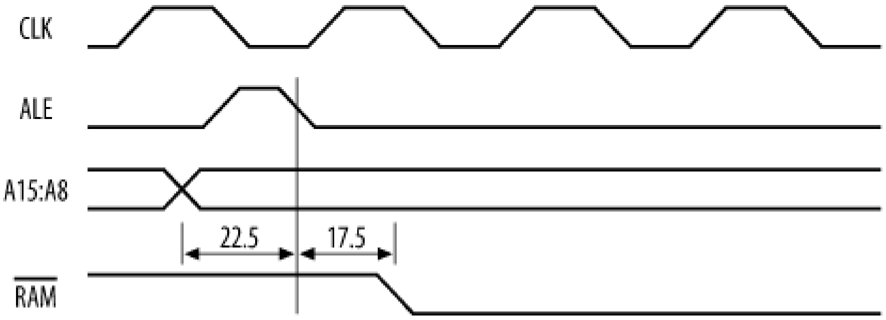
Timing for a read cycle to the RAM



During a read cycle, the processor will output a read strobe and an address, which in turn will trigger the address decoder. Some time later in the cycle, the processor will expect data from the RAM to be present on the data bus. It is critical that the signals that cause the RAM to output data will do so such that there will be valid data when the processor expects it. Meet this requirement, and you have a processor that can read from external memory. Fail this requirement, and you'll have an intriguing paperweight and a talking piece at parties.

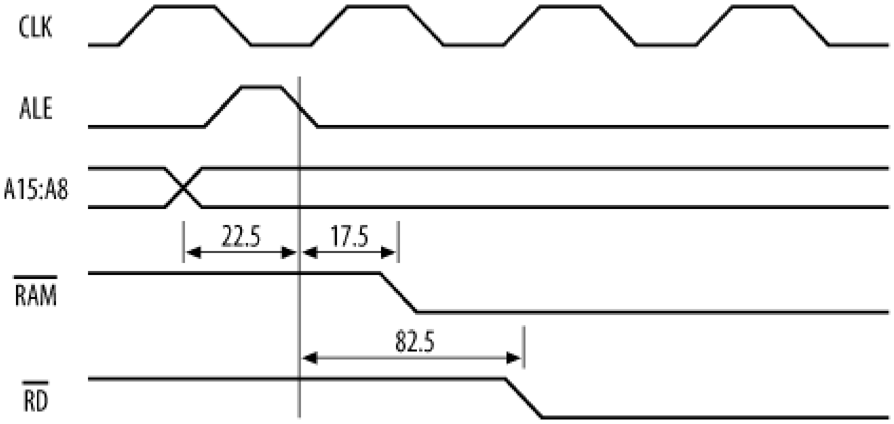
We start with the processor. I'm assuming that the processor's wait-state generator is disabled. For an AT90S8515 processor, everything is referenced to the falling edge of ALE. The high-order address bits, which feed our address decoder, become valid 22.5 ns prior to ALE going low on an 8 MHz AT90S8515. If we're using a 74HCT139 as an address decoder, this takes 40 ns to respond to a change in inputs. So, our chip select for the RAM will become valid 17.5 ns after ALE has fallen (Figure).

Timing for RAM chip select



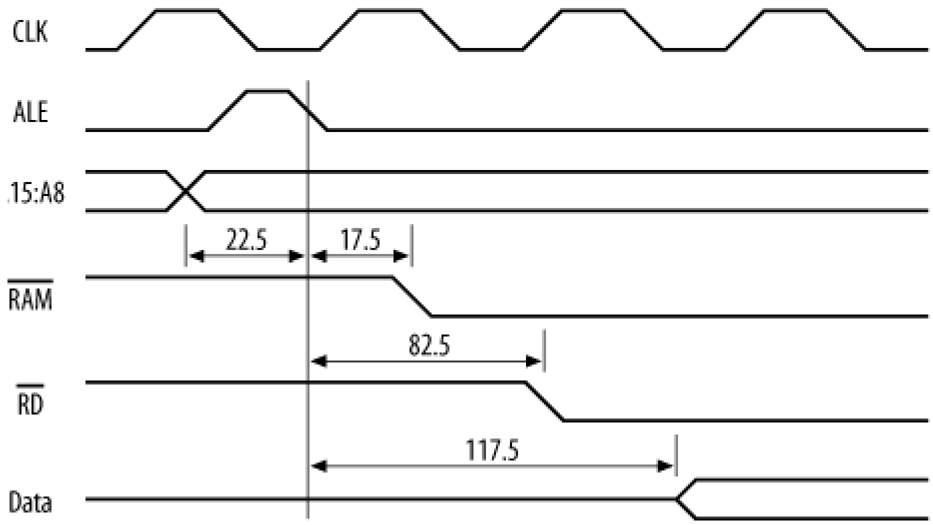
Now, RD will go low between 42.5 ns and 82.5 ns after ALE falls. Since the RAM will not output data until RD (OE) is low, we take the worst case of 82.5 ns (Figure).

Read strobe and chip select for RAM



The RAM will respond 70 ns after RAM and 35 ns after RD, whichever comes last. So, 70 ns from ALE low is 87.5 ns after ALE, and 35 ns after RD is 117.5 ns after ALE. Therefore, ALE is the determining control signal in this case. This means that the SRAM will output valid data 117.5 ns after ALE falls (Figure).

Valid data from the SRAM



Now, an 8 MHz processor expects to latch valid data during a read cycle at 147.5 ns after ALE. So our SRAM will have valid data ready with 30 ns to spare. So far, so good. But what about at the end of the cycle? Now, the processor expects the data bus to be released and available for the next access at 200 ns after ALE falls. The RAM takes 25 ns from when it is released by until it stops outputting data onto the data bus. This means that the data bus will be released by the RAM at 142.5 ns. So that will work too.

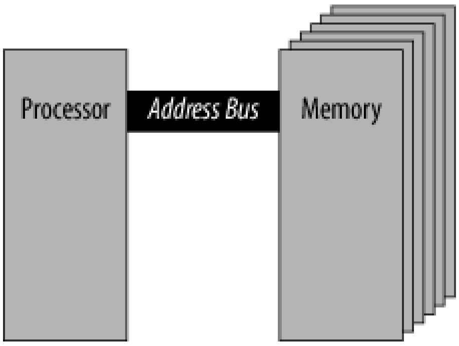
The analysis for a write cycle is done in a similar manner. It is important to do this type of analysis for every device interfaced to your processor, for every type of memory cycle. It can be difficult, because datasheets are notorious for leaving out information or presenting necessary data in a roundabout way. Working through it all can be time-consuming and frustrating, and it's far too easy to make a mistake. However, it is very necessary. Without it, you're relying on blind luck to make your computers run, and that's not good engineering.

Memory Management

In most small-scale embedded applications, the connections between a processor and an external memory chip are straightforward. Sometimes, though, it's advantageous to play with the natural order of things. This is the realm of memory management.

Memory management deals with the translation of logical addresses to physical addresses and vice versa. A logical address is the address output by the processor. A physical address is the actual address being accessed in memory. In small computer systems, these are often the same. In other words, no address translation takes place, as illustrated in Figure.

No address translation



For small computer systems, this absence of memory management is satisfactory. However, in systems that are more complex, some form of memory management may become necessary. There are four cases where this might be so:

Physical memory > logical memory

When the logical address space of the processor (determined by the number of address lines) is smaller than the actual physical memory attached to the system, it becomes necessary to map the logical space of the processor into the physical memory space of the system. This is sometimes known as banked memory. This is not as strange or uncommon as it may sound. Often, it is necessary to choose a particular processor for a given attribute, yet that processor may have a limited address space—too small for the application. By implementing banked memory, the address space of the processor is expanded beyond the limitation of the logical address range.

Logical memory > physical memory

When the logical address space of the processor is very large, it is not always practical to fill this address space with physical memory. It is possible to use some space on disk as virtual memory, thus making it appear that the processor has more physical memory than exists within the chips. Memory management is used to identify whether a memory access is to physical memory or virtual memory and must be capable of swapping the virtual memory on disk with real memory and performing the appropriate address translation.

Memory protection

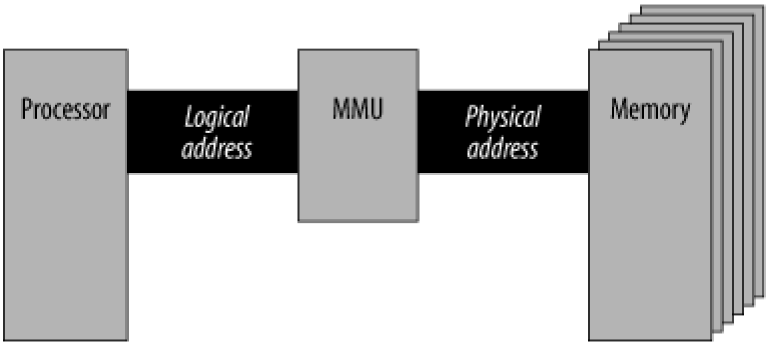
It is often desirable to prevent some programs from accessing certain sections of memory. Protection can prevent a crashing program from corrupting the operating system and bringing down the computer. It is also a way of channeling all I/O access via the operating system, since protection can be used to prevent all software (save the OS) from accessing the I/O space.

Task isolation

In a multitasking system, tasks should not be able to corrupt each other (by stomping on each other's memory space, for example). In addition, two separate tasks should be able to use the same logical address in memory, with memory management performing the translation to separate, physical addresses.

The basic idea behind memory management is quite simple, but the implementation can be complicated, and there are nearly as many memory-management techniques as there are computer systems that employ memory management. Memory management is performed by a Memory Management Unit (MMU). The basic form of this is shown in Figure. An MMU may be a commercial chip, a custom-designed chip (or logic), or an integrated module within the processor. Most modern, fast processors incorporate MMUs on the same chip as the CPU.

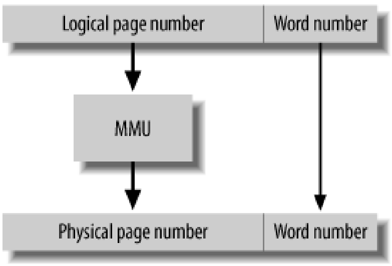
Address translation using an MMU



Page mapping

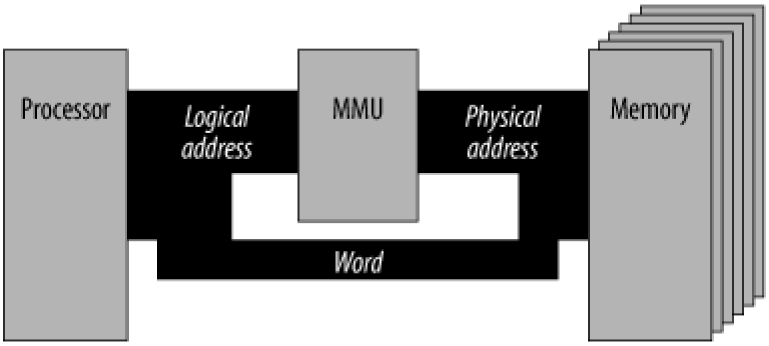
In all practical memory-management systems, words of memory are grouped together to form pages, and an address can be considered to consist of a page number and the number of a word within that page. The MMU translates the logical page to a physical page, while the word number is left unchanged (Figure). In practice, the overall address is just a concatenation of the page number and the word number.

Address translation



The logical address from the processor is divided into a page number and a word number. The page number is translated by the MMU and recombined with the word number to form the physical address presented to memory (Figure).

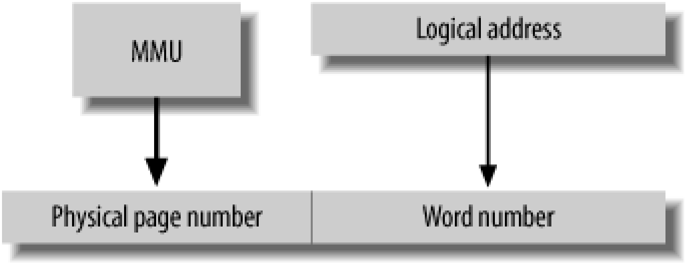
System using page address translation



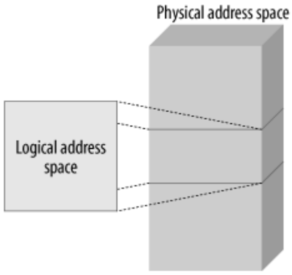
Banked memory

The simplest form of memory management is when the logical address space is smaller than the physical address space. If the system is designed such that the size of a page is equal to the logical address space, then the MMU provides the page number, thus mapping the logical address into the physical address (Figure).

MMU generation of page number

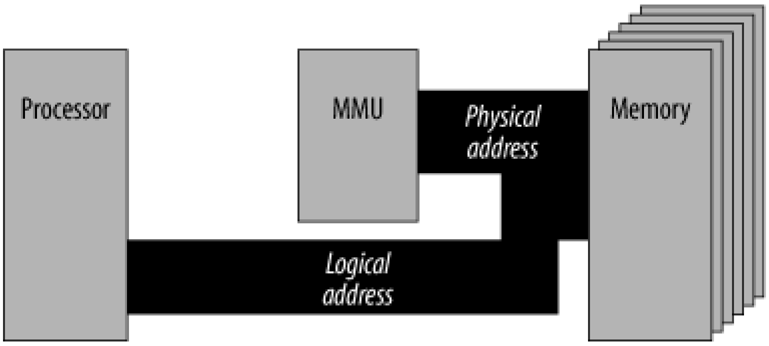


The effective address space from this implementation is shown in Figure. The logical address space can be mapped (and remapped) to anywhere in the physical address space. Mapping a smaller logical address space into a larger physical address



The system configuration for this is shown in Figure. This technique is often used in processors with 16-bit addresses (64K logical space) to give them access to larger memory spaces.

Generating a larger physical address



For many small systems, banked memory may be implemented simply by latching (acquiring and holding) the data bus and using this as the additional address bits for the physical memory (Figure). The latch appears in the processor's logical space as just another I/O device. To select the appropriate bank of memory, the processor stores the bank bits to the latch, where they are held. All subsequent memory accesses in the logical address space take place within the selected bank. In this example, the processor's address space acts as a 64K window into the larger RAM chip. As you can see, while memory management may seem complex, its actual implementation can be quite simple.

Simple banked-memory implementation

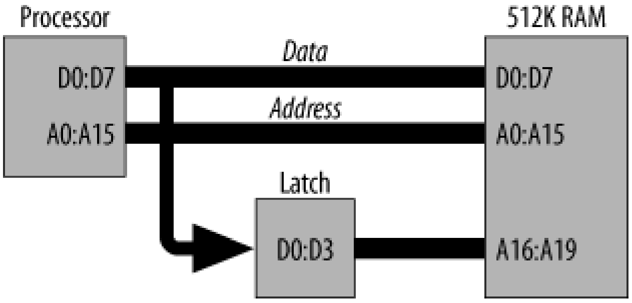


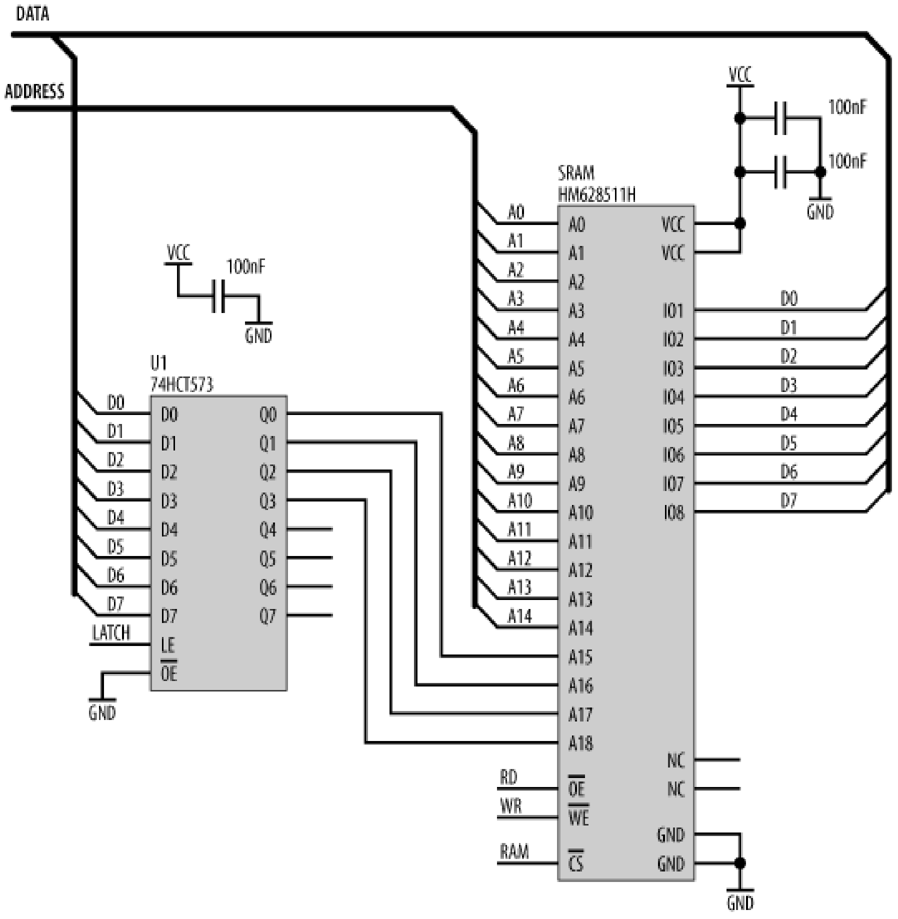
Figure shows the actual wiring required for a banked-memory implementation for our AT90S8515 AVR system, replacing the 32K RAMS with a 512K RAM.

