8 Introduction to *ADC, DAC and Timers*

1. Objectives

The aims of this experiment is Learned how to digitalize an analog signal, generate an analog signal from a digital value , and Understanding how to use timers with interrupts to obtain the needed resolution .

2.Overview

Analog-to-digital converters (ADCs) are needed in all those applications which, interfacing with the analogue world, exploit the digital processing of data. As digital processing is more and more gaining ground over analogue signal processing, the importance of ADCs correspondingly increases.

Digital signal processing systems are generally designed by considering analogue-to-digital converters (ADC's) as ideal components affected only by quantization and sampling errors. Viceversa, the effect of the ADC actual working conditions modifies the expected digital values and could compromise the effectiveness of the digital signal processing as a whole. ADC's are among the components that mostly influence metrological performance of digital measurement systems. ADC errors limit system dynamic, its frequency band and so on, by adding distortion and error effects to the output. Therefore, a deeper insight into the ADC characteristics is needed.

2.1 ADC

An analog-to-digital converter (abbreviated ADC, A/D or A to D) is a device that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude. The conversion involves quantization of the input, so it necessarily introduces a small amount of error. The inverse operation is performed by a digital-to-analog converter (DAC). Instead of doing a single conversion, an ADC often performs the conversions ("samples" the input) periodically. The result is a sequence of digital values that have converted a continuous-time and continuous-amplitude analog signal to a discrete-time and discrete-amplitude digital signal.

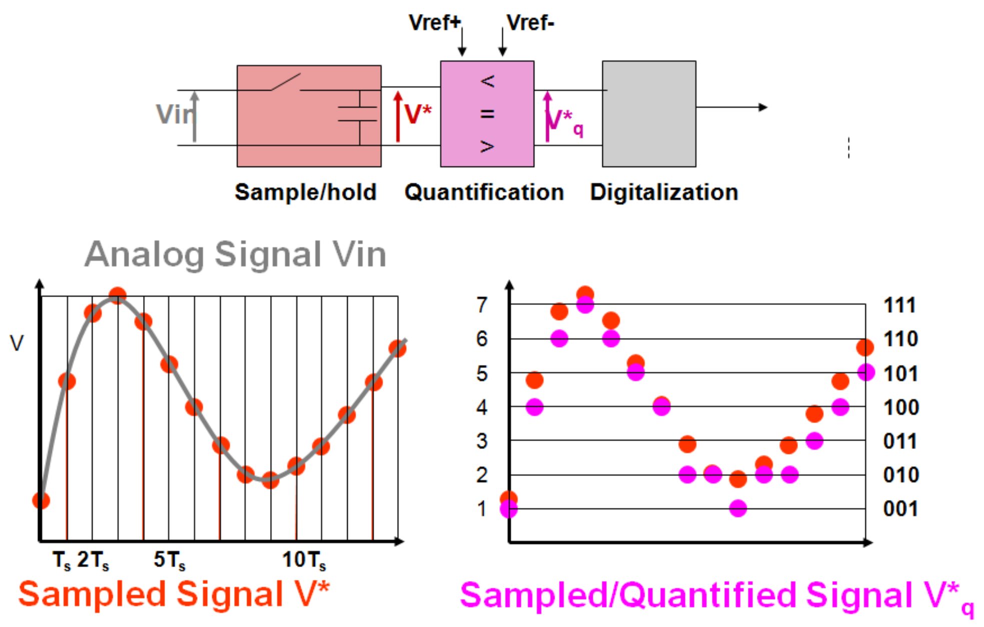


Figure 2.1 ADC: From the Analog Signal to the Digital Value

A successive-approximation ADC uses a comparator to successively narrow a range that contains the input voltage. At each successive step, the converter compares the input voltage to the output of an internal digital to analog converter which might represent the midpoint of a selected voltage range. At each step in this process, the approximation is stored in a successive approximation register (SAR) shows in figure.2.3. For example, consider an input voltage of 2 V and the initial range is 0 to 5 V. For the first step, the input 2 V is compared to 2.5 V (the midpoint of the 0–5 V range). The comparator reports that the input voltage is less than 2.5 V, so the SAR is updated to narrow the range to 0–2.5 V. For the second step, the input voltage is compared to 1.25 V (midpoint of 0–2.5). The comparator reports the input voltage is above 1.25 V, so the SAR is updated to reflect the input voltage is in the range 1.25 -2.5 V, and so on. The steps are continued until the desired resolution is reached.

