Unit -1

Introduction to Embedded System

EMBEDDED SYSTEM:

An embedded system is a microcontroller or microprocessor based system which is designed to perform a specific task. For example, a fire alarm is an embedded system; it will sense only smoke.

An embedded system has three components –

- It has hardware.
- It has application software.
- It has Real Time Operating system (RTOS) that supervises the application software and provide mechanism to let the processor run.

INTELLIGENT SYSTEM:

Intelligent System (IS) can be defined as the system that incorporates intelligence into applications being handled by machines. Intelligent systems perform search and optimization along with learning capabilities.

Intelligent systems are technologically advanced machines that perceive and respond to the world around them. Intelligent systems can take many forms, from automated vacuums such as the Roomba to facial recognition programs

EXPERT SYSTEM:

The expert systems are the computer applications developed to solve complex problems in a particular domain, at the level of extra-ordinary human intelligence and expertise. Examples – Flight-tracking systems, Clinical systems.

Characteristics of Expert Systems

- High performance
- Understandable
- Reliable
- Highly responsive

Capabilities of Expert Systems

The expert systems are capable of −

- Advising
- Instructing and assisting human in decision making
- Demonstrating
- Deriving a solution
- Diagnosing
- Explaining
- Interpreting input
- Predicting results
- Justifying the conclusion
- Suggesting alternative options to a problem

Artificial Intelligence	Expert System
AI is the ability of a machine or a computer program to think, work, learn and react like humans.	Expert systems represent the most successful demonstration of the capabilities of AI.
AI involves the use of methods based on the intelligent behaviour of humans to solve complex problems.	Experts systems are computer programs designed to solve complex decision problems.

ARCHITECTURE

1.SOFTWARE ARCHITECTURE

An embedded software architecture is a piece of software that is divided in multiple layers. The important layers in embedded software are

- Application layer
- Middleware layer
- Firmware layer

Application layer is mostly written in high level languages like java, C++, C# with rich GUI support. Application layer calls the middleware api in response to action by the user or an event.

The Middleware layer is mostly written in C++, C with no rich GUI support. The middleware software maintains the state machine of the device. And is responsible to handle requests from the upper layer and the lower level layer. The middleware exposes a set of api functions which the application must call in order to use the

services offered by the middleware. And vice versa the middleware can send data to the application layer via IPC mechanism.

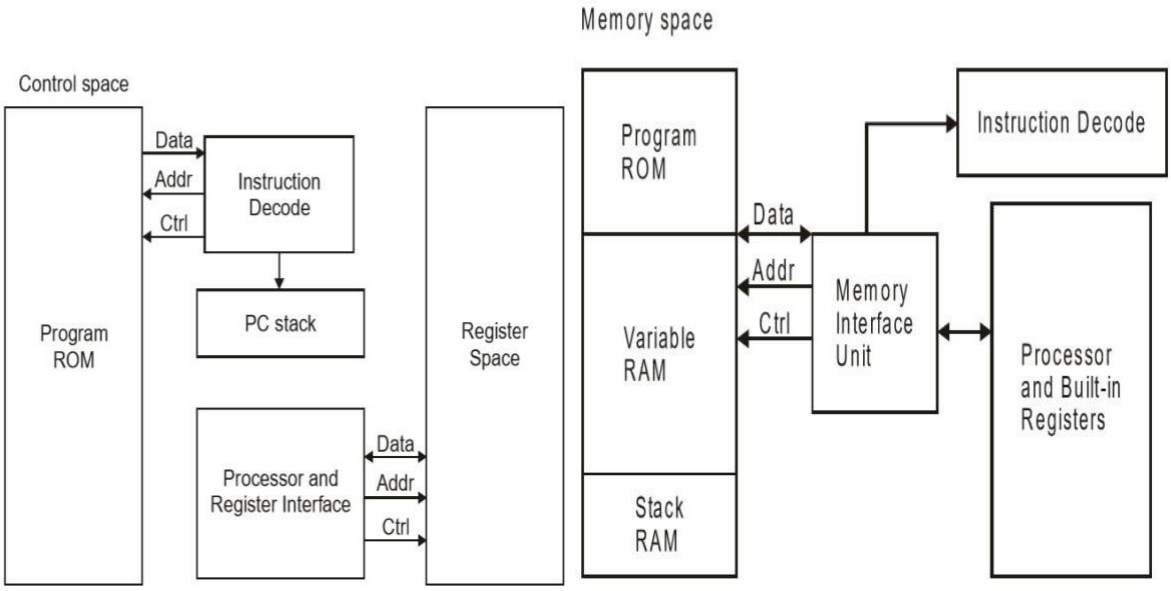
The Firmware layer is always written in C. The firmware is responsible for talking to the chipset either configuring registers or reading from the chipset registers. The firmware exposes a set of api's that the middleware can call. In order to perform specific tasks.