The RAM used is an HM628511H made by Hitachi. In this implementation, we still have the RAM allocated into the upper 32K of the processor's address space as before. In other words, the upper 32K of the processor's address space is a window into the 512K RAM. The lower 32K of the processor's address space is used for I/O devices, as before. Address bits A0 to A14 connect to the RAM as before, and the data bus (D0 to D7) connects to the data pins (IO1 to IO8) of the SRAM. Memory chip manufacturers often label data pins as "IO" pins, since they perform data input and output for the device.

Now, we also have a 74HCT573 latch, which is mapped into the processor's address space, just as we did with the LEDs latch. The processor can write to this latch, and it will hold the written data on its outputs. The lower nibble of this latch is used to provide the high-order address bits for the RAM.

Let's say the processor wants to access address 0x1C000. In binary, this is %001 1100 0000 0000 0000. The lower 15 address bits (A0 to A14) are provided directly by the processor. The remaining address bits must be latched. So, the processor first stores the byte 0x03 to the latch, and the RAM's address pins A18, A17, A16, and A15 see

Banked memory for an AVR computer



%0011 (0x03), respectively. That region of the RAM is now banked to the processor's 32K window. When the processor accesses address 0xC000, the high-order address bit (A15) from the processor is used by the address decoder to select the RAM by sending its input low. The remaining 15 address bits (A0 to A14) combine with the outputs of the latch to select address 0x1C000.

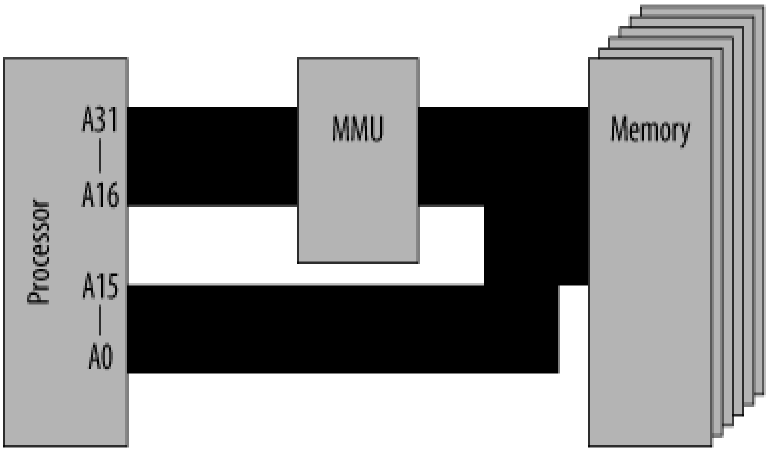
The NC pins are "No Connection" and are left unwired.

Address translation

For processors with larger address spaces, the MMU can provide translation of the upper part of the address bus (Figure).

The MMU contains a translation table, which remaps the input addresses to different output addresses. To change the translation table, the processor must be able to access the MMU. (There is little point in having an MMU if the translation table is unalterable.) Some processors are specifically designed to work with an MMU, while

Logical page-number translation



other processors have MMUs incorporated. However, if the processor being used was not designed for use with an MMU, it will have no special support. The processor must therefore communicate with the MMU as though it were any other peripheral device using standard read/write cycles. This means the MMU must appear in the processor's address. It may seem that the simplest solution is to map the MMU into the physical address space of the system. In real terms, this is not practical. If the MMU is ever (intentionally or accidentally) mapped out of the current logical address space (such that the physical page on which the MMU is located is not part of the current logical address space), it becomes impossible to access the MMU ever again. This may also happen when the system powers up, because the contents of the MMU's translation table may be unknown.

The solution is to decode the chip select for the MMU directly from the logical address bus of the processor. Hence, the MMU will lie at a constant address in the logical space. This removes the possibility of "losing" the MMU, but it introduces another problem. Since the MMU now lies directly in the logical address space, it is no longer protected from accidental tampering (by a crashing program) or illegal and deliberate tampering in a multitasking system. To solve this problem, many larger processors have two states of operation--supervisor state and user state--with separate stack pointers for each mode. This provides a barrier between the operating system (and its privileges) and the other tasks running on the system. The state the processor is in is made available to the MMU through special status pins on the processor. The MMU may be modified only when the processor is in supervisor state, thereby preventing modification by user programs. The MMU uses a different logical-to-physical translation table for each state. The supervisor translation table is usually configured on system initialization, then remains unchanged. User tasks (user programs) normally run in user state, whereas the operating system (which performs task swapping and handles I/O) runs in supervisor state. Interrupts also place the processor in supervisor state, so that the vector table and service routines do not have to be part of the user's logical address space. While in user state, tasks may be denied access to particular pages of physical memory by the operating system.

Summary:

- Like the PIC, the AVR is a RISC processor. Of the two architectures, the AVR is the faster in operation and arguably the easier for which to write code, in my personal experience.
- The AVR was developed in Norway and is produced by the Atmel Corporation. It is a Harvard-architecture RISC processor designed for fast execution and low-power consumption.
- The AVR has separate program and data spaces and supports an address space of up to 8M.
- ATtiy15 processor has 512 words of flash for program storage and no RAM! .This tiny processor, unlike its bigger AVR siblings, relies solely on its 32 registers for working variable storage.
- ATtiny15 processor also has 64 bytes of EEPROM (for holding system parameters), up to five general-purpose I/O pins, eight internal and external interrupt sources, two 8-bit timer/counters, a four-channel 10-bit analog-to-digital converter, an analog comparator, and the ability to be reprogrammed in-circuit.
- The first processor is the Atmel AT90S8535. This is a mid-range AVR with lots of inbuilt I/O, such as a UART, SPI, timers, eight channels of analog input, an analog comparator, and internal EEPROM for parameter storage. The processor has 512 bytes of internal RAM and 8K of flash memory for program storage.
- Memory management deals with the translation of logical addresses to physical addresses and vice versa.
- Memory management is performed by a Memory Management Unit (MMU).
- If the system is designed such that the size of a page is equal to the logical address space, then the MMU provides the page number, thus mapping the logical address into the physical address
- For many small systems, banked memory may be implemented simply by latching (acquiring and holding) the data bus and using this as the additional address bits for the physical memory.
- The MMU contains a translation table, which remaps the input addresses to different output addresses. To change the translation table, the processor must be able to access the MMU.
- Protection can prevent a crashing program from corrupting the operating system and bringing down the computer.

Questions:

- Write a brief notes on architecture of AVR controller?
- Explain features of ATtiny 15 processor?
- What is in system programming? Explain it in ATtiny15?
- Explain features of AT90s8535?
- How to design a data logger using AVR?
- Explain BUS interfacing?
- What is AT90S8515 Memory Cycle?
- Explain memory management in AVR?

References:

- Dhananjay Gadre - Programming and Customizing the AVR Microcontroller, McGraw-Hill, 2000.

- Richard H. Barnett, Sarah A. Cox, Larry D. O'Cull - Embedded C Programming and the Atmel AVR, Thomson Delmar Learning, 2002.

- John Morton - AVR: An Introductory Course, Newnes, 2002.

- Claus Kuhnel - AVR RISC Microcontroller Handbook, Newnes, 1998.

- Joe Pardue - C Programming for Microcontrollers, featuring ATMEL's AVR Butterfly and the free WinAVR Compiler, Smiley Micros, 2005. Smiley Micros

- Chuck Baird - Programming Microcontrollers using Assembly Language, Lulu.com, 2006. cbaird.net

13. 68HC11

Objective:

In this chapter, we'll look at the Free scale Semiconductor (formerly Motorola) 68HC11, a processor architecture that goes back to the very early days of microprocessors. I have a soft spot for this architecture. I first learned to write assembly language on a machine based on a 6802 processor, and I can still remember many of the opcodes by heart and can "read" raw 6800 machine code as though it were source.

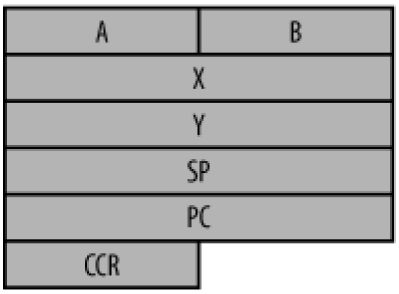
The architecture is far from cutting-edge. But it's easy to program, easy to build, and has been stable for a very long time. It's a good platform to start out on, and it's quick and easy to throw together a simple 8-bit computer using these chips. Let's start by taking a quick overview of the processor architecture.

Architecture of the 68HC11

The MC68HC11 is a member of the 8-bit, 6800 microprocessor family. The 68HC11 is a high-density, HCMOS microcontroller unit (MCU) featuring a fully static design. It is essentially a standard 6800 processor (with some enhancements) combined with inbuilt peripherals, such as an enhanced 16-bit timer with four-stage programmable pre-scaler, a serial peripheral interface (SPI), a serial communications interface (SCI), an 8-bit pulse accumulator, real-time interrupts, onboard static RAM, an eight-channel ADC, and onboard EEROM.

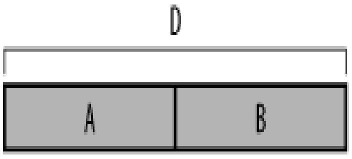
The main registers of the MC68HC11 are shown in Figure. (Note that this does not include the control registers associated with the various peripherals inside the chip.)

MC68HC11 registers



The MC68HC11 has two accumulators, A and B. The accumulators are both eight bits wide, but they may also be treated as a single, 16-bit accumulator, D (Figure).

MC68HC11 accumulators



The index registers (X and Y) are 16-bit registers that are used to hold addresses. As such, they can be used to point to locations in memory. They may also be used as 16-bit counters or temporary storage registers. The program counter (PC) is a 16-bit counter that points to the next location in memory from which an instruction is to be fetched. (In other words, it holds the address of the next instruction.) The condition code register (CCR) is a special 8-bit register that shows the status of the processor.

The stack pointer (SP) is a 16-bit register that points to the next free location on the stack. The stack is an area of memory defined for storage of data or addresses (treated as data). When a value is pushed onto the stack, the value is stored at the location pointed to by the stack pointer. The stack pointer then decrements automatically and points to the next available location. When something is pulled from the stack, the stack pointer is incremented automatically, and the data value is retrieved from that location. As a 68HC11's stack fills, it grows down through memory. When a 16-bit value is pushed onto the stack, the stack pointer is decremented twice (two 8-bit locations).

So that's the basic programmer's model for a 68HC11. While not overly powerful, it's nice and simple, and easy to master. Now let's see how to build a machine based on a 68HC11.

A Simple 68HC11-Based Computer

The computer will have 32K of static RAM, 16K of EPROM, a serial interface (internal to the 68HC11), and a latch controlling a bank of LEDs. While EPROM is old technology, I have chosen it for this system for two reasons. The first is that it is still common for 68HC11 machines to use EPROM, often for historical and legacy reasons. The second reason is that it allows me to show you how to use an EPROM in a design. Following the theme of "showing you how it's done," we'll also do the glue logic for the computer using discrete gates rather than a PAL.

The 68HC11 was designed to be used in a wide range of small applications, many relating to the monitoring or control of external devices. As such, it can run in several modes: single-chip mode, expanded multiplexed mode, bootstrap mode, and test mode. This last mode is used by Freescale during manufacturing and is not intended for user applications. In single-chip mode, the processor relies entirely on its internal features (small RAM, small

ROM, I/O, and timers) and has no external address or data bus. The majority of pins (known as ports A, B, C, and D in this particular processor) are therefore dedicated to digital I/O functions. In expanded multiplexed mode, the processor behaves like an ordinary, 8-bit processor, with ports B and C assuming the roles of the address and data buses. In bootstrap mode, the processor loads its vectors from the internal 192-byte ROM and initializes the internal serial interface. The processor can change from bootstrap mode to any of the other modes under software control. Two special pins on the processor (MODA and MODB) determine in which mode the processor will "come up." Table shows the settings for MODA and MODB and how these affect the 68HC11.

Since we wish to add external memory and a latch, the processor must be in expanded multiplexed mode. Hence, MODA and MODB must be tied high in our design.

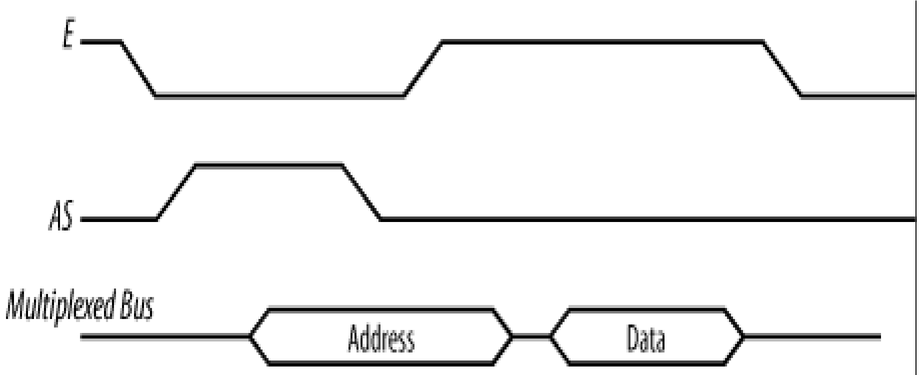
Boot modes for the 68HC11

MODB	MODA	Mode
1	0	Single chip
1	1	Expanded multiplexed
0	0	Special bootstrap
0	1	Special test

To reduce the number of external pins of the 68HC11, Freescale has multiplexed the address and data buses onto the same physical pins. This means the chip is smaller (and therefore cheaper), but it requires the system designer to add external logic to recover (separate) the address and data buses from the multiplexed bus. The data bus and the lower half of the address bus share port C, while the upper half of the address bus is on port B and requires no recovery. A special output, AS (address strobe), is provided to indicate whether address information or data is present on the bus.

The timing for a memory cycle is shown in Figure. The address becomes valid after AS goes high and remains valid as AS falls. AS can be used as the control input to a latch to recover the lower half of the address bus. Once latched, the address continues to be output by the latch and hence continues to be valid throughout the cycle. The data appears on the multiplexed bus later in the cycle.

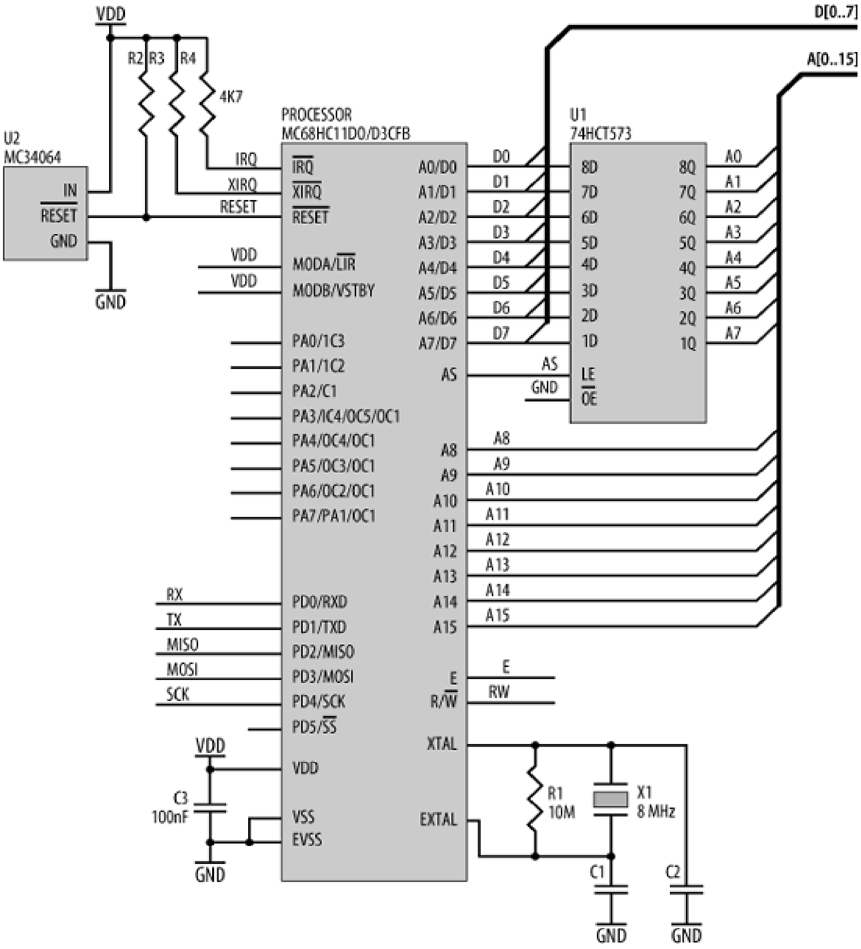
Timing of the multiplexed bus on a 68HC11



The upper address bits (A8:15) appear on port B 94 ns before E goes high (in the middle of the cycle) and remain valid for the whole cycle. No recovery is required for these address lines.