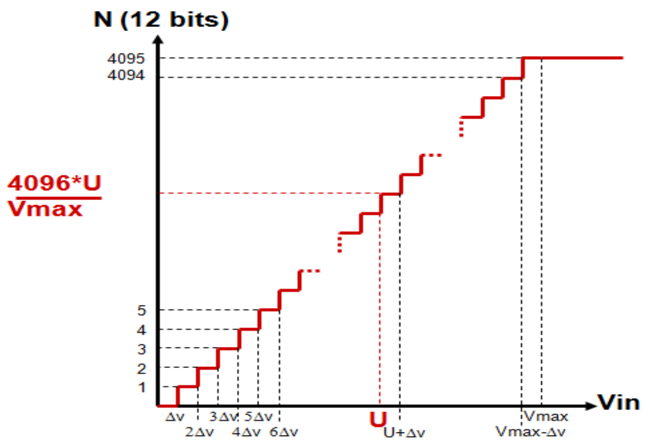


Figure 2.2: 12 bits ADC, resolution DV = Vmax/4096

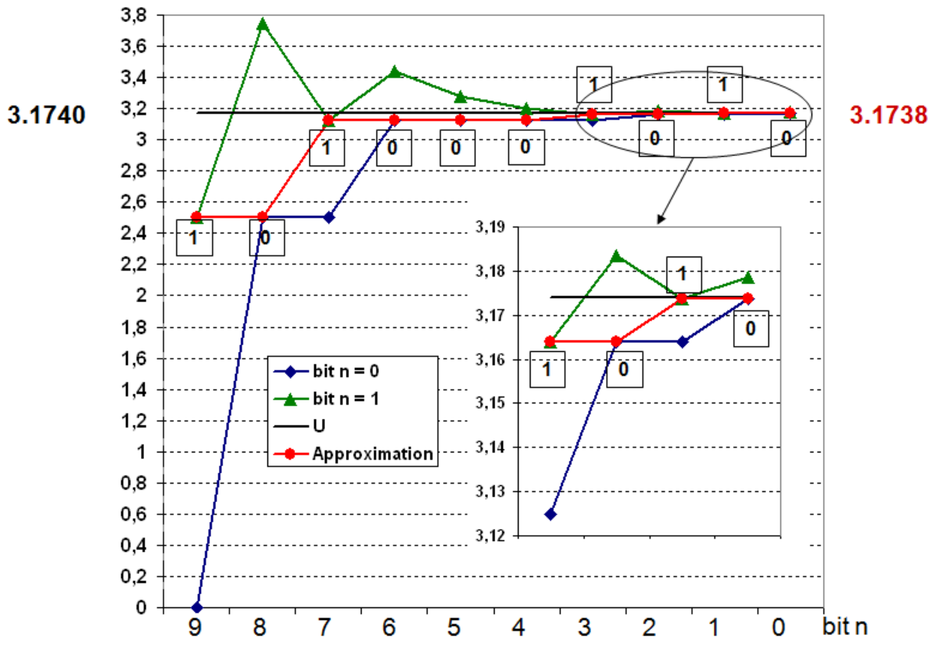


Figure 2.3 Successive approximation register (SAR)

2.2 DAC

In electronics, a digital-to-analog converter (DAC or D-to-A) is a device that converts a digital (usually binary) code to an analog signal (current, voltage, or electric charge). An analog-to-digital converter (ADC) performs the reverse operation. Signals are easily stored and transmitted in digital form, but a DAC is needed for the signal to be recognized by human senses or other non-digital systems.

A common use of digital-to-analog converters is generation of audio signals from digital information in music players. Digital video signals are converted to analog in televisions and mobile phones to display colors and shades. Digital-to-analog conversion can degrade a signal, so conversion details are normally chosen so that the errors are negligible.

