Embedded Systems

Workshop on Distributed Laboratory Instrumentation Systems

Abdus Salam ICTP, Trieste, November 26 – December 21, 2001

Chu Suan Ang
Kuala Lumpur
Malaysia

Email: csang@pc.jaring.my

Abstract

An overview of embedded systems is first given. This is followed by analysis, design and implementation. Development techniques relevant to embedded system are introduced. Embedded systems networking in distributed environment is discussed. An introduction of web-based embedded systems then follows. Examples of web-based systems are dealt with.

1 Overview of Embedded Systems

1.1 Introduction

Embedded systems have been around since 1950s. Early examples were a closed-loop control system at a Texaco refinery in Texas and a similar system at a Monsanto Chemical Company ammonia plant in Louisiana. When chemical plants used a mainframe computer for process control in 1950s and 1960s, the mainframe was really an embedded processor, albeit a big and expensive one. When a physicist used a PDP11 minicomputer in 1970s to control and monitor his cryogenics experiments, he had built an embedded system. However, in those days, the number of such systems was not very large, basically because of the cost of hardware. How many PDP11 can a cryogenics laboratory possess? From 1980 onwards PC and microcontroller based embedded systems have been widely used. There is an enormous proliferation of such systems in workplaces. Now we see the post-PC period where embedded systems are generally cheaper, smaller and more powerful. A recent high growth area is the wireless communications area where embedded systems again play a vital role.

The importance of embedded systems can be easily deduced from its huge market. Statistics from the Semiconductor Industry Association (SIA) show for the year 2000 the following market values: (1) Microcontrollers: $19 billion, (2) Microprocessors: $30 billion, (3) Entire Semiconductor Industry: $200 billion. The distribution is as follows: (1) America: 31%, (2) Asia Pacific: 25%, (3) Japan: 23% and (4) Europe: 21%. For comparison, the world population is now at 6 billion and the GDP at $39,128 billion. Three countries whose GDP is near the world semiconductor market are the Netherlands ($355 billion), Malaysia ($233 billion) and Switzerland ($191 billion).

The reasons behind the rapid growth of embedded systems are due to their rapid and continuous reduction in price and the increase in power and complexity. The cost change for embedded controller is phenomenal in the last three decades - from $100,000 in 1970s, $10,000 in 1980s, $1,000 in 1990s. It is below a thousand dollars now. There has been an order of magnitude change in price for systems carrying the same function in a decade.

Advances in telecommunications are another reason for the growth. Data communications equipment, cellular phones are two huge markets. The explosive growth of the Internet and the progresses in software technology bring in a wide variety of hitherto unknown devices and products. Web-based embedded systems are likely to be deployed in a wide range of applications, including the laboratories and the homes.

1.2 What are Embedded Systems?

An embedded system is one with a built-in or embedded processor or computer, typically for carrying out some kind of real-time applications. The computer in such a system is not used as a general purposed computing machine. An embedded processor may or may not have a standard keyboard and video monitor, but it will always have some kind of connection to the outside world, be it a synchrotron, an air-conditioner or a handphone. While it is possible to cite many examples for which the time of response is not critical, there are far more applications of embedded systems which are time critical. Thus the study of real-time aspects of embedded systems becomes an important issue and this workshop dedicates a significant amount of time on it.

It is the application rather than the hardware itself that defines an embedded system. A PC used as a general purposed computer is generally not considered an embedded system. The same type of PC used in the laboratory to log data or control and thus forming an integrated equipment is an embedded processor. In such a case, peripheral interface will be used. However, it may involve only the standard I/O (input/output) ports such as the COM Port, Printer Port, SCSI or USB interface.

There are numerous examples of embedded systems around us. Embedded processors can be found in a large number of applications and situations:

Laboratory - test equipment, data acquisition systems, control systems, dedicated equipment. The use of embedded systems in laboratories has been going on for a long time. In 1960s and 1970s researchers in laboratories used minicomputers as embedded processors. Now standard PC and microcontrollers are typically used. From 2000 onwards, the ubiquitous Ethernet and Internet make the use of distributed embedded systems in laboratory environment a reality. Test and laboratory equipment manufacturers are among the first major users of microprocessors in embedded systems. For example, Tektroniks and HP (now Agilent) used microprocessors in their equipment as early as 1970s. The predecessor of this Workshop was a college on the use of microprocessors and PC under real-time environment for laboratories.

Process industry - process control systems. This is the granddaddy of real-time embedded systems. Early examples were the closed-loop control system at a Texaco refinery in Texas in 1959 and a similar system at a Monsanto Chemical Company ammonia plant in Louisiana. As the industry was able to pay, they were the ones that use mainframe computers as embedded processors. It is interesting to note that the use of computers in the process industry more or less charts out the history of computer engineering and computer science. Practically

all the hardware and software techniques have been used by this industry in one way or the other.

Manufacturing industry - production line assembly equipment, automatic test equipment, robots. Manufacturing industry benefits tremendously from embedded processors especially in the area of automation or robotics. Without the use of embedded systems, you would not be paying the current price of about $1000 for your PC which is really more powerful than a minicomputers in 1970s, let alone the ENIAC (Pennsylvania, 1945, 19,000 vacuum tubes, 200kW, 10 decimal digits, 0.2 ms addition, 2.8 ms multiplication.) or the EDSAC (Cambridge, 1949, 3,800 vacuum tubes, 500kHz mercury delay lines, 256 words, 35 bits, 1.5 ms addition, 6 ms multiplication.) Assembly plants in Malaysia, Mexico, Philippines, Thailand, China and other countries are producing more than 4 billions microcontroller ICs this year. This is only possible when large amount of embedded systems with clever software are used in the assembly and production lines.

Automotive - engine controls, anti-lock braking, lamp, indicator and other controls. It turns out that the automotive industry is one of the most important customers of the embedded processors. The average amount spent by a car manufacturer on a car in microelectronics is more than one thousand dollars. New models and especially up-market cars engage more. The Mercedes S-series and the BMW 7-series both have more than 60 embedded processors in each vehicle. This industry stipulates high requirements; electronics used must be highly reliable while able to withstand severe conditions of temperature, vibration and electromagnetic interference. Some processors were initially specially designed for the automotive industry and latter only modified for general purposed use. A robust network CAN (Controller Area Network) was also first introduced in this market and is now adopted by many other industries.

Consumer Electronics - audio-visual equipment, household electronics (microwave ovens, washing machines, dishwashers, air-conditioners, etc.), Personal electronics (electronic toys, gadgets, etc.) The list of products in this category is very large and is expanding continuously as the costs of embedded controllers drop. It is inconceivable now to operate a new television set without an IR remote controller. This is of course easily made possible when the price of 4-bit microcontrollers drops below a dollar each.

Offices – office and other commercial equipments. Nowadays, fax machines, photocopiers, calculators, computers, scanners, printers, time attendance systems, access control systems, surveillances systems, amongst others, are used in most offices and commercial environment. In banks, more specialised equip-

ments including autoteller machines, counting machines are used. All these are embedded systems of varying complexities.

Telecommunications - pagers, telephones, wireless phones, cellular phones, switches, base stations and associated equipment. This is yet another major area of embedded processor application. With the rapid growth in the telecommunications especially in the area of cellular phones, telecommunications manufacturers have been pushing the advancement of embedded processors in terms of size, cost and complexity. With the requirement of integrating analogue and digital circuitries, they are encouraging the chip designer and manufacturer to push towards the limits of this technology.

Internet – PC and accessories. Within a PC system, there are many embedded system submodules: keyboards, visual display units, I/O units, and modems. Peripherals are embedded systems: disk drives, printers, scanners, cameras, etc. As mentioned earlier, the growth in the Internet provides a significant impetus for the development of embedded systems.

1.3 Embedded System Structure

Despite the many different types of applications, the principles of operation, system components and design methodologies of embedded systems are essentially the same. A typical system consists of a **Computer** and an **Interface** to the physical environment, which may be a chemical plant, a car engine or a keyboard, for example. In some applications, standard I/O (Input/Output) devices such as the VDU, keyboard and printer are present, as in the case of process controller in a chemical plant. In others there are no standard I/O devices, as in the case of car fuel injection control. In the former case, it is likely that a general purposed computer such as a PC or a more powerful workstation PC will be adapted as the embedded processor. In the latter, microcontrollers designed together with dedicated electronics will be used.

The computers generally fall into three categories: (1) Small, (2) Medium and (3) Large. Small computers are used in small applications such as TV remote controls. In such applications, 4-bit microcontrollers are sufficient. Medium sized computers are 8- or 16-bit microcontrollers or PCs. They are used in jobs like data acquisition systems in laboratories or factories. Large computers are typically high-end computers. A plant monitoring and control system calls for such a computer.

The interfaces in embedded systems may be grouped into the following types: (1) Common serial and parallel interface, (2) Industrial interface, (3) Networking interface, and (4) ADC & DAC. By far the most common interface to the out-

side world is the first group consisting of standard serial buses including RS-232, RS-423, RS-422 and RS-485. The parallel interface (Centronics or IEEE 1284), commonly referred to as the printer port, is also an important interface. More specific to embedded systems are a number of bus arrangements introduced by various semiconductor manufacturers. The common ones are I2C, SPI, Microwire and 1-Wire bus.

A second group of interfaces are used in industrial or other more special applications. They are the IEEE 488 (GPIB, HPIB), SCSI (small computer system interface), CAMAC (Computer Automatic Measurement And Control) and CAN (Controller Area Network). A number of newer interfaces may be grouped here as well. They are the IrDA (Infrared Data Association), USB (Universal Serial Bus) and IEEE 1394 (FireWire or iLink).

Embedded systems in a distributed environment such as the laboratory often have to be networked together. While some of the above are used to network embedded systems, the more common ones for this purpose are the Ethernet, modems, ISDN (Integrated Services Digital Network), DSL (Digital Subscriber Line) , ADSL (Asymmetric Digital Subscriber Line), and ATM (Asynchronous Transfer Mode). The Ethernet is used for intra-building networking. The others are used for networking embedded systems over longer distances, over the Internet in most cases.

1.4 Real-time Embedded Systems

It was mentioned earlier that embedded systems are typical used to carry out real-time applications. What are real-time systems? One definition is as follows: “Any system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness.”

The above definition covers a wide range of systems - from UNIX workstations to aircraft engine control systems. When a command is entered in a UNIX workstation, we typically get a response on the screen 'with a sufficiently small time lag'. In an aircraft engine control system, the response to commands and other input parameters has to be within certain time limits. There is however a subtle difference between the UNIX workstation and the aircraft engine control system in terms of timeliness.

An alternative definition of a real-time system is as follows: “A real-time system receives inputs and sends outputs to the target system at times determined by the

target system operational considerations - not at times limited by the capabilities of the computer system." This further defines the meaning of response time and it distinguishes between the UNIX workstation and the engine controller. In a UNIX workstation, occasionally when we issue a command, we may not get the response in a time to our liking because the CPU is running some other higher priority tasks or simply overloaded. In this case, the UNIX workstation no longer qualifies as a real-time system according to the more stringent definition mentioned above.

Thus we have (1) Hard real-time system – one that must satisfy deadlines on each and every occasion, e.g. temperature controller of a critical process, and (2) Soft real-time system – one where occasional failure to meet deadlines acceptable, e.g. autoteller machines.

While real-time embedded systems have received a lot of attention in recent years, the earliest proposal of using a computer in real-time applications for controlling a plant actually dates back to 1950 when Brown and Campbell published his paper:

> Brown, G.S., Campbell, D.P., "Instrument engineering: its growth and promise in process-control problems', *Mechanical Engineering*, 72(2): 124 (1950).

A couple of early industrial installations of embedded systems are listed below:

> A plant monitoring system was installed in September 1958 by Louisiana Power and Light Company at a power station in Sterling, Louisiana, USA.
>
> An industrial computer control installation was implemented by Texaco Company for a refinery at Port Arthur in Texas, USA in March 1959.

The above systems, as well as many other early systems were *supervisory control* systems that used steady-state optimisation calculations to determine the set points for standard analogue controllers. In other words, the digital computer was used to compute and to send simple command to many standard analogue controllers that had been in use for a longer time in the industry. These analogue controllers were generally expensive, complicated and required periodic calibrations. Later, *direct digital control* that allowed the direct control of plant actuators was added and analogue controllers were not required.

The early real-time programs were written in *machine code*, which was manageable when the tasks were well defined and the system small. However, in combining supervisory control with direct digital control the complexity of programming increased significantly. The two tasks have very different time scales and interrupting of the supervisory control is necessary. This led to the devel-

opment of general purposed real-time operation systems and high-level languages for such systems.

2 Analysis, Design & Implementation

Bringing up an embedded system is no different from bringing up any other computer based system and it is important that one applies good design and engineering methodology. Many different approaches have been advocated and there are many books written on the subject but basically the objective is to apply a systematic approach so that the target system may be built to specifications functionally and it is easy to maintain.

As in most other cases, the tasks are *analysis*, *design* and *implementation*.

2.1 Analysis

During the first phase, the functions, requirements and possible constraints of the target system are analysed. The *problem* must be well defined. Otherwise there is no *solution.* Difficulties in the later stage of a project often arise when the scope of the work is not rigidly known or when the designer is uncertain of the capabilities and constraints of the various hardware and software resources. Except for very small jobs, this analysis phase must be carried out with care.

In specifying the requirements of a system, the following list of questions form a basis to *freeze the specifications:*

What is the type of user interface required?

What are the data processing and storage requirements?

Communication requirements – does it connect to the Ethernet or the Internet?

What are the I/O requirements - serial, parallel, ADC/DAC?

What are the real-time requirements or constraints?

2.2 Choice of Hardware

As mentioned earlier, an embedded system typically consists of two modules – the processor module and the interface module. Before the actual design phase is entered, there is a decision to make. That is whether the hardware should be purchased or built? This normally hinges on economy consideration amongst others. Delivery time may be another consideration, for example.

Five or ten years ago, ready-made hardware modules for embedded system use were expensive and for many laboratory applications they are out of reach.

However, the past couple of years see the appearance of a range of affordable hardware modules specially targeted at the embedded system market. Many semiconductor manufacturers produce processor and interface modules besides IC components. A number of new manufacturers appear in the market offering very cost effective hardware that can no longer be ignored even for small applications in small laboratories.

One such manufacturer is Rabbit Semiconductor, Davis, California. In their recent catalogue, there is a module RabbitCore RCM2200 selling at $49 which has the following features:

- 8-bit powerful CPU based on the Z80
- 10Base-T Ethernet connection
- 128K SRAM
- 256K Flash
- 26 general purpose I/O lines

If the decision is to build your own hardware, it is likely that for laboratory applications, 8-bit microcontrollers (MCU) are the choice. Of course, for very large systems or for those requiring special data handling or processing, larger processors including the 32-bit CPU and DSP may be used. However, it is noted that the PCs or high-end PCs would generally be a more cost effective solution in this case, if they can handle the task in hand.

Indeed, for very small jobs, there are available off the shelf a range of small 4-bit MCUs. The costs of the bare MCUs are in the following range:

4-bit at $1

8-bit between $5 and $10

16- 32-bit between $50 and $100

Information on the various products in the market can be obtained from the manufacturers' websites. The major suppliers are: Motorola, Mitsubishi, NEC, Hitachi, Philips, Intel, SGS-Thomson, Microchip, Matshushita, Toshiba, NS, Zilog, TI, Siemens, Sharp.

Since the mid 1970s when microprocessors were first introduced, there have been a large number of processors produced. Many of them fall into families of devices which have the same core CPU and general characteristics. A few famous such families are:

Motorola: 6805, 6809, 6811, 683xx

Intel: 8051, 80186, 80386

Zilog: Z8, Z80180

Microchip: PIC family

If the decision is to buy ready made hardware, it is still necessary to choose the right host of products offered by many suppliers. If a PC can be used, it is probably still the simplest to implement. However, other than the desk top or notebook PCs, there are the SBCs (Single Board Computers) that are effectively an entire PC shrunk into a single board. These SBCs usually have all the standard I/O ports clustered around various connectors. RAM disks are often available. These are IC memories devices plugged into sockets on the SBC to replace floppy disks. Advantech is a major supplier for SBC but there are many others in the market offering similar products.