The basic circuit for a 68HC11 processor in expanded multiplexed mode, including the recovery of the lower address bits, is shown in Figure. Interrupt inputs, IRQ and XIRQ, require pull-up resistors as well. Motorola recommends the use of a special chip, MC34064, for generating a power-on reset. This simple three-pin device requires only power and ground, and will output a reset pulse on power up. This reset pulse is of an appropriate duration for a 68HC11. To provide a clock for our processor, we add an 8 MHz crystal to the processor's internal oscillator (pins XTAL and EXTAL). The crystal needs two bypass capacitors, C1 and C2, and also requires a resistor, R1, parallel.

MC68HC11 and support components



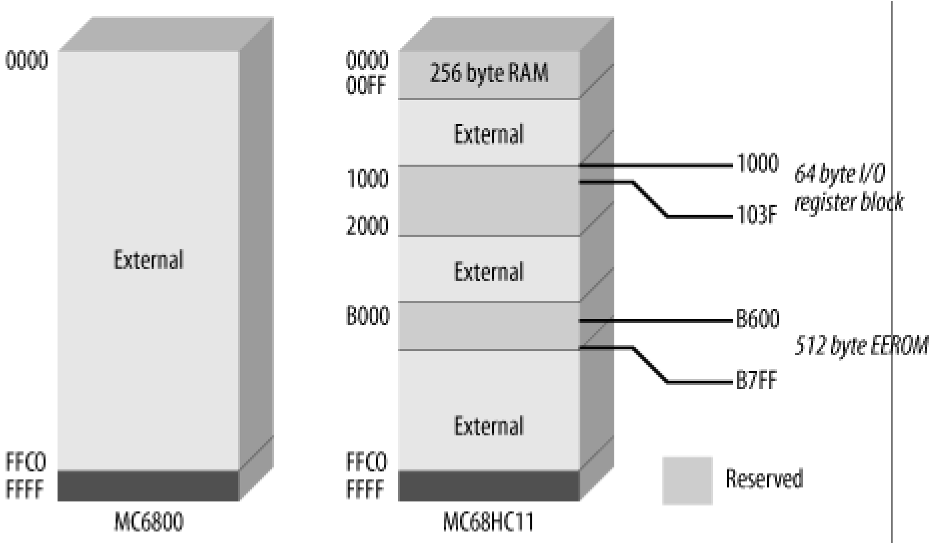
Port D is used for the internal serial interface, with bit 0 as the receive data input (Rx) and bit 1 as the transmit data output (Tx). Port D also contains the processor's SPI interface, allowing it to be interfaced easily to a variety of peripherals.

That completes the processor's basic requirements. The next task is to design the rest of the computer, which for our system with its one RAM, one ROM, and a latch is very simple. We start by allocating the memory space and then design the address decoder.

Address Decoding

The MC6800 and MC68HC11 address spaces are shown in Figure. They are both 64K spaces (16-bit address), but note the additional, internal features of the 68HC11 located in its memory map. The register block is not the accumulators or index registers that were mentioned previously. These do not appear in the memory map. The register block is an array of special registers that control the many internal peripherals this processor has, such as counter/timers, analog-to-digital converters, etc. Note that a 68HC11 has the ability, through software, to relocate the internal I/O registers and the 256-byte RAM to any 4K boundary. This means the designer can place these wherever is most appropriate for the design. The "external" designator in Figure means that addresses in this range are available externally for use by memory or other devices.

Comparison of 6800 and 68HC11 memory maps



The vectors are a table of addresses stored externally that point to routines in memory. The most important of these is the 16-bit RESET vector starting at address `0xFFFE`. This location contains a 16-bit pointer to the location in memory where the initialization code lies. The processor will load this pointer into its program counter after power-on or reset and thereby begin execution of the software. Therefore, since this vector needs to be valid at power-on, it must be nonvolatile (able to survive without power). For this reason, a ROM is usually located in the uppermost region of the address space.

Now, the 68HC11 has a 64K address space, so a 32K RAM is going to occupy half of this space. But which half? As mentioned earlier, for the vector table to be preserved during periods of no power, a ROM must be located in the uppermost part of the address space. Thus, our 32K RAM must be put in the lower half of the address space. However, the internal RAM and registers of the internal peripherals are mapped into the lower half of the address space. If we map our 32K RAM into this space, will it cause conflict? (In other words, will we need special logic to accommodate this?) The answer is no. The internal RAM and I/O registers take precedence, and accesses to their locations will not cause activity on the external buses of the processor. In effect, they are overlaid on top of the external RAM. From our point of view, this makes the design simple, as we don't need special logic to exclude those addresses from our memory space. And since the internal peripheral registers can be remapped under software control, these can be shifted elsewhere to an unused part of the address space.

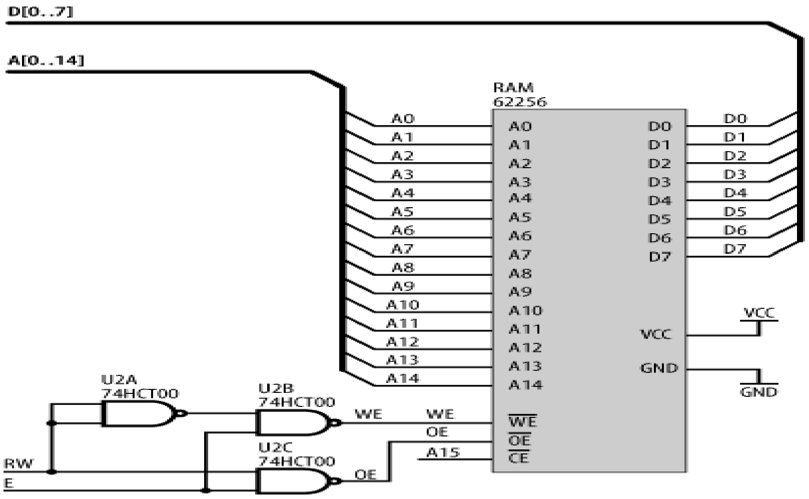
In the case of the 32K RAM, its address size ranges from `0x0000` to `0x7FFF`. In binary, this is `0000 0000 0000 0000` to `0111 1111 1111 1111`. Any combination of bits between these two addresses lies within the space allocated to the RAM. So address bits A0 to A14 relate to memory locations internal to the RAM, and hence they are not used by the address decoder. In other words, the address decoder "doesn't care" what they are since they go directly into the RAM. The address table for the RAM is shown in Table. The "X" means "don't care."

Address bit usage for the RAM

	A15	**A14**	**A13**	**A12**	**A11**	**A10**	**A9**	**A8**	**A7**	**A6**	**A5**	**A4**	**A3**	**A2**	**A1**	**A0**
RAM	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Look at the address range for the RAM: `0000 0000 0000 0000` to `0111 1111 1111 1111`. The only bit in common for all those address bits is the "0" at A15. As the RAM is 32K in size and this is half of the 64K address space of the processor, the only bit that needs to be taken into account when decoding for the RAM is A15. Since the RAM is in the lower half of the address space, the RAM will need to be selected when the most significant address bit is low. When it is high, the RAM will not be selected. The address decode for the RAM is therefore simply a direct connection between A15 and the RAM's CS. What could be easier than that? The circuit for the RAM, in this case a 62256 SRAM, is shown in Figure. This same generic circuit will work for any standard 32K x 8 SRAM.

RAM



When A15 is low, the chip enable (CE) is pulled low and is therefore asserted (since it is active low). Thus, the RAM is enabled when A15 is low, and not enabled when A15 is high.

Note the additional logic (the three NAND gates) in the circuit. The 68HC11 generates a R/W strobe indicating whether the cycle is a read cycle (R/W high) or a write cycle (R/W low). Now, the RAM has separate inputs to signify whether the access is a read or a write, and, in both cases, these are active low. The logic is used to convert the single R/ strobe into separate WE (write enable) and OE (output enable) inputs to the RAM. The R/W strobe is combined with the processor's E clock to ensure that the enables are active only during the valid part of the cycle (when E is high). Otherwise, OE would be active whenever it wasn't a write cycle. The NAND gate U2A is simply acting as an inverter.

The ROM is a 16K device, which is one half of the remaining address space. The only other external device is a latch (which need occupy only one byte). There are two ways of allocating the remaining 32K of memory to these two devices. The first is to use explicit address decoding in which every address line is accounted for. In this scheme, the latch would occupy exactly one byte of memory and no more. So if we decide to locate the latch at address `0x8000`, we have the address bits as shown in Table.

Address bits to select the latch at 0x8000

	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
Latch	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

If we are using explicit decoding, we must use all of these bits in our address decoder. Such logic would be both complex and slow.

A better way is to use partial address decoding. With this method, we divide our memory space among the devices, using just enough address bits to distinguish each device. It doesn't matter if the memory space allocated is much greater than that required by the device. Even if we allocate 16K of space to the latch, the latch will still work when we select it. It's true this is somewhat wasteful of address space, but it is a far more efficient method (in terms of logic) than explicit decoding. The logic required is much less, and if you are using discrete logic, the propagation delays are reduced. Timing is the most important consideration when designing any logic for a microprocessor system. If the timing isn't right, it simply won't work.

So our remaining 32K of address space needs to be divided between two devices. The address table for all three devices (RAM, latch, and ROM) is shown in Table.

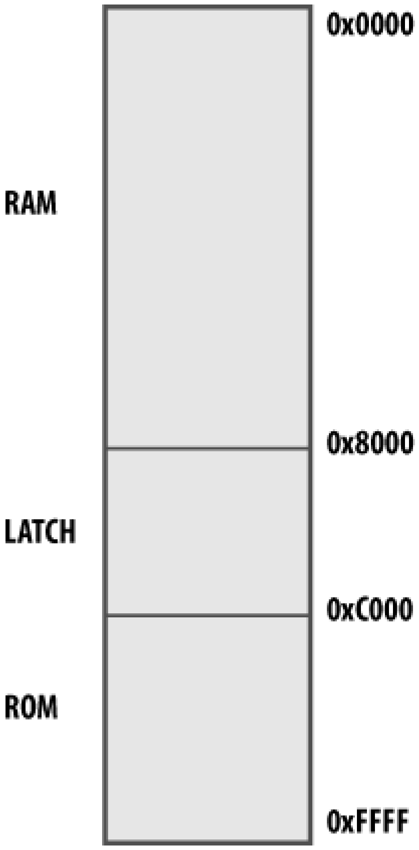
Address bit allocation for all devices

	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
RAM	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Latch	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X
ROM	1	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X

In this table, I have allocated the latch and the ROM 16K of space each. In other words, I have divided the remaining 32K of space equally between the two devices. This will make the address decoding much simpler. The resulting memory map for the computer (ignoring the internal I/O registers and memory for the moment) is shown in Figure.

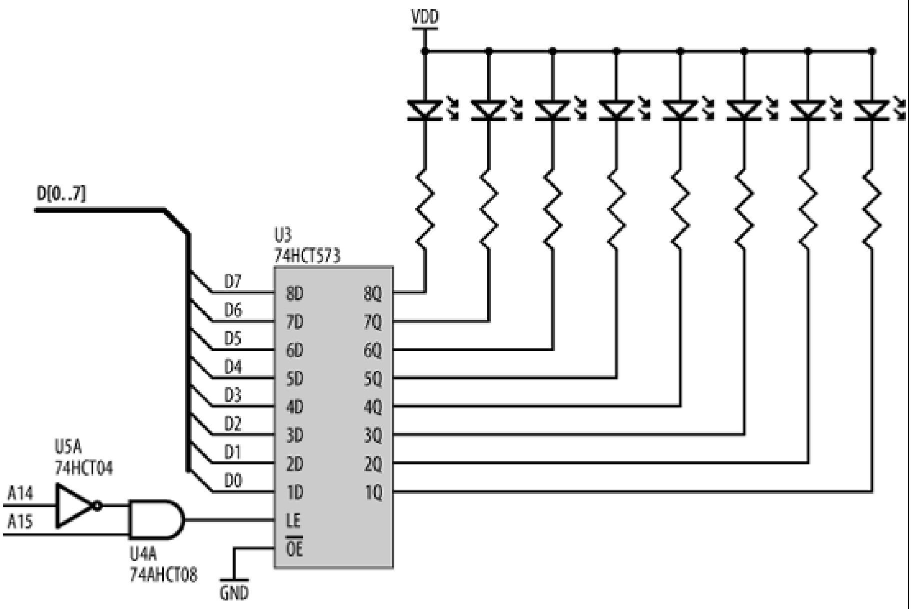
Memory map using partial address decoding



Note that because we have used partial decoding, the latch will appear multiple times in its allocated space. The latch represents one byte at address `0x8000`, but because we are looking at only A15 and A14 for its address decode, it is selected for all addresses in which A14 is low and A15 is high. Therefore, the latch appears throughout the address range `0x8000` to `0xBFFF`. For instance, if we access location `0x9401`, since A14 is low and A15 is high for that address, we will select the latch. A0 to A13 are not used by the decoder, so their state is irrelevant to the address decode.

The schematic for the LED latch circuit is shown in Figure (power and ground connections for U3 are present but are not shown for clarity). The latch has two control lines, LE and OE . LE going from high to low causes the latch to capture and hold the current input data, in this case from the processor's data bus. OE controls whether the latch outputs the data. Since we want the LEDs to always show their current state, we want the latch to permanently output the currently latched data. Hence, OE must always be asserted (tied low). The address decode for the latch is relatively simple. When the processor is writing data to the latch, A14 is low and A15 is high. A14 is inverted by U5A and ANDed with A15. The output of the AND gate (U4A) will be high; therefore, the latch will capture the data that is being written to it. In effect, the latch is acting as a single byte of write-only memory.

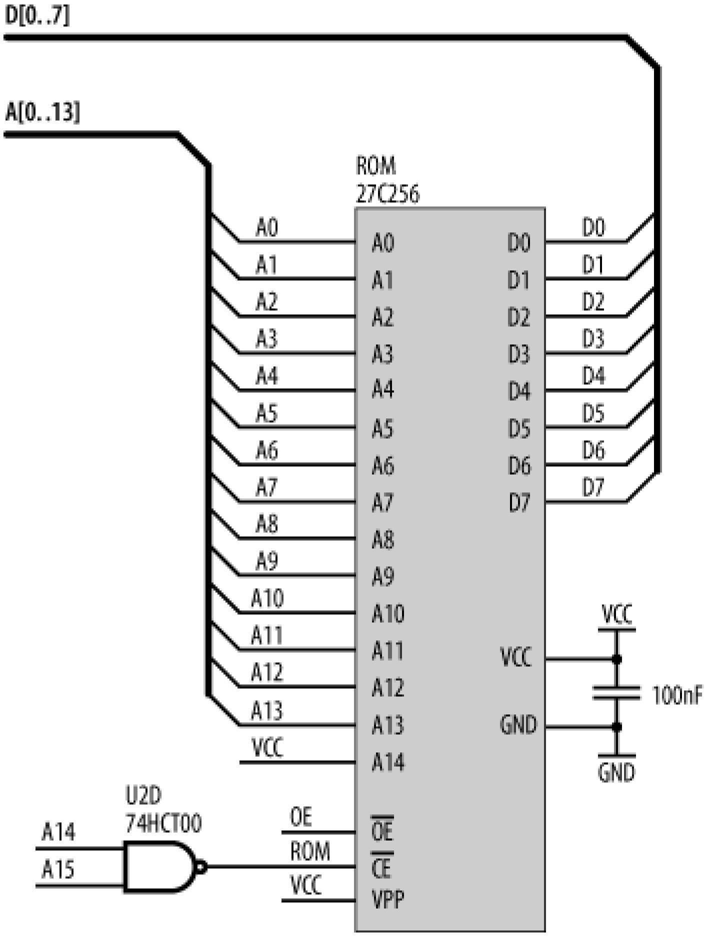
LEDs and latch



The address decode for the ROM is shown in Figure. When both A14 and A15 are high, the output from the NAND (U2D) gate will be low; therefore, the ROM will be selected. If either A14 or A15 are low, the ROM will not be selected. The ROM shown in the schematic is a 27256 (a 32K part), since these are easier to acquire than 16K devices. Because it is a 32K device, and we're using only half of its internal space, its address input A14 is tied high. In this way, we permanently select the upper half of the device's space. Note that this A14 input is not the same as the A14 address bit from the processor. The OE (output enable) input is connected to the same OE we generated for the RAM. Finally, the power pin VPP is used during programming (out of circuit) to load the chip with data. During normal operation, this pin is tied to VCC.

So that completes the basic design for the 68HC11 computer. Using the processor's SPI port, we could add a variety of additional peripherals or data-storage memories. Alternatively, we could add additional peripherals using the processor's buses by breaking up the address space allocated to the latch and providing support for more devices. We could also apply the memory-management techniques covered in the AVR chapter to increase the amount of RAM in the computer.

Address decode for ROM



Summary:

- The MC68HC11 is a member of the 8-bit, 6800 microprocessor family.
- It is essentially a standard 6800 processor inbuilt peripherals, such as an enhanced 16-bit timer with four-stage

programmable pre-scalar, a serial peripheral interface (SPI), a serial communications interface (SCI), an 8-bit pulse accumulator, real-time interrupts, onboard static RAM, an eight-channel ADC, and onboard EEROM.

