Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

School of Electrical Engineering

Laboratory Manual

MCOA506P

Real-Time Embedded Systems Lab

S No	Course Objectives
1	Acquire programming and hardware skills in typical embedded system development cycle
2	Demonstrate the different embedded system design concepts using cortex-M microcontroller

S No	Course Outcomes
1	Use modern software and hardware development tools for embedded system design
2	Develop embedded system to solve real world control and automation problems

List of Indicative Experiments

S No	Experiment Title
1	Implementation of Simple C Programming Concepts in IDE: Bitwise Operations, Control Blocks and Functions
2	GPIO Programming: Interfacing Input and Output Devices
3	Study of Polling and Interrupts using a Cortex-M Microcontroller
4	PWM based Permanent Magnet DC Motor Speed Control
5	Current Measurement using ADC in STM32F407 Microcontroller
6	I2C Communication based LCD Interface using an 8-bit I/O Expander
7	Interfacing a 3-axis MEMS Accelerometer using SPI Communication Protocol in ARM Cortex-M Microcontroller
8	Implementation of Controller Area Network (CAN) using Cortex-M Microcontroller
9	Modbus Communication between Commercial Meter and Cortex M Microcontroller
10	Data Acquisition System Design and FFT Analysis using ARM Cortex-M4 Microcontroller
11	Implementation of Motor Position and Speed Measurement using Cortex-M Microcontroller
12	Pre-Emptive Task Scheduling using RTOS Kernel for Multitasking Applications

Implementation of Simple C Programming Concepts in IDE: Bitwise Operations, Control Blocks and Functions

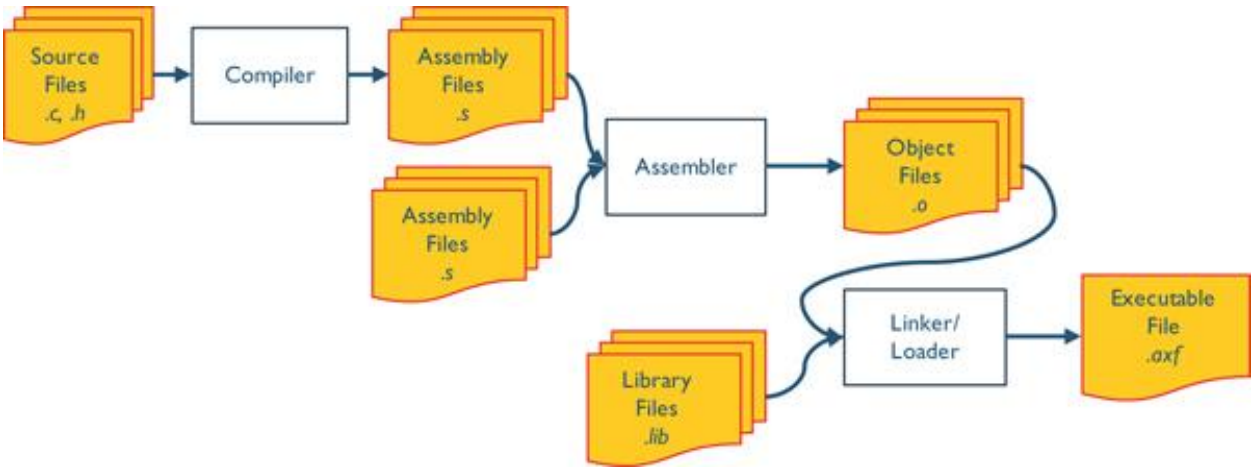
1. Aim:

To develop ‘C’ programs using STM32CubeIDE in the context of embedded systems.

2. Introduction:

Three major types of tools used for embedded software development are the program build toolchain, the programmer, and the debugger. The program build toolchain translates a program into a format that the MCU can understand, stored in an executable file. The programmer programs that information into the MCU’s program memory. This memory is nonvolatile (typically Flash ROM), so it will remain even after power is removed. The debugger enables the developer to control program execution and examine the program state (e.g., current instruction, values of processor registers, and data memory) as it runs on the processor. These tools are often grouped together in a single integrated development environment (IDE) to simplify development.

The program build toolchain is shown below.



The individual blocks of program build toolchain are explained below.