While SBCs are neat substitutes for PC, they are generally more expensive than the PC with the same capabilities. This is due to the smaller volume of production. Thus SBC are typically chosen when size constraint dictates its use. Or the

environment necessitates the use of industrial grade SBC. (Industrial grade PCs are more expensive.)

Many semiconductor manufacturers formerly producing only component ICs are now marketing board level modules which incorporate processors and I/O devices. These are sometimes referred to as core modules and are typically a fair bit less complex than the SBC. However, they are suitable candidates for a wide variety of embedded system applications. The attractions of these modules are cost and size. The RabbitCore series of Rabbit Semiconductor, from which one example was mentioned earlier, is an example. Dallas Semiconductor's TBM 390 (TINI Board Module) is yet another.

These modules typically have a modern MCU with serial and parallel I/O sufficient for most small to medium sized embedded system applications. Memories are in the range of about 1 Mbytes, half of which are RAM. Flash memories are getting common too. For larger storage, serial memories can usually be added. In the last year or so, Ethernet connection is incorporated making the module ready web enabled. This particular feature may prove to be an extremely important enhancement in core modules.

Software support for these core modules are generally good. Most manufacturers will provide development package consisting of both the hardware and the software environment. An IDE (integrated development environment) is usually provided. C compilers on top of assemblers are commonly available from either the manufacturer or a third party. The TBM is rather unique in that it is designed to run Java in the Internet environment.

2.3 Hardware Design

After the decision of whether to build or purchase the hardware is made, one can embark on the design proper. If the choice is on a standard PC or ready built hardware as the embedded processor, then the hardware design step is simplified to that of designing the interface board or circuitry to the target system. Although there can be an infinite variety of target systems, the interface requirements however can be grouped into just a few standard categories - digital I/O, analogue I/O, serial data communications and parallel data communications. Many of the interface requirements are normally provided for by the embedded processor hardware. Perhaps signal conditioning circuits (instrumentation amplifiers, precision attenuators, current drivers, etc.) are needed in the case of analogue I/O or special actuators or sensors.

What are steps taken if you have to design your own processor boards? Ten or fifteen years ago, one would build a microprocessor based system using a hand-

ful of chips including microprocessor, memory, peripheral devices and other glue chips to build an embedded processor. And to do that effectively, certain basic skills have to be acquired. In fact, the earlier Microprocessor College at ICTP spent four weeks trying to achieve just that.

A good example of building a processor board in this way is the 6809 system used in the workshop and earlier Colleges. There are many good reasons for doing so. First of all, it generally has more memory resources than a single chip microcontroller. This facilitates the use of more sophisticated resident firmware including a full-featured monitor or a real-time kernel, for example. Often, there is readily available software for a popular microprocessor such as the 6809. The designer may already be familiar with a well-known microprocessor and need not learn to use a new one.

The trend however, is to use single chip microcontrollers whenever possible. The beauty of designing embedded systems using microcontrollers is the relative ease and simplicity. In general, the overall cost of the system is also lower because of the lower component count.

Whether we use microprocessors or microcontrollers, there is a set of good design rules or practice that one should adhere to. Amongst them, one that has often been overlooked is that the design must incorporate facilities for debugging and testing. Small tests or diagnostics, switches or indicators, added during the designing stage cost very little, but help tremendously in the later stage.

2.4 Outline of Hardware Test Procedure

It would be nice if sophisticated tools such as development system, in-circuit emulator and logic analyzer are available. However, it is possible to test and debug with the basic electronics laboratory equipment such as multimeter, oscilloscope and function generator alone, if a systematic approach is adopted.

- Printed circuit board (PCB) inspection for track continuity and possible bridging. This is a step that is often overlooked. However, it is a vital step because easily locatable faults if left undetected, usually cause much more debugging efforts at a later stage.
- Power up the bare PCB and check voltages.
- If it is a microprocessor-based system, such as the 6809, or a microcontroller-based system operating in *expanded multiplexed* mode, test the address bus and (partially) the data and control bus on the *hardware kernel* which is the processor itself. This step is skipped if the system is single-chip, micocontroller-based.

In the case of 6809, this is done by forcing a `NOP ($12)` on the data bus by pulling up D1 and D4 to 5V via resistors and grounding all other data lines. It causes the continuous execution of `NOP` for all memory locations. This in turn results in A0 toggling at half the system clock rate, A1 toggling at half the rate of A0 and so forth. The address bus can thus be checked easily with an oscilloscope. In this test, data bus and control bus are partially verified.

The above test procedure is actually making use of the 1-byte instruction of the microprocessor in an unintended manner. For Z80, 8085 and 8088 similar techniques can be used. In Z80 and 8085, `RST 7 ($FF)` instruction is used whereas in 8088 either the 1-byte INT 3 or PUSH instructions may be similarly used.

- If a logic analyzer is not available, implement a tight loop program in the EPROM or EEPROM such as a branch-to-itself loop (`LOOP BRA LOOP`). For 6809, this consists of two bytes (`$20 $FE`) and takes three machine cycles to execute. A two-byte reset vector is also needed in the ROM. The execution of this very short program can be followed cycle by cycle on an oscilloscope and thereby confirming the proper operation, at least partially, of the data and control bus.

- It is a good idea to include DIP switches and LED indicators in the hardware even if they are not required in the final target system. Test routines for I/O ports which have these input switches and output indicators can be written and tested. Commonly used routines include incrementing the binary value of the output port at a slow rate for visual inspection, reading status of switches and sending it to the output port. This stage of testing serves to verify the operation of I/O ports and to provide users with function selection. Normally on power up the system is programmed to check the status of the input switches and jump to appropriate test routines or the main program.

- Small test routines for other components in the system are then implemented. This includes testing the serial link, the timers, ADC and the memories.

- In some embedded systems where the memory is not very small, a monitor program or kernel is then implemented.

- At this stage most of the hardware testing are done and the task moves on to application software testing and debugging. However, there is one type of hardware bug which is not detected by the testing mentioned above. These are problems caused by intermittent faults, glitches or external interference.

These are detected by means of logic analyzer or in-circuit emulator running in surveillance mode.

2.5 Some Hardware Development Tools

While one can get by with the basic tools for small embedded system development, nevertheless it will help if a number of other hardware development tools are available, especially when one is dealing with more sizeable projects or when problems such as intermittent faults, external electromagnetic interference, and glitches arise as mentioned above. It is impossible to give a thorough treatment of various hardware tools in detail here. However, a number of more important ones are introduced below.

Oscilloscope - The oscilloscope really needs no introduction other than listed here for completeness sake. It is noted that while the conventional dual-trace 20MHz cathode ray oscilloscope (CRO) is still the faithful workhorse in the lab, there exists in the market now digital oscilloscopes with liquid crystal display (LCD) at a reasonable price. Often it combines the function of a digital (memory) oscilloscope with a logic analyzer. The importance of the oscilloscope cannot be over-emphasized - after all the HP and Tektronix logic analyzer designers used their oscilloscopes to debug their embedded systems in the '70s!

Logic Analyzer - The two traces of an oscilloscope is really rather inadequate or impossible when it comes to *simultaneously* monitoring the 40 or so lines of a typical microprocessor or microcontroller circuit. Logic analyzers capture 48 or more signals and display them in multiple traces or in coded form. Being a powerful embedded system itself, the logic analyzer can perform a number of other things that expedite the debugging of embedded systems.

It allows a trigger condition (data, address and control bus pattern) to be set up and captures the cycle by cycle information in memory (typically few thousand cycles deep) when the trigger condition is met. The captured data can be viewed as traces, in binary/hex form or in mnemonics of the target processor after being disassembled. This provides a very powerful tool for monitoring what's going on at a very low level non-intrusively - at least while the embedded system is running at its normal speed.

Most logic analyzers also provide *timing analysis* whereby the traces are sampled at rates higher than the system clock and hence glitches or other irregular waveforms may be detected.

Emulator - First introduced by Intel, now in-circuit emulators are used in large number of embedded system development. This tool brings the debugging of

hardware one step higher than using the logic analyzer alone. Basically it not only allows the target system to be monitored, but also has the ability to stop execution in a controlled manner, change memory and register contents and resume execution. This is achieved by replacing the target system CPU with a more elaborate system typically containing the same type of CPU but having other resources which can carry out the actions mentioned above. In theory the system *emulates* all the CPU's functions in real time.

The major features of the in-circuit emulators are breakpoint, real-time trace, RAM overlay, and performance analysis. Breakpoint setting, as mentioned above, allows us to stop execution, for example, at the end of a function and monitor the return value. When the code does not behave as expected, real-time trace can be used to *look* at what the code is doing. Embedded systems often have their code stored in ROM or EPROM. To change the code during debugging is tedious. RAM overlay is a technique to circumvent this difficulty. Instead of running the code in the target system ROM or EPROM, RAM in the emulator which can be easily modified is used. Performance analysis deals with the problem of code not able to deliver the performance required, such as keeping up with external events. The analysis allows the programmer to scrutinize the execution of his code carefully and find remedies if possible.

In the case of microprocessor-based systems, the target microprocessor is replaced by an emulating processor which has overall control over the data, address and control bus and thus the operation of the entire system. In the case of microcontroller-based systems, it is more complicated. Typically, the emulator operates the microcontroller in the expanded mode so as to gain access to the internal bus. It must also have:

- extra RAM to hold the application software during development,
- a monitor program, and
- rebuilt ports to replace those lost in the expanded mode.

Other features available in an emulator are:

- communication facility between the monitor program and a host computer,
- ability to download object code from the host computer to the target system,
- ability to display and change RAM contents and processor status of the target system,
- single stepping and breakpoint features, and

- execution of the application program in full speed.

The emulator is almost an indispensable tool in the development of embedded systems but the downside is that it is generally not cheap. Good emulator can run to tens of thousands of dollars. Fortunately there are a number of low-cost emulators typically produced by chip manufacturers themselves to promote the sales of their microcontrollers. These are often sold under the name of evaluation board of system. They lack the sophistication of full featured emulators but nevertheless are very useful for small projects.

ROM Emulator - ROM emulators are like RAM overlays mentioned above, used to temporarily replace the target system firmware. A ROM emulator consists of RAM and associated circuit, a connection to the ROM socket in the target system and a link to a host computer. The host computer downloads the data into RAM which is then used by the target system as its ROM memory. This relatively simple tool is very effective in embedded system development because it reduces the iteration time significantly.

2.6 Software Design and Development

Since the late 1960s, the notion of software development as an engineering process has been universally accepted. For embedded systems, software development is basically a software engineering work. In essence, the aim is to produce well-engineered software, not just software that provides the required functionality. Software should be:

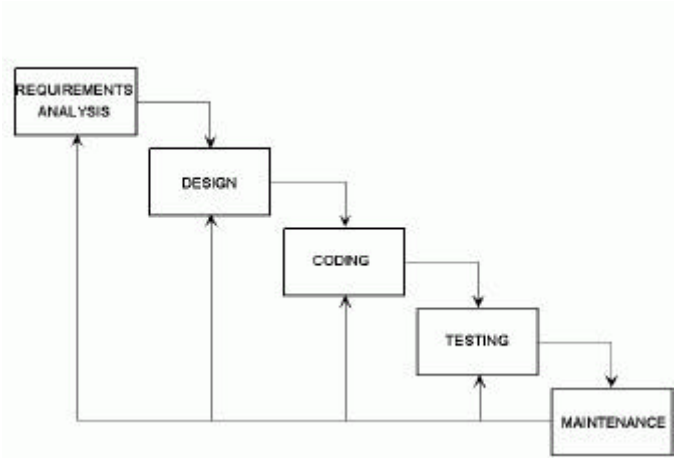
- Simple, clear and easily maintainable
- Reliable
- Efficient
- User friendly, i.e. having an appropriate user interface

Of the several models developed for software processes, we do not have to stick to a particular one religiously. For practical reasons, it is likely that we shall pick whichever model that appears to be more appropriate for the job in hand:

- The waterfall model or classic life-cycle approach
- The prototyping approach
- The spiral model

The waterfall model is the earliest and most widely used. In a way it is the most natural approach, being derived from other engineering disciplines and is considered a good model which offers a clear picture of the development process. Prototyping and the spiral models are also practised in other engineering disciplines and their use often encompasses the waterfall model within themselves.

2.7 The Waterfall Model of Software Development



The Waterfall Model depicted above is easy to understand and it emphasises the importance of the various phases of software development. In other words, software development is not just coding - thus you don't rush to the keyboard and start *programming* the moment you are given a job. A civil engineer does not build a bridge by sending some steel beams and concrete to the site, he designs it first. An electronic engineer does not go to the component rack and pick up a few resistors, capacitors and transistors and start soldering his amplifier, he designs it first. Similarly a software programmer should not write the software without first designing it.

Statistics on business software projects show the following cost breakdowns for the various phases:

Phase	Cost
Requirements/Design	44%
Coding	28%
Testing	28%

You do not consider mission completed when you have just finished writing your last line of code or pasting your last control on a form. You test your code, thoroughly! You are responsible for your own code just as a circuit designer is responsible for the functionality of his circuits. A circuit designer does not send

out an untested circuit and let the user come back with failure reports. The first principle in testing is of course *test as you write your code* as it is easier to find problems in small modules and the earlier the problem is found, the easier it is to fix. For a thorough account of various testing techniques, read chapter 6 of "The Practice of Programming" by Kernighan and Pike.

It is common for project groups to work long hours towards the deadline, sometimes round the clock. However, it is obvious that debugging convoluted code at 3 o'clock in the morning is unlikely to produce satisfactory results and thus no credit can be given even though you work seemingly long hours. The real solution is to avoid having to get yourself into such a situation - having to test and debug your code in the small hours because otherwise you are holding up the progress of the whole project. The difficulty or stress in many such situations can be avoided by having a well designed program and by having effective and thorough testing of code in the first place. That is, *design* and *test* your program for greater efficiency.

Design also allows you to eliminate ambiguities, in specifications and interface. Discuss your design with your fellow workers or supervisor. Assumptions you make in your design will be known in the early stage by them and can be modified with less effort if necessary. Difficulties you may have can also be addressed and solved much earlier. Modules that are new and difficult to you can be identified and tested first. Likewise critical modules are easily identified and can thus be dealt with accordingly.

Possible improvements of old designs are to be encouraged. However some improvements may have adverse effects on other part of the system or result in inconsistency. Without a proper design document, it will be too late when others realise your unacceptable improvements. And you will be reluctant to modify your code or design at that stage.

There are earlier work or code that may be reused or modified for reuse. You should find out as much as possible if you are new to the group. Often, your supervisor can suggest or tell you the existence of such prior art or point you to someone who has it, provided that you show him or her the design so that the necessary components can be identified.

2.8 Documentation

Documentation can never be overemphasized. Without good documentation, the requirements of the project itself are not known after a short while, even by you yourself. Needless to say, the effort of maintaining the software subsequently will be costly if at all possible.

Documentation should not be postponed till the end of the project, except for collation of documents and writing of manuals. Many of the documentation tasks should be carried out as the project progresses. In fact each and every phase of the waterfall model calls for documentation to be done, there and then. You should never wait till you finish the whole project to document because you almost never ever will, and even if you do, you will have difficulties in reconstructing past events.

2.9 Programming Languages

What programming language to use for embedded systems development? Most people agree that one should use a high level language (HLL) to develop embedded systems. Amongst the HLLs, C is known to be a good choice for embedded systems because of its versatility and widespread use. Java has become a strong candidate for its inherent object oriented programming features.

Besides knowing C, an embedded system programmer usually has to learn the assembly language as well. For very small projects, assembly language is still a good choice for speed and code efficiency. Even when one writes in C, a small amount of code such as the interrupt routines and sometimes the device drivers are still implemented in assembly language. Source code debugging is nice, but occasionally, one may have to debug at a lower level, especially when hardware debugger such as logic analyzer is used. In which case, a good knowledge of the assembly language is needed.

One important point in designing software for embedded system is to design with debugging in mind. More often than not, your code won't work the first time. Unlike hardware development, the time taken in testing and debugging during software development can be surprisingly long if you are not careful. Well organised code is a must if you want to minimize debugging time. Well commented code mentioned above is another cardinal virtue in programming.