Due to cost and the need for matched components, DACs are almost exclusively manufactured on integrated circuits (ICs). There are many DAC architectures which have different advantages and disadvantages. The suitability of a particular DAC for an application is determined by a variety of measurements including speed and resolution.

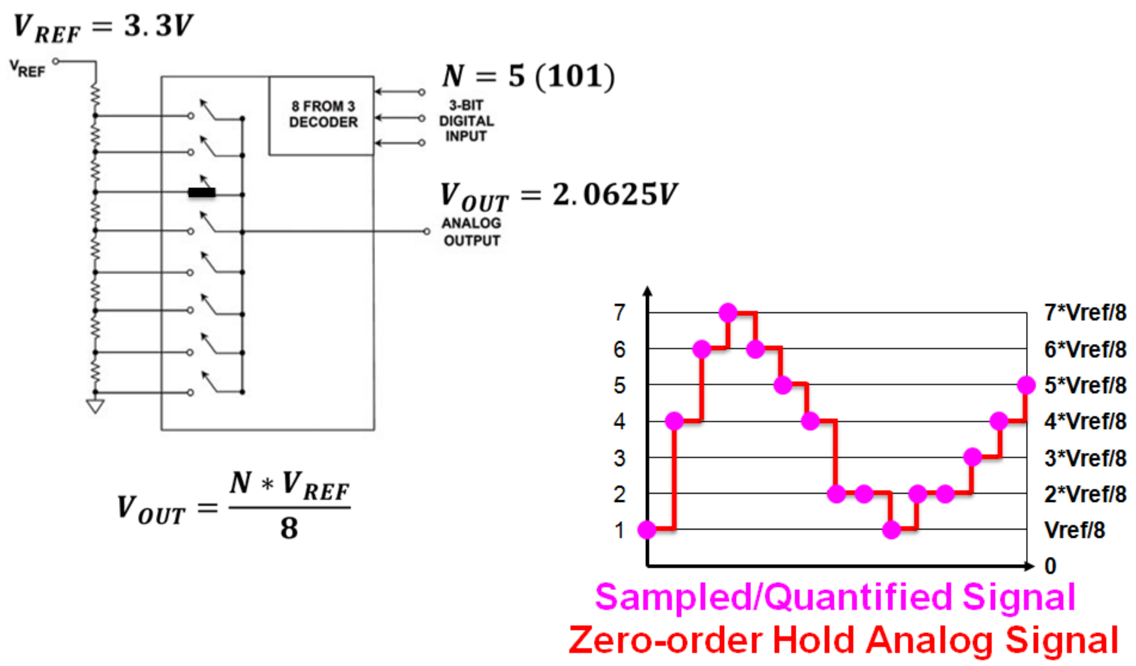
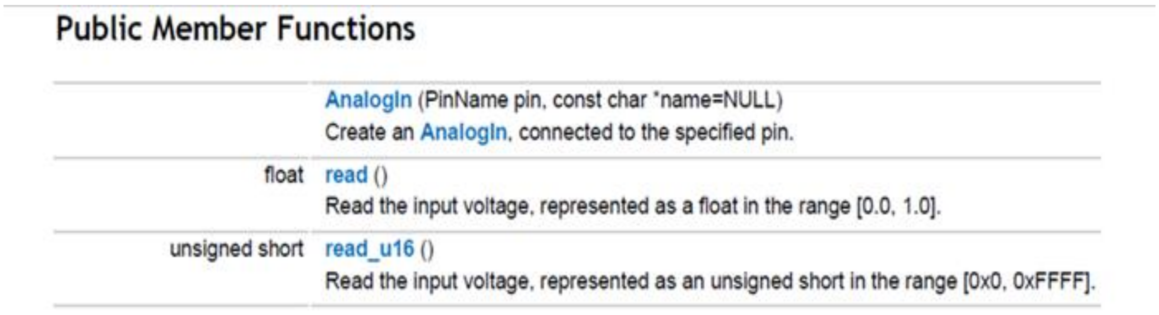


