Microcontroller Basics

2018

Contents

1 Embedded Systems - Overview

1.1 System

A system is an arrangement in which all its unit assemble work together according to a set of rules. It can also be defined as a way of working, organizing or doing one or many tasks according to a fixed plan. For example, a watch is a time displaying system. Its components follow a set of rules to show time. If one of its parts fails, the watch will stop working. So we can say, in a system, all its subcomponents depend on each other.

1.2 Embedded System

As its name suggests, Embedded means something that is attached to another thing. An embedded system can be thought of as a computer hardware system having software embedded in it. An embedded system can be an independent system or it can be a part of a large 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.

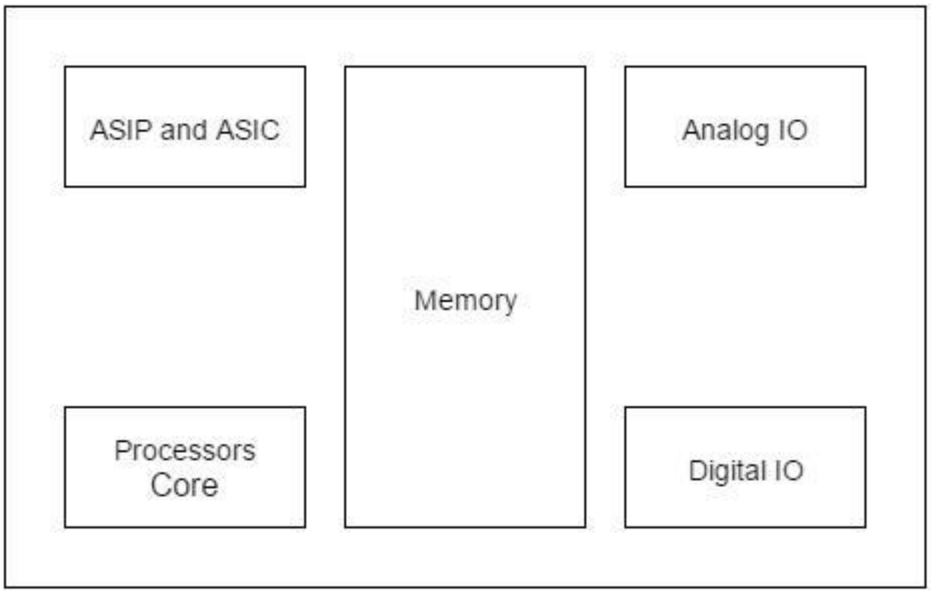
An embedded system has three components −

- 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 a process as per scheduling by following a plan to control the latencies. RTOS defines the way the system works. It sets the rules during the execution of application program. A small scale embedded system may not have RTOS.

So we can define an embedded system as a Microcontroller based, software driven, reliable, real-time control system.

1.3 Characteristics of an Embedded System

- Single-functioned − An embedded system usually performs a specialized operation and does the same repeatedly. For example: A pager always functions as a pager.
- Tightly constrained − All computing systems have constraints on design metrics, but those on an embedded system can be especially tight. Design metrics is a measure of an implementation's features such as its cost, size, power, and performance. It must be of a size to fit on a single chip, must perform fast enough to process data in real time and consume minimum power to extend battery life.
- Reactive and Real time − Many embedded systems must continually react to changes in the system's environment and must compute certain results in real time without any delay. Consider an example of a car cruise controller; it continually monitors and reacts to speed and brake sensors. It must compute acceleration or de-accelerations repeatedly within a limited time; a delayed computation can result in failure to control of the car.
- Microprocessors based − It must be microprocessor or microcontroller based.
- Memory − It must have a memory, as its software usually embeds in ROM. It does not need any secondary memories in the computer.
- Connected − It must have connected peripherals to connect input and output devices.
- HW-SW systems − Software is used for more features and flexibility. Hardware is used for performance and security.



1.1.1 Advantages

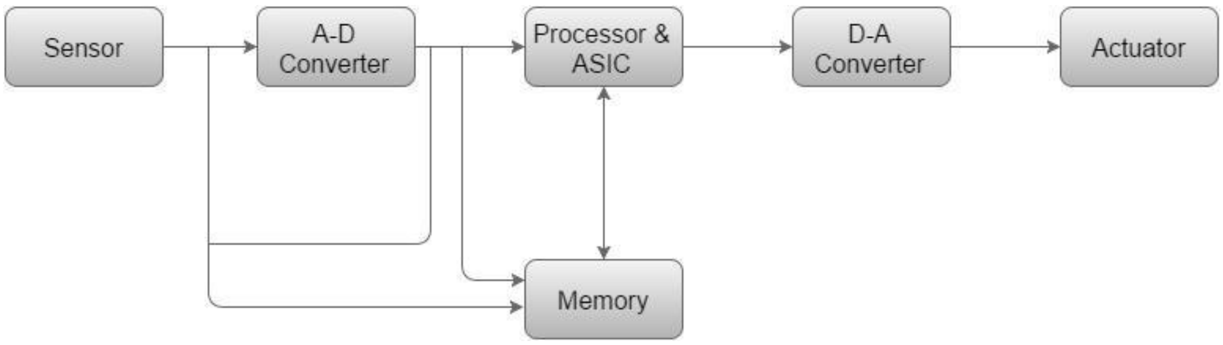
- Easily Customizable
- Low power consumption
- Low cost
- Enhanced performance

1.1.2 Disadvantages

- High development effort
- Larger time to market

1.4 Basic Structure of an Embedded System

The following illustration shows the basic structure of an embedded system −



- Sensor − It measures the physical quantity and converts it to an electrical signal which can be read by an observer or by any electronic instrument like an A2D converter. A sensor stores the measured quantity to the memory.
- A-D Converter − An analog-to-digital converter converts the analog signal sent by the sensor into a digital signal.
- Processor & ASICs − Processors process the data to measure the output and store it to the memory.
- D-A Converter − A digital-to-analog converter converts the digital data fed by the processor to analog data
- Actuator − An actuator compares the output given by the D-A Converter to the actual (expected) output stored in it and stores the approved output.

1.5 Embedded Systems - Processors

Processor is the heart of an embedded system. It is the basic unit that takes inputs and produces an output after processing the data. For an embedded system designer, it is necessary to have the knowledge of both microprocessors and microcontrollers.

1.6 Processors in a System

A processor has two essential units:

- Program Flow Control Unit (CU)
- Execution Unit (EU)

The CU includes a fetch unit for fetching instructions from the memory. The EU has circuits that implement the instructions pertaining to data transfer operation and data conversion from one form to another.

The EU includes the Arithmetic and Logical Unit (ALU) and also the circuits that execute instructions for a program control task such as interrupt, or jump to another set of instructions.

A processor runs the cycles of fetch and executes the instructions in the same sequence as they are fetched from memory.

1.7 Types of Processors

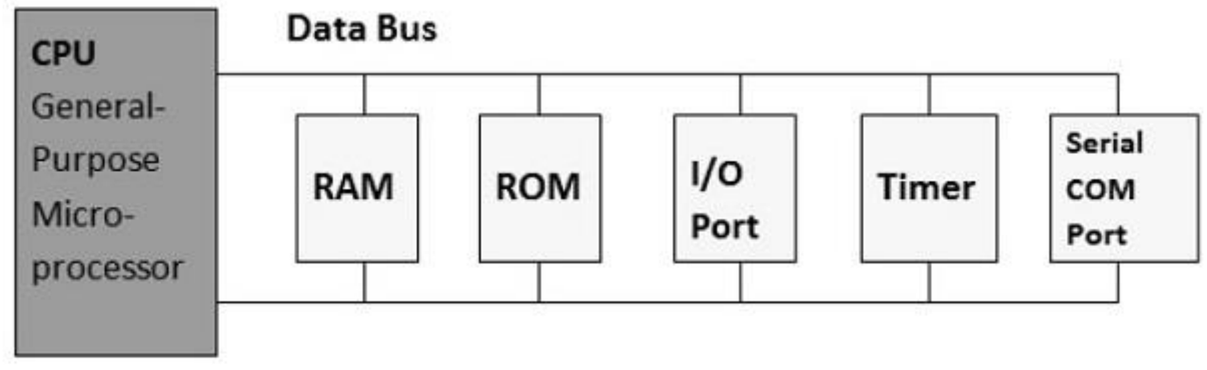
Processors can be of the following categories −

- General Purpose Processor (GPP)
  - Microprocessor
  - Microcontroller
  - Embedded Processor
  - Digital Signal Processor
  - Media Processor
- Application Specific System Processor (ASSP)
- Application Specific Instruction Processors (ASIPs)
- GPP core(s) or ASIP core(s) on either an Application Specific Integrated Circuit (ASIC) or a Very Large Scale Integration (VLSI) circuit.

1.8 Microprocessor

A microprocessor is a single VLSI chip having a CPU. In addition, it may also have other units such as coaches, floating point processing arithmetic unit, and pipelining units that help in faster processing of instructions.

Earlier generation microprocessors' fetch-and-execute cycle was guided by a clock frequency of order of ~1 MHz. Processors now operate at a clock frequency of 2GHz



A SIMPLE BLOCK DIAGRAM OF A MICROPROCESSOR

1.9 Microcontroller

A microcontroller is a single-chip VLSI unit (also called microcomputer) which, although having limited computational capabilities, possesses enhanced input/output capability and a number of on-chip functional units.

CPU	RAM	ROM
I/O Port	Timer	Serial COM Port

Microcontrollers are particularly used in embedded systems for real-time control applications with on-chip program memory and devices.

1.10 Microprocessor vs Microcontroller

Let us now take a look at the most notable differences between a microprocessor and a microcontroller.

Microprocessor	Microcontroller
Microprocessors are multitasking in nature. Can perform multiple tasks at a time. For example, on computer we can play music while writing text in text editor.	Single task oriented. For example, a washing machine is designed for washing clothes only.
RAM, ROM, I/O Ports, and Timers can be added externally and can vary in numbers.	RAM, ROM, I/O Ports, and Timers cannot be added externally. These components are to be embedded together on a chip and are fixed in numbers.

Designers can decide the number of memory or I/O ports needed.	Fixed number for memory or I/O makes a microcontroller ideal for a limited but specific task.
External support of external memory and I/O ports makes a microprocessor-based system heavier and costlier.	Microcontrollers are lightweight and cheaper than a microprocessor.
External devices require more space and their power consumption is higher.	A microcontroller-based system consumes less power and takes less space.

1.11 Infinite Loop

An infinite loop or an endless loop can be identified as a sequence of instructions in a computer program that executes endlessly in a loop, because of the following reasons −

- loop with no terminating condition.
- loop with a terminating condition that can never be met.
- loop with a terminating condition that causes the loop to start over.