Ideally a software development environment for embedded systems work should have the following three components:

Host computer - This is typically a PC which runs the editor, linker and compiler. PC has become the de facto standard as development platform for embedded systems because of its availability and the amount of commercial and public domain software tools obtainable. Traditional embedded system vendors have designed their development tools with the PC in mind. This also encourages a large number of third party software vendors to use the PC platform for their software tools.

Debugging engine - This refers to the component that allows you to *look* into your target system in terms of code execution. It may be in the form of an in-circuit emulator or in smaller projects a monitor program resident in the target system itself. This debugging engine allows you to open a window in the host computer and monitor the execution of your code or status of your processor in the target system. For any serious work, it is no longer acceptable to compile your code, program the EPROM, plug it in and hope that it will work!

Source-level debugger (SLD) - This is a piece of software running in the host PC which allows you to debug your code at source level, in conjunction with the debugging engine. Not only does it communicate with the debugging engine or target system, it also provides intelligent assistance in the debugging stage. For example it displays the source code (actual C statement instead of assembly code) at which the target is at, resolves symbolic references, examines in the high level format, allows breakpoint to be set at source level, single step through the code again at source code level, etc. Generally a good SLD will provide all these features in very neat multiple window environment, thus making debugging a much easier task than if it is done at assembly code or machine code level.

2.10 Cross Development

As mentioned above, mainly because of the ubiquitous position, the PC is almost universally used as the platform for embedded system development. In which one would be doing cross development running a host of cross software - cross assemblers and linkers, cross interpreters, cross compilers. Unless of course one is developing an embedded system with the same CPU as the PC used (e.g. 80186, 80188, 80386EX or the PC itself used an embedded processor.).

Cross development is necessary for a number of other reasons:

Many microcontrollers used in embedded systems are just too small to be used as processors in development systems. Native or resident assemblers and compilers may not be available for such systems.

Existing computer facilities are readily available and with the appropriate cross-development software tools, are suitable for carrying out the task of software development. This is considered an important advantage because no extra hardware is needed and software tools such as editors are already available.

Nowadays, one can find cross-development software tool for almost any processor in the market. Some manufacturers are supporting their products with a dial-up facility or through Internet which allows users to download cross-assemblers and cross-compilers to the PC.

Thus, cross assemblers are programs that run on a computer with a different processor from that of the target system, and *assemble* programs written for the target system into *re-locatable object code*. The linkers then relocate, usually with other object modules such as library modules, to the desired execution addresses for the target machine. Common features of cross assemblers are: (1) provision for using macros in program, thus *macro-assembler*, (2) conditional assembly, (3) assembly time calculations and (4) listing control.

Similarly, cross compilers are programs that run on a computer with a different processor from that of the target system, and *compile* high level language programs written for the target system typically into assembly language programs. The use of cross compiler can reduce program development time significantly for large project. It also makes programs more portable, since they are written in high level languages such as C. A typical cross compiler consists of: (1) macro pre-processor, (2) parser, (3) optimiser and (4) code generator.

2.11 Simulation

Simulation is a way of using software to model the target system including the target processor itself. A programmer can *see* his system running in the stable environment of his host computer which runs the simulation program. This is used when the target system is not available, when the target prototype is still unreliable, or when the programmer has to access the low level status of the system not normally accessible in embedded systems.

While it sounds like a great idea, unfortunately good simulators for embedded systems are not readily available. This is due to the fact that the simulator has to deal with real-time events and sometimes rather complex I/O. How can you get a general purposed simulator to understand your obtuse or ingenious interface to the solar tracking system? How do you simulate real-time, asynchronous events? To duplicate the data stream coming from the outside world is not easy either.

Nevertheless, there are simulators available for many processors. One successful category of simulators seems to be the microcontrollers such as the 8051. When many of the I/O are integrated on a single chip, they are well defined and thus can be simulated more readily.

2.12 Other Techniques for Embedded Systems

There are several other techniques that are found to be useful in developing or debugging software for embedded systems. A monitor program will help in the debugging process tremendously. In structuring your program, the following are

found to be very useful: (1) state machine technique and (2) task scheduler and (3) real-time kernel.

A monitor program in the case of MCU embedded systems is usually a small program of 1 to 2 Kbytes of memory for monitoring (and modifying) hardware related information. These may be memory contents, I/O status or even processor status. At low level debugging, a monitor program is an indispensable tool that allows a programmer to *look* into the hardware of the system.

3 Development Techniques

3.1 State Machine Method

For small systems, *sequential organization* of the program is often used. The entire function of an embedded system is represented by a flowchart and implemented accordingly using a single main loop. When external inputs or events arrive, the program branches off to some modules to carry out the required actions.

There are however a number of shortcomings using the above method:

- Testing of a monolithic program is often difficult.
- When the loop becomes large as more functions are added, life becomes complicated. When a single large loop is used, there is a tendency to produce *spaghetti* code.
- Subsequent modifications of system function, like adding another control switch, are tedious because the entire flowchart has to be revised and often re-implemented entirely.

For many embedded systems, the complexities often justify a more systematic approach of designing the software. Representing the function of a system by a **state machine** is such an approach. The power of state machine representation comes from the fact that it can subsequently be represented by a **state table** which is well suited for microcontroller and microprocessor implementation, even at assembly language level.

Using the state table method of implementing the functions of a system, it is natural that the job be broken down into small, more manageable and often independent modules, called the *action routines*. Such routines are more easily tested and often reusable.

However, the single most important advantage of state table implementation really lies in the ease of function modification. In most cases, only the state table is modified together with the necessary new routines, while most of the old code would be intact.

3.1 Example of State Machine Representation

A simple example of a system with key switches and display is given here to illustrate the method of state machine representation.

- Suppose we have a keypad with ten numeric keys 0 to 9 and two function keys ENTER and DELETE and a 4-digit numeric LED display.
- On power up, the display shall show 0.
- Numeric values can be entered on the keypad and as each digit is entered, it is scrolled into the display from the rightmost digit. During this mode, the display blinks to indicate *digit entering mode*.
- The *digit entering mode* is terminated with either the ENTER key or the DELETE key.
- If ENTER is pressed, the display stops blinking.
- If DELETE is pressed, the display stops blinking and shows 0.

There are 3 possible states in this example:

State	Name	Description
S0	Initial	Power-on state or after DELETE, display shows 0 in steady mode.
S1	Data Entry	Digit entry mode, display shows digits in blinking mode.
S2	Display	Final display mode, display shows final value in steady mode

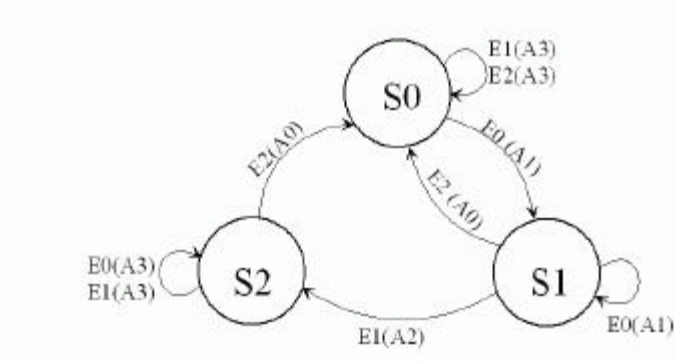
There are 3 types of event:

Event	Name	Description
E0	Number	Entry of any numeric key.
E1	Enter	ENTER key is pressed.
E2	Delete	DELETE key is pressed.

There are four action routines needed:

Action	Name	Description
A0	Reset	Display 0.
A1	Build digits	Build up display buffer from right while numbers are entered and blink display.
A2	Steady display	Show steady display.
A3	Null	No action.

The specification mentioned earlier is represented by a state diagram.



The above state diagram can be easily transformed into a state table representation as follows:

Present State	Event	Action	Next State
S0	E0	A1	S1
	E1	A3	S0
	E2	A3	S0
S1	E0	A1	S1
	E1	A2	S2
	E2	A0	S0
S2	E0	A3	S2
	E1	A3	S2
	E2	A0	S0

The complexity of the system has thus been broken down into:

- A number of action routines.
- A service routine to scan the keypad and update display.
- A state stable.
- A very small main program.

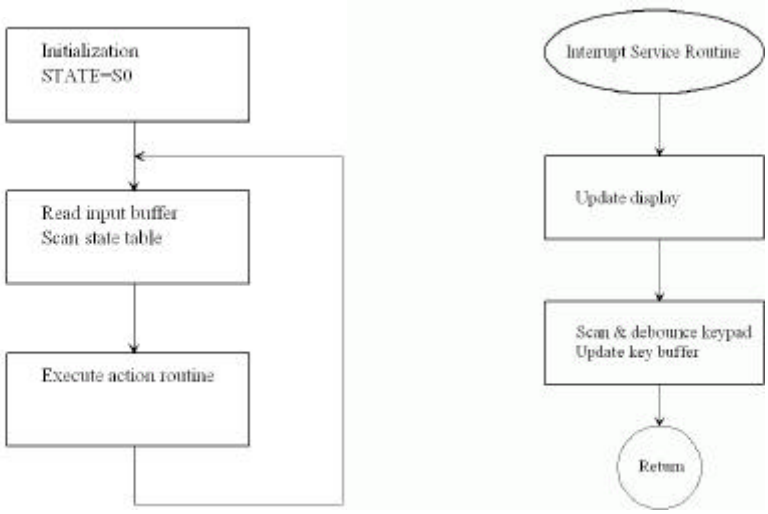
The main program has reduced to a trivial program consisting of initialization and an infinite loop of reading the input buffer to get a new event, scanning the state table to decide what is the next state and the action to be taken. The action is carried out simply by calling the action routine.

The advantage of this seemingly tedious process is as follows. First the state diagram technique allows a complete and systematic approach to the problem. All events can be analyzed at all possible states. This technique is often used to implement communications protocols which can be rather complex because of the large number of possible states.

The next important advantage of the state machine method is that subsequent modification of the functions of the system is relatively painless. Often, a rearrangement of the state diagram and state table is all that is necessary. In other case, the modification may involve the introduction of several new action routines. In either case, the modification is straightforward.

The keypad and display service routine may be implemented as an **interrupt service routine** based on 10-ms clock ticks from a programmable timer module, for example.

The main program: The interrupt service routine:



3.2 Task Scheduler in Embedded System

An application in real-time embedded system can always be broken down into a number of distinctly different tasks. For example,

- Keyboard scanning
- Display control
- Input data collection and processing
- Responding to and processing external events
- Communicating with host or others

Each of the tasks can be represented by a state machine. However, implementing a single sequential loop for the entire application can prove to be a formidable task. This is because of the various time constraints in the tasks – keyboard has to be scanned, display controlled, input channel monitored, etc.

One method of solving the above problem is to use a simple **task scheduler**. The various tasks are handled by the scheduler in an orderly manner. This produces the effect of simple multitasking with a single processor. A bonus of using a scheduler is the ease of implementing the *sleep* mode in microcontrollers which will reduce the power consumption dramatically (from mA to μA). This is important in battery operated embedded systems.

There are several ways of implementing the scheduler – preemptive or cooperative, round robin or with priority. In a cooperative or non-preemptive system, tasks cooperate with one another and relinquish control of the CPU themselves. In a preemptive system, a task may be preempted or suspended by different task, either because the latter has a higher priority or the time slice of the former one is used up. Round robin scheduler switches in one task after another in a round robin manner whereas a system with priority will switch in the highest priority task.

For many small microcontroller based embedded systems, a cooperative (or non-preemptive), round robin scheduler is adequate. This is the simplest to implement and it does not take up much memory. Ravindra Karnad has implemented such a scheduler for 8051 and other microcontrollers. In his implementation, all tasks must behave cooperatively. A task waiting for an input event thus cannot have infinite waiting loop such as the following:

```
While (TRUE)
{
 Check input
 …
}
```

This will hog processor time and reprieve others of running. Instead, it may be written as:

```
If (input TRUE)
{
 …
}
Else (timer[i]=100ms)
```

In this case, *task i* will check the input condition every 100 ms, set in the associated timer[i]. When the condition of input is false, other tasks will have a chance to run.

The job of the scheduler is thus rather simple. When there is clock interrupt, all task timers are decremented. The task whose timer reaches 0 will be run. The greatest *virtue* of the simple task scheduler ready lies in the smallness of the code, which is of course very important in the case of microcontrollers. The code size ranges from 200 to 400 byes.

3.3 Real-time Kernel in Embedded Systems

It would be ideal if we can incorporate a real-time operation system (RTOS) in the embedded system we build. Unfortunately, more often than not, the memory and other resources of most embedded systems do not permit this. There is however an alternative - that of using a subset of the RTOS to solve the problem of real-time requirements. If the I/O and file handling is removed from the fully fledged RTOS, we are left with a kernel which deals with tasks handling. This turns out to be a powerful tool in dealing with real life embedded system applications, such as the state machine technique.

In embedded systems, interrupts are used to respond to external events and in doing so avoid the waste of CPU time by constant polling for such events. However, interrupts handling can be rather complex if there are many processes to be handled simultaneously. In many situations, embedded systems run more or less independent programs that share some common resources. A very large intertwined program will result if we use simple interrupt handling technique. Real-time kernel (RTK) will help the programmer to deal with such circumstances by thinking in terms of concurrent tasks instead of individual routines that execute when certain events occur.

Real-time kernels come in a great variety of types. Many of the small RTKs are implemented in assembly language; others are implemented in HLL such as C. There are many RTK manufacturers producing kernels for 8-, 16- and 32-bit processors including proprietary and open market ones. The price tag of these commercial RTKs ranges from USD100 to USD10,000.

There are also a small number of real-time kernels appearing in journals, magazines and books, which are normally available in source code. One such example is an RTK designed by Jean J. Labrosse called µC/OS, which is implemented in C with full source code available to the user.

Jean J. Labrosse published an early version of µC/OS in *Embedded Systems Programming* magazine in June 1992. It was written in C with the initial goal for creating a small but powerful kernel for the 68HC11 microcontroller. It has since been extended to a portable system suitable for use with any microcontroller/microprocessor provided that it has a stack pointer and the processor status can be stacked and unstacked.

Labrosse has subsequently written a book describing µC/OS. The latest edition is:

Jean J. Labrosse, *MicroC/OS-II The Real-Time Kernel*, R & D Books, Lawrence, Kansas, 1999, ISBN 0-87930-543-6

The complete source listing of µC/OS is available in the book. It is also available in a companion disk.

The code is protected by copyright. However, you do not need a license to use the code in your application if it is distributed in object format. You should indicate in you document that you are using µC/OS.

3.4 Main Features of µC/OS

The main features of µC/OS are:

Portable - It is written in C, with a small processor specific code in assembly to create task, start multitasking and perform context switching. For 80186/80188 the assemble language code is less than 4 pages.

ROMable - The size and design of the kernel is such that it is suitable for storing in ROM or EPROM.

Priority driven - It always runs the highest priority task that is ready.

Pre-emptive - When a task makes a higher priority task ready to run, the current task is pre-empted or suspended and the higher priority task is immediately given control of the processor. Execution of the highest priority task is deterministic.

Multitasking - Up to 63 tasks may be set up.

Interrupt feature - Interrupts can suspend the execution of a task. If a higher priority task is awakened as a result of the interrupt, the higher priority task will run as soon as the interrupt completes. Interrupts can be nested up to 255 levels deep.