Embedded software is a combination of all the 3 layers mentioned above. It is created to perform some tasks or to behave in a predefined way.

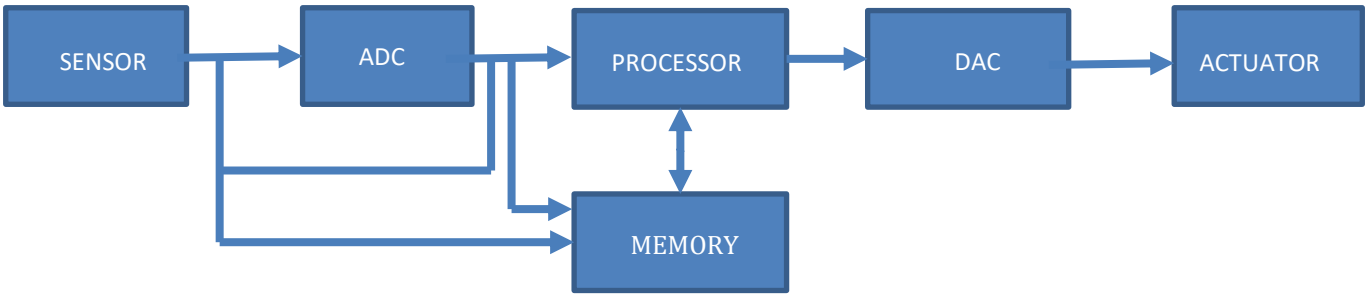
2.HARDWARE ARCHITECTURE

Embedded hardware are based around microprocessors and microcontrollers, also include memory, bus, Input/Output, Controller, where as embedded software includes embedded operating systems, different applications and device drivers. Basically these two types of architecture i.e., Harvard architecture and Von Neumann architecture are used in embedded systems.

Von-Neumann Architecture	Harvard Architecture
Single memory to be shared by both code and data.	Separate memories for code and data.
Processor needs to fetch code in a separate clock cycle and data in another clock cycle. So it requires two clock cycles.	Single clock cycle is sufficient, as separate buses are used to access code and data.
Higher speed, thus less time consuming.	Slower in speed, thus more time-consuming.
Simple in design.	Complex in design.



Architecture of the Embedded System includes Sensor, Analog to Digital Converter, Memory, Processor, Digital to Analog Converter, and Actuators etc.



CPU: Programming input and output:

The basic techniques for I/O programming can be understood relatively independent of the instruction set. In this section, we cover the basics of I/O programming and place them in the contexts of both the ARM and C55x.

We begin by discussing the basic characteristics of I/O devices so that we can understand the requirements they place on programs that communicate with them.

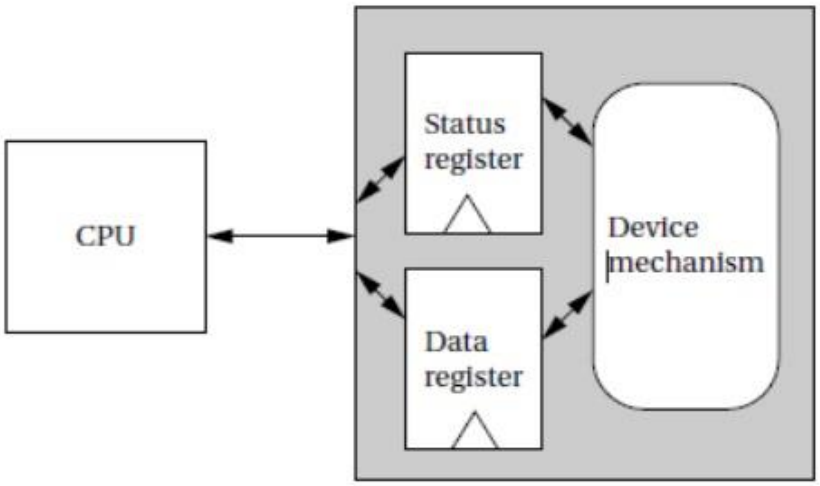


Fig 1.18 Structure of a typical I/O device.