Such infinite loops normally caused older operating systems to become unresponsive, as an infinite loop consumes all the available processor time. I/O operations waiting for user inputs are also called "infinite loops". One possible cause of a computer "freezing" is an infinite loop; other causes include deadlock and access violations.

Embedded systems, unlike a PC, never "exit" an application. They idle through an Infinite Loop waiting for an event to take place in the form of an interrupt, or a pre-scheduled task. In order to save power, some processors enter special sleep or wait modes instead of idling through an Infinite Loop, but they will come out of this mode upon either a timer or an External Interrupt.

1.12 Little Endian Vs Big Endian

Although numbers are always displayed in the same way, they are not stored in the same way in memory. Big-Endian machines store the most significant byte of data in the lowest memory address. A Big-Endian machine stores 0x12345678 as −

```
ADD+0: 0x12
ADD+1: 0x34
ADD+2: 0x56
ADD+3: 0x78
```

Little-Endian machines, on the other hand, store the least significant byte of data in the lowest memory address. A Little-Endian machine stores 0x12345678 as −

```
ADD+0: 0x78
ADD+1: 0x56
ADD+2: 0x34
ADD+3: 0x12
```

1.13 Questions

1. What is the difference between a microcontroller and a microprocessor?
2. State the microcontroller peripherals.
3. What is an infinite loop?
4. What is the difference between the Big Endian and Little Endian?
5. Draw a basic structure of an embedded system that reads a voltage from a power source, stores the data in memory and at the end is has connected an audio speaker.
6. What is an embedded system?

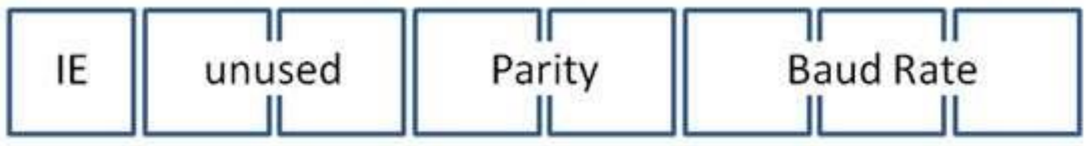
DAY-1

2. Device registers

Registers are used in the CPU to store information on temporarily basis which could be data to be processed, or an address pointing to the data which is to be fetched.
The broad issue is quite straightforward. A peripheral device is likely to have a number of internal registers, which may be read from or written to by software. These normally appear just like memory locations and can, for the most part, be treated in the same way. Typically a device register will have bit fields – groups of bits that contain or receive specific information. Such fields may be single bits, groups of bits, or a whole word. There may also be bits that are unused - reading from them or writing to them normally has no effect.

For example, a serial interface (UART) might have an 8-bit register, with the bits used like this:

- Bits 0-2: baud rate, where the values [0-7] have specific significance.
- Bits 3-4: parity, where 0=none, 1=even, 2=odd and 3 is an illegal setting.
- Bits 5-6: unused.
- Bit 7: interrupt enable.



Among the register there are roughly two groups of Registers. One is mostly used for storing (or assigning) a specific values that are used by ALU (Arithmatic Logic Unit). These registers are called General Purpose Registers. The other type of the registers is mostly used to control the various I/O pins or to read / write to those I/O pins. The one related to I/O pins are generally called I/O registers or Special Function Registers(SFR). Every CPUs and MCUs have various Registers in it but we don't usually care much of these registers when we are working on high level application programming. However, if you are an embedded system engineer working with Microcontrollers, you should be very familiar with the details of those Registers for the specific MCU (Microcontroller) that is used on your embedded board.
Then you may ask 'where/how I can get the register information about the MCU / CPU that is on my embedded Board ?'. The first document you need to see is the data sheet of the chipset.

2.1 Memory addressing

There are a few important matters to get right when accessing device registers from C. The first is data type. It is almost 100% certain that a register will be unsigned and its size (width in bits) must be accommodated. Using traditional C data types, our example register would be declared unsigned char. In newer versions of the language, there are ways to explicitly specify the bit width; in this case uint8_twould do the job.

If the devices registers are larger than 8 bits – perhaps 16 bits or 32 bits – endianity may be an issue; different CPUs locate most and least significant bytes differently. Any attempt to break down such registers into 8-bit units is likely to go awry, for example.

A really important issue to address is the suppression of unnecessary compiler optimizations.

Normally, a compiler will detect multiple accesses to a variable and make the code more efficient by accessing the memory location once, working on the data in a CPU register, and then writing it back to memory later. With a device register, it is essential that the actual device is updated exactly when the code when requires it to be. Similarly, repeated read accesses must be honored. For example, there may be some code that looks like this:

```
 unsigned char dev_reg;

while ((dev_reg & 1) == 0)
        wait();
```

The idea here is to continuously poll the variable dev_reg, which is actually a device register, waiting for the least significant bit to be set. Unless dev_reg is declared volatile, most compilers would optimize the code to a single read access and the loop would never end.

2.1.1 Using pointers

A common question from developers, working this close to the hardware for the first time, goes something like this: “I have a variable of the right type for my device register. How do I arrange for it to be mapped to the correct address?” This is a simple question, but the answer is less simple.

Some embedded software development toolkits do make it fairly simple by providing a facility whereby a variable (or a number of variables – a program section) can be precisely located using the linker. Although this is very neat and tidy, it renders the code somewhat toolchain-dependent, which is generally unwise. There is no standard way to achieve this result in the C language.

The usual solution is to think in terms of pointers, which are effectively addresses. So, instead of creating a variable of the appropriate type, you need a pointer to that type. Then, accessing the device register is simply a matter of dereferencing the pointer. So the above example may be re-written.

```
unsigned *char dev_reg;

while ((*dev_reg & 1) == 0)
  wait();
```

There is just the question of making sure that the pointer does point to the device register. So, for example, maybe the device register in the example is located at 0x8000000. The declaration of the pointer could include an initialization, thus:

```
unsigned *char dev_reg = (unsigned char *)0x80000000;
```

Notice that the address is expressed as an integer, which is type cast to be an appropriate pointer type.
There are two drawbacks of this approach:

- The use of pointers is fraught with potential errors.
- An additional variable – the pointer – is created, which uses up valuable RAM.

Both of these issues can be addressed by creating a pointer constant and dereferencing it. The resulting horrible syntax can be hidden in a macro, thus:

```
#define DEV_REG (*(unsigned char *)0x80000000)
```

Now, DEV_REG can be used anywhere that a normal variable is valid. So our example code becomes:

```
while ((DEV_REG & 1) == 0)
  wait();
```

2.1.2 Using bit fields

Since device registers usually contain fields of one or more bits, each of which corresponds to specific functionality (as shown in the example above), it would seem logical to use bit fields in a C structure, thus:

```
struct uart
```

```
{
   unsigned baud : 3;
   unsigned parity : 2;
   unsigned unused : 2;
   unsigned interrupt_enable : 1;
};
```

This is quite neat and enables clear code to be written like this:

```
struct uart myuart;

void main()
{
   myuart.baud = 2;
}
```

The good news is that this code might work just fine. But the bad news is that in might not. The method by which allocation of bit fields in a word is performed is compiler dependent. So, one compiler may produce exactly the result you expect, but another may not. It is even possible that a given compiler might produce different results depending on the optimization settings.

As compiler dependent code should be avoided, the simple rule is to avoid using bit fields for this purpose.

2.2 Representing Hardware Device Registers

A given hardware device often communicates through a sequence of device registers. Some registers communicate control and status information; others communicate data. A given device may use separate registers for input and output. Registers may occupy bytes, or words, or whatever the architecture demands. The simplest representation for a device register is as an object of the appropriately sized and signed integer type. For example, you might declare a one-byte register as a char or a two-byte register as a short int. Then you can receive data from a device by reading a value from its input data register, and send data to a memory-mapped device by storing a value into its output data register. A typical control/status register is really not an integer-valued object – it's a collection of bits. One bit may indicate that the device is ready to perform an operation, while another bit might indicate whether interrupts have been enabled for that device. A given device might not use all the available bits in its control/status register. Using the bitmask approach to manipulate control/status registers, you: • use bitwise operators to set, clear and test bits in the registers • use symbolic constants to represent masks for

isolating bits For example, here’s the bitmask representation for a simple 16-bit control/status register: typedef short int control; #define ENABLE 0x0040 /* bit 6: enable interrupts */ #define READY 0x0080 /* bit 7: device is ready */
For example, a program can define: control *const pcr = (control *)0xFF70;
In C, you can make a control register look more like an object by using a macro as follows: #define cr (*pcr) Then you can manipulate the device register by writing expressions such as: cr &= ~ENABLE;
Many devices use a control/status and a data register in tandem. The declarations for such a pair might look like: typedef short int control; typedef short int data; typedef struct port port; struct port { control cr; data dr; }; port *const p = (port *)0xFF70; The typedef: typedef struct port port; defines port as a type name that’s equivalent to struct port. This allows you to refer to the structure type port without also writing the keyword struct in front of it.

2.3 Representing Hardware Registers as Bitfields

Bitfields offer a slightly more elegant approach to modeling hardware registers. Using this approach, you:
• declare a register as a struct and its bits as bitfield members, and
• set, clear and test bits in registers by referring to each bitfield by name.
For example,

```
struct control
    {
    unsigned int /* unused */ : 6;
    unsigned int enable : 1;
    unsigned int ready : 1;
    unsigned int /* unused */ : 8;
    };
typedef short int data;

struct port
    {
    control volatile cr;
    data volatile dr;
    };
```

Using these declarations,

```
piop->out.cr.enable = 0;
```

disables output interrupts for that port and:

```
while (piop->out.cr.ready == 0)
    ;
```

waits until the output device is ready.

If you prefer, you can even think of each single-bit bitfield as a boolean, and write:

```
piop->in.cr.enable = false;
```

to disable output interrupts and:

```
while (!piop->in.cr.ready)
    ;
```

to waits until the output device is ready.

2.4 Embedded Systems - SFR Registers

A Special Function Register (or Special Purpose Register, or simply Special Register) is a register within a microprocessor that controls or monitors the various functions of a microprocessor. As the special registers are closely tied to some special function or status of the processor, they might not be directly writable by normal instructions (like add, move, etc.). Instead, some special registers in some processor architectures require special instructions to modify them.

2.5 Questions

1. What is a register?
2. What is the role of segment Register?
3. It is indicated to use pointers to registers? Why?
4. What SFR Register is meaning?
5. What term “bitfield” is referring to? Give an example.
6. What is the use of “volatile” keyword?
7. What is the difference between “volatile” keyword and “register” keyword?