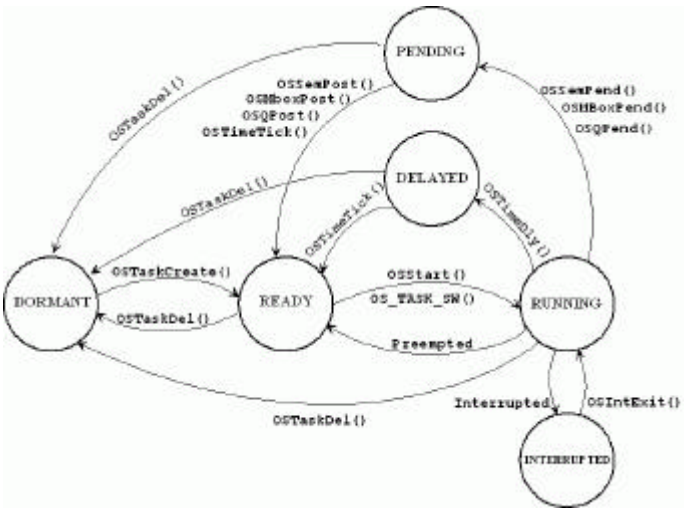
3.5 µC/OS Tasks

A *task* is an infinite loop function or one that deletes itself when it is finished. The infinite loop can be pre-empted by an interrupt that can cause a higher priority task to run as mentioned above. A task can also call the following µC/OS services: **`OSTaskDel(), OSTimeDly(), OSSemPend(), OSMboxPend(), OSQPend().`** Each task has a unique priority, ranging form 0 to 62. The lower the value the higher the task priority.

There are altogether six possible states for a task as listed below:

- DORMANT -The state when a task has not been made available to µC/OS.
- READY - When a task is created by calling **`OSTaskCreate()`**, it is in the READY state. Tasks may be created before multitasking starts or dynamically by a running task. If the created task has a higher priority than its creator, the created task is immediately given the control of the processor. A task can return itself or another task to the DORMANT state by calling **`OSTaskDel()`**.
- RUNNING - The highest priority task created is in the RUNNING state when multitasking is started by calling **`OSStart()`**.
- DELAYED -The running task may call OSTimeDly() and enters the DELAYED state. The next highest priority task then runs. The delayed task is made ready to run by **`OSTimeTick()`** when the desired delayed time expires.
- PENDING - The running may have to wait for an event by calling **`OSSemPend()`**, **`OSMboxPend()`** or **`OSQPend()`**. It then enters the PENDING state. The next highest priority task then runs. The task is made ready when the event occurs. The occurrence of an event may be signalled by another task or by an interrupt service routine (ISR).
- INTERRUPTED - A task may be interrupted and enters the INTERRUPTED state. The ISR then runs. The ISR may make one or more tasks ready to run. When all tasks are either waiting for events or delayed, an idle task **`OSTaskIdle()`** is executed.

µC/OS Task State Transition Diagram:



3.6 Task Scheduling

Task scheduling is done by **`OSSched()`** which determines which task has the highest priority and thus will be the next to run. Each task has a unique priority number between 0 and 63. Priority 63, the lowest, is assigned to the idle task when µC/OS is initialised.

Each task that is ready to run is placed in a ready list. The task scheduling time is constant irrespective of the number of tasks created. **`OSSched()`** looks for the highest priority task and verifies that it is not the current task to prevent unnecessary context switch. A context switch is then carried out by **`OS_TASK_SW()`.**

`OSSched()` runs in a critical section to prevent ISR from changing the ready status of a task.

3.7 Interrupt Processing

µC/OS requires an *interrupt service routine* (ISR) written in assembly language. Interrupts are enabled early in the ISR to allow other higher priority interrupts to enter. **`OSIntEnter()`**is called on entering and **`OSIntExit()`**on leaving the ISR to keep track of the interrupt nesting level. There may be 255 levels.

µC/OS's worst case interrupt latency is 550 MPU clock cycles (80186/80188). µC/OS's worst case interrupt response time is 685 MPU clock cycles (80186/80188).

3.8 Clock Tick

Time measurement in suspending execution and in waiting for an event is provided by **`OSTimeTick()`**, which supplies the *clock ticks* or the heartbeats. **`OSTimeTick()`** also decrements the **`OSTCBDly`** field for each **`OS_TCB`** that is not zero. The time between tick interrupts is application specific and is typically between 10 ms and 200 ms. **`OSTimeTick()`** increments a 32-bit variable OSTime since power up. This provides a system time.

3.9 Communication and Synchronisation

µC/OS supports message *mailboxes* and *queues* for communication. A task can deposit, through a kernel service, a message (the pointer) into the mailbox. Similarly, one or more tasks can receive messages through a service provided by the kernel. Both the sending and receiving task have to agree as to what the pointer is pointing to. A message queue is an array of mailboxes. µC/OS supports *semaphore* (0-32767) for synchronisation and coordination.

The above services are *events*. Thus, a task can signal the occurrence of an event (**`POST`**) or wait for an event to occur (**`PEND`**). However, the ISR can **`POST`** an event but cannot **`PEND`** on an event. When an event occurs, the highest priority task waiting for the event is made ready to run.

3.10 Memory Requirements

The memory required for the program is less than 3150 for the 80186/80188 microcontroller. This can be reduced if some of the services are not required. The RAM or data memory is as follows:

200

+ ((1 + OSMAX_TASK) * 16)

+ (OS_MAX_EVENTS * 13)

+ (OS_MAX_QS * 13)

+ SUM(Storage requirements for each message queue)

+ SUM(Storage requirements for each task stack)

+ (OS_IDLE_TASK_STK_SIZE)

3.11 Kernel Services

For reference the kernel services are given in the following table:

#	Service	Description
1	**OSInit()**	Initialize μC/OS
2	**OSIntEnter()**	Signal ISR entry
3	**OSIntExit()**	Signal ISR exit
4	**OSMboxCreate()**	Create a mailbox
5	**OSMboxPend()**	Pend for message from mailbox
6	**OSMbox Post()**	Post a message to mailbox
7	**OSQCreate()**	Create a queue
8	**OSQPend()**	Pend for message from queue
9	**OSQPost()**	Post a message to queue
10	**OSSchedLock()**	Prevent rescheduling
11	**OSSchedUnlock()**	Allow rescheduling
12	**OSSemCreate()**	Create a semaphore
13	**OSSemPend()**	Wait for a semaphore
14	**OSSemPost()**	Signal a semaphore
15	**OSStart()**	Start multitasking
16	**OSTaskChangePrio()**	Change a task's priority
17	**OSTaskCreate()**	Create a task
18	**OSTaskDel()**	Delete a task
19	**OSTimeDly()**	Delay a task for n system ticks
20	**OSTimeGet()**	Get current system time
21	**OSTimeSet()**	Set system time
22	**OSTimeTick()**	Process a system tick

4 Embedded Systems Networking in Distributed Environment

4.1 Introduction

There are two broad methods of connecting embedded systems in a distributed environment:

(1) Non-Internet connection
(2) Internet connection

Before the proliferation of the Ethernet and Internet, connecting embedded systems in a distributed environment such as a laboratory or a factory was usually achieved by conventional parallel or serial connections. Parallel connections were suitable for short-range use, typically within a few meters, limited by difficulties in driving synchronous multiple signals and cost of cabling. The advantage of parallel connection was obviously higher data rates on a parallel bus. Parallel bus standards at Mbps rates were implemented when the standard serial link was still at Kbps.

While there are dedicated parallel bus standards designed for specific industries or environments such as the General Purpose Interface Bus (GPIB), the Computer Automated Measurement And Control (CAMAC) and the Small Computer System Interface (SCSI), the most common method is the parallel port, used typically for printer connection. This is simple to use and is readily available, being a standard I/O port in all PCs. The newer version (IEEE 1284) available in most new PCs is being used for many purposes other than connecting to a printer.

However, by far the most important networking technique in a distributed environment beyond the size of a test bench is, of course, the serial bus, exemplified by the RS-232-C (Recommended Standard number 232, revision C from the Electronic Industry Association) and its related standards. Like the parallel port, the use of this serial technique is extremely handy. There are two serial ports available in most standard desktop PCs – COM1 and COM2. Users and manufacturers alike are taking advantage of these two serial ports for almost every type of equipment.

The limitations of the standard PC serial ports are data rate and distance, being 20 Kbps and <50 feet. Related standards such as RS-485 eliminate such limitations to a great extent while still maintaining the simplicity of the standard. For

example, RS-485 is capable of 4000 feet and 10 Mbps performance. It has an added advantage of a bus configuration which allows it to connect to 32 transmitters and 32 receivers. (For comparison, the RS-232 is a point-to-point connection which can only connect two machines together.) Although existing OS in PCs would not be able to handle continuous 10 Mbps data rate, the 4000-foot transmission distance is a welcome feature in distributed laboratory environment.

As newer equipment and gadgets (especially video devices) demand higher and higher data rates, there are a couple of high-speed serial communication standards that have been introduced to the PC world in recent years. Universal Serial Bus (USB) provides for peripheral speeds of up to 1.5 Mbps for low-speed devices and up to 12 Mbps for full-speed devices. The latest USB 2 specification increases the data rate to 480 Mbps. The cable length is limited to 5 metres, extendable to 25 metres using 4 hubs or with active cables. A total of 127 devices may be connected.

A competing standard to USB is the IEEE 1394 (FireWire or iLink) which has a maximum data rate of 400 Mbps and is cable of connecting up to 63 devices. At this dazzling speed, full frame rate video can be transmitted from cameras to PCs directly through a thin cable. The cable length is 4.5 metres, again extendable with hubs and repeaters.

Some applications call for wireless connection. This is provided by IrDA, a standard defined by the Infrared Data Association. It specifies wireless transfer via infrared radiation at 875 nm. IrDA 1.1 has a maximum data rate of 1.152 Mbps. At this rate, the range is small, typically less than 20 cm.

The recent ubiquitous deployment of the Ethernet and Internet and the availability of low cost embedded systems, capable of functioning as web server, have changed everything in networking equipments in a distributed environment. It is now the age of the Internet! In this respect there are two approaches. One is to forget your old equipment and design new web-based ones and network them. The other is to add a web-enabled front-end or intermediary to your existing equipment and thus make it accessible via the Internet. Prudence and reality tell us that there are many situations where existing equipment cannot be abandoned or redesigned. Thus the second approach is an important one. Indeed, many companies dedicated to providing such hardware and software have appeared in the last couple of years. Their solutions are typically very attractive and cost effective, and we shall look at a couple of them.

If you embark on an entirely new project then you have two options in hardware for your embedded systems. If you are lucky enough to have a large budget, then use a PC or one of its derivatives (SBC and embedded PC) as your embed-

ded processor. The interface hardware can also be purchased as standard plug-in boards on the PCI (Peripheral Component Interconnect) bus. This eliminates hardware design completely. There are numerous suppliers producing a plethora of I/O boards. You can almost always find what you want provided that you can afford it. We shall not look into this area in these lectures.

Sometimes a PC is an overkill for your applications or your budget simply cannot afford using PCs. Until recently, building web-based applications under such a situation has been difficult. One often settled for the non-Internet option. Rather affordable hardware is currently available in the market either as components or ready built modules that are web-enabled. We shall look into this area.

4.2 Non-Internet Connection

The hardware or physical layer specifications of a few common methods of connecting embedded systems are described in this section. They are

(1) The parallel port
(2) The serial port
(3) Selected high-speed ports

4.3 The Parallel Port

Originally intended for printer connection, this port has undergone several revisions and it now provides bi-directional connection to a host of equipment. The initial printer port in the IBM PC in 1981 was really designed for unidirectional connection to the relatively slow dot-matrix printers at the time. It transferred data at 150 kilobytes/second and was software intensive. Lack of design standards forced a cable limitation to 6 feet. In 1987, IBM PS/2 was introduced and this PC enhanced the parallel port by adding bi-directional data flow.

Printer and computer manufacturers started to enhance the parallel port standard in 1991 and this resulted in two improved standards: EPP (Enhanced Parallel Port) and ECP (Extended Capability Port). EPP was introduced by Intel, Xircom and Zenith Data Systems and was used primarily by non-printer peripherals such as CD ROM, tape drives, hard disks etc. ECP was introduced by Hewlett Packard and Microsoft and was used primarily by new printers and scanners.

In 1994 the current IEEE 1284 standard, “Standard Signalling Method for a Bi-directional Parallel Peripheral Interface for Personal Computers”, was released. This standard encompasses all the other modes of parallel port, viz.

(1) Compatibility mode – Centronics or standard mode
(2) Nibble mode – 4 bits at a time using status line for data
(3) Byte mode – 8 bits at a time using data line
(4) EPP – bi-directional, up to 2 Mbytes/second
(5) ECP – bi-directional, DMA transfer, up to 2 Mbytes/second

The standard specifies a cable length of 10 metres.

The original parallel port consists of 17 signal lines falling into three groups – Data (8 lines), Status (5 lines) and Control (4 lines). The remaining 8 lines in the DB25 connector port are grounded. For compatibility, the 17 signal lines are maintained even in the advanced EPP and ECP mode. However, the function and designation of the lines are modified according to the new requirements of the modes.

In the original PC, the data lines are output-only lines and thus they cannot be used to read data from peripheral devices. (The nOC output control pin of the 74LS374 D-type flip-flops are permanently grounded to enable all output lines.) To get round this problem, the Nibble mode uses 4 of the 5 status (input) lines to read 4 bits at a time from the peripheral devices. While this is more complicated and slow, it has the advantage of compatibility with the oldest PCs.

Most PCs produced after 1987 have the ability to control the direction of the data port thus making bi-directional transfer of 8-bit data using the same port possible. This is the Byte mode.

For the first three modes of communication – Compatibility, Nibble and Byte – handshaking of data between the PC and peripheral is carried out by the CPU and it takes several clock cycles to transfer a single byte or nibble. This limits the data rate to 150K bytes per second for byte-wide transfer and to 50K bytes per second for nibble-wide transfer. These data rates are however quite respectable for many laboratory applications. Because of their simplicity, they are used in many embedded system applications. Hence, we shall look into the actual transfer cycles in these modes.

The EPP mode enhances the performance of the parallel port by increasing the data rate to 2M bytes per second while still maintaining compatibility with the standard port. A read or write cycle is done within one ISA I/O cycle. This is achieved with a local protocol between the host and peripheral using a state machine and the necessary hardware to carry out interlocking handshakes. To enhance the performance, the EPP protocol defines four types of data transfer cycles: (1) data write, (2) data read, (3) address write and (4) address read. While data cycles and address cycles are intended for transferring data and ad-

dress/command respectively, they are really just two transfer channels and can be used flexibly by device manufacturers.

The EPP data rates are sufficient for use with network adapters, hard disks, other tape storage devices and CD ROMs. Thus many device manufacturers adopted the EPP mode swiftly as an optional data transfer method.

The ECP came even later as another high performance parallel port mode. It is specifically designed with printers or similar devices in mind. The protocol allows for (1) data and (2) command cycles in both forward (write) and reverse (read) directions. In this respect it is similar to the EPP. However, it includes several features that are very specific to implementations but powerful. They are (1) Run Length Encoding (RLE) data compression, (2) FIFOs on both forward and reverse channels and (3) DMA and programmed I/O for the host register interface.

The RLE, with a compression ration up to 64:1, is useful for printers and scanners that handle raster images with large amounts of identical data (blank or black). The channel addressing concept is intended for addressing multiple logical devices using a single physical port. A good example of such a requirement is the combined Fax/Scanner/Printer machine.

The following tables show the pin designations and other relevant information for the first three modes of operation. These are modes that are widely used by embedded systems implementers in the laboratory. The remaining two modes, EPP and ECP, while more advanced, are not as commonly used in simple laboratory applications. When high data rates are required, it is now advisable to implement high-speed serial link interfaces such as the USB or FireWire.