1. Input and Output Devices:

Input and output devices usually have some analog or non electronic component for instance, a disk drive has a rotating disk and analog read/write electronics. But the digital logic in the device that is most closely connected to the CPU very strongly resembles the logic you would expect in any computer system.

Figure 1.18 shows the structure of a typical I/O device and its relationship to the CPU. The interface between the CPU and the device's internals (e.g., the rotating disk and read/write electronics in a disk drive) is a set of registers. The CPU talks to the device by reading and writing the registers.

Devices typically have several registers:

1. *Data registers* hold values that are treated as data by the device, such as the data read or written by a disk.
2. *Status registers* provide information about the device's operation, such as whether the current transaction has completed.
3. Some registers may be read-only, such as a status register that indicates when the device is done, while others may be readable or writable.

2. Input and Output Primitives:

Microprocessors can provide programming support for input and output in two ways: ***I/O instructions*** and ***memory-mapped I/O***.

Some architectures, such as the Intel x86, provide special instructions (in and out in the case of the Intel x86) for input and output. These instructions provide a separate address space for I/O devices.

But the most common way to implement I/O is by memory mapping even CPUs that provide I/O instructions can also implement memory-mapped I/O.

As the name implies, memory-mapped I/O provides addresses for the registers in each I/O device. Programs use the CPU's normal read and write instructions to communicate with the devices.

3. Busy-Wait I/O:

The most basic way to use devices in a program is *busy-wait I/O*. Devices are typically slower than the CPU and may require many cycles to complete an operation. If the CPU is performing multiple operations on a single device, such as writing several characters to an output device, then it must wait for one operation to complete before starting the next one. (If we try to start writing the second character before the device has finished with the first one, for example, the device will probably never print the first character.) Asking an I/O device whether it is finished by reading its status register is often called polling.

SUPERVISOR MODE, EXCEPTIONS, AND TRAPS:

These are mechanisms to handle internal conditions, and they are very similar to interrupts in form. We begin with a discussion of supervisor mode, which some processors use to handle exceptional events and protect executing programs from each other.

1. Supervisor Mode:

As will become clearer in later chapters, complex systems are often implemented as several programs that communicate with each other. These programs may run under the command of an operating system. It may be desirable to provide hardware checks to ensure that the programs do not

interfere with each other—for example, by erroneously writing into a segment of memory used by another program. Software debugging is important but can leave some problems in a running system; hardware checks ensure an additional level of safety.

In such cases it is often useful to have a ***supervisor mode*** provided by the CPU. Normal programs run in ***user mode***. The supervisor mode has privileges that user modes do not. Control of the memory management unit (MMU) is typically reserved for supervisor mode to avoid the obvious problems that could occur when program bugs cause inadvertent changes in the memory management registers.

Not all CPUs have supervisor modes. Many DSPs, including the C55x, do not provide supervisor modes. The ARM, however, does have such a mode. The ARM instruction that puts the CPU in supervisor mode is called SWI:

SWI CODE_1

It can, of course, be executed conditionally, as with any ARM instruction. SWI causes the CPU to go into supervisor mode and sets the PC to 0x08.The argument to SWI is a 24-bit immediate value that is passed on to the supervisor mode code; it allows the program to request various services from the supervisor mode.

In supervisor mode, the bottom 5 bits of the CPSR are all set to 1 to indicate that the CPU is in supervisor mode. The old value of the CPSR just before the SWI is stored in a register called the ***saved program status register (SPSR)***. There are in fact several SPSRs for different modes; the supervisor mode SPSR is referred to as SPSR_svc.

To return from supervisor mode, the supervisor restores the PC from register r14 and restores the CPSR from the SPSR_svc.

2. Exceptions:

An ***exception*** is an internally detected error. A simple example is division by zero. One way to handle this problem would be to check every divisor before division to be sure it is not zero, but this would both substantially increase the size of numerical programs and cost a great deal of CPU time evaluating the divisor’s value.

The CPU can more efficiently check the divisor's value during execution. Since the time at which a zero divisor will be found is not known in advance, this event is similar to an interrupt except that it is generated inside the CPU. The exception mechanism provides a way for the program to react to such unexpected events.