DAY-2

2 What is GPIO?

Stands for "General Purpose Input/Output." GPIO is a type of pin found on an integrated circuit that does not have a specific function. While most pins have a dedicated purpose, such as sending a signal to a certain component, the function of a GPIO pin is customizable and can be controlled by software.
Not all chips have GPIO pins, but they are commonly found on multifunction chips, such as those used in power managers and audio/video cards. They are also used by system-on-chip (SOC) circuits, which include a processor, memory, and external interfaces all on a single chip. GPIO pins allow these chips to be configured for different purposes and work with several types of components.
A popular device that makes use of GPIO pins is the Raspberry Pi, a single-board computer designed for hobbyists and educational purposes. It includes a row of GPIO pins along the edge of the board that provide the interface between the Raspberry Pi and other components. These pins act as switches that output 3.3 volts when set to HIGH and no voltage when set to LOW. You can connect a device to specific GPIO pins and control it with a software program. For example, you can wire an LED to a GPIO and a ground pin on a Raspberry Pi. If a software program tells the GPIO pin to turn on, the LED will light up.

Most computer users will not encounter GPIO pins and do not need to worry about configuring them. However, if you are a hobbyist or computer programmer, it can be helpful to learn what chips have GPIO pins and how to make use of them.

A GPIO pin is a 'general purpose input/output' pin. This is by default only high or low (voltage levels, high being the micro controller's supply voltage, low usually being ground, or 0V). But the levels of 'high' and 'low' are usually given as voltages as a proportion of the supply voltage. So anything usually above 66% of the supply voltage is considered a logic level 'high' which means some lower voltage devices can talk with high voltage devices as long as the levels fall within what is considered 'high'. A 1.8–2.7V low power microcontroller or GPS receiver for example will have trouble communicating directly to a 5V microcontroller because what the low voltage device sees as 'high' the higher voltage device will not think it's high at all. This is for using GPIO as an input pin, and output is basically the same - the output high is based on the supply of the controller, where it will drive current out and set the voltage of that pin to VCC, or sink current and pull the pin to 0V for a logic 'low'.

Sometimes you can use a SINGLE pin for 'analog' values, by configuring the GPIO pin to be used by other onboard devices like an 'analog to digital' (ADC) converter. The pin is set to a channel on the ADC and this acts as an input to the ADC now, not a normal

GPIO pin. You can then set the ADC to take a sample, and read the ADC's result register value for numbers like 0-1024 if it's 10-bit resolution.

As someone has mentioned, a GPIO pin could be used in software to give the effect of a Pulsed Width Modulation (PWM) signal, usually at low speeds for GPIO toggling. Most microcontrollers have dedicated PWM generators which can be configured to use a GPIO pin as an output pin, and these are very fast and far more stable than using software to control GPIO for generating a PWM signal. PWM are used for 'average' or '%' style signals and allow you to do things like dim lights and control a motor's speed.

GPIO pins are usually arranged in groups, called Ports. In small controllers, they might be 8-bit architecture, so ports are often grouped into lots of 8, and their values can be read all at the same time by reading a 'data register' that represents the logic high/low values of those pins. Similarly, you can set pins to be outputs and then write 8-bits into a data register, and the microcontrollers GPIO controller will read the register's changed values, and drive the pin high or pull the pin low depending what value you just set.

GPIO pins are organized in ports

Depending on the architecture we can have 8/16/32 pin ports

Ports "numbered" in letters: PORTx, x ε [A..Z]

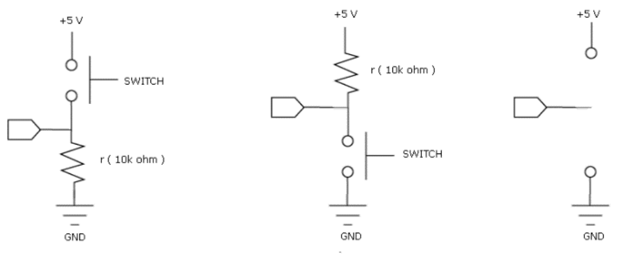
They can be either entry or exit.

Obs. GPIOs are implemented with 3-state logic

1. '0' logic
2. '1' logic
3. 'Z' - high impedance

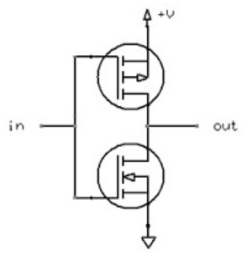
Generally, input pins can be configured in 3 ways

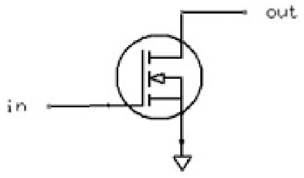
1. Pull-down (internal resistors pull the pin to GND)
2. Pull-up (internal resistors pull the pin to Vcc)
3. High impedance (pin is disconnected from Vcc and GND and floats)



Generally, output pins can be configured in 2 ways

1. Push-Pull (pinii sunt trasi activ la Vcc xor la GND)
2. Open-Drain (pinii sunt trasi la GND sau sunt lasati liberi)





How pull-up and pull-down resistors work (see site from bellow) :

https://medium.freecodecamp.org/a-simple-explanation-of-pull-down-and-pull-up-resistors-660b308f116a

Questions:

1. What is a GPIO? Definition
2. How can be used?
3. How can be configured?
4. How a pull up resistor work?
5. How a pull down resistor work?

3 Embedded Systems - Timer/Counter

A timer is a specialized type of clock which is used to measure time intervals. A timer that counts from zero upwards for measuring time elapsed is often called a stopwatch. It is a device that counts down from a specified time interval and used to generate a time delay, for example, an hourglass is a timer.

A counter is a device that stores (and sometimes displays) the number of times a particular event or process occurred, with respect to a clock signal. It is used to count the events happening outside the microcontroller. In electronics, counters can be implemented quite easily using register-type circuits such as a flip-flop.

3.1 Difference between a Timer and a Counter

The points that differentiate a timer from a counter are as follows −

Timer	Counter
The register incremented for every machine cycle.	The register is incremented considering 1 to 0 transition at its corresponding to an external input pin (T0, T1).
Maximum count rate is 1/12 of the oscillator frequency.	Maximum count rate is 1/24 of the oscillator frequency.
A timer uses the frequency of the internal clock, and generates delay.	A counter uses an external signal to count pulses.

Prescale in Microcontrollers

For a timer, you can control two thing.
A. Till what value it goes to from where.
B. How fast it covers all the values in between.

What a timer does is,(assuming a up timer) starts from a value and 'counts' up to another value. However it does not does this counting randomly. It won't count one today and next tomorrow. Rather, it will do it in a multiple of the time taken by one clock cycle that depends upon the micro controller.

So say one clock cycle takes 1 nanosecond.
A timer can time up every one nano second, and thus reach from 0 to 1000 in one micro second(1 nano x 1000 =1 micro)
Or, if you instruct it to time every up after every 2 nano seconds, it will reach from 0 to 1000 in two micro seconds.
This instruction can be changed by you, by use of prescalar. Given you have a 5 bit pre scalar, you can make your timer count every 1 clock cycle(CC) , every 2CC,4CC,8CC,16CC,32CC.
That means you can slow down the timer at max by 32 times the normal rate.
Also understand, it doesn't mean you can pick up any value from 1 to 32. Only those that are power of 2.

Another way of thinking is that your timer is a shift register and the clock for it it first passed through a frequency divisor. The divisor can take values from 2^0 , 2^1,2^2,2^3,2^4 and 2^5.

3.2 Questions

1. What is watchdog timer?
2. What is the difference between a Timer and a Counter?
3. What is a prescaler and how can be used?

DAY -3

4 Interrupts

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. Whenever an interrupt occurs, the controller completes the execution of the current instruction and starts the execution of an Interrupt Service Routine (ISR) or Interrupt Handler. ISR tells the processor or controller what to do when the interrupt occurs. The interrupts can be either hardware interrupts or software interrupts.

4.1 Hardware Interrupt

A hardware interrupt is an electronic alerting signal sent to the processor from an external device, like a disk controller or an external peripheral. For example, when we press a key on the keyboard or move the mouse, they trigger hardware interrupts which cause the processor to read the keystroke or mouse position.

4.2 Software Interrupt

A software interrupt is caused either by an exceptional condition or a special instruction in the instruction set which causes an interrupt when it is executed by the processor. For example, if the processor's arithmetic logic unit runs a command to divide a number by zero, to cause a divide-by-zero exception, thus causing the computer to abandon the calculation or display an error message. Software interrupt instructions work similar to subroutine calls.

Other example will be a function call .

4.3 What is Polling?

The state of continuous monitoring is known as polling. The microcontroller keeps checking the status of other devices; and while doing so, it does no other operation and consumes all its processing time for monitoring. This problem can be addressed by using interrupts.

In the interrupt method, the controller responds only when an interruption occurs. Thus, the controller is not required to regularly monitor the status (flags, signals etc.) of interfaced and inbuilt devices.

4.4 Interrupts v/s Polling

Here is an analogy that differentiates an interrupt from polling −

Interrupt	Polling
An interrupt is like a shopkeeper. If one needs a service or product, he goes to him and apprises him of his needs. In case of interrupts, when the flags or signals are received, they notify the controller that they need to be serviced.	The polling method is like a salesperson. The salesman goes from door to door while requesting to buy a product or service. Similarly, the controller keeps monitoring the flags or signals one by one for all devices and provides service to whichever component that needs its service.

4.5 Interrupt Service Routine

For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler. When an interrupt occurs, the microcontroller runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its interrupt service routine, ISR. The table of memory locations set aside to hold the addresses of ISRs is called as the Interrupt Vector Table.

4.6 Steps to Execute an Interrupt

When an interrupt gets active, the microcontroller goes through the following steps −

- The microcontroller closes the currently executing instruction and saves the address of the next instruction (PC) on the stack.
- It also saves the current status of all the interrupts internally (i.e., not on the stack).
- It jumps to the memory location of the interrupt vector table that holds the address of the interrupts service routine.
- The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine, which is RETI (return from interrupt).

- Upon executing the RETI instruction, the microcontroller returns to the location where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then, it start to execute from that address.

4.7 Edge Triggering vs. Level Triggering

Interrupt modules are of two types − level-triggered or edge-triggered.

Level Triggered	Edge Triggered
A level-triggered interrupt module always generates an interrupt whenever the level of the interrupt source is asserted.	An edge-triggered interrupt module generates an interrupt only when it detects an asserting edge of the interrupt source. The edge gets detected when the interrupt source level actually changes. It can also be detected by periodic sampling and detecting an asserted level when the previous sample was de-asserted.
If the interrupt source is still asserted when the firmware interrupt handler handles the interrupt, the interrupt module will regenerate the interrupt, causing the interrupt handler to be invoked again.	Edge-triggered interrupt modules can be acted immediately, no matter how the interrupt source behaves.
Level-triggered interrupts are cumbersome for firmware.	Edge-triggered interrupts keep the firmware's code complexity low, reduce the number of conditions for firmware, and provide more flexibility when interrupts are handled.