Figure 2.4 DAC: From the Digital Value to the Analog Signal

2.3 **MBED Functions of the AnalogIn Class**

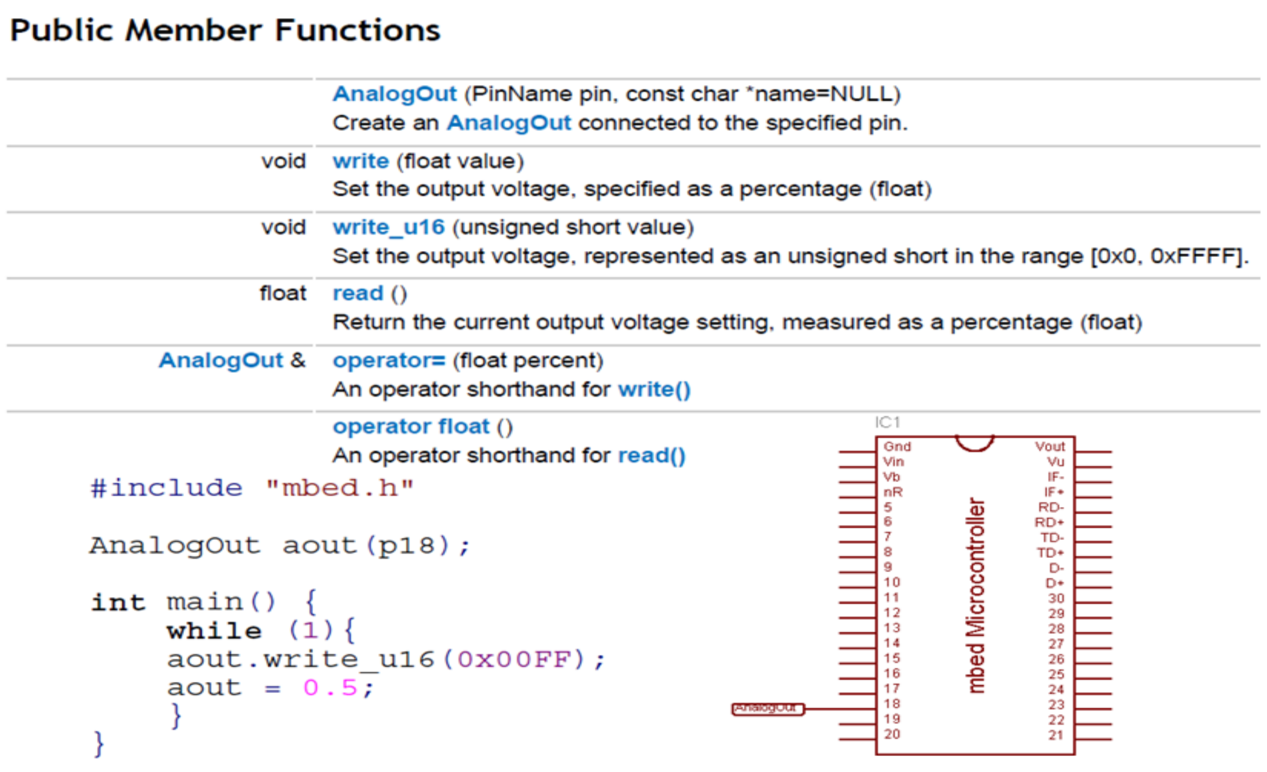
Figure 2.5 shows MBED analog Read functions, where figure 2.6 shows the write functions.



Public Member Functions

	AnalogIn (PinName pin, const char *name=NULL) Create an AnalogIn, connected to the specified pin.
float	read () Read the input voltage, represented as a float in the range [0.0, 1.0].
unsigned short	read_u16 () Read the input voltage, represented as an unsigned short in the range [0x0, 0xFFFF].