- The MC68HC11 has two accumulators, A and B. The accumulators are both eight bits wide.
- The stack pointer (SP) is a 16-bit register that points to the next free location on the stack.
- The stack is an area of memory defined for storage of data or addresses (treated as data).

Questions:

- Explain the architecture of MC68HC11?
- Design a simple computer using MC68HC11?
- Explain Memory mapping in MC68HC11 controller?
- What are the different booting modes of 68HC11?
- How to decode address in 68HC11?
- Explain the interfacing of RAM in 68HC11?
- Is it possible to interface with LEDs using 68HC11?Explain.

References:

- Data Acquisition and Process Control with the MC68HC11 Micro Controller by Frederick F. Driscoll, Robert F. Coughlin, Robert S. Villanucci.
- Introduction to Microprocessors and Microcontrollers by John Crisp.
- Microprocessors and Microcomputers: Hardware and Software (6th Edition) by Ronald J. Tocci and Frank J. Ambrosio.

14. MAXQ

Objective:

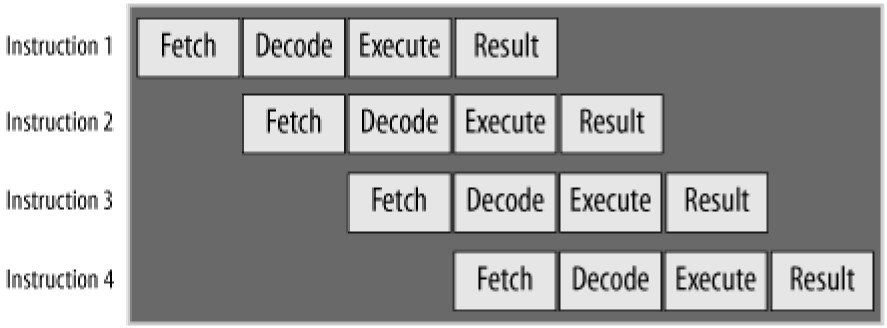
In this chapter, we'll look at an innovative new processor architecture introduced to the world in 2004. Dallas Semiconductor, a subsidiary of Maxim (http://www.maxim-ic.com), developed the 16-bit MAXQ microcontrollers to target the low-cost, low-power embedded applications market. The architecture is aimed directly against Microchip's PIC, Atmel's AVR, Texas Instruments' MSP430, and the 8051 architecture offered by many manufacturers (including Dallas Semiconductor itself). The MAXQ is an interesting contender for top RISC microcontroller. It's fast, has a lot of functionality, and is very low-powered. At the time of writing, the User's Guide for the MAXQ is 230 pages long. Obviously, this processor has a lot of features, too many to be thoroughly covered here. Therefore, I'm going to simply concentrate on the basic design for a MAXQ-based system. Let's start by seeing what makes the MAXQ so different and so interesting.

Architectural Overview

The stated design goal for the MAXQ was to achieve a high performance-to-power ratio. In other words, the aim was to maximize the processor's throughput of instructions while minimizing the current draw. Many RISC processors achieve single-cycle execution but do so through the use of an instruction pipeline. In a pipelined architecture, the execution unit is comprised of many stages. At any one time, several instructions will be in the process of being decoded and executed. Thus, with a pipeline, although a given instruction may take several cycles to execute, the processor is able to have an instruction terminate on each cycle (Figure).

Each cycle moves each instruction further along the pipeline, from fetch to termination (and result). The disadvantage of a pipeline is that a call or jump instruction means that all instructions following in the pipeline are not needed and the pipeline must be reloaded from a new location (where the jump/call was directed). So, while

Four-stage instruction pipelining



Pipelining can achieve single-cycle execution; it falls down in a big way unless the code is linear.

The MAXQ does not have an instruction pipeline, yet it still achieves single-cycle execution, with the exception of long jumps and calls and some extended register accesses. Now, you may say, "So what's the difference?" since pipelined processors have problems with jumps and calls too. The difference is that a pipelined processor executing a jump means that not only is the jump not single-cycle, but it will cause a disruption to the pipeline affecting the following instructions. With the MAXQ, this is not the case. Only the jump or call is not single-cycle, and all subsequent instructions execute without incurring the delay of a pipeline reload.

The MAXQ achieves this by having instruction-decode and execution units that are much simpler than those found on many other processors. How simple? Well, the MAXQ has only one instruction (`move`), but that one instruction has multiple functions, depending on the source and destination operands. By having only one instruction, a classical decode unit is not required. (You already know what the instruction is going to be, so what's there to decode about it?) Hence, the execution of instructions is reduced to determining source and destination, and whether additional hardware operations are triggered as part of the move. The source and destination bits of an instruction merely activate internal data paths, and this happens as the instruction is fetched.

The basic format for a MAXQ instruction is `fdd dddd ssss ssss`, where `f` is the format bit, `d` represents the destination-field bits, and `s` represents the source-field bits. When the format bit is a `1`, the instruction moves data from one index module to another. When the format bit is a `0`, an immediate 8-bit value is loaded into an index module (Table).

For mat bit	**7-bit destination**	**8-bit source**
1	Index module	Index module
0	Index module	Immediate byte data

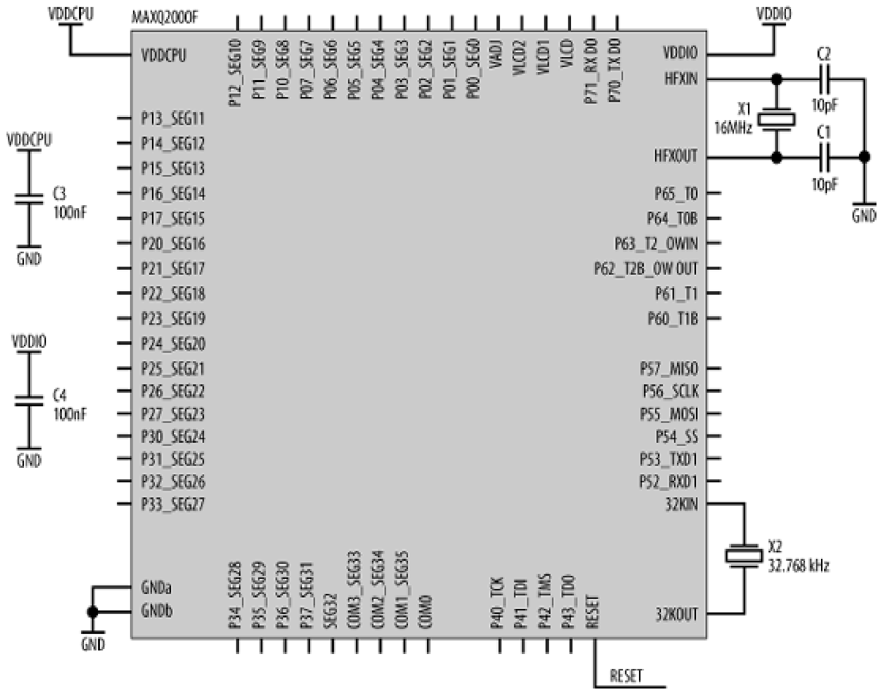
In the MAXQ architecture, the index modules are not necessarily specific registers, and herein lies the flexibility of the architecture. A given working accumulator may actually be represented by more than one index module. One index module will target the accumulator and perform an addition operation, while another index module will target the same accumulator yet perform a subtraction. In this way, two different operations are specified by two index modules, even though they both target the same register.

Like the PIC and AVR, the MAXQ is a Harvard-architecture processor, with separate code and data spaces. Overall, the MAXQ is a very nice processor, and one that I'm sure will gain market share as time passes.

Schematics

A major problem common in utilizing a microcontroller in a mixed-signal environment (one that combines both digital and analog components) is the noise the digital subsystem introduces. Higher processor performance normally results in greater noise in the analog section, unless great pains are undertaken to minimize these effects. Thus, achieving high throughput is often contrary to the goal of keeping the analog circuits as noise-free as possible. The MAXQ implements intelligent clock management that reduces noise by enabling clocks only to those subsystems that require them, and only when they require them. In this way, the overall digital noise is reduced considerably. The MAXQ processor requires two crystals, a 16 MHz crystal (X1) for the main CPU clock and a 32.768 kHz watch crystal (X2) for the timers. Figure shows the MAXQ2000F processor with its support components.

MAXQ processor and support components

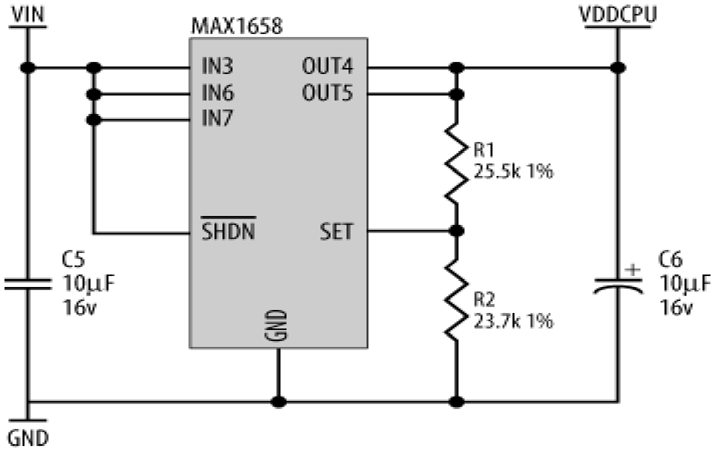


The MAXQ processor requires two power supplies, VDDCPU (2.5 V) and VDDIO (3.6 V), each decoupled to ground with 100 nF ceramic capacitors. The 2.5 V supply may be generated using a MAX1658 (Figure). This is a general-purpose regulator, the output of which is adjustable via bias resistors. These resistors, R1 and R2, set the output to +2.5 V. It is important that these resistors are precise, so choose resistors with 1% tolerance. The input and output of the regulator are each decoupled with 10 uF capacitors. The MAX1658 can operate on an input voltage (VIN) of between 2.7 V and 16.5 V, supplying up to 350 mA to the embedded computer system.

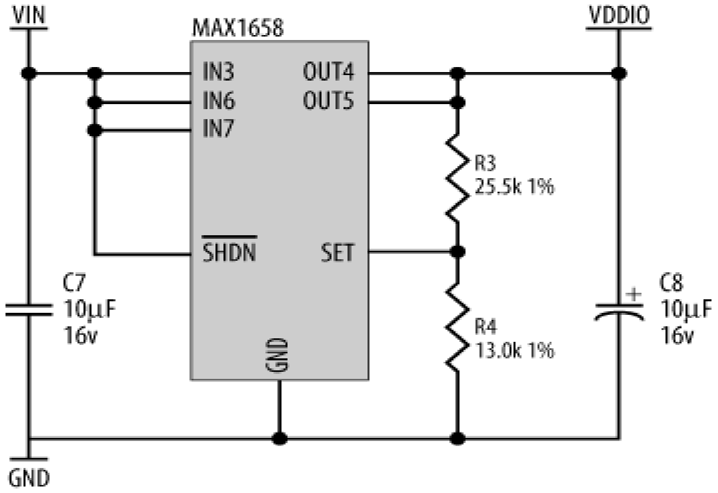
A similar circuit is used to generate the 3.6 V supply required by the MAXQ's I/O subsystems (Figure). Note the different resistor values required to generate 3.6 V rather than 2.5 V.

The MAXQ has an internal power-on reset generator, kicking the processor to life at power-up. No external reset circuit is required. If a manual reset is required, a push- button switch may be used to pull the RESET line low. However, it is important to note that RESET is a bidirectional line. The MAXQ also uses this signal as an output to indicate that a reset condition (possibly generated internally) is being serviced.

Generating 2.5 V for VDDCPU



Generating 3.6 V for VDDIO

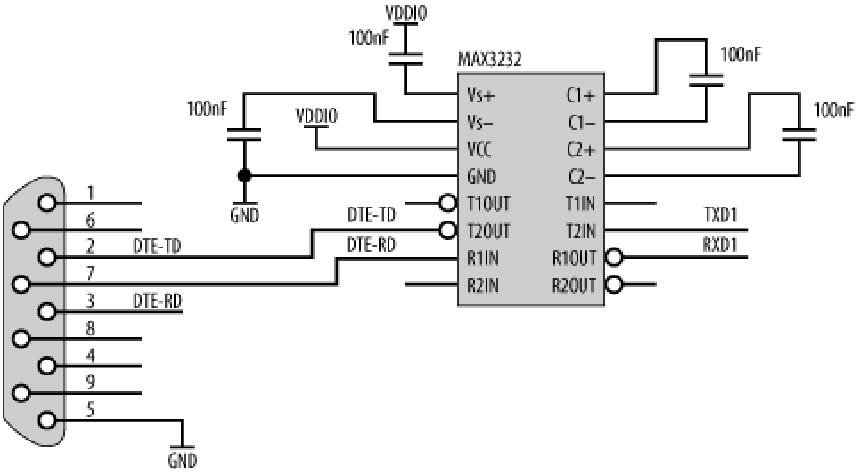


This can be used by the system designer to reset external peripherals as well (if required).

The various ports, labeled Px, provide access to the MAXQ's I/O. As well as providing digital I/O, they also serve dual purposes. Port 5 provides a SPI interface as well as a serial port. The SPI interface may be connected to any SPI-based peripheral. The serial port requires a level shifter such as a

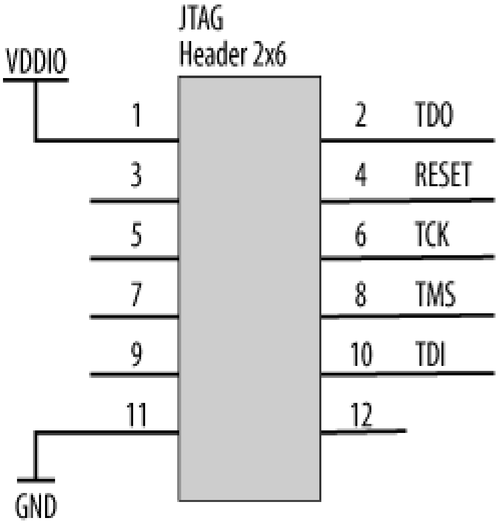
MAX3232, as shown in Figure. The transmitter (TXD1) and receiver (RXD1) of the MAXQ connect to the receiver and transmitter pins on the MAX3232.

Serial port



Port 5 of the MAXQ provides access to JTAG signals for in-system programming and debugging. Figure shows the pinout for a JTAG header. This is the same pinout used on the Maxim MAXQ development system, allowing you to use the same environment for your embedded computer.

JTAG interface



The MAXQ is a versatile and fast 16-bit processor, and the family is due to be expanded by Maxim. If you're looking for a low-powered, yet very capable processor for an embedded application, take a close look at the MAXQ. You'll find it's an impressive little processor.

Summary:

- Dallas Semiconductor, a subsidiary of Maxim (http://www.maxim-ic.com), developed the 16-bit MAXQ microcontrollers to target the low-cost, low-power embedded applications market.
- The MAXQ design is goaled to achieve a high performance-to-power ratio.
- The disadvantage of a pipeline is that a call or jump instruction means that all instructions following in the pipeline are not needed and the pipeline must be reloaded from a new location.
- The basic format for a MAXQ instruction is `fdd dddd ssss ssss`, where `f` is the format bit, `d` represents the destination-field bits, and `s` represents the source-field bits.
- Like the PIC and AVR, the MAXQ is a Harvard-architecture processor, with separate code and data spaces.
- The MAXQ processor requires two power supplies, VDDCPU (2.5 V) and VDDIO (3.6 V), each decoupled to ground with 100 nF ceramic capacitors.

Questions:

- Explain the architecture of MAXQ IC?How it differs from pic,AVR & 8051's architectures?
- Define the format of MAXQ controller instruction?
- Explain MAXQ with its schematics?
- Explain how a Numerical LCD is connected with its pin diagram?

References:

- Analog Interfacing To Embedded Microprocessors – STUART R. BALL

- C Programming for Embedded Systems – KIRK ZURELL
- Design with 8051- FRONTLINE ELECTRONICS
- Embedded Controller Hardware Design - Ken Arnold
- Embedded Software The Works – colin walls
- Embedded Systems Firmware Demystified - Ed Sutter
- Embedded_Controller_Hardware_Design – KEN ARNOLD
- Programming Embedded Systems in C and C++ - Michael Barr
- The Art of Designing Embedded Systems - Jack G. Ganssle

15. 68000-Series Computers

Objective:

In this chapter, we'll take a look at a 32-bit processor that has been around for quite some time and has evolved into a plethora of controllers and embedded processors. The 68000 (also known as the "68k") is produced by Free scale Semiconductor and is licensed by several other manufacturers. The range of 68000-based processors is large (check out the manufacturers' web sites for a list of processors and their features). The number of applications that the 68000 has found its way into is enormous. You can even get 68000s as soft cores for FPGAs, which means you place a 68000 CPU in the midst of your programmable logic, all on the one chip.