4.8 Enabling and Disabling an Interrupt

Upon Reset, all the interrupts are disabled even if they are activated. The interrupts must be enabled using software in order for the microcontroller to respond to those interrupts.

IE (interrupt enable) register is responsible for enabling and disabling the interrupt. IE is a bitaddressable register.

4.9 Interrupt Enable Register

EA	-	ET2	ES	ET1	EX1	ET0	EX0

- EA − Global enable/disable.
- - − Undefined.
- ET2 − Enable Timer 2 interrupt.
- ES − Enable Serial port interrupt.
- ET1 − Enable Timer 1 interrupt.
- EX1 − Enable External 1 interrupt.
- ET0 − Enable Timer 0 interrupt.
- EX0 − Enable External 0 interrupt.

To enable an interrupt, we take the following steps −

- Bit D7 of the IE register (EA) must be high to allow the rest of register to take effect.
- If EA = 1, interrupts will be enabled and will be responded to, if their corresponding bits in IE are high. If EA = 0, no interrupts will respond, even if their associated pins in the IE register are high.

4.10 Interrupt Priority in 8051

We can alter the interrupt priority by assigning the higher priority to any one of the interrupts. This is accomplished by programming a register called IP(interrupt priority).

The following figure shows the bits of IP register. Upon reset, the IP register contains all 0's. To give a higher priority to any of the interrupts, we make the corresponding bit in the IP register high.

-	-	-	-	PT1	PX1	PT0	PX0

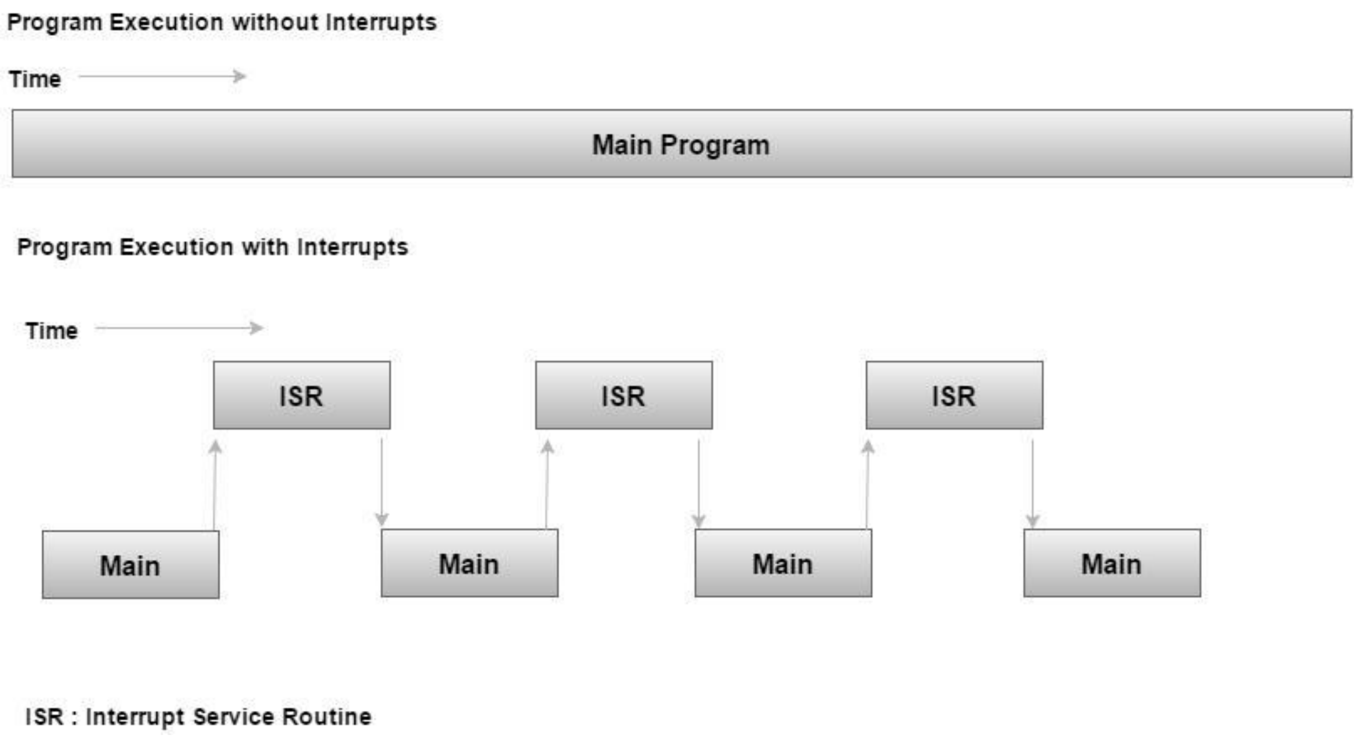
-	IP.7	Not Implemented.
-	IP.6	Not Implemented.
-	IP.5	Not Implemented.
-	IP.4	Not Implemented.
PT1	IP.3	Defines the Timer 1 interrupt priority level.
PX1	IP.2	Defines the External Interrupt 1 priority level.
PT0	IP.1	Defines the Timer 0 interrupt priority level.
PX0	IP.0	Defines the External Interrupt 0 priority level.

4.11 Interrupt inside Interrupt

What happens if the 8051 is executing an ISR that belongs to an interrupt and another one gets active? In such cases, a high-priority interrupt can interrupt a low-priority interrupt. This is known as interrupt inside interrupt. In 8051, a low-priority interrupt can be interrupted by a high-priority interrupt, but not by any another low-priority interrupt.

4.12 Triggering an Interrupt by Software

There are times when we need to test an ISR by way of simulation. This can be done with the simple instructions to set the interrupt high and thereby cause the 8051 to jump to the interrupt vector table. For example, set the IE bit as 1 for timer 1. An instruction SETB TF1 will interrupt the 8051 in whatever it is doing and force it to jump to the interrupt vector table.


Program Execution without Interrupts
Time
Main Program
Program Execution with Interrupts
Time
ISR
ISR
ISR
Main
Main
Main
Main
ISR : Interrupt Service Routine

4.13 Questions

1. What is the difference between timer and counter?
2. What is an interrupt?
3. What means interrupt latency?
4. What are the causes of interrupt latency?
5. How to reduce the interrupt latency?
6. What is a nested interrupt?
7. What is the main difference between Interrupts and Pooling Mode?
8. Give two examples of Hardware Interrupts. One example of sw interrupt.
9. How can the interrupts be disabled? Why is needed to disable the interrupts?

5 ADC

5.1 The Analog world with digital Electronics

Few years back the entire electronics devices that we use today like phones, computers, Televisions etc were analog in nature. Then slowly the landline phones were replaced by modern mobile phones, CRT Televisions and monitors were replaced by LED displays, computers with vacuum tubes evolved to be more powerful with microprocessors and microcontrollers inside them and so on..

In today's digital age we are all surrounded by the advanced digital electronic devices, this might deceive us to think that everything around us is digital in nature, which is not true. The world has always been analog in nature, for instance everything that we humans feel and experience like speed, temperature, air velocity, sunlight, sound etc are analog in nature. But our electronic devices which run on microcontrollers and microprocessors cannot read/interpret these analog values directly since they run only on 0's and 1's. So we need something which will convert all these analog values into 0's and 1's so that our microcontrollers and microprocessors can understand them. This something is called the **Analog to Digital Converters or ADC for short**. In this article we will learn **everything about ADC and how to use them**.

5.2 What is ADC and how to use it?

As said earlier ADC stands for Analog to digital conversion and it is used to convert analog values from real world into digital values like 1's and 0's. So what are these analog values? These are the ones that we see in our day to day life like temperature, speed, brightness etc. But wait!! Can an ADC convert temperature and speed directly into digital values like 0's and 1's?

No defiantly not. An ADC can only **convert analog voltage values into digital values**. So which ever parameter we wish to measure, it should be converted into voltage first, this conversion can be done with the help of **sensors**. For example to convert temperature values into voltage we can use a Thermistor similarly to convert brightness to voltage we can use a LDR. Once it is converted to voltage we can read it with the help of ADC's.

In order to know how to use an ADC we should first get familiar with some basic terms like, channels resolution, range, reference voltage etc.

5.3 Resolution (bits) and channels in ADC

When you read the specification of any Microcontroller or ADC IC, the details of the ADC will be given using the terms channels and Resolution (bits). For instance **an Arduino UNO's ATmega328 has a 8-channel 10-bit ADC**. Not every pin on a microcontroller can read Analog voltage, the term 8-channel means that there are 8 pins on this ATmega328 microcontroller which can read Analog voltage and each pin can read the voltage with a resolution of 10-bit. This will vary for different types of Microcontrollers.

Let us assume that our ADC range is from 0V to 5V and we have a 10-bit ADC this means our input voltage 0-5 Volts will be split into 1024 levels of discrete analog values(2^{10} = 1024). Meaning 1024 is the resolution for a 10-bit ADC, similarly for a 8-bit ADC resolution will be 512 (2^{8}) and for a 16-bit ADC resolution will be 65,536 (2^{16}).

With this if the actual input voltage is 0V then the MCU's ADC will read it as 0 and if it is 5V the MCU will read 1024 and if it somewhere in between like 2.5V then the MCU will read 512. We can use the below formulae to calculate the digital value that will be read by the MCU based on the Resolution of the ADC and Operating voltage.

(ADC Resolution / Operating Voltage) = (ADC Digital Value / Actual Voltag e Value)

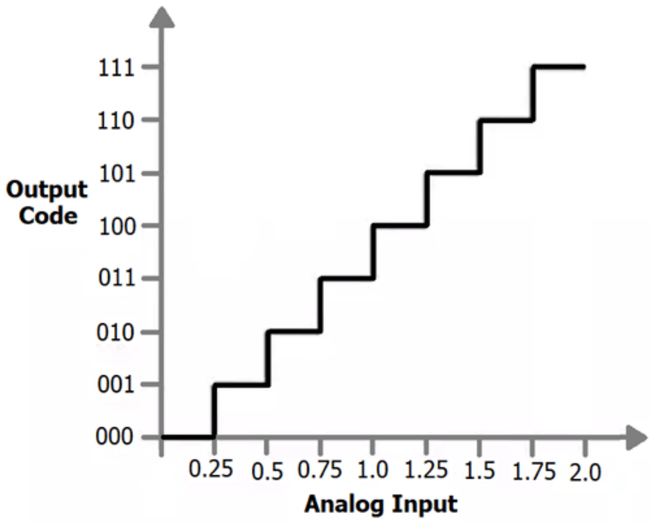
5.4 Reference Voltage for an ADC

Another important term that you should be familiar with is the reference voltage. During an ADC conversion the **value of unknown voltage is found by comparing it with a known voltage, this is known voltage is called as Reference voltage**. Normally all MCU has an option to set **internal reference voltage,** meaning you can set this voltage internally to some available value using software (program). In an Arduino UNO board the reference voltage is set to 5V by default internally, if required user can set this reference voltage externally through the Vref pin also after making the required changes in the software.