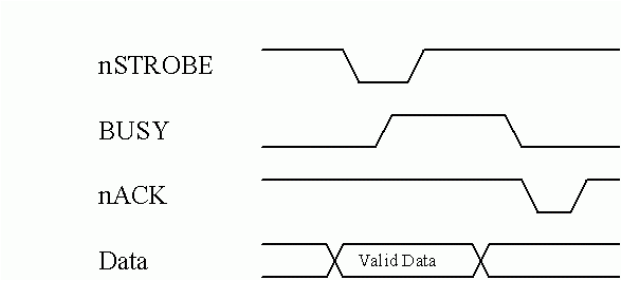
Compatibility Mode (or Standard Parallel Port Mode)

Pin	Signal	I/O	Reg	Description
1	nSTROBE	Out	CR-0	Falling edge strobes data byte into printer
2	D0	Out	DR-0	Bit 0 of data
3	D1	Out	DR-1	Bit 1 of data
4	D2	Out	DR-2	Bit 2 of data
5	D3	Out	DR-3	Bit 3 of data
6	D4	Out	DR-4	Bit 4 of data
7	D5	Out	DR-5	Bit 5 of data
8	D6	Out	DR-6	Bit 6 of data
9	D7	Out	DR-7	Bit 7 of data
10	nACK	In	SR-6	Low pulse indicates byte received
11	BUSY	In	SR-7	High indicates printer busy
12	PE	In	SR-5	High indicates out of paper
13	SELECTED	In	SR-4	High indicates printer online

14	nAUTOFEED	Out	CR-1	Low to insert line-feed on each carriage return
15	nERROR	In	SR-3	Low to indicate printer error
16	nINIT	Out	CR-2	Low to reset printer
17	nSELECT	Out	CR-3	Low to select printer
18-25	Ground			Ground

CR - Control Register
DR - Data Register
SR - Status Register

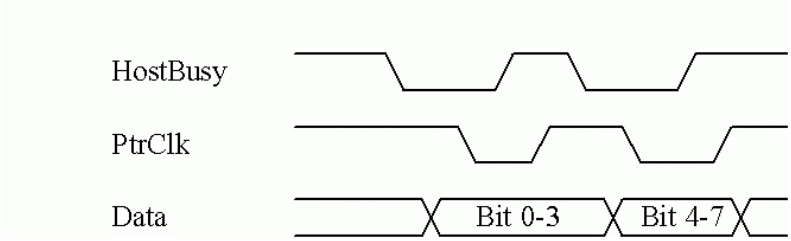
Compatibility Mode Data Transfer Cycle



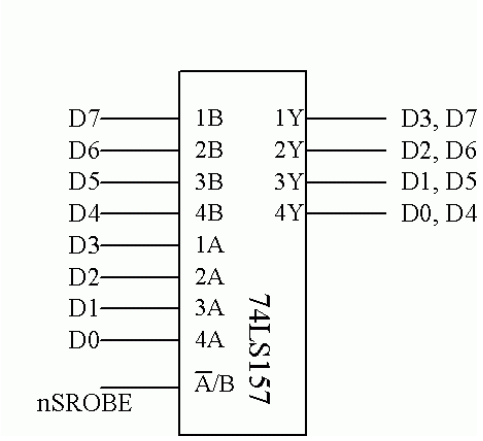
Nibble Mode (Reverse Transfer)

Pin	SPP Signal	Nibble Mode	I/O	Description
1	nSTROBE	Not used	Out	Not used
2	D0	Not used	Out	Not used
3	D1		Out	
4	D2		Out	
5	D3		Out	
6	D4		Out	
7	D5		Out	
8	D6		Out	
9	D7		Out	
10	nACK	PtrClk	In	Low indicates valid nibble High in response to HostBusy going high
11	BUSY	PtrBusy	In	Data bit 3 then bit 7
12	PE	AckDataReq	In	Data bit 2 then bit 6
13	SELECTED	Xflag	In	Data bit 1 then bit 5
14	nAUTOFEED	HostBusy	Out	Low indicates host ready for nibble High indicates nibble received
15	nERROR	nDataAvail	In	Data bit 0 then bit 4
16	nINIT	NINIT	Out	Not used
17	nSELECT	1284Active	Out	High indicates 1284 mode
18-25	Ground	Ground		Ground

Nibble Mode Data Transfer Cycle



On the peripheral side, the reverse transfer in the Nibble mode can be implemented with a quad 2-line-to-1 multiplexer such as the 74LS157 as shown in the following diagram.

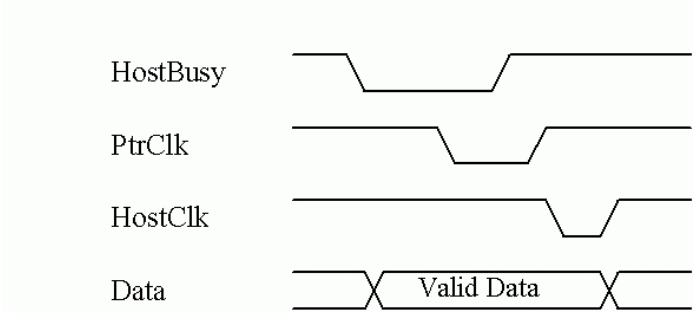


Byte Mode

Pin	SPP Signal	Byte Mode	I/O	Description
1	nSTROBE	HostClk	Out	Pulse low to indicate byte received
2	D0	D0	I/O	Bit 0 from peripheral to host
3	D1	D1	I/O	Bit 1 from peripheral to host
4	D2	D2	I/O	Bit 2 from peripheral to host
5	D3	D3	I/O	Bit 3 from peripheral to host
6	D4	D4	I/O	Bit 4 from peripheral to host
7	D5	D5	I/O	Bit 5 from peripheral to host
8	D6	D6	I/O	Bit 6 from peripheral to host
9	D7	D7	I/O	Bit 7 from peripheral to host
10	nACK	PtrClk	In	Low indicates valid data High in response to HostBusy going high
11	BUSY	PtrBusy	In	Printer busy
12	PE	AckDataReq	In	Follows nDataAvail
13	SELECTED	Xflag	In	Extensibility flag, not used
14	nAUTOFEED	HostBusy	Out	Low indicates host ready for nibble High indicates nibble received

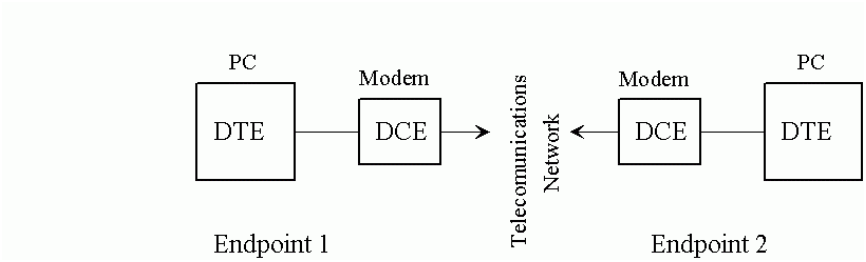
15	nERROR	nDataAvail	In	Low indicates data available
16	nINIT	NINIT	Out	Not used
17	nSELECT	1284Active	Out	High indicates 1284 mode
18-25	Ground	Ground		Ground

Byte Mode Data Transfer Cycle



4.4 The Serial Ports

The two serial communication ports in the PC follow the RS-232 standard of the Electronics Industry Association. It was introduced in early 1960s and has since been the most widely used serial communication technique in the laboratory. The title of the standard, "Interface Between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange" indicates the original intention. That is, to connect Data Terminal Equipment (DTE) to Data Communications Equipment (DCE). DTE and DCE are the two pieces of equipment at the two endpoints of a telecommunication channel. The DTEs used to be dumb terminals with which the users interact. The DCEs were modems that connected to telephone lines. In most laboratory and office environments now, the DTEs are more likely to be PCs. The DCEs may still be modems, but they may also be non-existent because the other endpoint is nearby and it is easier to use straight-through cables instead of the telephone network. In this case, the DCEs are called Null Modems.

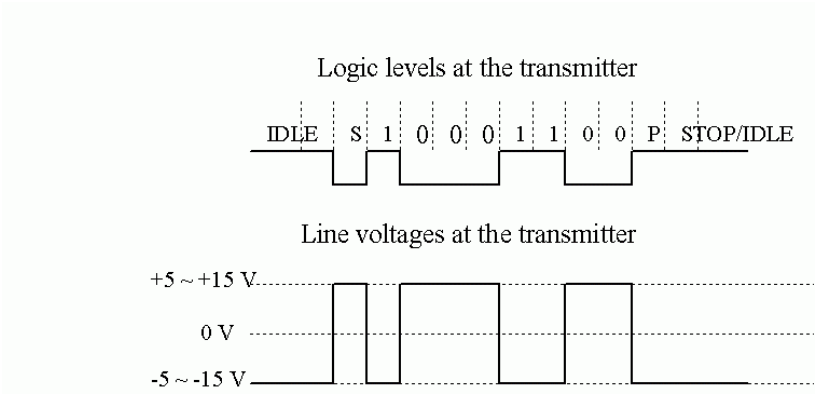


The RS-232 uses an asynchronous mode. Typically, data is transferred in symbols or words of 5 to 8 bits, one bit at a time, least significant bit first, at a predetermined speed called the baud rate. However, from one word to the next, the time interval is not fixed and thus asynchronous. A long string of words may be transferred in a contiguous manner, one immediately trailing the previous one, or they may be sent intermittently, with varying pauses between words.

No clock or other timing signal is sent. It thus requires a start bit to indicate the beginning of a byte and a stop bit to signify the end. It is the receiver's job to sample the bits based on these marking bits. For the purpose of error checking, a parity bit (either even or odd parity) may be added at the end of each word or symbol. The transmitter computes the parity bit based on all the data bits. The receiver checks if the actual parity bit value corresponds to the calculated value.

The actual signals sent on the line are positive and negative voltages with respect to ground potential. A bit 0 or logical 0 is sent as SPACE or +5 to +15 volts; a bit 1 or logical 1 is sent as MARK or –5 to –15 volts. The receiver recognises –3 to –25 volts as MARK and +3 to +25 as SPACE. The region between –3 volts to +3 volts is a transition region where the state is undefined. The wide range of voltages for the two valid states are very useful in actual implementation, taking care of line drop, supply fluctuation, ground potential difference etc. A number of restrictions imposed by the standard further enhance the robustness. For example, the output of the transmitter may be shorted to other lines without causing any damage to itself and the receiver must be able to withstand voltages in the range of –25 volts to 25 volts.

When a line is idle, it is in the MARK state. The start bit is SPACE, stop bit(s) is MARK. The timing diagram of sending numeral "1" or 0x31 with even parity is shown below.



The standard specifies a total of 21 control signals or 'circuits' on top of the two data lines (Transmit Data and Receive Data). The original specification thus requires a DB-25 connector with 25 circuits. The control lines are used to connect to a DCE or modem for control or test functions. It turns out that many of the control lines are not needed for standard modem connections. In fact they may not be used at all if an RS-232 port is connected to another RS-232 port on two pieces of equipment. In this case, the only requirement is of course to connect the Transmit Data of one DTE (the PC) to the Receive Data of the other. This crossing of two wires, from pin 2 to pin 3, constitutes a Null modem.

In most PCs, a subset of the lines is used for the serial port. Six control signals are chosen, making it possible to house the port in a DB-9 connector with 9 circuits. The connector on the PC or DTE is a male (or plug) while the corresponding one on the modem or DCE is a female (or socket). If two PCs are connected using the serial COM ports, the cable (or the Null modem) obviously has two female DB-9 connectors.

Three types of connectors are used for RS-232: DB-9 (9-pin D-shell connector), DB-25 (25-pin D-shell connector) and RJ45 (modular telephone connector). The various signal assignments on the DTE (or PC) side are as follows:

Name	Direction	Description	DB-9	DB-25	RJ45
CD	←	Carrier Detect	1	8	2
RxD	←	Receive Data	2	3	5
TxD	→	Transmit Data	3	2	6
DTR	→	Data Terminal Ready	4	20	3
Gnd		Signal Ground	5	7	4
DSR	←	Data Set Ready	6	6	(1)
RTS	→	Request To Send	7	4	8
CTS	←	Clear To Send	8	5	7
RI	←	Ring Indicator	9	22	1

Direction is DTE (PC) relative to DCE (modem)

The common bit rates of RS-232 are at 1200, 2400, 4800, 9600, 19200 bps. With modern drivers, higher bit rates, while non-standard, are possible.

The standard specifies a cable length of 50 feet or 2500 pF in cable capacitance, at 20K bps. If cables with lower capacitance are used, it is possible to transmit at distances greater than 50 feet. For example the UTP Cat-5 cable has a capacitance of 17 pF/ft. Thus about 150 feet of cable may be used. Likewise, reducing the transmission speed increases the maximum cable length. Laboratory tests showed that a cable length of 500 feet is possible at 9600 bps.

Flow control in RS-232 if needed is carried out either in hardware or software to prevent buffer overflow on the receiver. In the hardware or the RTS/CTS flow

control method, the transmitter asserts a Request To Send signal when it has data to transmit. The receiver asserts a Clear To Send signal only when it is ready to receive data. In the software or XON/XOFF control mode, the receiver tells the transmitter to send more data by sending an XON control character (0x11). When the receiver buffer is nearly full, it stops the transmitter by sending an XOFF control character (0x13). Both the XON and XOFF are sent as data and no hardware control signals are involved.

Very small embedded systems connected to the serial port may get their power from the serial port itself although this is not part of the standard. A serial port mouse is such an example. Since DTR and RTS are often not used as intended, they may act as a source of small power supply. These pins are designed to drive a load of 3~7 kOhm at 7 to 11 volts typically and they can be used to drive a regulator to produce regulated 5-volt output, provided that the load current is small, typically less than 10 mA, depending on the type of driver used in the port. An output pin from a standard 1488 RS-232 driver chip can supply up to 6.5 mA at 5 volts and that for the MAX232 is 5 mA. Devices operated in this manner are called pin powered devices.

4.5 Differential Drive Serial Communication Standards (RS422, RS485)

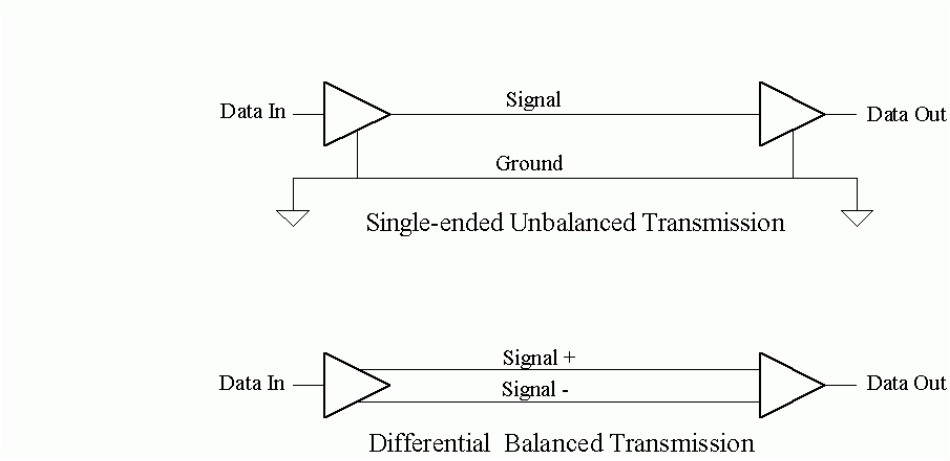
While RS-232 is simple and readily available, there are some shortcomings that reduce its speed and transmission distance. First, it assumes common ground potential between the two endpoints, the result of the single-ended (or unbalanced) drive. This may not be so when the separation of the devices increases. A receiver at a different ground potential will sense the signal voltages with an undesired offset.

A single signal cable makes screening or shielding of electromagnetic noise difficult if not impossible. While a separate screen sheath may be used on the cable, internal noise generated by other signal lines is a problem. This crosstalk effect becomes more and more severe as transmission speed goes up.

A different EIA standard, the RS-423 attempts to eliminate the shortcomings of RS-232 to a certain extent. By using different electrical parameters including a controlled slew rate on the signal (rise and fall times at 30% of unit interval at >= 1K baud), a speed of 100K bps at 4000 feet can be achieved.

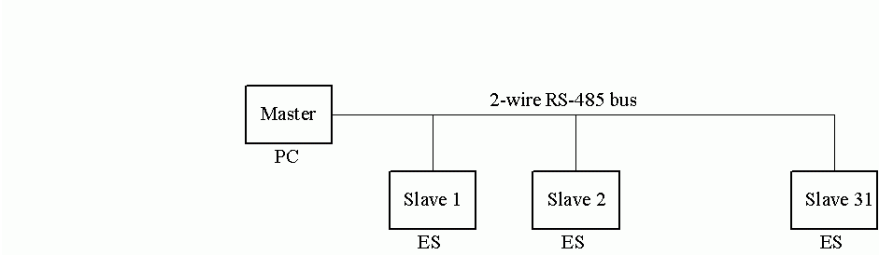
However, a more significant enhancement of the serial communication technique is achieved by using differential or balanced drive. Two famous EIA standards that operate in this mode are the RS-422 and RS-485. A pair of wires is used for

transmitting each signal. The voltage difference between the two wires is unaffected by difference in ground potentials and the receiver should function properly provided that the common mode voltage limit is not exceeded. If these two balanced wires are twisted together the difference voltage is unaffected by external electromagnetic noise, including crosstalk. Noises picked up by the two wires are likely to be identical and hence they are cancelled at the receiver input which is a differential voltage sensor.