2.1 Cross Compilers:

Executable files created by compilers are usually platform dependent. An executable file compiled for one type of microprocessors, such as ARM Cortex-M3, cannot directly run on a platform with a different kind of microprocessors that support a different set of machine instructions, such as PIC or Atmel AVR microcontrollers. When we migrate a program written in a high-level language to a processor of a different instruction set, we usually have to modify and recompile the source programs for the new target platform. The binary machine program follows a standard called executable and linkable format (ELF). ELF, as its name suggests, provides two interfaces to binary files:

(i) a linkable interface that is used at static link time to combine multiple files when compiling and building a program.

(ii) an executable interface that is utilized at runtime to create a process image in memory when a program is loaded into memory and then executed.

2.2 Assembly Code:

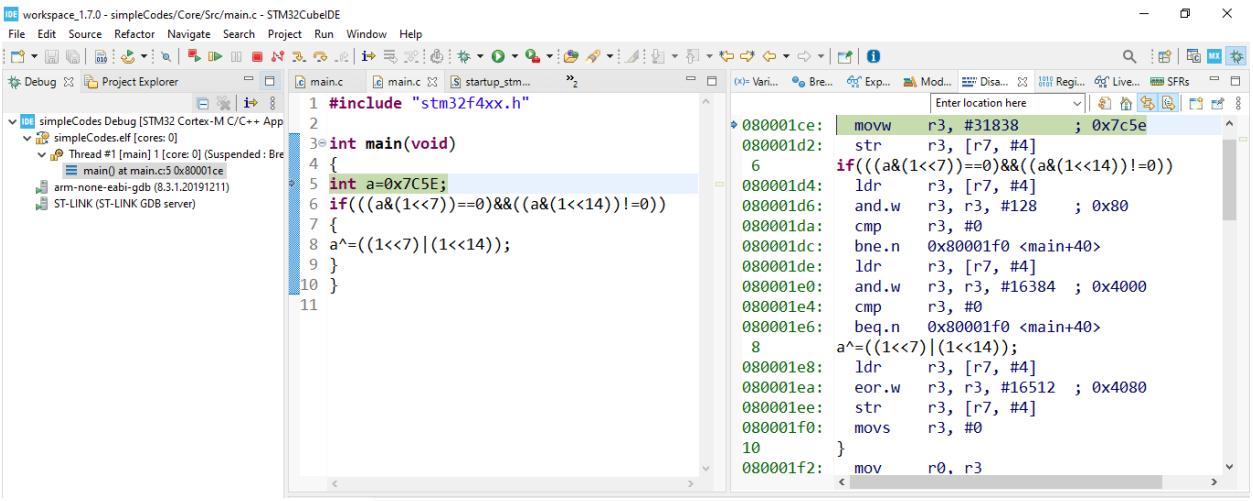
To optimize our application for maximum execution speed or minimum memory size, writing pieces of our code in assembly language is one approach. The goal of the software is to combine two 8-bit variables into one 16-bit variable. If you cannot see the bug, look up the precedence of the two operators << and +.

```
combine                                    uint32_t combine(
  MOV  r2,r0      ;R0=msb                    uint8_t msb,
  ADD  r3,r1,#0x08 ;lsb+8                    uint8_t lsb){
  LSL  r0,r2,r3   ;msb<<(8+lsb)            return msb<<8 + lsb;
  BX   lr                                  }
```

Good understanding of the assembly code generated by our compiler makes us better programmers.

2.3 Debugger:

Debugger enables the developer to control program execution and examine the program state (e.g., current instruction, values of processor registers, and data memory) as it runs on the processor. It allows user to observe and control program execution, variables, and processor registers.



3. Variables:

Obvious considerations for creating a variable are the size and format of the data. Scope of the variable defines which software modules can access the data. A system is easier to design (because the modules are smaller and simpler), easier to change (because code can be reused), and easier to verify (because interactions between modules are well-defined) when we limit the scope of our variables. Since modules are not completely independent, we need a mechanism to transfer information from one to another.

3.1 Declarations:

Describing a variable involves two actions. The first action is declaring its type and the second action is defining it in memory (reserving a place for it). Although both of these may be involved, we refer to the C construct that accomplishes them as a *declaration*. If the declaration is preceded by **extern,** it only declares the type of the variables, without reserving space for them. Here, the definition must exist in another source file. Failure to do so, will result in an unresolved reference error at link time.