Always remember that the measured analog voltage value should always be less than the reference voltage value and the reference voltage value should always be less than the operating voltage value of the microcontroller.

5.5 Example

Here we are taking example of ADC which has 3 bit resolution and 2V reference voltage. So it can map the 0-2v analog voltage with 8 (2^3) different levels, like shown in the below picture:



So if analog voltage is 0.25 then the digital value will be 1 in decimal and 001 in binary. Likewise if analog voltage is 0.5 then the digital value will be 2 in decimal and 010 in binary.

Some microcontroller has inbuilt ADC like Arduino, MSP430, PIC16F877A but some microcontroller don't have it like 8051, Raspberry Pi etc and we have to use some external Analog to digital converter ICs like ADC0804, ADC0808.

Below you can find various examples of ADC with different microcontrollers:

- How to use ADC in STM32F103C8

5.6 ADC types and working

There are many types of ADC, the most commonly used ones **are Flash ADC, Dual Slope ADC, Successive approximation and Dual Slope ADC**. To explain how each of these ADC's work and the difference between them would be out of scope for this article as they are fairly complex. But to give a rough idea ADC has an internal capacitor which will get charged by the analog voltage that is to measured. Then we measure the voltage value by discharging the capacitor over a period of time.

5.7 Some commonly arising questions on ADC

How to measure more than 5V using my ADC?
As discussed earlier an ADC module cannot measure voltage value more than the operating voltage of the microcontroller. That is a 5V microcontroller can measure only a maximum of 5V with its ADC pin. If you want to measure anything more than that say, you want to measure 0-12V then you can map the 0-12V into 0-5V by using a potential divider or voltage divider circuit. This circuit will use a pair of resistors to map down the values for a MCU, you can know more about voltage divider circuit using the link. For our above example we should use a 1K resistor and 720 ohm resistor in series to the voltage source and measure the voltage in between the resistors as discussed in the link above.

How to convert Digital Values from ADC into actual Voltage Values?
When using an ADC converter to measure analog voltage the result obtained by the MCU will be in digital. For example in a 10-bit 5V microcontroller when the actual voltage that is to be measure is 4V the MCU will read it as 820, we can again use the above discussed formulae to convert the 820 to 4V so that we can use it in our calculations. Lets cross-check the same.

(ADC Resolution / Operating Voltage) = (ADC Digital Value / Actual Voltag e Value)

Actual Voltage Value = ADC Digital Value * (Operating Voltage / ADC Res olution)

= 820 * (5/1023)

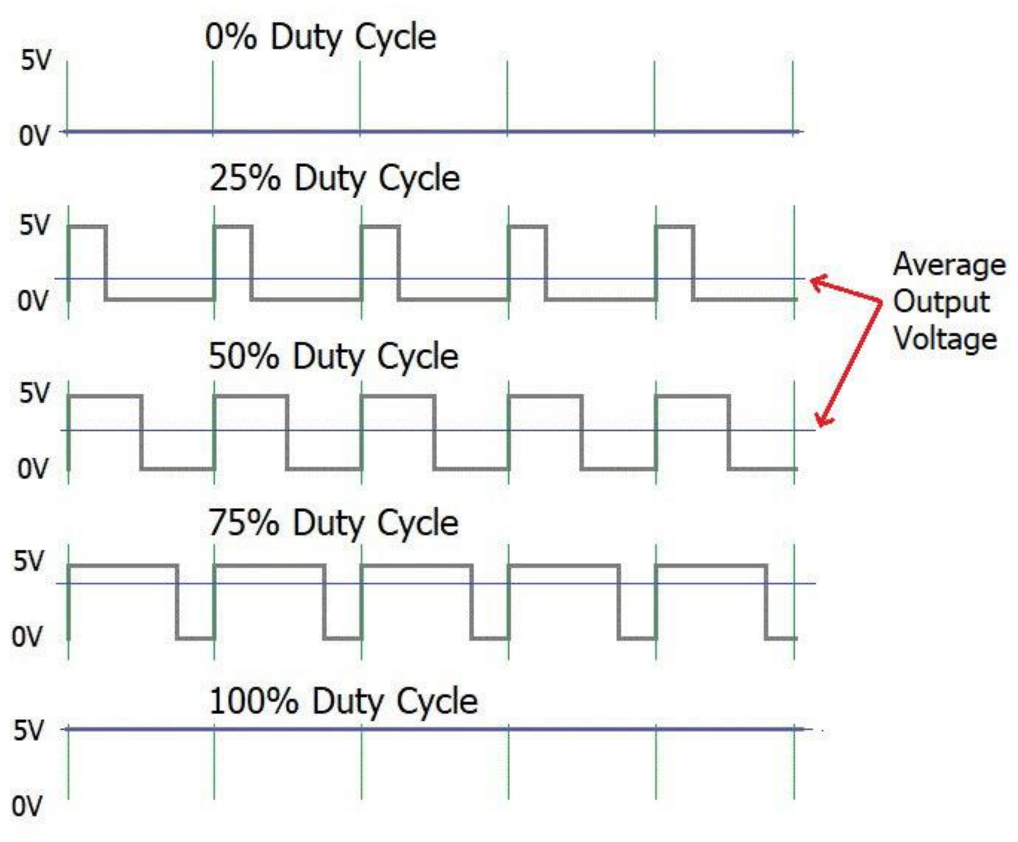
= 4.007

= ~4V

5.8 Questions

1. What is ADC?
2. For what is used the Resolution in an ADC?
3. Usually an ADC can measure the voltage from 0 to 5V (or from 0 to 3.3V). How a voltage of 7V can be measured?
4. Describe in couple of sentences the “Successive Approximation” ADC Conversion Method.
5. What is a Channel for an ADC?

6 Pulse Width Modulation



6.1 What is PWM: Pulse Width Modulation

Inverters, Converters, SMPS circuits and Speed controllers.... One thing that is common in all these circuits is that it consists of many electronic switches inside it. These switches are nothing but Power electronic devices like MOSFET, IGBT, TRIAC etc. In order to control such power electronic switches we commonly use something called PWM signals (Pulse Width Modulation). Apart from this, PWM signals are also used for driving Servo motors and also for other simple tasks like controlling the brightness of a LED.

In our previous article we learnt about ADC, while ADC is used to read Analog signals by a digital device like microcontroller. A PWM can be considered as an exact opposite of it, **PWM is used to produce Analog signals from a digital device like microcontroller**. In this article we will learn about **what is PWM**, PWM signals and some parameters associated with it, so that we will be confident in using them in our designs.

6.2 What is PWM (Pulse Width Modulation)?

PWM stands for Pulse Width Modulation; we will get into the reason for such a name later. But, for now understand PWM as a type of signal which can be produced from a digital IC such as microcontroller or 555 timer. The signal thus produced will have a train of pulses and these pulses will be in form of a square wave. That is, at any given instance of time the wave will either be high or will be low. For the ease of understanding let us consider a 5V PWM signal, in this case the PWM signal will either be 5V (high) or at ground level 0V (low). The duration at which the signals stays high is called the "*on time*" and the duration at which the signal stays low is called as the "*off time*".

For a PWM signal we need to look at two important parameters associated with it one is the PWM duty cycle and the other is PWM frequency.

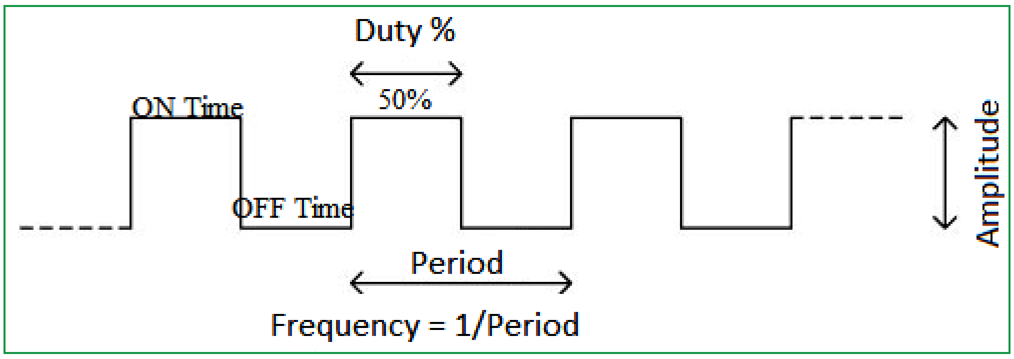
6.3 Duty cycle of the PWM

As told earlier, a PWM signal stays on for a particular time and then stays off for the rest of the period. What makes this PWM signal special and more useful is that we can set for how long it should stay on by controlling the duty cycle of the PWM signal.

The percentage of time in which the PWM signal remains HIGH (on time) is called as duty cycle. If the signal is always ON it is in 100% duty cycle and if it is always off it is 0% duty cycle. The formulae to calculate the duty cycle is shown below.

Duty Cycle =Turn ON time/ (Turn ON time + Turn OFF time)

The following image represents a PWM signal with 50% duty cycle. As you can see, considering an entire **time period** (on time + off time) the PWM signal stays on only for 50% of the time period.



By controlling the Duty cycle from 0% to 100% we can control the "*on time*" of PWM signal and thus the width of signal. Since we can modulate the width of the pulse, it got its iconic name "*Pulse width Modulation*".

6.4 Frequency of a PWM

The frequency of a PWM signal determines how fast a PWM completes one period. One Period is the complete ON and OFF time of a PWM signal as shown in the above figure. The formulae to calculate the Frequency is given below

Frequency = 1/Time Period

Time Period = On time + Off time

Normally the PWM signals generated by microcontroller will be around 500 Hz, such high frequencies will be used in high speed switching devices like inverters or converters. But not all applications require high frequency. For example to control a servo motor we need to produce PWM signals with 50Hz frequency, so the frequency of a PWM signal is also controllable by program for all microcontrollers.