Using this differential drive technique, transmission speed up to 10M bps can be achieved. The cable length is specified at 4000 feet. The RS-422 standard also specifies a multi-drop (party-line) arrangement where one driver (master) is connected to 10 receivers (slaves) that listen simultaneously. The master transmit lines are connected to all the receivers of the slaves. The slave transmit signal is carried on another pair. They are all connected to the receiver of the master. At any one time, only one of the slave transmitters is turned on to avoid garbling of data.

The RS-485 is a true multipoint arrangement whereby all the transmitters and receivers are connected to a single pair of wires, the bus. This is possible because the transmitter can be put into a high impedance or tri-state mode. Up to 32 pairs of transmitters and receivers may be connected together. The standard does not specify the method for orderly transmission in this bus system. One simple implementation method is for the master to initiate a communications request to a slave station by addressing it. It turns off its transmitter immediately and the slave can then transmit on the bus.



A 9-bit protocol is often used in the multidrop network shown above. An extra 9^{th} bit is added to the typical 8-bit data in the serial transmission to indicate it is an address frame. When the master wants to transmit a block of data to a particular slave, it first sends an address byte as data and turns on the 9^{th} bit. The particular slave, whose address matches that sent by the master, knows that it is being addressed. It thus receives the subsequent block of data based on whatever protocol was agreed upon earlier. Other slaves ignore the block of data not intended for them.

As standard transceivers (or UART) for RS-232 do not support 9^{th} bit addressing, the parity bit is often used as the 9^{th} bit. This is the case if the PC is used. In this case, the parity checking capability is lost of course. However, this is often not a problem as many existing applications ignore the parity checking in the first place. Other block checking techniques including the CRC are used.

A summary of the RS-232 and RS-485 electrical specifications are shown below.

RS-232

Parameter	Conditions	Min	Max	Units
Output (Open Circuit)			25	V
Output (Loaded)	3 K <=R_L<= 7 K	5	15	V
Output Resistance	-2V<=Vo<=2V		300	Ohm
Output Current (Short-Circuit)			500	mA
Output Slew Rate			30	V/µs
Maximum Load Capacitance			2500	pF
Receiver Input Resistance	3V <= V_{IN}<=25V	3000	7000	Ohm

RS-485

Parameter	Conditions	Min	Max	Units
Output (Open Circuit)		1.5 -1.5	6 -6	V V
Output (Loaded)	R_{LOAD} = 54 Ohms	1.5 -1.5	5 -5	V V

Output Current (Short-Circuit)	Output to +12V or -7V		±250	mA
Output Rise Time	R_{LOAD} = 54 Ohms C_{LOAD} = 50 pF		30	% (bit width)
Output (Common Mode)	R_{LOAD} = 54 Ohms	-1	3	V
Receiver Sensitivity	$-7 <= V_{cm} <= +12$		±200	mV
Receiver (Common-Mode)		-7	+12	V
Receiver Input Resistance		12K		Ohm

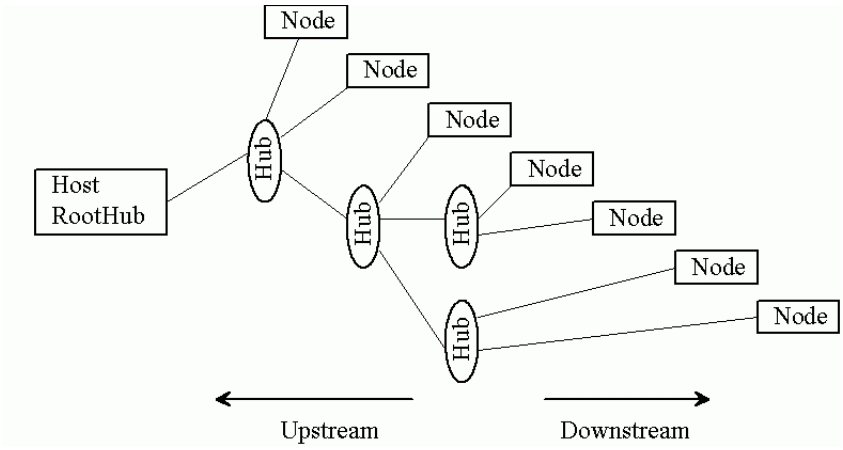
4.6 Universal Serial Bus (USB)

The Universal Serial Bus (USB) was standardised in 1995 by a group of companies to introduce an advanced serial bus to the PC. In the short period since its appearance, it has been widely accepted and all new PCs now incorporate the USB ports. There are a myriad of USB peripheral devices produced (over 1000 products have passed the compliance test), including mice, scanners, printers, digital cameras, hard disks, multimedia equipment, etc.

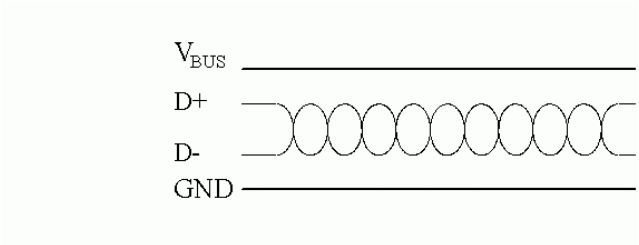
For embedded system developers, the USB provide an alternative and more powerful way of interfacing to the PCs. It is a serial bus with hot-swap capability, meaning a device may be connected or disconnected without turning off the power.

USB 1.1 offers two data rates, full speed at 12 Mbps and low speed at 1.5 Mbps. USB 2.0 offers high speed at 480 Mbps. Devices of different speeds may be mixed and the bandwidth may be divided into asynchronous and isochronous streams. The latter offers guaranteed or reserved bandwidth and is useful for streaming devices, such as audio and video equipment.

A total of 127 devices may be connected to a single host. This is done by connecting multiple USB hubs in a daisy chain. Devices are connected to the downstream port of a hub while the upstream port is connected to the host or another hub (in the upstream of the bus topology). Devices are either self powered or powered from the bus. A typical USB hub may provide 100 mA on each downstream port in the Bus Powered mode, and 500 mA on the Self Powered mode.



USB devices are connected using a simple cable consisting of a twisted pair of wires for signal and an untwisted pair for power (V_{bus} at 5 V and ground). Cable segments can be up to several metres depending on the wire gauge used.



The clock speeds for high-, full- and low-speed operations are 480, 12, and 1.5 Mbps respectively. The voltage level for logical 1 at the final target connector is defined by (D+) - (D-) > 200 mV and that for logical 0 is (D-) - (D+) > 200 mV.

The USB uses a polled bus protocol whereby the host (or PC) initiates all data transfers. A transaction begins by the host sending a token packet. It contains information on the type and direction of transaction, device address, and end-point number. Data transfer than occurs between the source and the destination by the source sending either a data packet or a packet indicating no data. The destination closes the transaction by returning a packet indicating where the previous packet has been received successfully.

The pipe concept is used to refer to the channel between an endpoint in a device and that in a host. Two types of pipes are defined: stream and message. Streams have no defined data structure while messages have. A message pipe called Default Control Pipe is set up when a device is powered. It provides access to the device configuration, status and control information. Pipes have associations of data bandwidth, transfer service type, and endpoint characteristics like direc-

tions and buffer sizes. A flow control mechanism allows the construction of flexible schedules. This allows concurrent servicing of a heterogeneous mix of streams, i.e. multiple streams can be served at different intervals with packet of different sizes.

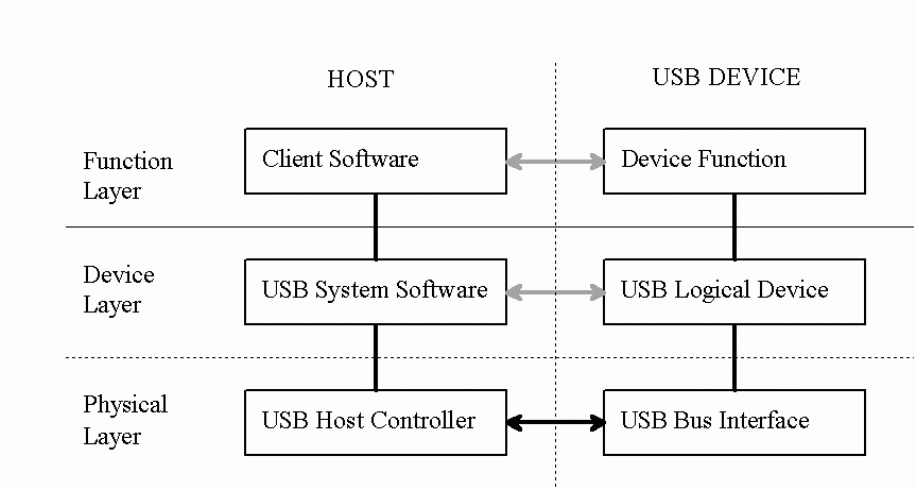
There are four types of data transfers: (1) Control Transfer, (2) Bulk Data Transfer, (3) Interrupt Data Transfer and (4) Isochronous Data Transfer.

Control transfer, a lossless data transfer method, is used to configure a device at attach time. Bulk data transfer is used for transferring asynchronously large volume of data. Data integrity is ensured at hardware level by error detection and retries if necessary. An example is the transfer from the host to a printer.

Interrupt data transfer is used for transferring small amount of data with small latency or rate specified by the device. Data are typically event notification, character or coordinates in one or more bytes. An example is the pointing device.

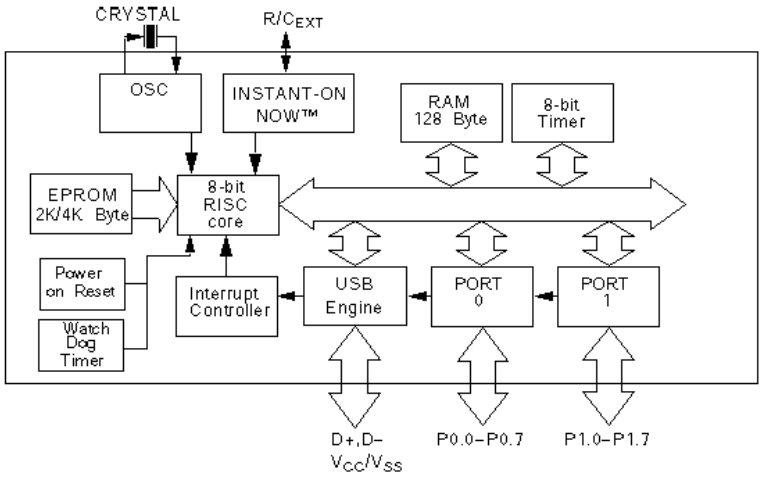
Isochronous data transfer is designed for real-time streams such as voice and video. A dedicated bandwidth is allocated for this transfer. The timeliness is ensured at the expense of occasional transient data losses. In other words, hardware retries are not used in case of data loss. This is known to be better for real-time voice and video transmission.

The USB standard implements a data flow model that can be divided intro three layers: (1) Function Layer, (2) USB Device Layer, and (3) Physical Layer as shown below.



The physical layer provides physical, signalling and packet connectivity function between the host and a device. The device layer carries out general USB operation between the host and a device through a logical link. In the host, this is typically carried out in the operating system, as API for example. The function layer is used for high level function communication between the host and a device, again through a logical link as viewed by the software.

There are a number of manufacturers producing components specially designed for implementing embedded systems using the USB. The functions of the physical and device layers are taken care off by the USB chips. An example is the CY7C630/1XXA series produced by Cypress Semiconductor Corporation. This particular series of chips conforms to the USB 1.1 specifications operating at 1.5 Mbps. Members of the family have an 8-bit RISC microcontroller with a built-in 1.5-Mbps USB Serial Interface Engine (SIE). Memories are in the range of 128 bytes RAM and 2 to 4 Kbytes EPROM. Besides the USB interface, two 8-bit parallel ports are available. Such USB chips are suitable for small to medium scale applications including serial port conversion, keyboard, mouse, joystick and many others. The block diagram of a CY7C63000 chip is shown below.



4.7 IEEE 1394 Bus

This is an emerging high speed serial bus originally developed by Apple (called FireWire) and used by Sony in digital video equipment (called iLink). It was accepted as an industrial standard in 1995 (called IEEE 1394). Before the introduction of USB 2.0 specifications, the IEEE 1394 was the highest speed serial bus at 400 Mbps. A 1.6 Gbps and 3.2 Gbps versions are being defined and developed. The following is a partial of peripherals available with IEEE 1394

interface: digital video camcorders, digital cameras, printers, mass storage devices, ADCs.

It is similar to USB in principle. It has both asynchronous mode isochronous mode of transmission suitable for both bursty data and real-time video streams. At 400 Mbps, full frame rates digital video streams can be transmitted. It has hot swapping capability. It allows up to 63 devices to be connected. Devices can also be bus powered or self powered.

One significant difference between IEEE 1394 and the USB is the bus topology. The IEEE 1394 uses a peer-to-peer configuration, instead of the master-slave configuration. This means a PC is not needed to act as a host or master for communication. An IEEE 1394 digital video camera can be connected to a video player directly. Similarly more than one PC can be connected to a digital camera to use the same video stream.

This, together with the speed and other requirements, makes the specifications rather complex. The specifications document is a few hundred pages divided into many areas of applications, including instrumentation, industrial and automotive, on top of the more familiar video related specifications of digital video, MPEG, digital TV, etc. In the instrumentation area, one application is to replace the now rather slow IEEE 488 GPIB with an IEEE 1394 bus. There is an effort to define how the IEEE 488.2 command structure can be transmitted over the IEEE 1394.

As in the USB, the IEEE 1394 uses a three layers model for communication: Physical, Link and Transaction. The physical layer handles the signals required by the bus; the link layer formats data into packets. Packets are passed on to the transaction layer which then present them to the application above it. Semiconductor chips are available for handling the functions of the physical and link layers. Part of the transaction layer functions is also handled by the dedicated IEEE 1394 chip. The rest are in software, in the OS and the user programs.

In the Linux environment, version 2.2 onwards support IEEE 1394. The subsystem has been included with the standard kernel sources since version 2.3.40. It is distributed as a patch to version 2.2. IEEE 1394 is also supported on the Windows and Macintosh platforms.

The IEEE 1394 cable consists of two differential signal pairs, a power and a ground line. Two types of connectors are defined: a six-wire version (measures 10 mm by 5 mm) and a four-wire version (measures 5 mm by 3 mm) without the power pair. The maximum cable length for speed greater than 200 Mbps 4.5 m. At lower speed, cables up to 14 m long can be used. Multiple devices can be

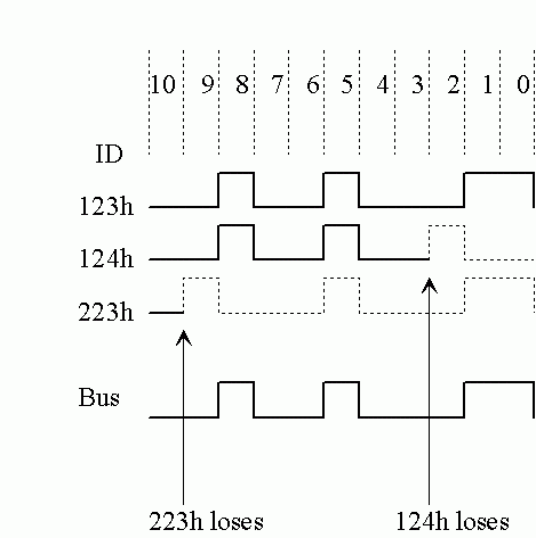
daisy-chained using repeaters to extent communication distance. It is interesting to note that the future IEEE 1394b specifications also support cable length of 100 m using plastic optical fibre and multiple kilometre using glass optical fibre. CAT-5 cables can be used at 100 Mbps at distances up to 100 m.