3.2 Typical 'C' data types:

Data Type	Size (bits)	Alignment	Data Range
bool	8	byte	0 or 1. Bits 1 – 7 are ignored
char	8	byte	-128 to 127 (signed), or 0 to 255 (unsigned)
int	32	word	-2,147,483,648 to 2,147,483,647 (signed), or 0 to 4,294,967,296 (unsigned)
short int	16	halfword	-32,768 to 32,767 (signed), or 0 to 65,536 (unsigned)
long int	32	word	same as int
long long	64	word	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (signed), or 0 to 18,446,744,073,709,551,616 (unsigned)
float	32	word	+/- (1.4023×10^{-45} to $3.4028 \times 10^{+38}$), always signed
double	64	word	+/- (4.9406×10^{-324} to 1.7977×10^{308}), always signed
long double	96	word	Enormous range
pointer	32	word	0 to 4,294,967,296

3.3 Variable storage classes:

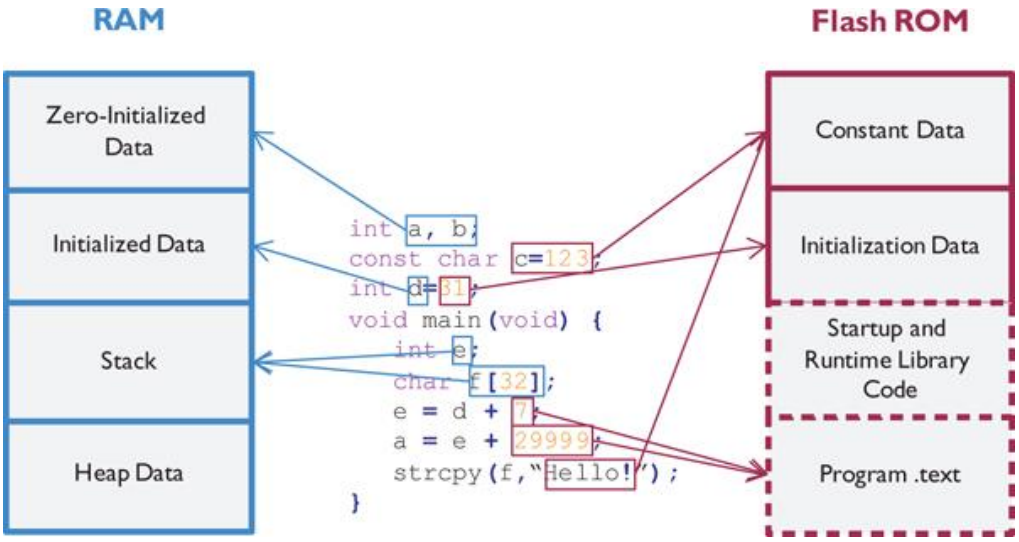
Modifier	Comment
auto	automatic, allocated on the stack
extern	defined in some other program file
static	permanently allocated
register	attempt to implement an automatic using a register instead of on the stack
volatile	can change value by means other than the current program
const	fixed value, defined in the source code and can not be changed during execution
unsigned	range starts with 0 includes only positive values
signed	range includes both negative and positive values

3.4 Memory Allocation:

A *variable* is a named object that resides in RAM memory and is capable of being examined and modified. It is used to hold information critical to the operation of the A *constant* is a named object that resides in memory (usually in ROM) and is only capable of being examined. *literal* is the direct specification of a number character or string. The difference between a literal and a constant is that constants are given names so that they can be accessed more than once.

```
short MyVariable;          /* variable allows read/write access */
const short MyConstant=50; /* constant allows only read access */
#define fifty 50
```

A program's memory requirement:



3.5 Global vs. Static Variables:

In an embedded system we normally wish to place all variables in RAM and constants in ROM.

The fact that these types of variables exist in permanently reserved memory means that static variables exist for the entire life of the program. When the power is first applied to an embedded computer, the values in its RAM are usually undefined. Therefore, initializing global variables requires special run-time software consideration. A **static global** is very similar to a regular global. In both cases, the variable is defined in RAM permanently. The assembly language access is identical. **The only difference is the scope. Static global** cannot be referenced by modules in other files.

3.6 Static Variables:

Different from local variables, a *static* C variable has a lifetime over the entire program runtime.

It is initialized only once at the compiling time no matter how many times this function is called.

A static variable can be either global or local. A static local variable can only be available within the scope of the function in which this variable is declared. A static global variable can only be accessed within the source file in which this variable is declared.