6.5 Some commonly arising questions on PWM

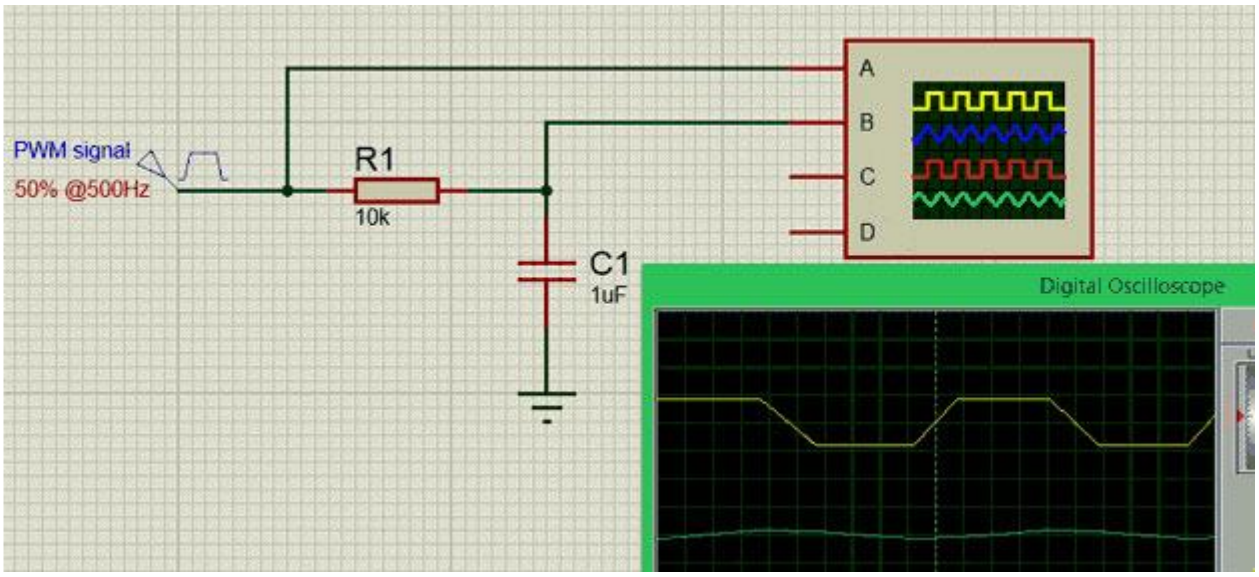
What is the difference between the Duty cycle and Frequency of a PWM signal?

The duty cycle and frequency of a PWM signals are often confused upon. As we know a PWM signal is a square wave with a particular on time and off time. The sum of this *on time* and *off time* is called as one time period. The inverse of one time period is called frequency. While the amount of time the PWM signal should remain on in one time period is decide by Duty cycle of the PWM.

To put it simple, how fast the PWM signal should turn on and turn off is decided by **the frequency of the PWM signal**and in that speed how long the PWM signal should remain turned on is decided by the **duty cycle of the PWM signal**.

How to convert PWM signals into Analog Voltage?

For simple applications like controlling the speed of a DC motor or adjusting brightness of an LED we need to convert the PWM signals into analog voltage. This can be easily done by using a RC filter and is commonly used where a DAC feature is required. The circuit for the same is shown below



In the graph shown above, the Yellow coloured one is the PWM signal and the blue colour one is the output analog voltage. The value of the resistor R1 and capacitor C1 can be calculated based on the frequency of the PWM signal but normally a 5.7K or 10K resistor and a 0.1u or 1u Capacitor is used.

How to calculate the output voltage of PWM signal?

The output voltage of a PWM signal after converting it to analog will be the percentage of Duty cycle. For example if the operating voltage is 5V then the PWM signal will also have 5V when high. In such case for a 100% duty cycle the output voltage will be 5V for a 50% duty cycle it will be 2.5V.

Output Voltage = Duty cycle (%) * 5

Examples:

We have previously used PWM with various microcontroller in many of our projects:

- Pulse width Modulation (PWM) in STM32F103C8

6.6 Questions

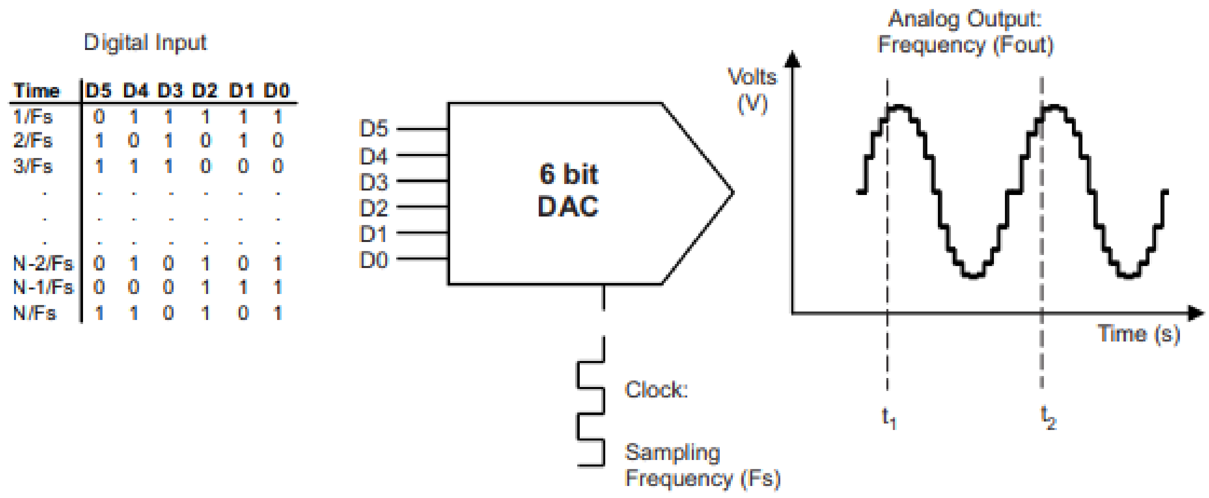
1. What is PWM? Give an example.
2. What is a Duty Cycle for a PWM signal?
3. What is the difference between Frequency and Period?
4. What is the difference between Duty Cycle and Frequency on a PWM signal?

7 DAC

7.1 What is a DAC?

A Digital to Analog Converter (DAC or D-to-A) is a device that converts digital codes to an analog signal. There are many categories of DACs. Some of these categories include sigma-delta DACs, pulse width modulators, interpolating and high speed DACs. In this paper the focus will be on the interpolating and high speed DACs.

Figure 1 provides the basic block diagram, functionality and common terminology for DACs. This figure shows a digital word applied to the inputs of the DAC, which is then converted to an analog signal at the sampling frequency (Fs) applied to the DAC clock. Figure 1 is a time domain representation of the DACs input and output signals.



Time	D5	D4	D3	D2	D1	D0
1/Fs	0	1	1	1	1	1
2/Fs	1	0	1	0	1	0
3/Fs	1	1	1	0	0	0
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
N-2/Fs	0	1	0	1	0	1
N-1/Fs	0	0	0	1	1	1
N/Fs	1	1	0	1	0	1

Fout: Analog Output Frequency = $1/(t_2 - t_1)$
Fs: Clock Frequency
N: Number of digital samples
n: Number of output bits; in this 6 bit DAC example n = 6

Figure 1. Basic DAC Diagram and Terminology

Time domain representations are often described as real world signals. In Figure 1, notice the analog output's amplitude is shown in volts (linear) and seconds (linear). Also notice the digital input codes are listed with time stamps (1/Fs, 2/Fs, 3/Fs…). Time domain representations are often easy to visualize and help with understanding gross concepts. However the time domain is poor when it comes to measuring performance of DACs and other signal processing devices. Measuring performance is best done in the frequency domain. Therefore, it is important to understand how the time domain and frequency domain relate.

7.2 Questions

1. Define in couple of sentences the Digital to Analog Converter.

DAY – 4

8 Communication Bus

Please see the following trainings:

http://www.eapbg.com/home/category/Communication

8.1 Questions

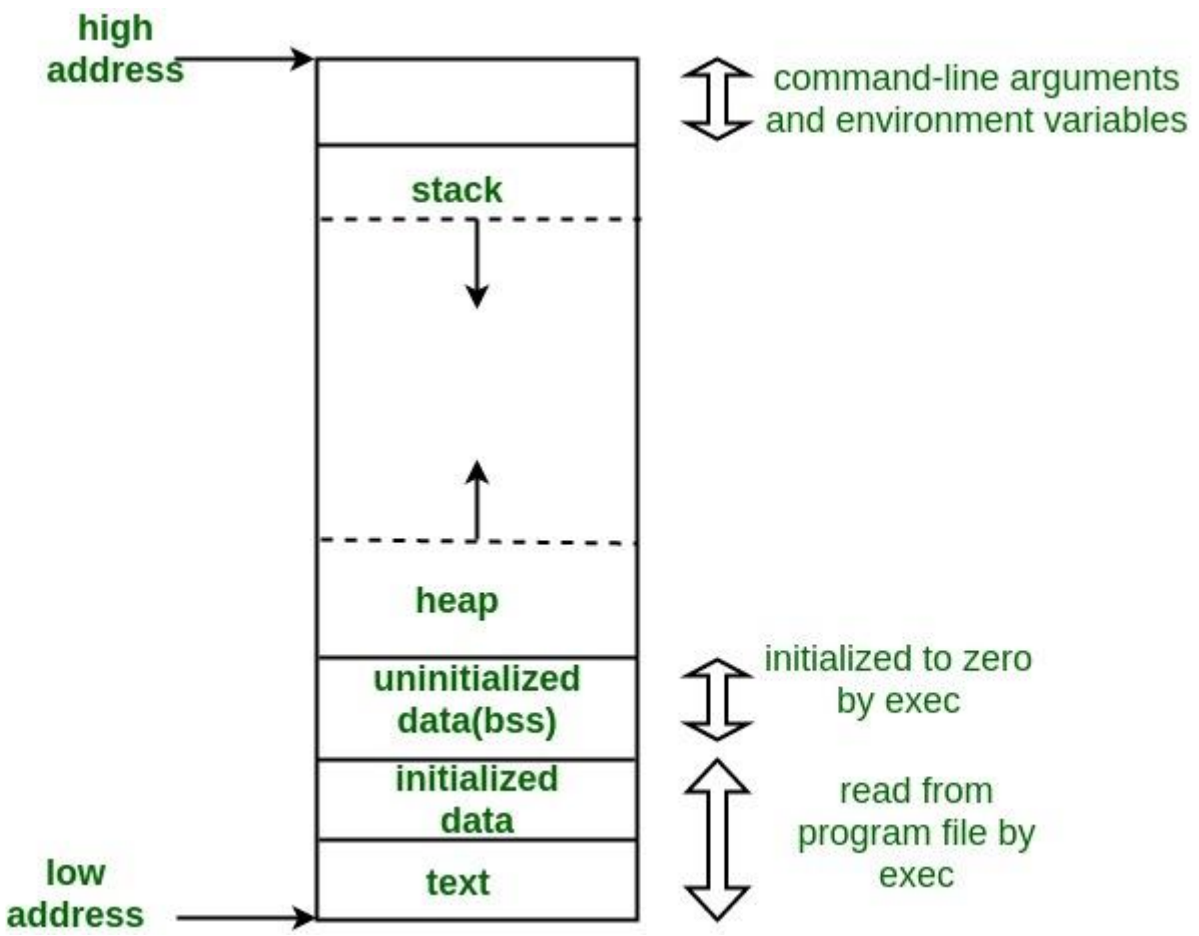
1. What is the format of a CAN data Frame?
2. CAN is a syncron or asyncron protocol? Why?
3. How SPI work?
4. How UART work?