Figure 2.4 AnalogRead Functions



Public Member Functions

	AnalogOut (PinName pin, const char *name=NULL) Create an **AnalogOut** connected to the specified pin.
void	**write** (float value) Set the output voltage, specified as a percentage (float)
void	**write_u16** (unsigned short value) Set the output voltage, represented as an unsigned short in the range [0x0, 0xFFFF].
float	**read** () Return the current output voltage setting, measured as a percentage (float)
AnalogOut &	**operator=** (float percent) An operator shorthand for **write()**
	operator float () An operator shorthand for **read()**

```
#include "mbed.h"

AnalogOut aout(p18);

int main() {
    while (1){
    aout.write_u16(0x00FF);
    aout = 0.5;
    }
}
```

Figure 2.6 AnalogWrite Functions

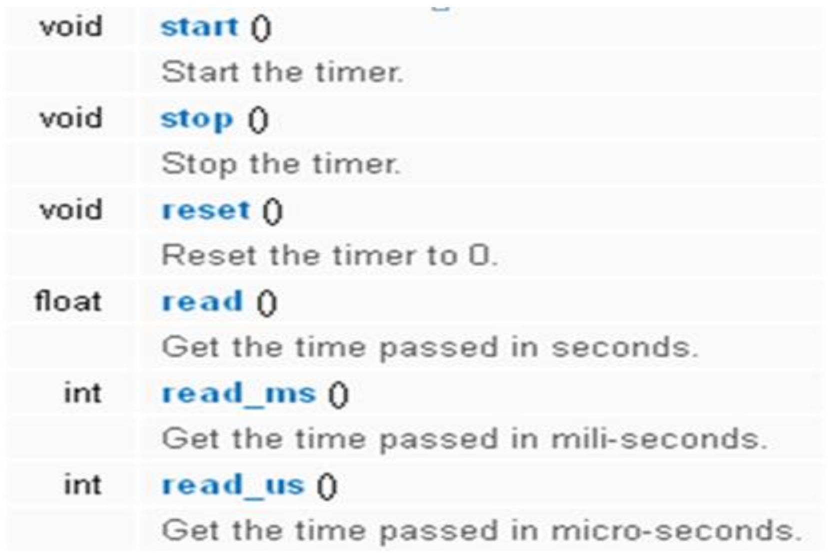
2.4 Timers and interrupts

Many embedded systems need high precision timing control and the ability to respond urgently to critical requests. For example an automotive system needs to be able to respond rapidly to a crash detection sensor in order to activate the passenger airbag.

Interrupts in embedded systems can be thought of as functions which are called by specific events rather than directly in code. Interrupts allow code execution to be halted while another, higher priority section of software executes. ISR can be programmed to execute on timed events or by events that occur externally in hardware.

The Timer interface is used to create, start, stop and read a timer for measuring small times (between microseconds and seconds). Figure 2.7 shows public member functions of the Timer object to perform scheduled programming.

Hardware interrupts: Microprocessors can be set up to perform specific tasks when hardware events are incident. This allows the main code to run and perform its tasks, and only jump to certain subroutines or functions when something physical happens. Figure 2.8 shows public member functions of the external interrupt.



void	**start** ()
	Start the timer.
void	**stop** ()
	Stop the timer.
void	**reset** ()
	Reset the timer to 0.
float	**read** ()
	Get the time passed in seconds.
int	**read_ms** ()
	Get the time passed in mili-seconds.
int	**read_us** ()
	Get the time passed in micro-seconds.

Figure 2.7: timer object

InterruptIn	**A digital interrupt input, used to call a function on a rising or falling edge**
Functions	**Usage**
InterruptIn	Create an InterruptIn connected to the specified pin
rise	Attach a function to call when a rising edge occurs on the input
rise	Attach a member function to call when a rising edge occurs on the input
fall	Attach a function to call when a falling edge occurs on the input
fall	Attach a member function to call when a falling edge occurs on the input
mode	Set the input pin mode

Figure 2.8 functions of the external interrupt

With scheduled programming, we need to be careful with the amount of code and how long it takes to execute. For example, if we need to run a task every 1 ms, that task must take less than 1 ms second to execute, otherwise the timing would overrun and the system would go out of sync.