The Motorola MC68000 was introduced in 1979 as the successor to its 8-bit 6800 family. It featured a large address space, 32-bit registers, a large number of addressing modes, and an enlarged instruction set with over 1,000 opcodes. It was designed with the intention of running multitasking operating systems, specifically Unix. Its use in Unix machines has now long since passed, having been usurped by more advanced RISC processors. The 68000 processor was also used in the original Macintosh computers, as well as in the Atari ST, the Commodore Amiga, and Jef Raskin's CAT computer, all long extinct. Motorola Semiconductor is now known as Free scale Semiconductor.

The processor's wide range of software and reasonable computing power are now encouraging its extensive use in embedded systems. It now forms the basis of a family of microcontrollers designed for embedded systems, industrial control, networking, and PDAs. The 683xx series is the primary family of microcontrollers specifically tailored to embedded applications. These processors combine a CPU32 core (68020-based) with various integrated functions (such as UARTs, SPI, ADCs, etc.). Additional 68000 processors have been developed for specialized applications. The original Palm PDA has a 68EZ328 Dragon Ball processor, also based on a CPU32 core, which incorporates an LCD controller along with many of the common functions found in PDAs. The Dragon Ball is essentially a PDA on a chip—just adds memory. The ucLinux fraternity uses a Dragon Ball processor in its small embedded controller board.

The 68000 architecture was upgraded to RISC with the Cold Fire series of processors. These see extensive use in industrial control and network interfaces.

The 68000 series of processors are good general-purpose processors. They have a nice instruction set, are easy (and fun) to write code for, and are relatively easy to build computers around. They have large address spaces and asynchronous operation, allowing them to be interfaced to a wide variety of memory and peripherals of varying operating speeds. They are used in industrial control and monitoring, and also in consumer electronics.

In this chapter, we'll look at the standard 68000 processor. More than likely, this is not the processor you will use in a design. Rather, you will choose a 68000-based integrated controller that better suits your needs. So, why look at a standard 68000 and not one of the derivatives? First, there are far too many diverse 68000-based processors to cover. Second, since these processors are all based on the 68000, understanding the basic 68000 is a great starting point. Finally, all the derivatives are generally easier to use than the original, so if you can design around a standard 68000, then you can design for a derivative processor as well.

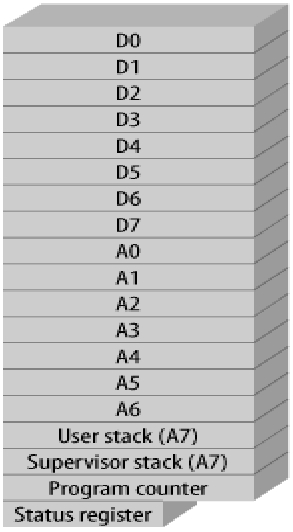
Understanding the 68000 gives you access to a wide range of available processors. There are dozens of commercial C compilers and assemblers available for the 68000 family, as well as a number of public-domain compilers as well. The 68000 is fully supported by the gnu development suite. Both Linux and BSD are also available for the 68000, as well as for numerous commercial operating systems.

The 68000 Architecture

The 68000 has eight 32-bit data registers (D0-D7), eight 32-bit address registers (A0-A7), a 32-bit program counter, two 32-bit stack pointers, and a 16-bit status register (Figure). The processor is capable of handling data as 32-bit long words, 16-bit words, bytes, or bits.

The processor has two modes of operation: supervisor mode (operating system) and user mode (applications). The mode of operation is made available to external hardware, thereby giving the address decoder the ability to have separate supervisor and user spaces.

68000 programmer's model

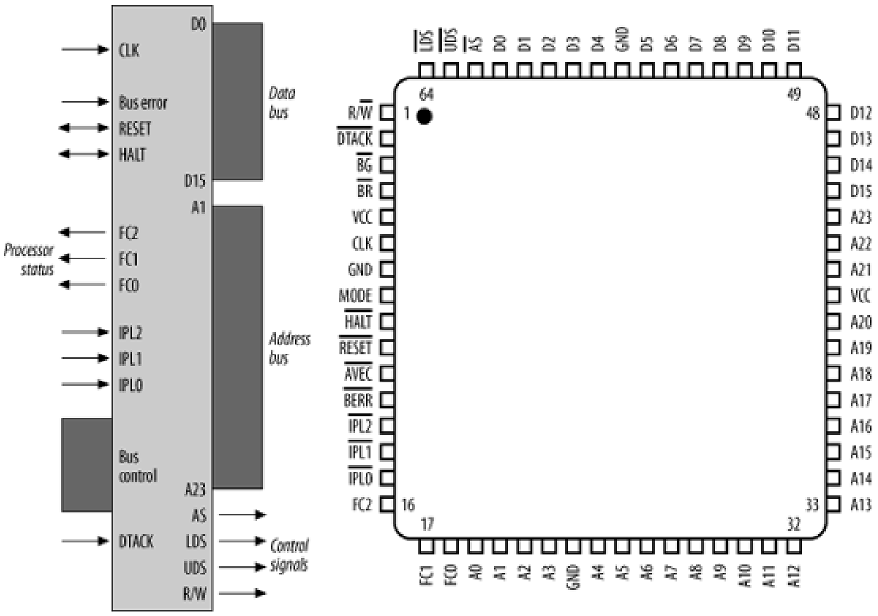


The standard 68000 is just a conventional, bus-based processor. A block diagram of a generic 68000-series processor is shown in below Figure. The figure also shows the pins for a sample 68000-series processor. The pins and signals of 68000s can vary from one device to another, but they all have the same core functionality. The embedded controllers add to this basic functionality with additional I/O capability. We'll look at the pins for the

MC68EC000 shortly. The original 68000 has a 23-bit address bus (A1 to A23), giving it access to a memory space of 16M, and a 16-bit data bus. Most other processors based on the 68000 architecture have address and data buses of 32 bits and can therefore access up to 4G of memory.

The processors have an input clock that drives all processor operation. Memory accesses typically take eight input clock cycles, provided that wait states are not introduced. Many processors based on the 68000 incorporate in-built address coding and software-configurable wait-state generation, making interfacing much simpler.

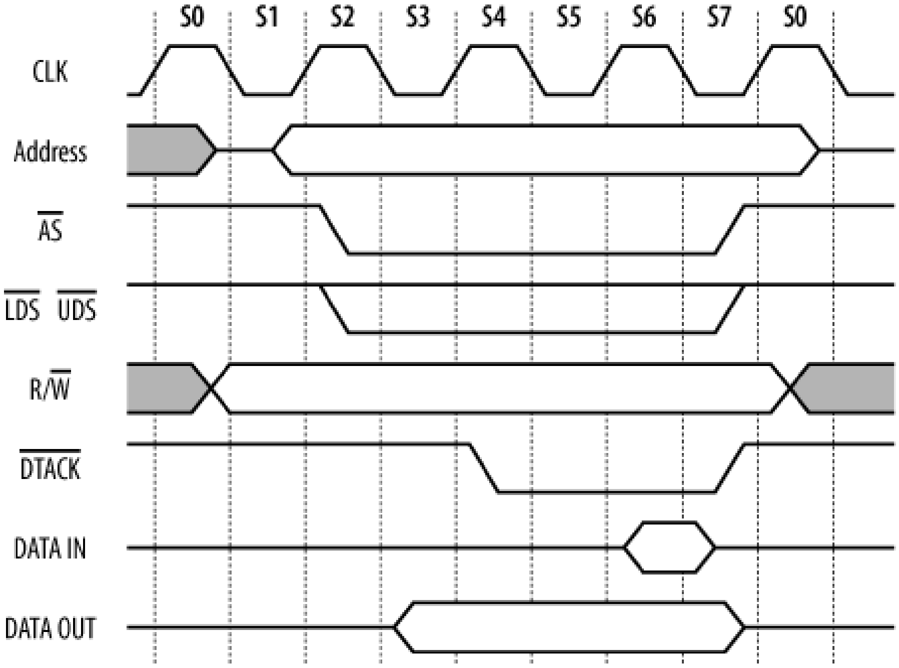
MC68000 block diagram and pinout



The processors have an address strobe (AS) indicating when a valid address is present on the bus, data strobes (LDS, UDS) indicating valid data, and a R/W line that shows the direction of the transfer. In addition, a Data Transfer Acknowledge input, DTACK, is used by external devices to indicate to the processor that it may terminate its current memory cycle. (Some 68000 processors call their Data Transfer Acknowledge DTACKB.) The function code outputs (FC0, FC1, and FC2) indicate the current operating mode (supervisor or user) of the processor. Bus Error (BERR) is used by an external address decoder to indicate an error condition. This allows the system to trap out accesses to unused regions of memory space, or in combination with the status lines, to detect user access to memory space allocated for supervisor use only. For example, if a program crashes and, in the process of crashing, attempts to access a region of memory to which no device is allocated, the address decoder is able to signal that fault back to the processor. An assertion of BERR causes the processor to execute an interrupt and take appropriate action. HALT is used to suspend processor operation without generating a reset. Three interrupt inputs (IPL0, IPL1, and

IPL2) are used to generate seven levels of external interrupt handling. Bus Grant (BG) and Bus Request (BR) are DMA control signals by which another processor can arbitrate to acquire the computer's buses. The MODE pin, present on only some 68000 processors, determines whether the 68000 uses its data bus as 16 bits or 8 bits. MODE is sampled as the processor comes out of reset. AVEC, also found in only some 68000 processors, determines whether the processor uses auto-vectoring for its interrupts. If auto-vectoring is enabled, the processor will expect the interrupting peripheral to supply the appropriate vector. This allows a peripheral to specify what type of action the processor needs to take when a given interrupt is generated. Other 68000 processors may have other signals as well, but these are the main ones. The basic timing diagram for a 68000 memory access is shown in Figure.

MC68000 timing diagram



The memory cycle of a 68000 is divided into a number of clock states, S0 through S7. The cycle begins with state S0. The processor validates R/ for the coming cycle, sending it low for a write access, driving it high for a read access. The processor also tristates its address bus from the previous memory access. By S2, the processor has output a valid address and drives the address strobe (AS) low, indicating that a valid address is present. The lower and upper data strobes (LDS and UDS) go low as appropriate and indicate the width of the memory access taking place. For a 16-bit transfer, both LDS and UDS assert. For an 8-bit transfer, only one of LDS or UDS asserts, depending on whether the upper byte or lower byte is being transferred. If the current memory access is a write cycle, the processor outputs valid data in state S3. At this point, all outputs from the processor are valid, and the processor waits for the device being accessed to respond.

At the falling edge of the clock in S4, the processor begins checking the state of the DTACK input. If DTACK is high, the processor inserts wait states and continues to do so until DTACK is found to be low on the falling edge of the clock. (You'll learn how to generate wait states in "Wait States," later in the chapter.) When DTACK is low, the processor recognizes this as an indication that the device being accessed has had sufficient time to respond and prepares to terminate the cycle. If the cycle is a read cycle, the processor will latch data on the falling edge of the clock in state S6. If it is a write cycle, the device being accessed will latch data as the data strobes go high in S7.

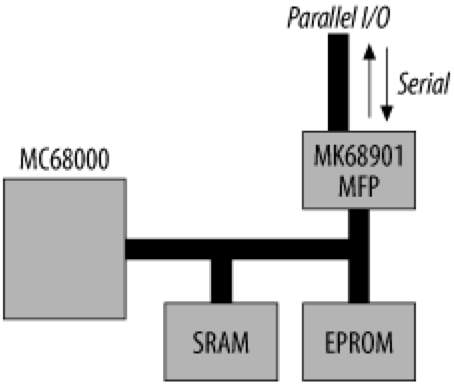
Support for synchronous operation is also provided for, using control signals found in the old 6800 series of processors. Since 6800s have long since passed into history, and 6800-based peripherals are now exceptionally rare, just ignore the 6800 control signals. Most 68000-based derivative processors no longer include support for 6800 peripherals.

A Simple 68000-Based Computer

Objective:

Let's look now at a small 68000-based computer. For simplicity, we'll give it just a small amount of memory and a single peripheral, an MK68901 MFP (Multi-Function Peripheral) produced by ST Electronics. The MFP gives us a UART, parallel I/O, and interrupt control. A block diagram of the system is shown in Figure.

68000-based computer



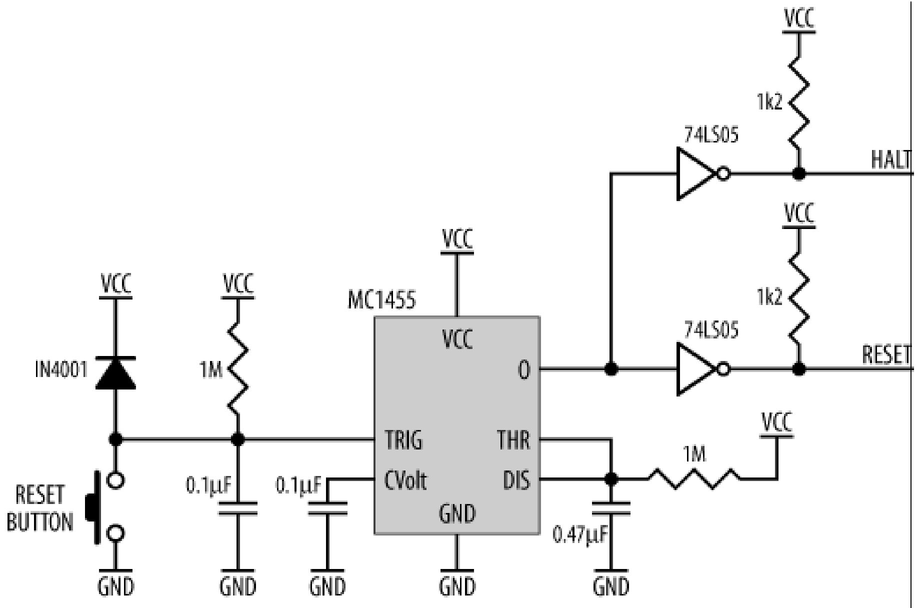
This system is designed with only a small amount of memory, to keep the design uncomplicated. While this is not much compared to many desktop machines, it is sufficient for many small, control applications. This design could be used for a number of simple applications. The counters of the MK68901 may be used to monitor external event pulses or to generate PWM for motor control. This computer could also be used to accept commands through its serial port and activate (or deactivate) external subsystems using the parallel I/O pins of the MK68901. This basic design could also be adapted to provide a bridge between an RS-232C interface and a parallel port. You could use this to interface a parallel-port printer to a serial-port-only computer. Alternatively, you could use it to put a serial modem on your PC's parallel port. Using the bus-interfacing techniques we learned in the AVR chapter, you could add additional peripherals such as ADCs and DACs, Ethernet, or a whole range of other devices. The list of possible applications is endless. And it all starts with this core design.

So, let's start our tour of a 68000-based computer system. We'll look at the reset circuit, address decoder, I/O, and memory.

Reset Circuit

To reset an MC68000, both RESET and HALT must be driven low simultaneously. In addition, both of these signal lines may also act as outputs from the processor. Therefore, both must be independently driven by the reset circuit through open-collector gates. The conventional way of doing a 68000 reset circuit is shown in Figure.

Reset circuit



The MC1455 will respond to a disruption on VCC by sending its output low. This output is used to drive RESET and HALT low simultaneously. In normal operation, RESET is held high by the pull-up resistor, unless pulled low through the reset switch being pressed. The diode is present to remove any glitches that might send RESET above VCC.

A better reset circuit is shown in Figure, using a MAX825 integrated reset controller. Again, both RESET and HALT need to be driven low.

Address Decoder

Logic to perform address decoding and the generation of separate read and write strobes is implemented in a PAL. In each case, (Address Strobe) of the processor is used as an indication of a valid address present on the bus. The address-decode equations are as follows:

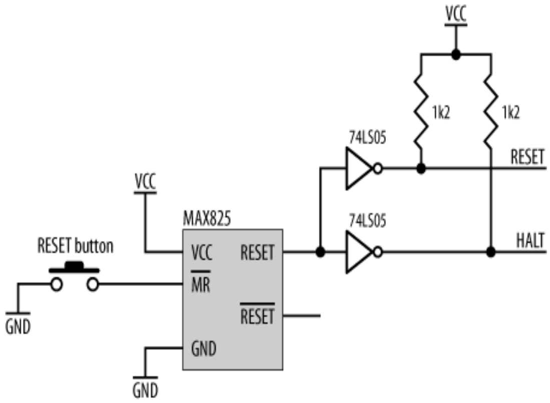
ROM = /(/AS * /A23 * /A22)

RAM0 = /(/AS * /A23 * A22 * /LDS)

RAM1 = /(/AS * /A23 * A22 * /UDS)

MFP = /(/AS * A23 * /A22)

MAX825 reset circuit for a 68000



With the exception of the MFP, which generates its own DTACK, DTACK for all other devices is generated as part of the address decoding. Since DTACK from the PAL must be OR-tied with DTACK from the MFP, it must be driven from an open-collector gate. Therefore, we generate an active-high acknowledge (which we'll designate TACK) from the PAL and invert this through a 74LS05 open-collector inverter.