DAY - 5

9 Memory Layout of C Programs

A typical memory representation of C program consists of following sections.

1. Text segment
2. Initialized data segment
3. Uninitialized data segment
4. Stack
5. Heap



high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address

A typical memory layout of a running process

9.1 Text Segment

A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

As a memory region, a text segment may be placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

9.2 Initialized Data Segment

Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer. Note that, data segment is not read-only, since the values of the variables can be altered at run time.

This segment can be further classified into initialized read-only area and initialized read-write area.

For instance the global string defined by char s[] = “hello world” in C and a C statement like int debug=1 outside the main (i.e. global) would be stored in initialized read-write area. And a global C statement like const char* string = “hello world” makes the string literal “hello world” to be stored in initialized read-only area and the character pointer variable string in initialized read-write area.

Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

9.3 Uninitialized Data Segment

Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing
uninitialized data starts at the end of the data segment and contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

For instance a variable declared static int i; would be contained in the BSS segment.
For instance a global variable declared int j; would be contained in the BSS segment.

9.4 Stack

The stack area traditionally adjoined the heap area and grew the opposite direction; when the stack pointer met the heap pointer, free memory was exhausted. (With modern large address spaces and virtual memory techniques they may be placed almost anywhere, but they still typically grow opposite directions.)
The stack area contains the program stack, a LIFO structure, typically located in the higher parts of memory. On the standard PC x86 computer architecture it grows toward address zero; on some other architectures it grows the opposite direction. A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack. The set of values pushed for one function call is termed a “stack frame”; A stack frame consists at minimum of a return address.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

9.5 Heap

Heap is the segment where dynamic memory allocation usually takes place. The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size (note that the use of brk/sbrk and a single “heap area” is not required to fulfill the contract of malloc/realloc/free; they may also be implemented using mmap to reserve potentially non-contiguous regions of virtual memory into the process’ virtual address space). The Heap area is shared by all shared libraries and dynamically loaded modules in a process.

9.6 Examples

5. Let us add one global variable in program, now check the size of bss (highlighted in red color).

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/ /* BSS -
Uninitialized Data Segment */

int main(void)
{
    return 0;
}
```

6. Let us add one static variable which is also stored in bss.

```
#include <stdio.h>

int global;       /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i; /* Uninitialized static variable stored in bss */
    return 0;
}
```

3. Let us initialize the static variable which will then be stored in Data Segment (DS)

```
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

4. Let us initialize the global variable which will then be stored in Data Segment (DS)

```
 #include <stdio.h>

int global = 10; /* initialized global variable stored in DS*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

5. Consider the following example of a program containing all three forms of memory:

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

int x;

int main(void)
{
    int y;
    char *str;

    y = 4;
    printf("stack memory: %d\n", y);

    str = malloc(100*sizeof(char));
    str[0] = 'm';
    printf("heap memory: %c\n", str[0]);
    free(str);
    return 0;
}
```

The variable **x** is static storage, because of its global nature.
Both **y** and **str** are dynamic stack storage which is deallocated when the program ends. The function **malloc()** is used to allocate 100 pieces of of dynamic heap storage, each the size of char, to **str**. Conversely, the function **free()**, deallocates the memory associated with **str**.

9.7 Questions

1. Which are the five elements that typically represents a memory in a C Program?
2. What is the difference between the Text Segment and Data Segment?
3. Also in Stack and Data Segment are stored the variables from a C Program. What is the main difference between these two?
4. What is the start-up code?
5. What are the start-up code steps?
6. Give an example of a variable that is stored on Heap Segment.

10 Microcontroller Memory

10.1 RAM

RAM: we know that RAM (Random Access Memory) which is a volatile memory used for storing the data temporarily in its registers. RAM memory is divided in to Banks, in each banks we have number of registers. The RAM registers is divided into 2 types. They are General purpose registers (GPR) and Special purpose registers (SPR).

1. **GPR:** general purpose registers as the name implies for general usage. For example if we want to multiply any two numbers using PIC we generally take two registers for storing the numbers and multiply the two numbers and store the result in other registers. So general purpose registers will not have any special function or any special permission, CPU can easily access the data in the registers.
2. **SPR:** Special function registers are having the specific functions, when we use this register they will act according to the functions assigned to them. They cannot be used like normal registers. For example you cannot use STATUS register for storing the data, STATUS registers are used for showing the status of the program or operation. User cannot change the function of the Special function register; the function is given by the vendor at the manufacturing time.

10.2 ROM

ROM: we know that ROM (Read Only memory) is a non volatile memory used for storing the data permanently. In microcontroller ROM will store the complete instructions or program, according the program microcontroller will act. Rom is also called program memory in this memory user will write the program for microcontroller and save it permanently and get executed by the CPU. According to the instruction executed by the CPU the PIC microcontroller will perform the task. In ROM there are different types which are used in different PIC microcontrollers.

- **EEPROM:** In the normal ROM we can write the program for only one time we cannot reuse the Microcontroller for another time where as in the EEPROM (Electrically Erasable Programmable Read Only Memory) we can program the ROM for number of times.
- **Flash Memory:** flash memory is also PROM in which we can read write and erase the program more than 10,000 times. Mostly PIC microcontroller uses this type of ROM.

10.3 ROM vs RAM

The differences between ROM (Read Only Memory) and RAM (Random Access Memory) are:

1. ROM is a form of permanent storage while RAM is a form of temporary storage.
2. ROM is non-volatile memory while RAM is volatile memory.
3. ROM can hold data even without electricity, while RAM needs electricity to hold data.

10.3.1 Storage Capacity

A ROM chip usually stores no more than a few megabytes of data (4 MB ROM is quite common these days). In contrast, a RAM chip can store as much as 16 Gigabytes' (or more) worth of information.

10.3.2 Ease of writing data

It's easier to write data in RAM than in ROM, since the latter is a place for storing very limited, albeit immensely important and permanent information.

The next time you find yourself in a circle of computer geeks, make sure that you bring this information to the table. They might already know it, but they'll certainly be impressed!

10.3.3 Speed

RAM trumps ROM in terms of speed; it accesses data much faster than ROM, and boosts the processing speed of the computer.

10.3.4 Data retention

This is the most noteworthy difference between these two forms of memory. ROM is a form of non-volatile memory, which means that it retains information even when the computer is shut down. RAM, on the other hand, is considered volatile memory. It holds data as long as your computer is up and running. After that…

10.4 Flash Memory

Definition: **Flash memory** (Known as Flash Storage) is a type of non-volatile storage memory that can be written or programmed in units called “Sector” or a “Block.” Flash Memory is **EEPROM** (*Electronically Erasable Programmable Read-Only Memory*) means that it can retain its contents when the power supply removed, but whose contents can be quickly erased and rewritten at the byte level by applying a short pulse of higher voltage. This is called flash erasure, hence the name. Flash memory is currently both too expensive and too slow to serve as mainmemory.

Flash memory (sometimes called "Flash RAM") is a distinct EEPROM that can read block-wise. Typically the sizes of the block can be from hundreds to thousands of bits. Flash Storage block can be divided into at least two logical sub-blocks.

Flash memory mostly used in consumer storage devices, and for networking technology. It commonly found in mobile phones, USB flash drives, tablet computers, and embedded controllers.

Flash memory is often used to hold control code such as the basic input/output system (BIOS) in a personal computer. When BIOS needs to be changed (rewritten), the flash memory can be written to in block (rather than byte) sizes, making it easy to update. On the other hand, flash memory is notusedas random access memory (RAM) because RAM needs to be addressable at the byte (not the block) level.

10.5 EEPROM

10.5.1 What is an EEPROM?

EEPROM, pronounced as Double-E-PROM, stands for Electrically Erasable Programmable Read-Only Memory. This kind of memory devices is re-programmable by the application of electrical voltage and can be addressed to write/read each specific memory location.
The EEPROM memory devices have evolved from the old EPROM memories. Which was the previous technology in this area. A typical EPROM has a window on the top side of the IC to allow the ultraviolet rays to reach the memory cells in order to erase the memory. The EPROMs had to be exposed to ultraviolet light for a convenient time period in order to get fully-erased.

You should also know that EEPROM memories are non-volatile. Which means it won’t lose the data contents when the power goes OFF. It can only be erased electrically whether it’s internal within a microcontroller (By Code) or external IC (By Electronic Device). On the contrary, the RAM memory is volatile. Which means it does lose all of its contents when the power goes OFF.

EEPROMs are fabricated and shipped as standalone IC chips and Microchip Technology has a significant share of this area in the market. External EEPROMs are manufactured to be interfaced in two different ways. There are parallel & serial address/data lines versions of EEPROMs.

The internal EEPROM memories (Built-in Within Microcontrollers) can be accessed for reading/writing operations by code. Writing a few lines of code will enable you of storing and/or retrieving data from the built-in EEPROM memory. And this is going to be our task in this tutorial. To develop the required firmware to drive this memory module.

10.5.2 Applications for EEPROMs

Let me give you a quick recap of the features of EEPROM memories before discussing their applications in real-life. A typical EEPROM device regardless of its type (internal/external) has the following features:

- Electrical erase-ability
- Electrical re-programmability
- Non-volatile memory locations
- Serial/Parallel interfaces for address/data lines (For External EEPROMs)
- Easy programmatically-controlled memory interface (For Internal EEPROMs)

With all of these features in mind. What applications do you think are a good fit for EEPROMs?
Well, there are many situations indeed. I'll list down some of these and you can think of or search for any further applications. There is a ton of ways in which we can take advantage of the features provided by EEPROMs. And there are countless situations in which we use this kind of memory devices.

- If you're building a computer system that needs to switch between programs (partially), you should have a memory to store these program instructions. Then you can easily load these instructions to your flash memory when you need to.
- Building an embedded system that needs to remember a user-given data even after a power-restart or a power-down condition. Such as a digital lock password-protected money locker. When the power goes OFF, the user's password must be stored in a safe place.
- When you're building a robot that's doing a specific mission (scanning, searching, trolling around) where it's collecting some sort of critical data. It's a good idea to have a backup copy of this data on your local (internal) EEPROM. In an emergency, you can stream this data via a serial bus to your computer anytime you want.
- For a multi-computer system where there are many controllers working together in a robotic system or the like. If there is a sharable data, it'd be a

good idea to store it on shared EEPROM via the serial bus (e.g. I2C). Any device can update/read this data without adding the overhead of sending/receiving data requests/updates between all the computers involved.

- And much more…

There are millions of EEPROM chips are being shipped every single year. These devices are actually embedded in almost all electronic devices you might ever know. If it's not existent as a standalone IC chip, it should be integrated within the embedded computer (e.g. MCU) which controls your device behind the scenes.