3.Procedure

3.1 **Reading an Analog Input**

3.1.1 Connect input as per connections given in Figure 3.1

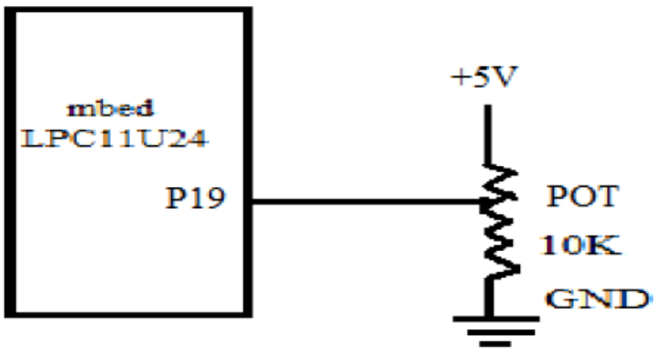


Figure 3.1: connect an analog input to mbed

Remark:- Range for analog inputs data should be in the range of 0.1 to 1.0 Volts only.

3.1.2 Write the desired C++ program and compile it using mbed compiler.

```
;Program written using C++ for mbed LPC11U24 board
; Analog input given at GPIO19
;Output seen at all 4 onboard LEDs

#include "mbed.h"
 BusOut ledout(LED1, LED2, LED3, LED4);
AnalogIn ain(p19);

int main()
 {
   float c = ain.read();
   while(1){
     c= ain.read();
   if (c<0.1)
    {       ledout = 0;        }
  else if (c<0.2)
    {       ledout = 1;        }
   else if (c<0.3)
   {       ledout = 2;        }
   else if (c<0.4)
   {       ledout = 3;        }
 else if (c<0.5)
   {       ledout = 4;        }
   else if (c<0.6)
   {       ledout = 5;        }
   else if (c<0.7)
   {       ledout = 6;        }
   else if (c<0.8)
   {       ledout = 7;        }
   else if (c<0.9)
   {       ledout = 8;        }
   else if (c<1.0)
   {       ledout = 9;        }
  }}
```

3.2 **Using Timer Object**

Create a square wave output using scheduled programming and verify the timing accuracy with an oscilloscope. Write the following C++ program and compile it using mbed compiler

```
#include "mbed.h"
Timer timer1; // define timer object
DigitalOut output1(X); // digital output
//You have to find a digital output pin and
assign it to variable X
void task1(void); // task function prototype
//*** main code
int main() {
timer1.start(); // start timer counting
while(1) {
if (timer1.read_ms()>=200) // read time in ms
{
task1(); // call task function
timer1.reset(); // reset timer
}
}
}
void task1(void){ // task function
output1=!output1; // toggle output
```

3.3 Using Timeout interface

The Timeout interface is used to setup an interrupt to call a function after a specified delay. Write a simple program to setup a Timeout to invert an LED after a given time

```
#include "mbed.h"

Timeout flipper;
DigitalOut led1(LED1);
DigitalOut led2(LED2);

void flip() {
    led2 = !led2;
}

int main() {
    led2 = 1;
    flipper.attach(&flip, 2.0); // setup flipper t
o call flip after 2 seconds

    // spin in a main loop. flipper will interrupt
 it to call flip
    while(1) {
        led1 = !led1;
        wait(0.2);
    }
}
```

3.4 Using Ticker interface

The Ticker interface is used to setup a recurring interrupt to repeatedly call a function at a specified rate. Write a simple program to setup a Ticker to invert an LED repeatedly.

```
#include "mbed.h"

Ticker flipper;
DigitalOut led1(LED1);
DigitalOut led2(LED2);

void flip() {
    led2 = !led2;
}

int main() {
    led2 = 1;
    flipper.attach(&flip, 2.0); // the address of
the function to be attached (flip) and the interva
l (2 seconds)

    // spin in a main loop. flipper will interrupt
 it to call flip
    while(1) {
        led1 = !led1;
        wait(0.2);
    }
```

3.5 **External interrupts on the mbed**

Use the mbed InterruptIn library to toggle an LED whenever a digital pushbutton input goes high.

```
#include "mbed.h"

InterruptIn button(p5); // Interrupt on
digital pushbutton input p5
DigitalOut led1(LED1); // digital out to
LED1
void toggle(void); // function prototype
int main() {
button.rise(&toggle); // attach the
address of the toggle
} //
function to the rising edge
void toggle() {
led1=!led1;
}
```

TODO: Combine the timer and hardware interrupt programs to show that a scheduled program and an event driven program can operate together. Flash a LED at rate defined by a timer and allow a hardware interrupt to write your name on LCD when a pushbutton is pressed.