The PAL equation to generate TACK is simply:

TACK = (/AS * MFP)

Therefore, TACK is active (high) whenever the processor accesses its address space, so long as it is not accessing the MFP. If the address strobe is high, or if there is an access to the MFP, then TACK is low. The TACK output from the PAL is inverted through an open-collector 74LS05 and "OR-tied" (directly connected together) with DTACK from the MFP. DTACK requires a pull-up 1 kΩ resistor, since this input must have a sharp rise time. A block diagram is shown in Figure.

No provision for generating a BERR is made because our simple address decoding allocates all of the address space. If we had any unused regions of the memory space, we would use our address decoder to generate a BERR when accesses to the unused regions were made.

The PAL equations to generate separate read and write strobes for the memory chips are:

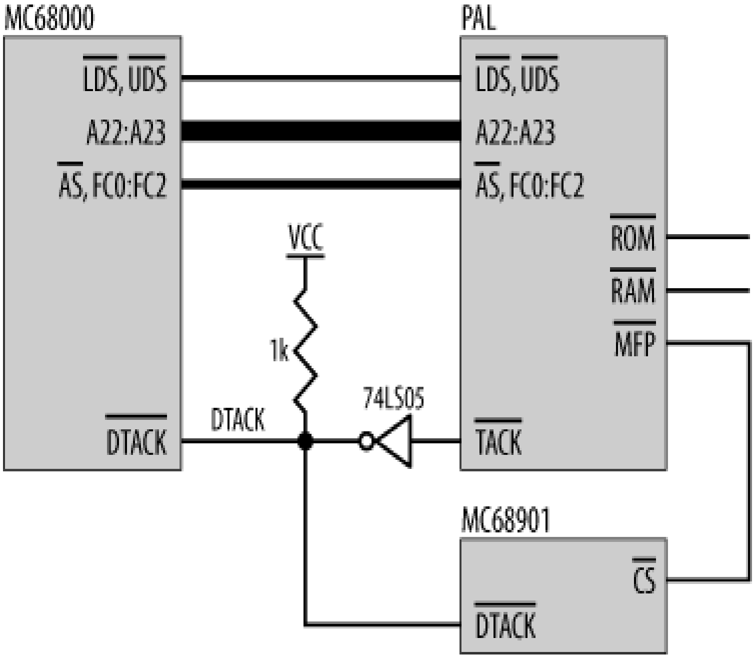
UWE = /(/UDS * RW)

LWE = /(/LDS * RW)

UOE = /(/UDS * /RW)

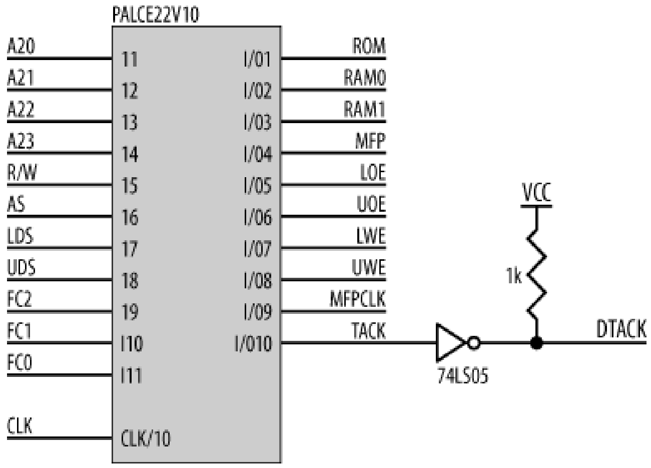
LOE = /(/LDS * /RW)

Address decode and DTACK generation


MC68000
$\overline{\text{LDS}}$, $\overline{\text{UDS}}$
A22:A23
$\overline{\text{AS}}$, FC0:FC2
$\overline{\text{DTACK}}$
PAL
$\overline{\text{LDS}}$, $\overline{\text{UDS}}$
A22:A23
$\overline{\text{AS}}$, FC0:FC2
$\overline{\text{ROM}}$
$\overline{\text{RAM}}$
$\overline{\text{MFP}}$
$\overline{\text{TACK}}$
VCC
1k
74LS05
DTACK
MC68901
$\overline{\text{CS}}$
$\overline{\text{DTACK}}$

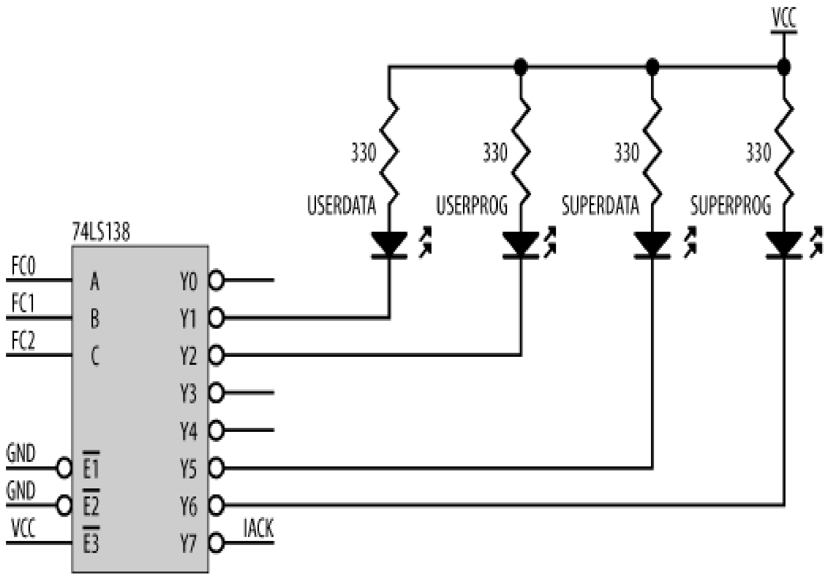
The connections for the PAL are shown in Figure. Additional addresses are brought into the PAL to allow for future changes to the memory map. The processor's clock (CLK) is used by the PAL to generate the clock for the MFP (MFPCLK).

Address decode and system-logic PAL



The function code outputs (FC0-FC2) can be decoded using a 74LS138 demultiplexer to drive three LEDs (Figure). These provide a visible indication of processor status. The function codes could also be used by the address decoder if you wanted to have separate user and supervisor address spaces. Many of the more sophisticated peripheral chips (such as the MFP) require the processor to acknowledge when they have generated an interrupt. The 74LS138 also uses the function codes to generate an interrupt acknowledge (IACK) for peripherals, since the function codes also indicate an IACK condition.

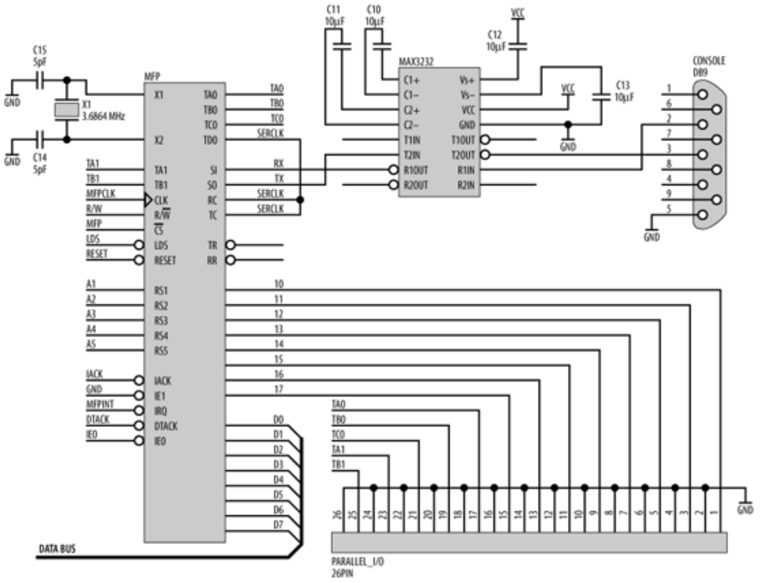
Status LEDs indicating processor mode



I/O

The MK68901 Multifunction Peripheral (MFP) provides a serial port, as well as basic parallel I/O functions, a 16-source interrupt controller, and four 8-bit timers. The MK68901 has an internal oscillator that drives the internal timers. A timer output (TD0) is fed back into the MFP as the clock for the serial interface. The internal oscillator must therefore run at a frequency appropriate for RS-232C. An external 3.6864 MHz crystal drives the oscillator. This input clock can be divided up by the MFP, providing the appropriate baud rates for the serial port. The serial lines from the MFP are converted to RS-232C voltage levels by a MAX3232 level shifter. A 9-pin, D-type connector provides access to the RS-232C signals. The parallel I/O lines and timer inputs and outputs are also made available through a 26-pin IDC connector. The schematic for the MFP is shown in Figure.

Multifunction Peripheral

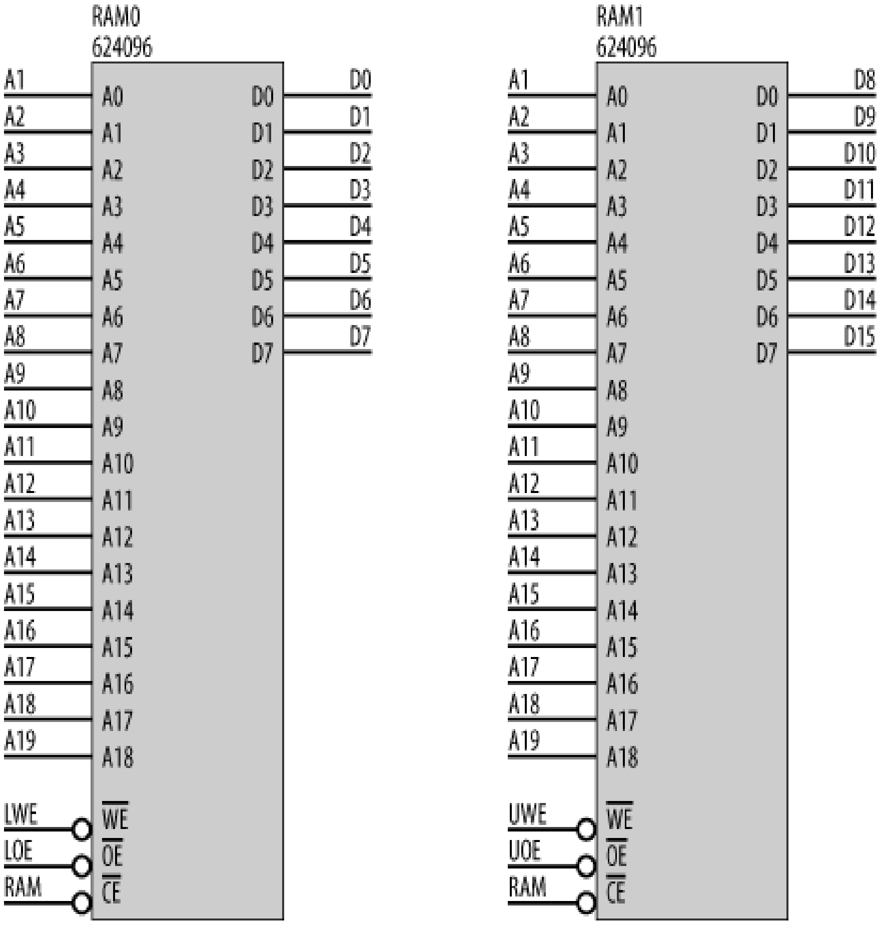


Memory

The system is designed with 256K of EPROM and 512K of static RAM. The connections to the SRAM are shown in Figure. Note that since the data bus of a 68000 is 16 bits wide, two SRAMs are required. For 68000-based derivatives with 32-bit external data buses, four memory chips would be required in parallel. Note how half the data bus goes to one chip and the other half goes to the other chip.

Now, note the address lines going to the SRAMs. The lowest address bit from the processor is A1, and this is connected to the A0 inputs of the SRAMs, and so on. Since the processor accesses external memory in 16-bit words, A1 represents the least significant address bit. In other words, as you move from word to subsequent word in memory, it is A1 that increments. However, A0 is the least significant address bit of the SRAMs, but since the two SRAMs together form a 16-bit word of memory, the A0 of the SRAMs must connect to A1 of the processor. The other address bits follow on from that starting point.

Interfacing to SRAM

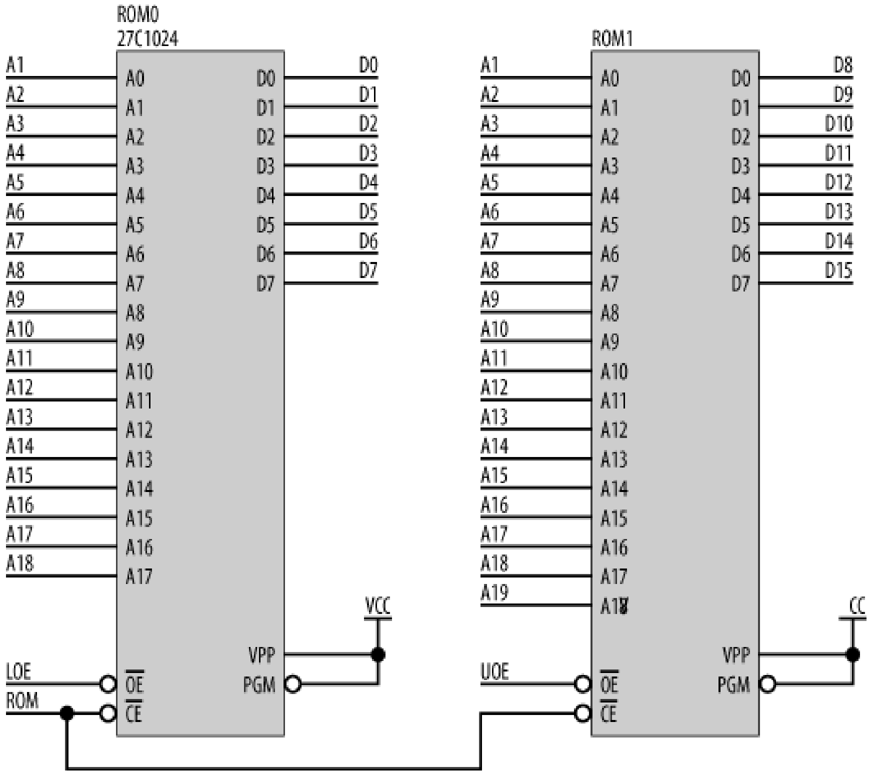


Similarly, the connections for the ROMs are shown in Figure.

Wait States

Depending on the speed of your processor and the access times of your memory and peripheral chips, it may be necessary to introduce wait states into the 68000's memory cycle. Wait-state generation follows basically the same principle for processors that support asynchronous memory cycles. The processor will have an input (sometimes more than one) that will cause it to delay the memory cycle, giving slower

Interfacing to EPROMs

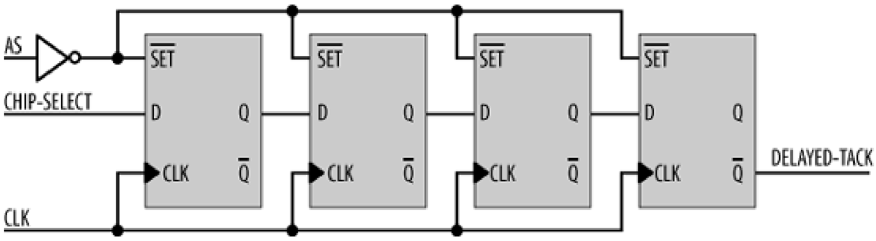


devices time to respond. In the case of the 68000, that input is DTACK . To insert a wait state for a given device, we need to detect an access to that device and hold DTACK inactive for the required additional clock cycles. In other words, we need to use the chip select for a given device to delay DTACK going low. The circuit to do this is simple and is best done inside a PAL or other programmable logic device. This facilitates changing the wait-state generator if faster parts are used in the design at a later stage. The wait-state generator consists of a series of D-type flip-flops(Figure). Each flip-flop represents an additional clock cycle that the transfer acknowledges is delayed.

A flip-flop is a logic element that feeds the D input through to the Q output on the changing edge of a clock pulse.

Between memory cycles, the address strobe, AS, goes high. It is first inverted and then connected to the active-low SET input of each of the flip-flops. Thus, the output of each of the flip-flops is driven high between each memory cycle. This resets them from any previous cycle. The address decoder generates a chip select for the particular device, and this is connected to the D input of the first flip-flop. So, on each successive clock pulse, the `0` provided by the chip select is clocked through from one flip-flop to the next. After four clock pulses, the `0` has arrived at the Q output of the last flip-flop. The inverted output of this flip-flop, , becomes a `1`. This is then output by the PAL to be inverted by the 74LS05 open-collector inverter to provide DTACK for the processor. For additional wait states, add more flip-flops. For several devices requiring different numbers of wait states, use their combined chip selects to feed the D input of the first flip-flop; then "tap" into the wait-state generator at different stages for the required delay. Each of these taps is gated with the respective chip select to enable/disable that output before they are all recombined to generate a unified acknowledge for the processor.

Wait-state generator



Most processors that support wait states now include inbuilt, software-configurable wait-state generators. This makes the task of designing the system logic much simpler.