10.5.3 Serial bus devices

The common serial interfaces are SPI, I²C, Microwire, UNI/O, and 1-Wire. These use from 1 to 4 device pins and allow devices to use packages with 8-pins or less.

A typical EEPROM serial protocol consists of three phases: OP-Code Phase, Address Phase and Data Phase. The OP-Code is usually the first 8-bits input to the serial input pin of the EEPROM device (or with most I²C devices, is implicit); followed by 8 to 24 bits of addressing depending on the depth of the device, then the read or write data.

Each EEPROM device typically has its own set of OP-Code instructions mapped to different functions. Common operations on SPI EEPROM devices are:

- Write Enable
- Write Disable
- Read Status Register
- Write Status Register
- Read Data
- Write Data

Other operations supported by some EEPROM devices are:

- Program
- Sector Erase
- Chip Erase commands

10.5.4 Parallel bus devices

Parallel EEPROM devices typically have an 8-bit data bus and an address bus wide enough to cover the complete memory. Most devices have chip select and write protect pins. Some microcontrollers also have integrated parallel EEPROM.

Operation of a parallel EEPROM is simple and fast when compared to serial EEPROM, but these devices are larger due to the higher pin count (28 pins or more) and have been decreasing in popularity in favor of serial EEPROM or flash.

10.6 Questions

1. What is the difference between the RAM and ROM?
2. Describe in couple of sentences the EEPROM.
3. Describe in couple of sentences the FLASH Memory.
4. What is the difference between FLASH Memory, EPROM and EEPROM?
5. What is the difference between the volatile and non-volatile memory?
6. Where is stored a local variable?
7. Where is stored a static local variable?
8. Where is stored a global variable?

11 Operating Systems

11.1 What is A RTOS?

Dedicated FreeRTOS developers have been working in close partnership with the world's leading chip companies for more than 15 years to provide you market leading, commercial grade, and completely free high quality **RTOS** and tools ...but what is an RTOS?

This page starts by defining an operating system, then refines this to define a real time operating system (RTOS), then refines this once more to define a real timer kernel (or real time executive).

See also the FAQ item "why an RTOS" for information on when and why it can be useful to use an RTOS in your embedded systems software design.

11.2 What is a General Purpose Operating System?

An operating system is a computer program that supports a computer's basic functions, and provides services to other programs (or *applications*) that run on the computer. The applications provide the functionality that the user of the computer wants or needs. The services provided by the operating system make writing the applications faster, simpler, and more maintainable. If you are reading this web page, then you are using a web browser (the application program that provides the functionality you are interested in), which will itself be running in an environment provided by an operating system.

11.3 What is an RTOS?

Most operating systems appear to allow multiple programs to execute at the same time. This is called multi-tasking. In reality, each processor core can only be running a single thread of execution at any given point in time. A part of the operating system called the scheduler is responsible for deciding which program to run when, and provides the illusion of simultaneous execution by rapidly switching between each program.

The type of an operating system is defined by how the scheduler decides which program to run when. For example, the scheduler used in a multi user operating system (such as Unix) will ensure each user gets a fair amount of the processing time. As another example, the scheduler in a desk top operating system (such as Windows) will try and ensure the computer remains responsive to its user. [Note: FreeRTOS is not a big operating system, nor is it designed to run on a desktop computer class processor, I use these examples purely because they are systems readers will be familiar with]

The scheduler in a Real Time Operating System (RTOS) is designed to provide a predictable (normally described as *deterministic*) execution pattern. This is particularly of interest to embedded systems as embedded systems often have real time requirements. A real time requirements is one that specifies that the embedded system must respond to a certain event within a strictly defined time (the *deadline*). A guarantee to meet real time requirements can only be made if the behaviour of the operating system's scheduler can be predicted (and is therefore deterministic).

Traditional real time schedulers, such as the scheduler used in FreeRTOS, achieve determinism by allowing the user to assign a priority to each thread of execution. The scheduler then uses the priority to know which thread of execution to run next. In FreeRTOS, a thread of execution is called a *task*.

11.4 What is FreeRTOS?

FreeRTOS is a class of RTOS that is designed to be small enough to run on a microcontroller - although its use is not limited to microcontroller applications.

A microcontroller is a small and resource constrained processor that incorporates, on a single chip, the processor itself, read only memory (ROM or Flash) to hold the program to be executed, and the random access memory (RAM) needed by the programs it executes. Typically the program is executed directly from the read only memory.

Microcontrollers are used in deeply embedded applications (those applications where you never actually see the processors themselves, or the software they are running) that normally have a very specific and dedicated job to do. The size constraints, and dedicated end application nature, rarely warrant the use of a full RTOS implementation - or indeed make the use of a full RTOS implementation possible. FreeRTOS therefore provides the core real time scheduling functionality, inter-task communication, timing and synchronisation primitives only. This means it is more accurately described as a real time kernel, or real time executive. Additional functionality, such as a command console interface, or networking stacks, can then be included with add-on components.

11.5 RTOS

A Real Time Operating System (RTOS) is an operating system developed for real-time embedded applications evolved around processors or controllers. It allows priorities to be changed instantly and data to be processed rapidly enough that the results may be used in response to another process taking place at the same time, as in transaction processing [1]. It has the ability to immediately respond in a predetermined and predictable way to external events. Overall a mode of action and reaction by RTOS and application software handles the entire embedded application. Let us consider the role of RTOS in the mobile phone. A cell phone has several features like call processing, notifying a waiting call, maintain a phone directory, messages and other utilities like web browsers, calculator, games, apps etc. RTOS handles each of these features as a separate task. Suppose a user is playing game on his cell phone and a call arrives, immediately the caller's ID starts flashing on the screen. After completion of the call, the user can resume the game from the level/ point it has got suspended. It is observed that in this case RTOS handled the tasks using priorities and performed multitasking using context switching and scheduler. This paper gives quick overview of complete architecture of RTOS. The paper also discusses the various aspects and issues of design and implementation of RTOS for controllers and processors.

11.5.1 Task

A task also called a thread is a simple program that thinks it has the CPU all to itself. The design process for a real time application involves splitting the work to be done in different tasks. Each task is assigned a priority, its own set of CPU registers and its own stack area. Each task is typically in an infinite loop that can be in any of the five states: Dormant, Ready, Running, Waiting and Interrupted. The first step in designing of the RTOS is to decide on the number of tasks that the CPU of the embedded appication can handle.

11.5.2 Multitasking

It is the process of scheduling and switching the CPU between the several tasks. It maximizes the CPU utilization and makes programming efficient for designing and debugging [2]. Processors and Controllers with inbuilt pipe lining architecture and can further increase the speed of the execution of a particular task.

11.5.3 Context Switch

When a multitasking kernel decides to run different tasks, it simply saves the current task's context storage area (Task Stack). Once this operation is performed, new task's context is restored from its storage area (Stack) and execution of new task's code is resumed.

11.5.4 Task Priority

Priority is assigned to a task depending on its importance such as static priority. Static priority means priority of the task does not change during run time. Dynamic priority means the priority of the task can change during run time.

11.5.5 Assigning Priority

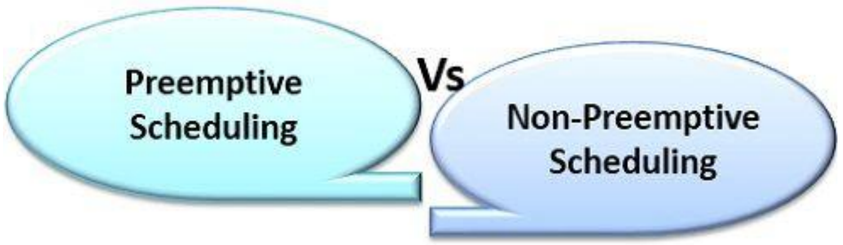
Assigning priority to task is required to know which task is more important than other to facilitate scheduling.

11.5.6 Ready Table

Each task is assigned a unique priority level between highest priority and lowest priority. Lowest priority is assigned to the idle task. Each task ready to run is placed in the ready list. Task priorities does not depend on maximum number of tasks. Tasks priorities can also be grouped. When task is ready to run, it sets its corresponding bit in the ready table declaring its ready status. Keeping ready list allows reducing the amount of RAM needed, when application requires few task priorities. Thus, by checking the ready status from the ready table and priority scheduler decides which task will run.

11.6 Preemptive vs Non-Preemptive Systems

11.6.1 Difference Between Preemptive and Non-Preemptive Scheduling in Operating Systems



11.6.2 Preemptive Scheduling

Preemptive Scheduling means once a process started its execution, the currently running process can be paused for a short period of time to handle some other process of higher priority, it means we can preempt the control of CPU from one process to another if required.

A computer system implementing this supports multi-tasking as it gives the user impression that the user can work on multiple processes.

It is practical because if some process of higher priority is encountered then the current process can be paused to handle that process.

Examples:- SRTF, Priority, Round Robin, etc.

11.6.3 Non-Preemptive Scheduling

Non-Preemptive Scheduling means once a process starts its execution or the CPU is processing a specific process it cannot be halted or in other words we cannot preempt (take control) the CPU to some other process.

A computer system implementing this cannot support the execution of process in a multi task fashion. It executes all the processes in a sequential manner.

It is not practical as all processes are not of same priority and are not always known to the system in advance.

11.6.4 Difference between Preemptive and Non-Preemptive Scheduling in OS

Preemptive Scheduling	Non-Preemptive Scheduling
Processor can be preempted to execute a different process in the middle of execution of any current process.	Once Processor starts to execute a process it must finish it before executing the other. It cannot be paused in middle.
CPU utilization is more compared to Non-Preemptive Scheduling.	CPU utilization is less compared to Preemptive Scheduling.
Waiting time and Response time is less.	Waiting time and Response time is more.
The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.	When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.
If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.

11.7 Questions

5. What is an RTOS?
6. What Task Priority refer to in a RTOS?
7. What is the difference between the Preemptive and Non-Preemptive Systems?
8. What “context switch” is referring to?

12 References

No	Element
1	https://www.wisc-online.com/learn/career-clusters/stem/cis6208/digital-to-analog-converters
2	http://cs.curs.pub.ro/wiki/pm/lab/lab2
3	https://www.arlabs.com/help.html
4	http://www.dauniv.ac.in/downloads/EmbsysRevEd_PPTs/Chap_2Lesson17EmsysNew.pdf
5	https://www.youtube.com/watch?v=ELl3abwYQ90
6	https://www.youtube.com/watch?v=NBaB98yEW3s
7	https://www.wisc-online.com/learn/career-clusters/stem/cis6208/digital-to-analog-converters
8	http://cs.curs.pub.ro/wiki/pm/lab/lab2
9	https://www.arlabs.com/help.html