4.8 Controller Area Network (CAN)

The Control Area Network (CAN), originally developed for the automotive industry by Robert Bosch in 1980s, is a real-time serial data communications bus now being deployed in many distributed environment. It is now accepted as an international standard (ISO 11898 and ISO 11519 for high and low speed applications respectively). Many semiconductor manufacturers (e.g. Atmel, Fujitsu, Intel, Microchip, Motorola, Philips, Siemens, etc.) produce a wide range of controller and interface chips for CAN.

CAN controllers are physically small and relatively cheap devices to be used for real-time data acquisition and control applications. Data are transmitted on a two-wire bus using 5-V differential mode using a multicast protocol. Large number of devices may be connected to the bus. The limit is imposed by the drive capabilities of the control chips and up to 64 nodes is common. Unlike most other bus arrangements, devices are not identified by their device address. Data messages transmitted on the bus do not carry source or destination address. Unique message identifiers are used instead. A node sends out a message with an identifier and all the nodes on the bus receive it. It's up to the receiving node to decide whether to process the incoming message. The message identifier also serves as a priority indicator. The lower identifier has a higher priority.

Contention is solved by Carrier Sense, Multiple Access with Collision Detect (CSMA/CD) as in the Ethernet. However, an enhanced feature for non-destructive bitwise arbitration is used to resolve collision. This is achieved by a wired-AND mechanism. A logical 0 in the identifier bit pattern is considered dominant and it overwrites a logical 1 which is recessive as shown below. Thus a node sending out message with an identifier 123h will overwrite node with message identifier 223h and that then with identifier 124h. Once a node transmitter with a lower priority loses, it turns into a receiver and listens to the message at a higher priority (or lower identifier value).



A useful feature in the message-based protocol of the CAN is the Remote Transmit Request (RTR). This allows a node to request information from other nodes. The protocol also permits additional nodes to be added without having to reprogram the existing nodes in the network.

Four different types of messages or frames are defined: (1) Data Frame, (2) Remote Frame, (3) Error Frame and (4) Overload Frame.

A data frame consists of the data field and several other fields to provide additional information. The whole frame can be divided into Arbitration Field, Control Field, Data Field, CRC Field, Acknowledge Field and End of Frame Field. The Arbitration Field is the message ID mentioned above and is used for prioritising messages on the bus. The prioritising field has 11 (standard frame) or 29 (extended frame) identifier bits and one RFR bit. If the RFR is set, the frame becomes a remote frame. The control field consists of a bit that signifies extended frame and a four-bit Data Length Code (DLC). The value of DLC is between 0 and 8 representing 0 to 64 bits. The remote frame has no data field, irrespective of the value of the DLC.

The CRC is a 15-bit CRC field with a delimiter. The acknowledge field is used to indicate if the message was received correctly, irrespective of whether it was processed or ignored. This is done by asserting a dominant bit on the bus at the ACK slot bit time.

When a node detects an error, it sends an error frame. When a node is not ready to receive additional messages, it sends an overload frame on to the bus. CAN

was designed with error tolerant and robustness in mind. Thus, there are many types of errors being used: (1) CRC Error, (2) Acknowledge Error, (3) Form Error, (4) Bit Error, and (5) Stuff Error.

Although CAN is not a standard bus in PCs, CAN interface cards are readily available. As mentioned earlier, a wide range of CAN-based controller chips are available as standard semiconductor parts. There are two hardware implementations: (1) Basic and (2) Full. The former uses a standalone CAN controller connected to an existing microcontroller. The latter integrate CAN controller, CPU and RAM in a single package.

Data rates of CAN bus depends on the length of the bus. For ISO11898 compliant devices, 1 Mbps is guaranteed for lengths up to 40 m. 500 m cable may be used if data rate is reduced to 125 Kbps.

An example of a small CAN controller chip is the MCP2510 by Microchip. It is a 18-pin standalone CAN controller featuring an industry standard SPI serial interface. On-board features include interrupt capability, message masking and filtering, message prioritisation, and multi-purpose I/O pins. Multiple transmit/receive buffers significantly offload the microcontroller overhead required to handle CAN message traffic. Applications for the MCP2510 include device control, sensor monitoring, meter interfacing, automotive electronics, and instrument control.

4.9 Inter-IC Serial Buses

Working with embedded systems, one often comes across several serial buses used between ICs or modules within a system. These are typically de facto standards originally developed by IC manufacturers for specific functions and subsequently expanded to more general use. Even though they are only used for short interconnections, they are getting increasing important to embedded systems designers as miniaturisation and simplicity in designs are being emphasised. For example, using a serial bus between a flash memory device and the microcontroller not only reduces the complexity in circuit design, but also brings down the overall cost as serial devices are typically cheaper because of the smaller package and the PCB would be significantly smaller. The only drawback in using a serial bus is of course transmission speed reduction. However, in many laboratory scale embedded systems, the data rates of ~100 Kbps to ~10 Mbps available in serial buses are more than sufficient.

4.10 I2C (Inter Integrated Circuit Bus)

This is a bidirectional two-wire serial bus introduced by Philips Semiconductor in 1980s for their television and other audiovisual products. Today it is widely used in the industry for embedded systems.

The two active lines are SDA (serial data) and SCL (serial clock). A device has an unique address (7 or 10 bits) and may be receiver-only, or transmitter/receiver. Each device may be either a master or slave depending on whether it initiates a data transfer. Multiple masters are allowed in a bus. Over the years, data transfer speed has been increased from 100 Kbps (standard) to 400 Kbps (fast) and 3.4 Mbps (high speed).

A bus master places the address of the destination (slave) on the bus. All devices on the bus listen and the one being address communicates with the master. If more than one master transmit, an arbitration scheme is used to decide the priority.

4.11 SPI (Serial Peripheral Interface)

This is a bidirectional full-duplex four-wire serial bus used originally by Motorola in their microcontrollers. The four wires are a clock, a data in, a data out and a select signal. The generic names of the SPI input/output (I/O) pins are:

- SS (slave select)
- SPSCK (SPI serial clock)
- MOSI (master out slave in)
- MISO (master in slave out)

In full duplex operation, the MISO pin of the master SPI module is connected to the MISO pin of the slave SPI module. The master SPI simultaneously receives data on its MISO pin and transmits data from its MOSI pin. Slave output data on the MISO pin is enabled only when the SPI is configured as a slave. To support a multiple-slave system, a logic 1 on the SS pin puts the MISO pin in a high-impedance state.

The MOSI pin of the master SPI module is connected to the MOSI pin of the slave SPI module. The master SPI simultaneously transmits data from its MOSI pin and receives data on its MISO pin.

The serial clock signal from master to slave synchronizes data transmission between the two. A byte of data is exchanged in 8 clock cycles.

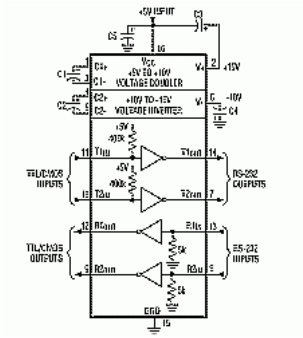
The SS is used to select a slave when a device is configured in slave mode. When an SPI is configured as a master, the SS input can be used in conjunction with a flag to prevent multiple masters from driving MOSI and SPSCK thus resolving contention.

4.12 SCI (Serial Communications Interface)

This is the Motorola implementation of the RS-232 asynchronous serial communications bus. It is a partial implementation in that only the two I/O data lines are implemented and that they are at TTL voltage level instead of the standard bipolar signals as in the RS-232 specifications.

For short distance connections, such as within an equipment, direct TTL levels are sufficient. If long transmission distance is necessary, or if the other end of the connection is a standard RS-232 port such as the PC COM port, voltage translation between TTL and RS-232 can be done.

Several manufacturers produce chips for this purpose. An example is the MAX232A by Maxim. It operates from a 5V power supply and generates typical voltages of +8 V and –8 V using charge pump voltage doubler and voltage inverter.



4.13 Microwire

This is a serial interface bus defined by National Semiconductor and used in many of their products including the COPS microprocessors. It is a three-wire (SO, SI and CK) interface very similar to the SPI. In fact many the SPI interface may be used to connect to Microwire memories and peripheral devices.

4.14 1-Wire Bus

This is a serial bus defined by Dallas Semiconductor. 1-Wire devices lower system cost and simplify design with an interface protocol that supplies control, signalling, and power over a single-wire connection. A variety of identification, sensor, control, and memory functions are available in conventional IC packages, and stainless-steel-clad casing called iButtons.

A 1-Wire network consists of a master and one or more slaves devices connected together through the 1-Wire interface. The master initiates and control half duplex data transfer on the bus. It uses conventional TTL voltage levels of maximum 0.8 V for logical 0 and minimum 2.2 V for logical 1. The master has a weak resistive pull-up open drain output. A slave shorts circuit the data line to change logical state to 0. For large network, the weak pull-up is supplemented by a controlled strong pull-up.

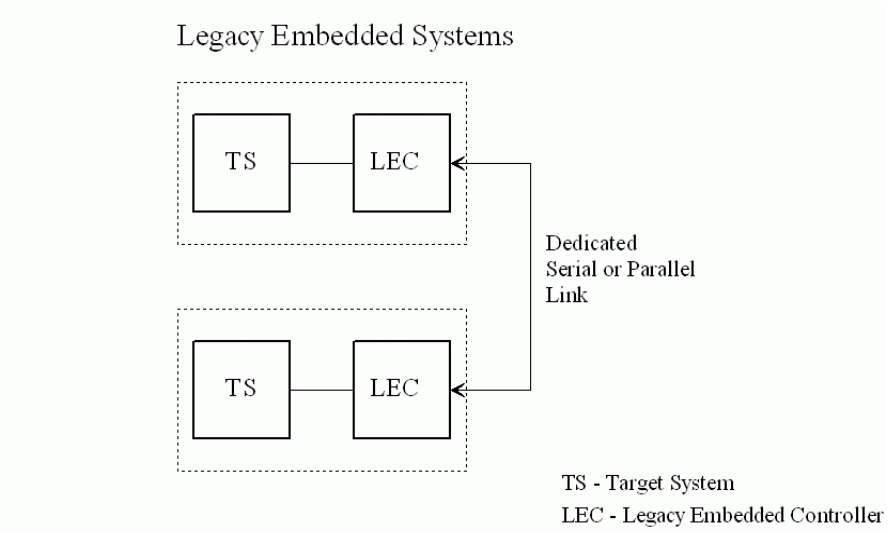
Every slave device has a unique 64-bit address, which consists of an 8-bit family code, a 48-bit serial number and an 8-bit CRC of the first seven bytes.

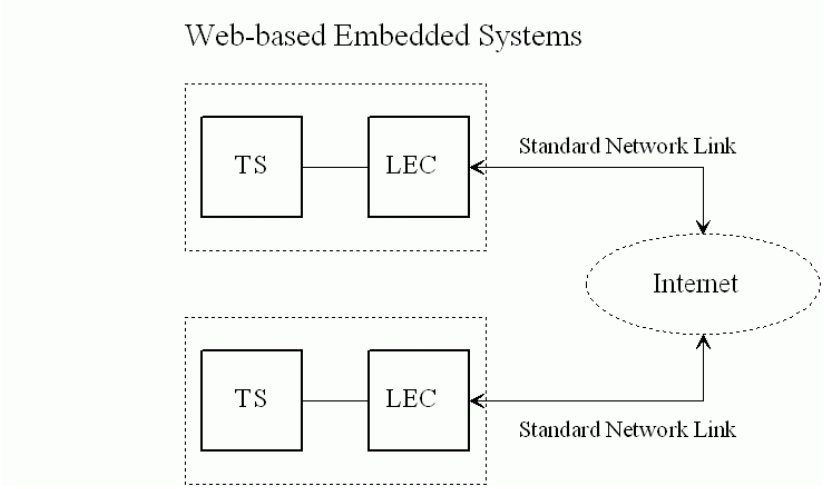
Slave devices must have its own timing circuit and this is synchronised by the falling slope of the signal on the bus. A slave typically obtains its power from the 1-Wire bus by means of an on-chip capacitor.

5 Web-based Embedded Systems

5.1 Introduction

As mentioned earlier, the proliferation of the Ethernet and Internet brings in another dimension in networking embedded systems. In a nutshell the suite of Internet protocols and the cost reduction in embedded Ethernet hardware change the way embedded systems are connected in a distributed environment.





It is now cheaper and simpler to connect an embedded system, even a small one such as a simple thermometer, to another equipment via the Internet (or its variation the Intranet) than to link the two together using traditional direct connection. When the equipment is within the range of a LAN (typically ~ 100 m), the Ethernet is the choice for physically connection. To cover large distances, the

Ethernet will typical have one of the several ways to connect to the public telecommunications network. Thus the embedded system is connected to the WAN (wide area network). The links to the telecommunications network can be modems, ISDN, DSL, or leased lines. These links are not the job of the embedded systems designers. The job of the designers boils down to adding the Ethernet connection to the equipment, or a serial link to the modem if Ethernet is not used.

The suite of Internet protocols (UDP and TCP/IP, and application protocols FTP, HTTP, SNMP etc.) are covered by other lectures in this workshop, we shall not delve into this topic in any depth. However, the Ethernet interface and the serial link that connect the embedded system to the Internet shall be discussed here.

Last few years saw the appearances of experimental, hobbyist and commercial real-time embedded systems on the Internet that can be accessed easily using standard browsers. However, one of the earlier implementations is the Cambridge coffee pot webcam that appeared in 1991 (predated world wide web) and subsequently put on the www. (http://www.cl.cam.ac.uk/coffee/coffee.html) Now there are numerous webcams on the Internet. At the height of its fame, there were more than 2 million people viewing the plain coffee pot in Trojan Room in Computer Laboratory in Cambridge.

While it is fascinating and significant to be able to view video frames from any place in the world with an Internet connection, the basic principle is straightforward and simple. A server gets data (video frames in this case) ready in a form meaningful to the client (the web browser) and the client accesses it via the Internet. It turns out that the server side effort is relatively simple (writing an HTML document, with SSI server side includes). The client (the browser) is a complex program but it is a standard application in all platforms. In fact, the earliest Cambridge coffee pot server and client implementation was done in one day, at that initial stage, for fun!

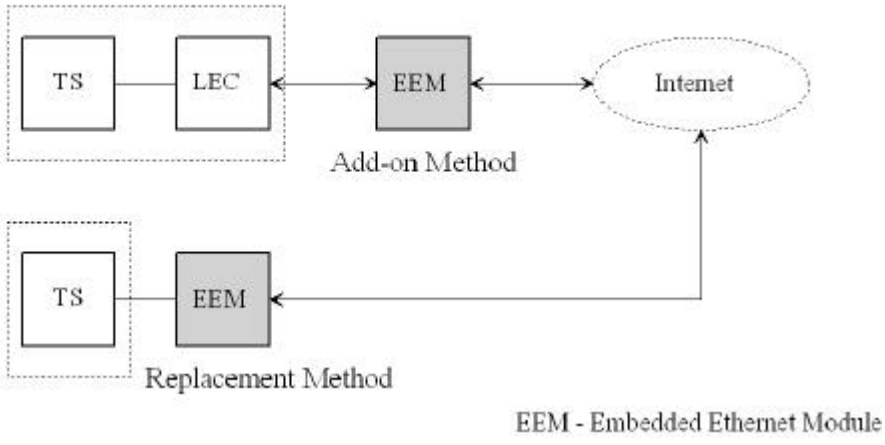
The more recent implementations of web-based that are of interest to the embedded community is in the implementation of small and lean embedded web servers. They are standalone servers that have their own IP (Internet address) and are connected to the Internet without using a PC. One of the tiniest is a match head sized web server using an 8-pin PIC microcontroller (PIC12C509A). (http://www-ccs.cs.umass.edu/~shri/iPic.html) It has TCP/IP stack and a HTTP web-server. The TCP/IP stack is fitted into 256 bytes. This project proves the point that putting up a web-based embedded system, albeit a rather basic one, can be done with very little hardware, and software!