Summary:

- The Motorola MC68000 was introduced in 1979 as the successor to its 8-bit 6800 family.
- The 68000 has eight 32-bit data registers (D0-D7), eight 32-bit address registers (A0-A7), a 32-bit program counter, two 32-bit stack pointers, and a 16-bit status register.
- The original 68000 has a 23-bit address bus (A1 to A23), giving it access to a memory space of 16M, and a 16-bit data bus.
- The MODE pin, present on only some 68000 processors, determines whether the 68000 uses its data bus as 16 bits or 8 bits.
- Depending on the speed of your processor and the access times of your memory and peripheral chips, it may be necessary to introduce wait states into the 68000's memory cycle.
- A 9-pin, D-type connector provides access to the RS-232C signals. The parallel I/O lines and timer inputs and outputs are also made available through a 26-pin IDC connector.
- The MK68901 Multifunction Peripheral (MFP) provides a serial port, as well as basic parallel I/O functions, a 16-source interrupt controller, and four 8-bit timers.

- The 74LS138 also uses the function codes to generate an interrupt acknowledge (IACK) for peripherals, since the function codes also indicate an IACK condition.

Questions:

- What is programmer’s model of 68000?
- Explain the timing diagram of MC68000 for memory access?
- How to design a basic computer using MC68000?
- How MK68901 is interfaced with MC68000?
- How to interface SRAM with MC68000?

References:

- Analog Interfacing To Embedded Microprocessors – STUART R. BALL
- C Programming for Embedded Systems – KIRK ZURELL
- Design with 8051- FRONTLINE ELECTRONICS
- Embedded Controller Hardware Design - Ken Arnold
- Embedded Software The Works – colin walls
- Embedded Systems Firmware Demystified - Ed Sutter
- Embedded_Controller_Hardware_Design – KEN ARNOLD
- Programming Embedded Systems in C and C++ - Michael Barr
- The Art of Designing Embedded Systems - Jack G. Ganssle

16. The DSP56800

Objective:

Unlike the conventional DSP56000 with its 24-bit architecture, the DSP56800 series has a 16-bit architecture better suited to small-scale control applications. It is fixed-point (integer) only, which is fine for most control applications. If necessary, floating- point arithmetic can be synthesized in software.

The architecture is based on four functional units, each with their own registers, operating independently and in parallel with the other units. These functional units are the program controller, which is responsible for software execution; the Address Generation Unit (AGU), which handles bus accesses; the Data ALU, which performs the arithmetic operations; and the bit-manipulation unit for efficient and rapid bit-based operations.

The independent operation of these units allows for very efficient and fast software execution. While the Data ALU or bit-manipulation unit are performing an operation specified by an instruction, the AGU can be generating addresses for the execution of another instruction, while the program controller can be fetching yet another instruction for execution. The instruction set directly supports this parallelism. To accomplish this high internal throughput, the processor has not one but three internal address buses and four internal data buses (three data buses for the core and one for peripherals). Two operands may be sourced from the internal memory and operated on in a single instruction. The result is that the architecture achieves a throughput of 40 MIPS on an 80 MHz clock. That's RISC-like performance with a CISC-like instruction set. In other words, that's a lot of punch.

It has hardware looping using the DO and REP instructions. DO allows you to specify a block of code (of any size) and have the processor execute it as a loop in hardware. You don't need a counter test and conditional branch instruction at each iteration, saving processor execution overhead. REP allows the repetition of a single instruction, and REPs can be nested inside DO loops. As such, you have very versatile looping capability with no overhead. Loops on a DSP are fast.

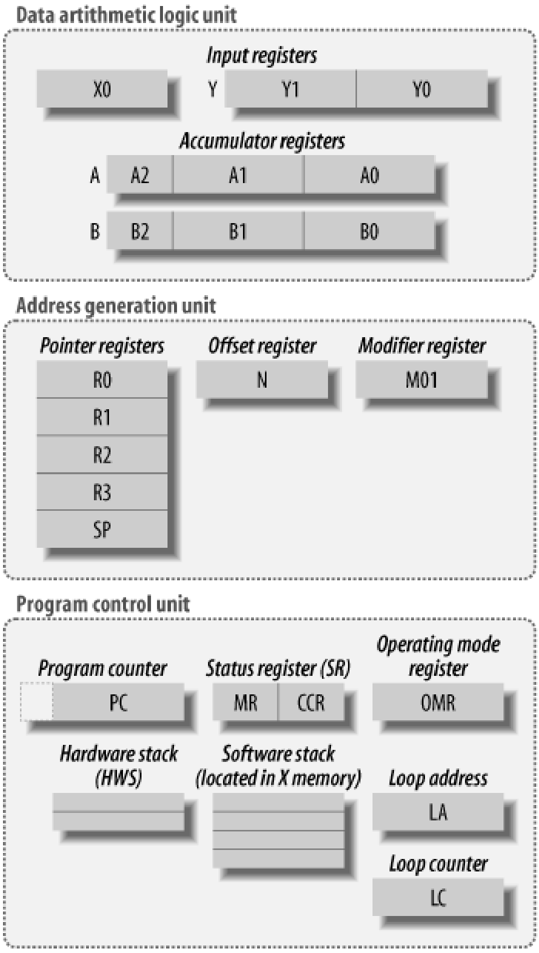
The programmer's model for the DSP56800 core is shown in Figure.

The processor has two 36-bit accumulators, a 16 x 16-bit multiply and Accumulate (MAC) unit, and a 16-bit barrel shifter. The MAC allows you to multiply two numbers and then add the result to a growing total, all with a single instruction. MACs allow for efficient execution of many signal-processing algorithms, as well as neuro-fuzzy code.

The barrel shifter allows you to shift up to 16 bits in either direction in a single cycle. So, if you want to shift an operand 15 bits to the left, a conventional processor would require 15 separate shift-left instructions (or one shift-left, a loop, a counter variable, and a conditional test for the loop). The DSP56800, like many DSPs, can perform this operation in just one cycle.

In short, the DSP56800 has very tight and efficient code with high functionality that it executes exceptionally quickly. It is a fast processor around which it is easy to design a powerful embedded computer system.

We'll look at how you design a system based on the DSP56805 processor, a member of the DSP56800 family specifically designed for industrial control. The DSP56805 has an internal 1K program RAM, 4K of bootstrap ROM (for loading boot software from an external memory or peripheral, 63K of program flash, 8K of data flash, and 4K of data RAM. The processors also have external data and address buses, so the processor's memory can be expanded well beyond its internal resources. It has a 64K x 16-bit address space, giving access to 128K (bytes) of external memory.



The DSP56800 processors also provide the ability to separate data and program spaces, thereby doubling the external address space. The processor also has a programmable wait-state generator, simplifying interfacing to external devices. The generator may be programmed to provide 0, 4, 8, or 12 wait states for accesses to a given device.

DSP56800s in general come with a range of inbuilt peripherals, including SPI ports (sometimes two), several 16-bit general-purpose timers, a watchdog timer (called a Computer Operating Properly, or COP, timer by

Free scale Semiconductor), a timer for real-time operation, a Synchronous Serial Interface (SSI) for accessing audio codec's (combined ADCs and DACs) and other DSPs, and general-purpose I/O lines. The DSP56805 adds two 6-channel Pulse Width Modulation (PWM) units for motor control and other uses, two 4-channel ADCs at a resolution of 12 bits per channel, and two quadrature decoders for measuring motor positions. It also has a CAN networking module, two serial ports (called Serial Communication Interfaces, or SCIs, by Free scale Semiconductor), and 14 dedicated and 18 shared I/O lines.

The processors operate from a supply voltage of between 3.0 V and 3.6 V but have 5 V-tolerant inputs, making interfacing to a wide variety of devices easy. (Other DSP56800s may operate on a supply voltage of between 4.57 V and 5.5 V, depending on the particular chip.) The processor has several low-power and sleep modes, making it ideal for battery-powered systems.

All DSP56800 processors incorporate a JTAG (Joint Test Action Group) port for interfacing to specialized debugging instruments. The JTAG port also allows direct access to the processor's onboard flash program memory, making the job of downloading new code simple and fast. All in all, quite a nice processor. So, let's look at how you build a system based on one. For simplicity, I'll look at each subsystem in turn.

A DSP56805-Based Computer

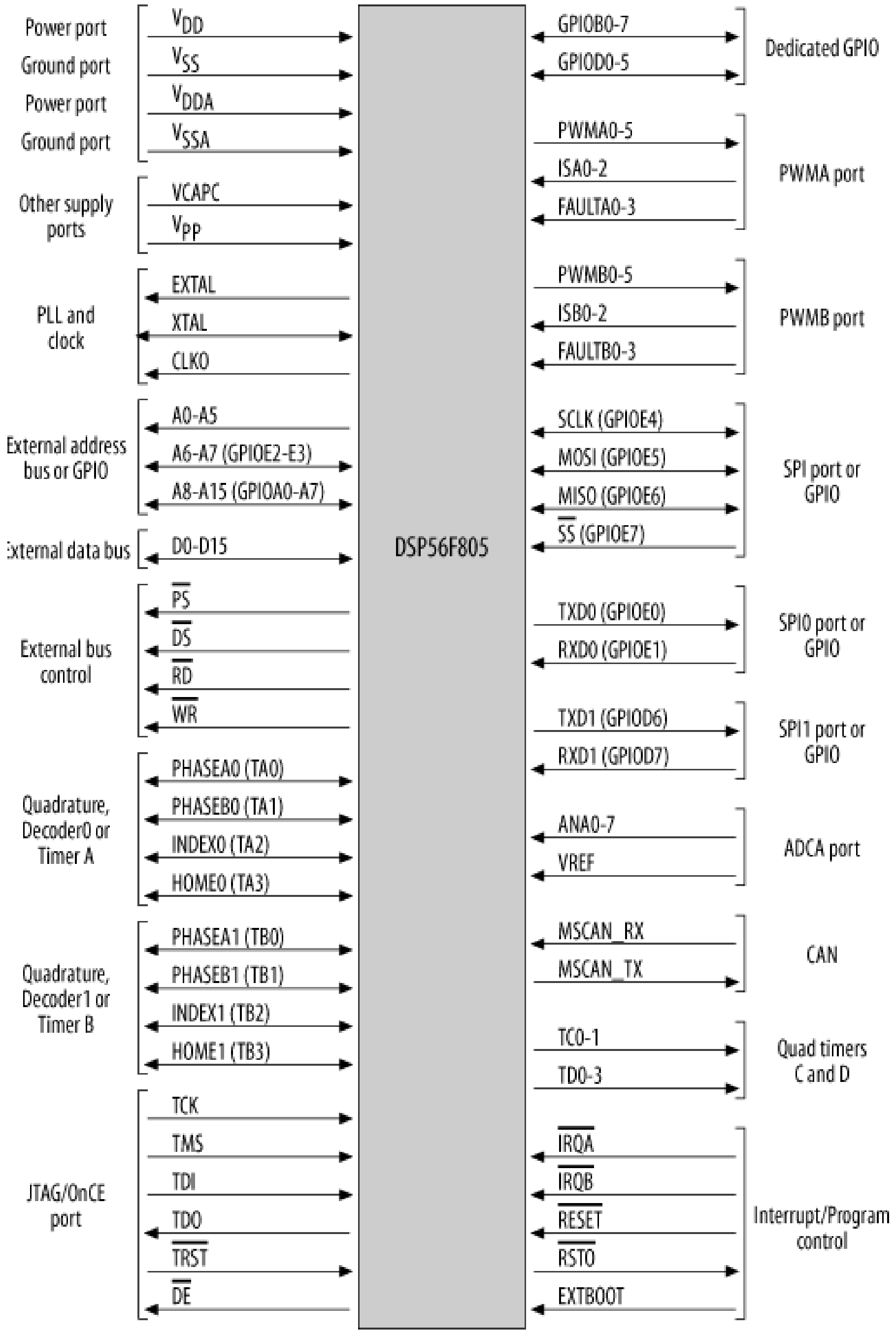
The DSP56805 has nine power pins. Each of these must be decoupled to ground using 100 nF ceramic capacitors. Each capacitor should be placed as close as possible to its respective power pin. Since this processor can operate at a relatively high speed, and can therefore generate a lot of noise, a four-layer circuit board is the preferred option for construction. As with any design, any unused inputs must be tied inactive. A block diagram of the DSP56805 is shown in Figure in next page.

Oscillator

Like all processors, the DSP56805 requires a clock signal. The processor can operate from an oscillator frequency of up to 80 MHz (giving 40 MIPS) or as slow as a few MHz to save power. The processor may even have its clock completely stopped (so-called "DC operation," meaning the clock is no longer an AC signal) to further save power. (This processor's sibling, the DSP56801, has a complete internal oscillator and so requires no external clock-generation circuit).

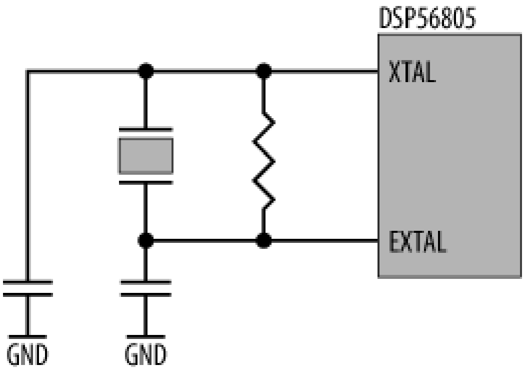
The processor has an inbuilt oscillator circuit, requiring only an external crystal in the range of 4 MHz to 8 MHz and support components. From this low crystal frequency, the processor internally synthesizes a clock speed of between 40 MHz and 110 MHz under software control. Note that while the clock-generation circuit is able to produce 110 MHz, the processor isn't able to operate at that speed. So keep the speed below 80 MHz, and the processor, your software, and you will all be happy.

DSP56805 block diagram



In a typical application, the crystal frequency is 8 MHz, with a resistor value of 10 MΩ. Decoupling capacitors are approximately 15 pF or so. However, the values of the resistor and capacitors required can vary, so make sure you check the technical data from the crystal manufacturer. It will tell you specifically what values to use for a particular crystal.

Crystal oscillator circuit

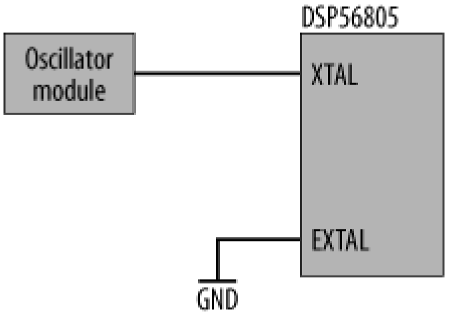


Alternatively, you could use an external oscillator module to generate the processor's clock (Figure). The module's output is connected to the XTAL input of the processor. When operating in this configuration, EXTAL must be connected to ground.

Reset and Interrupts

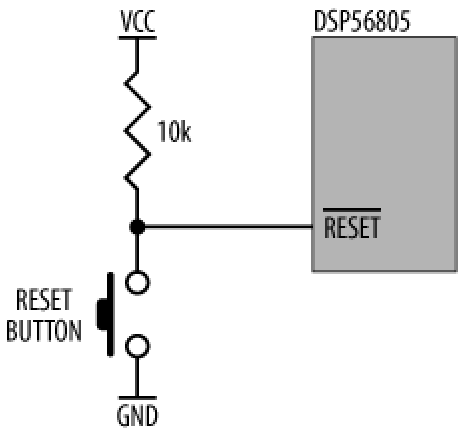
The DSP56805 has an internal power-on circuit to correctly start up the processor. It also has a watchdog reset circuit, driven by an internal timer, to recover the processor from a software crash. So, all we need to do is to provide our system with an

Oscillator module



external reset so we can manually restart the machine by pressing a button. Normally, such a reset circuit would need to debounce the button press and also ensure that the reset state was held for a minimum period of time. On the DSP56805, life is much simpler. The processor incorporates internal debounce circuitry on its input. Further, it has circuitry that ensures that a reset is held for the appropriate duration. So, our external reset circuit is simply a push-button and a pull-up resistor (Figure).

External reset on a DSP56805



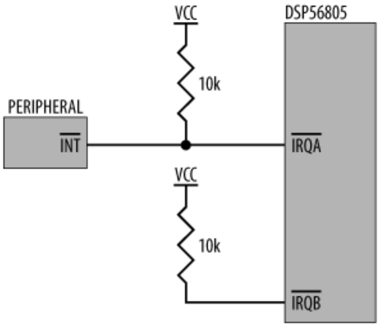
The DSP56805 can boot from external memory or from its internal ROM for single-chip operation. An input pin, EXTBOOT, is sampled as the processor comes out of reset. If EXTBOOT is pulled low, the processor executes code from the internal ROM. This is known as Mode 0 operation. There are two forms of Mode 0. Mode 0A maps all memory as internal, whereas Mode 0B maps the lower 32K words (64K bytes) of the address space as internal and the upper 32K words as external. Mode 0A is the default mode, and Mode 0B may be entered only under software control.

If the EXTBOOT pin is high upon exiting reset, then the processor boots from external memory. This is known as Mode 3 operation. (There is no Mode 1 or Mode 2, as these are reserved for ROM-based DSP56800 processors.) Once operational, the processor can toggle from one mode to the other under software control.

Other DSP56800 processors have variations of the operating modes and memory maps, so, as always, check the datasheet for the particular processor you are using.