Just as interrupts can be seen as an extension of the subroutine mechanism, exceptions are generally implemented as a variation of an interrupt. Since both deal with changes in the flow of control of a program, it makes sense to use similar mechanisms. However, exceptions are generated internally.

Exceptions in general require both prioritization and vectoring. Exceptions must be prioritized because a single operation may generate more than one exception for example, an illegal operand and an illegal memory access.

The priority of exceptions is usually fixed by the CPU architecture. Vectoring provides a way for the user to specify the handler for the exception condition.

The vector number for an exception is usually predefined by the architecture; it is used to index into a table of exception handlers.

3. Traps:

A trap, also known as a software interrupt, is an instruction that explicitly generates an exception condition. The most common use of a trap is to enter supervisor mode.

The entry into supervisor mode must be controlled to maintain security—if the interface between user and supervisor mode is improperly designed, a user program may be able to sneak code into the supervisor mode that could be executed to perform harmful operations.

The ARM provides the SWI interrupt for software interrupts. This instruction causes the CPU to enter supervisor mode. An opcode is embedded in the instruction that can be read by the handler.

The process of converting the source code representation of your embedded software into an executable binary image involves three distinct steps:

1. Each of the source files must be compiled or assembled into an object file.

2. All of the object files that result from the first step must be linked together to produce a single object file, called the relocatable program.

3. Physical memory addresses must be assigned to the relative offsets within the relocatable program in a process called relocation.

The result of the final step is a file containing an executable binary image that is ready to run on the embedded system.

BOOTLOADER:

Bootloader is the first piece of firmware which gets executed once the Embedded System is turned-on/reset. The primary objective of the Bootloader is to initialise the Embedded System and provide control to the Application/RTOS. The other objective of the Bootloader is also to support the data loading feature.

Working of bootloader:

After a computer is turned on, information about the installed hardware comes up on the screen. The bootloader places its operating system into the memory. The basic input/output system (BIOS) carries out tests before transferring control to the Master Boot Record (MBR), which contains the boot loader.

A lot of bootloaders are configured to give users different booting options. The options include different operating systems, different versions of the same operating system, operating system loading options, and programs that run without an operating system.

In certain cases, a device may have two operating systems. Bootloaders can be used on these devices to start the correct operating system that users prefer automatically. A bootloader can also be used to boot the operating system into safe mode for recovery.

You can use a bootloader to boot into a program without having to start the operating system. This can be useful with devices such as game consoles. After the game disc is inserted into the console and the console is turned on, the user is taken straight to the game instead of the welcome screen.

Example: Arduino UNO Bootloader
A good example of a basic bootloader is the bootloader programmed into the Arduino UNO.

On power-up or reset, the bootloader watches the serial receive pin for a few seconds - waiting to receive the special sequence of bytes that indicates an upload attempt from the IDE. If that magic byte sequence is received, the bootloader will respond with an acknowledgement and the IDE will start sending the code. The bootloader will receive the code from the IDE and burn it into flash memory.

If the bootloader does not see the magic byte sequence, after a few seconds it will proceed to start running whatever code is already burned into flash.

DEVICE DRIVER

A device driver is a special kind of software program that controls a specific hardware device attached to a computer. Device drivers are essential for a computer to work properly. These programs may be compact, but they provide the all-important means for a computer to interact with hardware, for everything from mouse, keyboard and display (user input/output) to working with networks, storage and graphics.

How Do Device Drivers Work?

Device drivers generally run at a high level of privilege within the operating system runtime environment. Some device drivers, in fact, may be linked directly to the operating system kernel, a portion of an OS such as Windows, Linux or Mac OS, that remains memory resident and handles execution for all other code, including device drivers. Device drivers relay requests for device access and actions from the operating system and its active applications to their respective hardware devices. They also deliver outputs or status/messages from the hardware devices to the operating system (and thence, to applications).

Purpose of Device Drivers

Device drivers are necessary to permit a computer to interface and interact with specific devices. They define the messages and mechanisms whereby the computer (OS and applications) can access the device or make requests for the device to fulfill. They also handle device responses and messages for delivery to the computer.

Device tree :

The device tree is a set of text files in the Linux kernel source tree that describe the hardware of a certain platform. They are located at arch/arm/boot/dts/ and can have two extensions:

- *.dtsi files are device tree source include files. They describe hardware that is common to several platforms which include these files on their *.dts files.
- *.dts files are device tree source files. They describe one specific platform.