More realistic embedded web servers are built with microcontrollers having more memories than that of the PIC12C509A, which has a mere 1536 bytes of ROM and 41 bytes of RAM. Almost every microcontroller manufacturer now has application notes on how to implement TCP/IP stack on their range of microcontrollers.

Vendors and suppliers of embedded web servers are mushrooming in the Internet because of the potential market they see. At the moment embedded Ethernet modules that have memories in the range of 256 Kbytes to 1 Mbytes, an Ethernet link, several serial links, and parallel I/Os are available in the price range of $50 ~ $100. For most laboratory applications when the volume is low, it is the most cost effective way for implementing embedded web-based systems.

5.2 Migration to Web-based Systems

The embedded systems designer faces a dilemma – whether to build the complete application on the new embedded web-based modules or to retain the existing or legacy systems. As in the case of other technological innovation, there are several approaches to handling the migration to the new technology. In embedded systems at this time, there are two ways as shown below.



The first decision that has to be made is whether to use the new embedded web-based microcontroller module as:

(1) An *add-on* to the legacy system or
(2) A *replacement* of the legacy electronics.

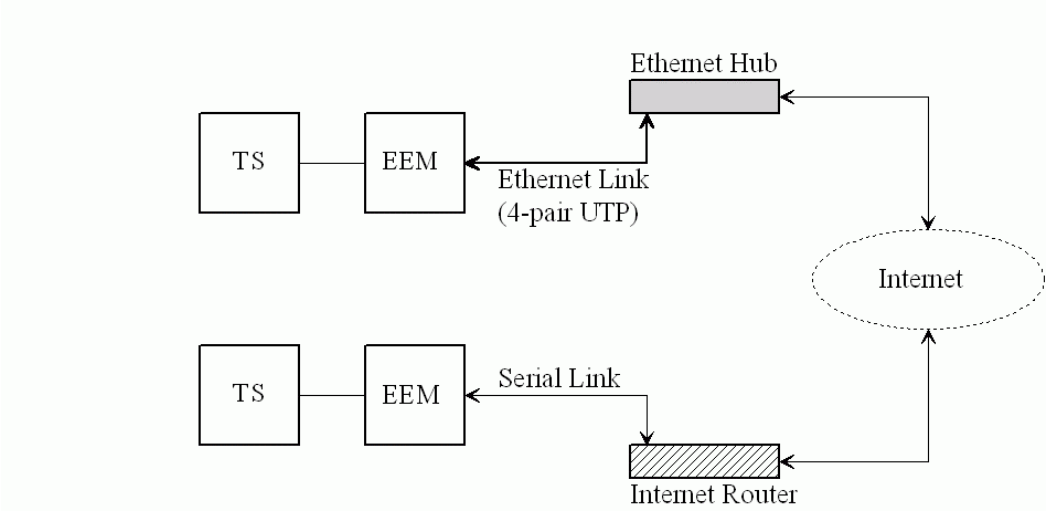
The add-on option has the advantage of little or no disruption to the existing system. In many cases where the existing systems have standard serial or parallel I/Os, no hardware or software modifications are needed. In others, appropriate modification to bring the existing system peripheral acceptable to the embedded Ethernet module has to be done. This is straightforward in many laboratory type systems. If the legacy system is a complex piece of equipment, this is also a good option to adopt.

The shortcoming of this add-on option is the lacking in functional enhancement of the final system other than the Internet connection. The Internet connection may be all that is needed. In which case it is obviously the simplest solution.

The replacement option is to replace the part or all the existing controller electronics with the embedded module. This is an attractive solution in some cases because new functions may be added to the existing system using the more powerful embedded microcontroller. Cost may be another factor. The new crops of embedded microcontroller modules are substantially cheaper than the older microprocessor-based systems. Thus if additional systems are needed, the old hardware may be expensive or obsolete. However, the effort and cost of rewriting or porting the software and firmware have to be weighed carefully against the gain achieved through hardware replacement.

5.3 Hardware Connection to the Internet

Irrespective of the ways - add-on or replacement - the new embedded web-based module functions in the legacy system, there are still two alternatives for connecting to the Internet world:



(1) Ethernet connection
(2) Serial connection

The Ethernet is now so widely deployed that there is hardly any laboratory or office that does not use it to network PCs or other equipment. Thus, for embedded systems, *Ethernet capable* feature would simplify connection to the Internet. A 10Base-T or 100Base-T connection from an embedded system will be most convenient; the connection to the outside world then reduces to linking the system to a hub or switch with a simple CAT-5 UTP (unshielded twisted pair) cable.

Until recently, the rather complex CSMA/CD physical layer protocol (MAC) and the large buffer size (~ 1500 bytes) required for the Ethernet frame meant that implementation was costly. Interface to the Ethernet controller chips was typically designed for 16- or 32-bit buses which were rather uncommon in small and medium scaled embedded systems. Many small embedded systems thus cannot justify having an Ethernet interface. The situation has changed in the last two years. Both cost reduction and simplification of driving circuitries make it possible to tug an Ethernet interface on an embedded system now. There are only two key components needed – an Ethernet controller chip and a line driver transformer. An 8-bit microcontroller can now be used to interface to the controller chip. The cost of the Ethernet controller and the line driver is in the range of $10 ~ $20.

While the Ethernet interface is the ubiquitous link to the Internet, it is not the only way. An embedded system can be connected to the outside world via a serial link like the RS-232 COM port on the PC. There are a number of situations where this method of connection is preferred. First, it is cheaper to use a serial port, which is a standard item in most microcontrollers. This const reduction refers to the embedded system side. It is noted that a matching serial port (in the form of Internet router) is needed on the network side. This may increase the overall system cost because an Internet router with serial ports has to be installed. Such Internet routers are not as common as the Ethernet hubs in most laboratories. Dispensing with the Ethernet controller chip reduces both the hardware and software complexity needed on the microcontroller host. In the tiny web server project mentioned earlier using a small PIC controller, there is simply not enough hardware resources on the microcontroller to drive an Ethernet controller.

Another reason for deploying a serial link is to connect to a modem for Internet access. This is a very common mode of connection. Most of us use a modem dial-up to access the Internet at home. Embedded systems to be used at locations without an Ethernet can use the serial mode in a similar manner.

In a very small implementation of a single embedded system connected to a PC acting as a host processor or controller, the serial link is also the choice. In this

case, the PC host's COM port is simply connected to the serial port of the embedded system using a crossover cable (null modem).

5.4 Ethernet Connection

While there are several ways of Ethernet connections and several high speed Ethernets (e.g. 10 Gbps) being introduced and standardized, the type that concerns the embedded systems designer is the plain 10Base-T, which is 10 Mbps, base-band, with twisted pair. This is the most widely used method of accessing the Ethernet in most laboratories. In a typically network, switches or hubs are used to link the various Ethernet nodes. An embedded system with a 10BaseT port only needs an Ethernet cable (CAT-5, UTP with modular RJ45 plugs at both ends, up to 100 m) to connect to an available port on a hub.

In the event that you have to build your own Ethernet for your embedded systems network in your laboratory, what are the hardware components involved? First you typically need a PC with an Ethernet connection. Many new PCs in the market have a built-in Ethernet port. For older PCs, a PCI Ethernet card (less than $20) can be used. Then you need a Ethernet hub which connects all nodes in your network. Hubs come in various forms. A small one with 4 to 8 ports ($20 ~ $50) would normally be sufficient. Then of course you need the cables with RJ45 terminations.

As in the case of serial link, it is possible to connect a host PC to an embedded system with just a crossover cable. No Ethernet hub is necessary in this case. This turns out to be a handy set-up for testing and debugging your embedded system. Before you deploy your embedded system in the Internet, you may want to test your system by just connecting it to a standalone host PC with a simple crossover cable.

5.5 Ethernet Controller

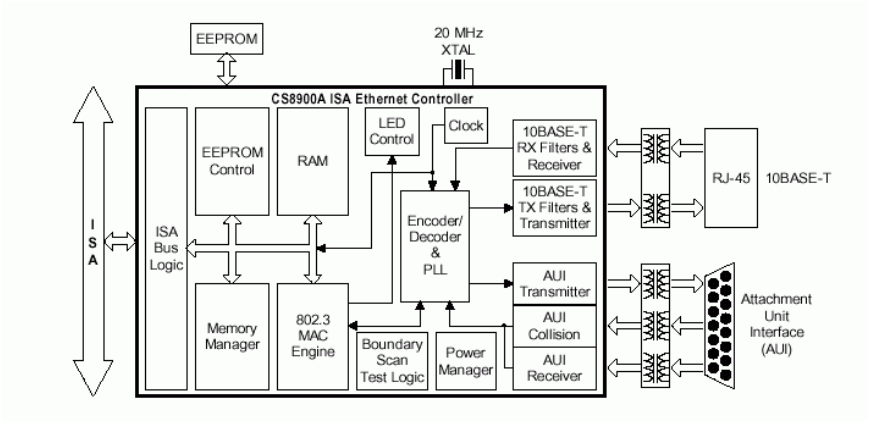
Most PC users are familiar with the NIC (network interface card) which provides the interface to the Ethernet. Two widely used cards are the 3Com 3C509 and the Novell NE2000. These cards, typically connected to either an ISA or PCI bus, use Ethernet controller chips which are rather sophisticated. The full functions of these Ethernet controller chips are often not required in embedded systems. Although the controller chips are rather complex, they nevertheless can be implemented in small 8-bit microcontroller systems with relative ease.

One of such controllers is the RTL8019AS produced by Realtek which is the most widely used with 70% world market share in 1999. Both controllers are simple to use and relatively low cost. Packaged in a 100-pin PQFP (plastic quad

flat pack) measuring 14 by 20 mm the RTL8019AS has the following main features:

- 16 Kbytes SRAM
- IEEE802.3 compliant
- Software compatible with NE2000
- PnP
- Full-duplex
- Power down modes
- Supports BROM
- Diagnostics LED outputs

Another similar controller is the CS8900A by Cirrus Logic. Originally designed as an ISA-bus Ethernet controller, it can be adapted for 8-bit microcontroller use. A complete Ethernet circuit can be designed on a PCB of about 10 cm^2. A block diagram of this chip is shown below:



The controller is accessed through either 8- or 16-bit ports. There are 8 registers which can be memory mapped into usual address space. Each register is 16 bits. For embedded system applications, these registered are accessed directly by the microcontroller.

Address Offset	I/O	Register
0h	R/W	Receive/Transmit Data (Port 0)
2h	R/W	Receive/Transmit Data (Port 1)
4h	W	TxCMD (Transmit Command)
6h	W	TxLength (Transmit Length)

8h	R	Interrupt Status Queue
Ah	R/W	PacketPage Pointer
Ch	R/W	PacketPage Data (Port 0)
Eh	R/W	PacketPage Data (Port 1)

The Ethernet frame transmission is carried out as follows:

(1) To transmit a frame, first write a transmit command (00C0h) to the TxCMD port and the length to the TxLength port. In the case of 8-bit connection, each is done in two steps, a low byte and then a high byte to the appropriate address. The BusTX register is then checked to see if the transmit buffer is available. This is done by setting the Packetpage Pointer with the correct value (in this case 0138h) and check the Packet-Page Data (Port 0) for the appropriate status, in this case bit 8 (Rdy4TxNow flag).

(2) If transmit buffer is available (Rday4TxNow flag set), data are written, one byte at a time, to the Receive/Transmit Data (Port 0). Again two bytes in two consecutive locations of the port.

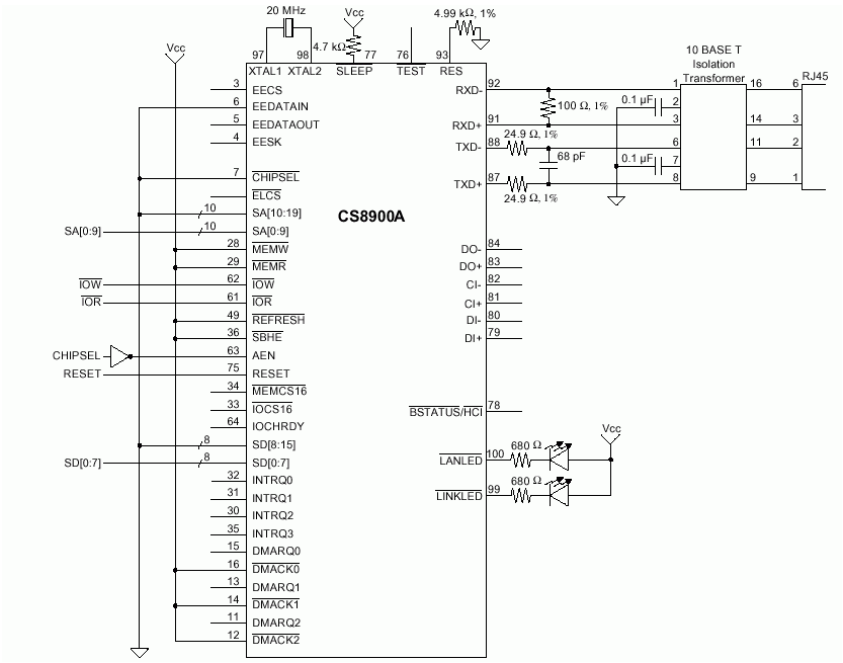
The steps for frame reception are:

(1) Poll Rx Event Register to determine if a frame is ready to be read. This is done by reading the RxStatus word from data port 0, high order byte first.

(2) The frame length is then read from data port 0, again high order byte first.

(3) The frame data is then read from data port 0, low order byte first. Repeat until the whole frame is read.

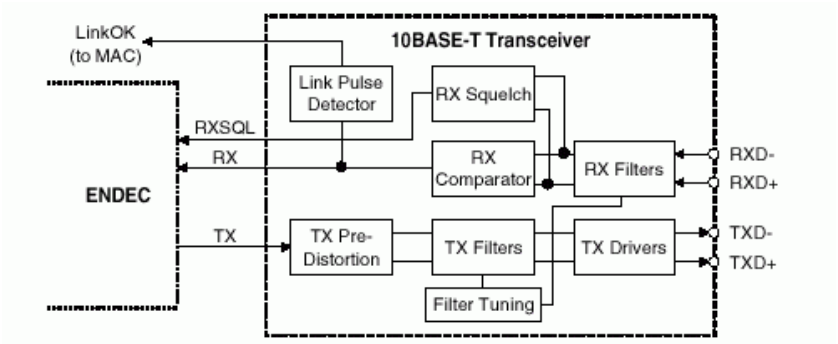
One can see the simplicity of interfacing the controller from the following reference design for an 8-bit microcontroller using the CS8900A.



Two LED outputs are usually provided by the Ethernet controller to monitor line status as shown in the diagram above. A LANLED is turned on (logical low) when the controller transmits or receives a frame, or when a collision is detected. It remains low until there has been no line activity for 6 ms.

A LINKLED is controller by either the controller or the host. In the former, this LED is turned on whenever there are valid 10Base-T pulses. In the latter case, this LED is turned on whenever a HCB0 bit (a bit in a SelfCTL register) is set.

Ethernet controllers such as the CS8900A have a built-in 10Base-T transceiver including analogue and digital circuitry needed to connect to a simple isolation transformer. A block diagram of the transceiver is as follows:



Fifth-order Butterworth low-pass filters are used for the receiver and transmitter. The nominal 3 dB cutoff frequency of the filters are 16 MHz and the attnuation at 30 MHz is –27 dB.

In 10Base-T transmitter, Manchester encoded data from the ENDEC (encoder decoder) pass through a predistortion circuit for wave shaping and preequalization. The signal is then filtered before being fed to differential drivers and finally brought out to the TxD+ and TxD- pins. Link pulses are sent in the absence of transmit packets. These are positive pulses, one bit time wide, generated every 16 ms.

In the receiver, a squelch circuit determines if the incoming filtered signal is valid (reaches the threshold). The receiver pair RxD+ and RxD- is monitored continuously. If a packet or link pulse is not received within a time limit (150 ms), transmission of packet is disabled. The received signals are also checked for polarity. In the case of polarity reversal, it is possible to set the controller to correct for the reversal.