Aside from numerous internal sources of interrupts (from the onboard peripherals), the DSP56805 has two external interrupt sources, IRQA and IRQB. These may be used by external-interface peripherals (or even external systems) to gain the processor's attention. Whether they are connected to an external interrupt source or not, they require an external pull-up resistor. In the example given (Figure), IRQA has an interrupt source from a peripheral, while IRQB is unused.

External interrupt sources

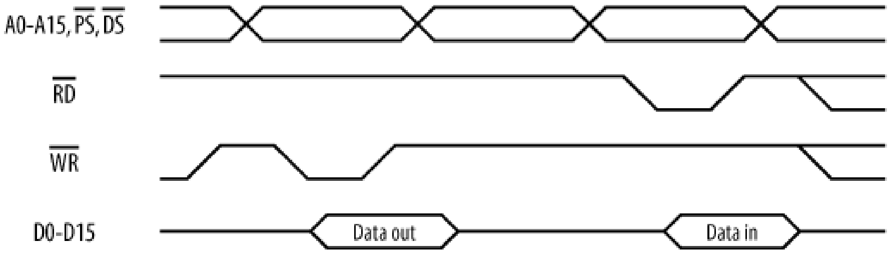


External Memory

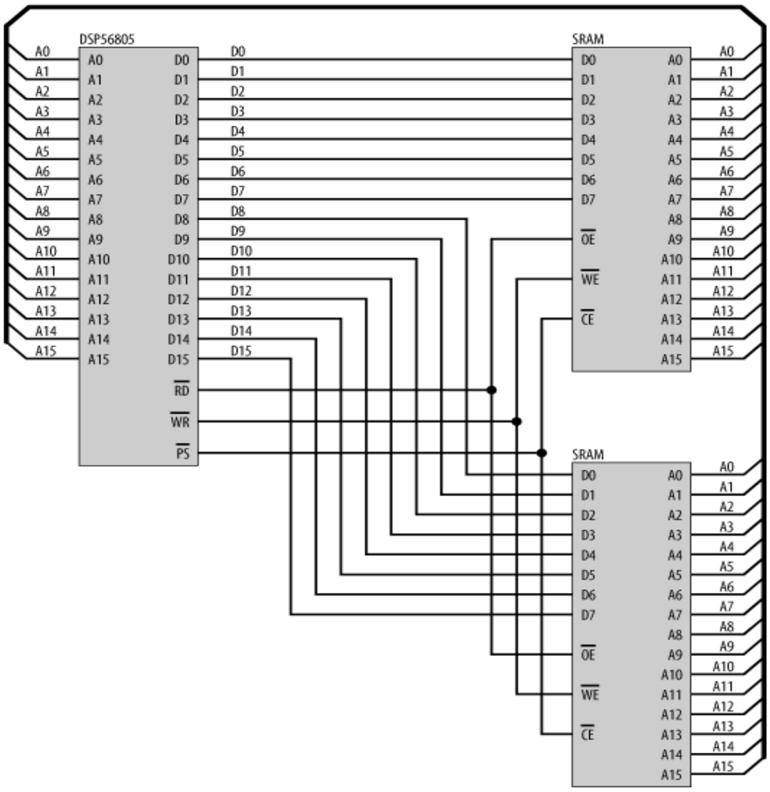
The processor has an external 16-bit data bus that serves for accesses to both external program memory and external data memory. Data and program memory can exist within the same memory chips, or separate data and program address spaces may be implemented. The processor has two outputs, (Program Strobe) and (Data Strobe), which indicate the type of memory access.

The timing for a DSP56805 write cycle followed by a read cycle is shown in Figure. Since the processor has a programmable wait-state generator, external memory devices or peripherals of varying response times may be accommodated.

DSP56805 memory cycle

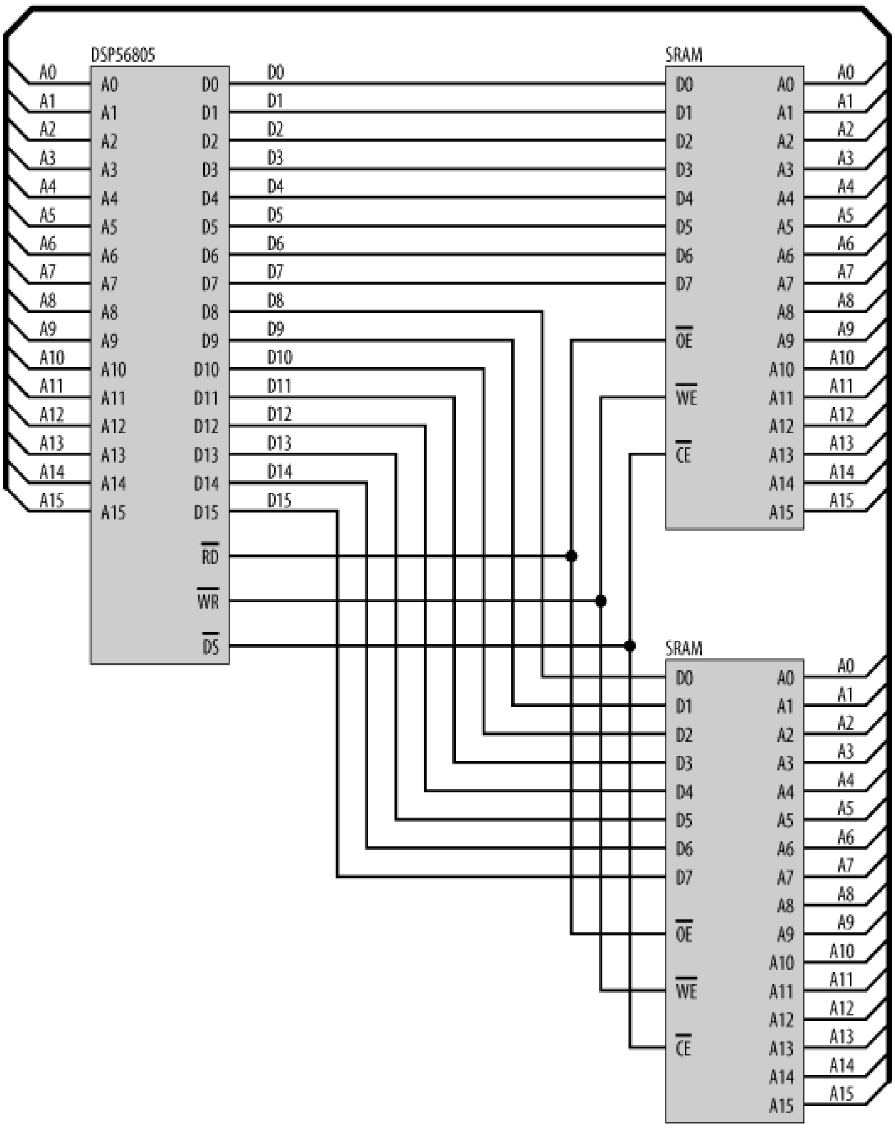


The DSP56805 may be connected to memory using a "glueless" interface. This means no external logic is required. The connections for interfacing a DSP56805 to two 64K program SRAMs are shown in Figure.



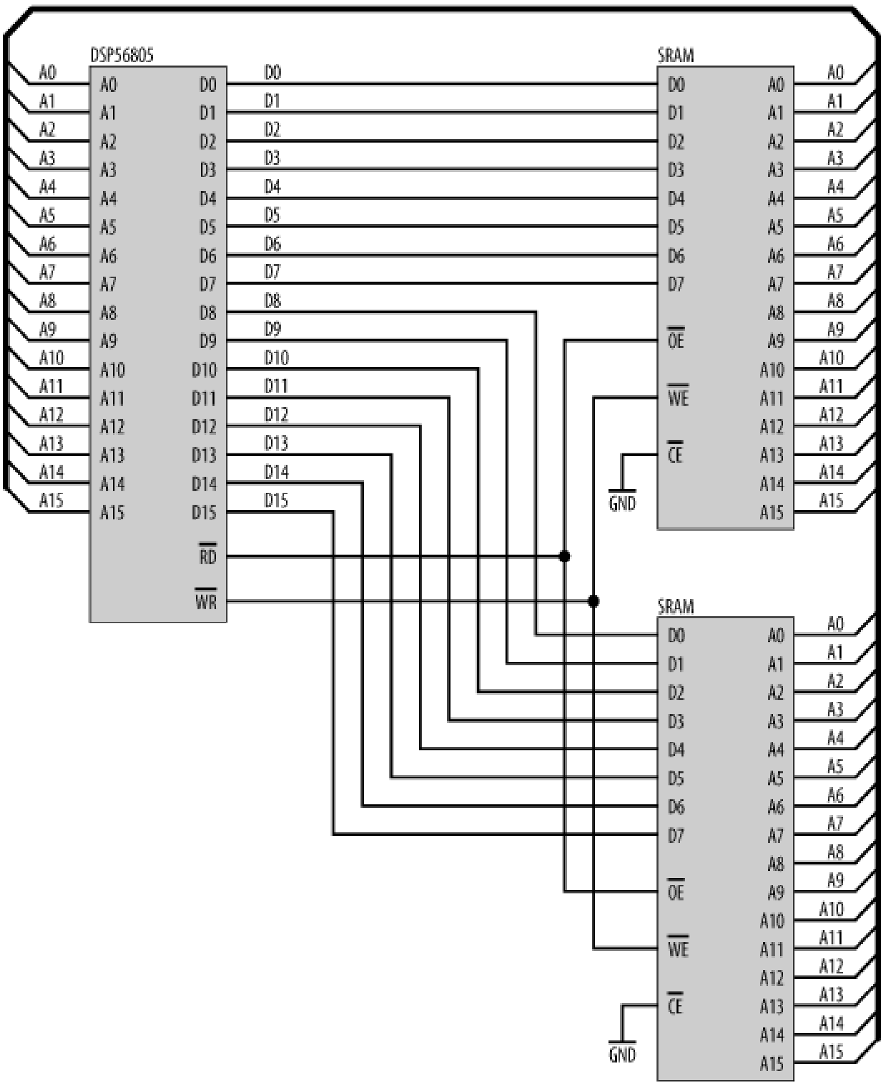
Interfacing the DSP56805 to program SRAM

When accessing the program address space, PS is low, and so this may be used as a chip select to the SRAMs. Similarly, the same configuration may be used for data memory, except that in this case, DS becomes the chip select (Figure). Note that when I say "program memory" or "data memory," I'm simply referring to the intended use of these chips, not distinguishing between different types of memory chip. The same type of SRAM chips will suffice for both regions.


DSP56805
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15
D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 D10 D11 D12 D13 D14 D15
RD
WR
DS
SRAM
D0 D1 D2 D3 D4 D5 D6 D7
OE
WE
CE
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15
SRAM
D0 D1 D2 D3 D4 D5 D6 D7
OE
WE
CE
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12 A13 A14 A15

Interfacing the DSP56805 to data SRAM

So, our DSP56805 computer has four SRAM chips in total, evenly divided between program memory and data memory. Each region has 64K x 16 bytes (two 19-bit memory chips), giving a total of 128K bytes of program space and 128K bytes of data memory. The total memory for our system is therefore 256K bytes. If more data memory is required, memory banking may be used to increase the available space. Note that you do not necessarily have to have separate program and data spaces. You can just as easily have two SRAMs total, with the program and data spaces coexisting in the same chips (Figure).



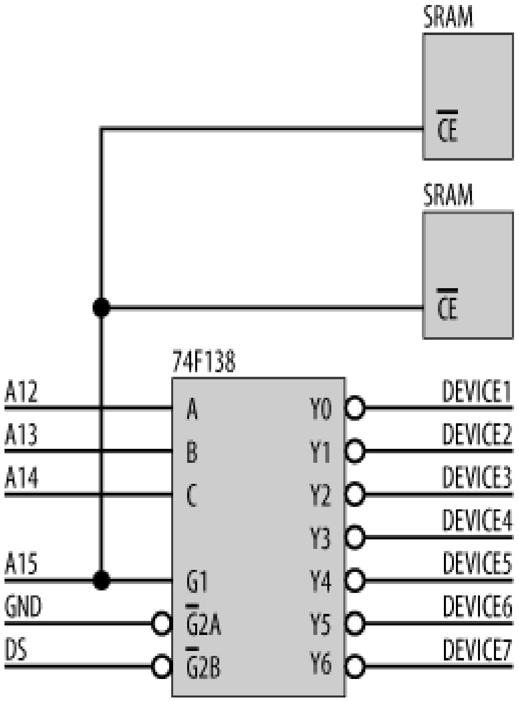
Shared program and data memory

In this case, both PS and DS are ignored, since we are no longer distinguishing between data and program spaces. The chip enable (CE) inputs of the SRAMs are simply tied to ground, so that these devices are permanently enabled. This will work because an SRAM will respond only if CE is low and either the output enable (OE) or the write enable (WE) go low as well. So in this example, it is the output enable or write enable that will activate the SRAMs. Note that permanently enabling an SRAM will increase its power consumption. Of course, we could just as easily combine DS and PS such that either going low will enable the SRAMs, but this requires extra logic, and it really isn't necessary.

If you have different types of devices within your memory space, such as a smaller data SRAM and some peripherals, then you must include DS as part of the chip enable for the SRAMs and peripherals. The most logical way to do this is to use DS as the enable to your address decoder, which in turn selects the appropriate device. Note that it must be DS for accessing peripherals, since you can't execute code directly out of a peripheral.

A sample address decoder is shown in Figure. This will select either two 32K SRAMs or one of eight peripherals within the data space.

Address decoder for two 32K SRAMs and eight peripherals



When A15 is low, the SRAMs are selected. When A15 is high and is low, the address decoder is enabled and one of the eight peripherals is selected, depending on the state of A12, A13, and A14.

Using this address-decode scheme, you can add up to eight bus-based peripherals. The processor also has a SPI interface, so that opens up another avenue for expansion. Using SPI, you can add extra ADCs, DACs, real-time clock calendars, nonvolatile data memories, as well as a host of other devices. Of course, the DSP56805 has a range of inbuilt peripherals already. Its SPI, parallel I/O, and serial port interfaces are used just as we saw with the smaller microcontrollers. The DSP56805 has a wide variety of onboard peripherals, making this an exceptionally capable processor.

JTAG

The DSP56805 has a JTAG port to aid system debugging. A JTAG port consists of four dedicated signals (Table 19-1).

Signal name	**Function**
TDI	Test data input
TDO	Test data output
TMS	Test mode select
TCK	Test clock

Free scale Semiconductor adds additional signals to the standard JTAG set. Specifically, it adds RESET (Test Reset) to reset the JTAG state machine and DE(Debug Event), which is equivalent to an interrupt output, indicating that an event (such as a breakpoint) has happened in the OnCE (On-Chip Emulation) module.

JTAG is principally intended for debugging purposes, but since it gives you complete control of the processor's internals, it can also be used for reprogramming the internal program flash. The Free scale Semiconductor application note (AN1935/D) Programming On-Chip Flash Memories of DSP56F80x DSPs Using the JTAG/OnCE Interface, available from the Free scale Semiconductor web site, contains full details on the process involved, as well as sample source code and examples.

The Free scale Semiconductor Software Development Kit, based on the CodeWarrior C compiler, for the DSP56800 series provides both software and hardware tools for programming these processors.

Summary:

- Unlike the conventional DSP56000 with its 24-bit architecture, the DSP56800 series has a 16-bit architecture better suited to small-scale control applications.
- The architecture is based on four functional units, each with their own registers, operating independently and in parallel with the other units. These functional units are the program controller, which is responsible for software execution.
- The processor has two 36-bit accumulators, a 16 x 16-bit multiply and Accumulate (MAC) unit, and a 16-bit barrel shifter.
- The DSP56805 adds two 6-channel Pulse Width Modulation (PWM) units for motor control and other uses, two 4-channel ADCs at a resolution of 12 bits per channel, and two quadrature decoders for measuring motor positions.

- The DSP56800 has very tight and efficient code with high functionality that it executes exceptionally quickly.
- The DSP56805 has an internal 1K program RAM, 4K of bootstrap ROM (for loading boot software from an external memory or peripheral, 63K of program flash, 8K of data flash, and 4K of data RAM.
- The DSP56805 has an internal power-on circuit to correctly start up the processor. It also has a watchdog reset circuit, driven by an internal timer, to recover the processor from a software crash.
- The processor has an external 16-bit data bus that serves for accesses to both external program memory and external data memory.

Questions:

- Explain the architecture of DSP56800?
- What is programmer's model of DSP56800?
- What are the features of DSP56800?
- Explain the pin diagram of DSP56805?
- What is purpose of external 16bit data bus?
- How SRAM is interfaced with DSP56805?
- What is JTAG?

References:

- Analog Interfacing To Embedded Microprocessors – STUART R. BALL
- C Programming for Embedded Systems – KIRK ZURELL
- Design with 8051- FRONTLINE ELECTRONICS
- Embedded Controller Hardware Design - Ken Arnold
- Embedded Software The Works – colin walls
- Embedded Systems Firmware Demystified - Ed Sutter
- Embedded_Controller_Hardware_Design – KEN ARNOLD
- Programming Embedded Systems in C and C++ - Michael Barr
- The Art of Designing Embedded Systems - Jack G. Ganssle