3.7 Static Global Variables:

Always avoid global variables. If a variable is accessed by more than one subroutine within a source file, then it can be declared as global static variable.

```
static int counter = 0;

void increase(void){
  counter++;
}

void decrease(void){
  counter--;
}
```

3.8 Static Local Variables:

All static variables are allocated in the data memory. A local variable is often stored in a register or the heap region of the data memory. A static variable is always loaded from memory first and then is stored back to the memory before exiting the subroutine. Therefore, if the subroutine is called again, the static variable keeps its previous value, instead of its initial value. The initialization is carried out at compile time instead of at runtime. The variable is defined in the data region with an initial value. No matter how many times subroutine runs, the variable is never initialized.

Example of a local non-static variable:

C Program	Assembly Program

`int foo();`	`AREA static_demo, CODE`
	`EXPORT __main`
	`ALIGN`
	`ENTRY`

`int main(void) {`
`  int y;`
`  y = foo(); // y = 6`
`  y = foo(); // y = 6`
`  y = foo(); // y = 6`
`  while(1);`
`}`
	`__main PROC`
	`BL foo ; r0 = 6`
	`BL foo ; r0 = 6`
	`BL foo ; r0 = 6`
	`stop B stop`
	`ENDP`

`int foo() {`
`  int x = 5; // x is a local variable`
`  x = x + 1;`
`  return(x)`
`}`
	`foo PROC`
	`MOV r0, #5`
	`ADD r0, r0, #1`
	`BX lr`
	`ENDP`
	`END`

Example of a local static variable:

C Program	Assembly Program

`int foo();`	`AREA myData, DATA`
	`ALIGN`
	`// Reserve space for x`
	`x DCD 5`
	
	`AREA static_demo, CODE`
	`EXPORT __main`
	`ALIGN`
	`ENTRY`

`int main(void) {`
`  int y;`
`  y = foo(); // y = 6`
`  y = foo(); // y = 7`
`  y = foo(); // y = 8`
`  while(1);`
`}`
	`__main PROC`
	`BL foo ; r0 = 6`
	`BL foo ; r0 = 7`
	`BL foo ; r0 = 8`
	`stop B stop`
	`ENDP`

`int foo() {`
 	
`  // local static variable`
`  // x is initialized only once`
`  static int x = 5;`
 	
`  x = x + 1;`
`  return(x)`
`}`
	`foo PROC`
	`; load address of x`
	`LDR r1, =x`
	`; load value of x`
	`LDR r0, [r1]`
	`ADD r0, r0, #1`
	`; save value of x`
	`STR r0, [r1]`
	`BX lr`
	`ENDP`
	`END`

3.9 External Storage Class:

The compiler knows an external variable by the keyword **extern** that must precede its declaration. Only global declarations can be designated extern and only globals in other modules can be referenced as external. This global can be referenced by any function from any file in the software system. You normally use **extern** when you want to refer to something that is *not* in the current translation unit, such as a variable that's defined in a library you will be linking to it.

This tells the compiler, **"There's a variable defined in another module called myGlobalVariable, of type integer. I want you to accept my attempts to access it, but don't allocate storage for it because another module has already done that."**

extern int myGlobalVariable;

3.10 Scope of a Variable:

The scope of local variables is the block in which they are declared. Local declarations must be grouped together before the first executable statement in the block-at the head of the block. If we declare a local variable with the same name as a global object or another local in a superior block, the new variable temporarily supersedes the higher level declarations

```
unsigned char x;   /* a regular global variable*/
void sub(void){
    x=1;
    {   unsigned char x;   /* a local variable*/
        x=2;
        {   unsigned char x;  /* a local variable*/
            x=3;
            PORTA=x;}
        PORTA=x;}
    PORTA=x;}
}
```

3.11 Volatile Variable:

When the compiler optimizes a C program, a hard-to-find hidden error is that the program mistakenly reuses the value of a variable stored in a register, instead of reloading it from memory

each time. To avoid such compilation error, the program should declare the variable as volatile, such as:

volatile int variable

A volatile variable is a variable that may be changed by an external input or an interrupt handler. Therefore, the processor should not use a register to cache this variable to avoid using stale data. The keyword volatile forces the compiler to generate an executable, which always loads the variable value from the memory whenever this variable is read, and always stores the variable in memory whenever it is written.

This code illustrates the necessity of declaring a variable **counter as volatile,** shared by two concurrently running tasks (**main function** and **SysTick_Handler**).

Main Program (*main.c*)

```
//volatile uint32_t counter; // correct
uint32_t counter;          // incorrect
extern void task();
extern void SysTick_Init();

int main(void) {
   counter = 10;
   SysTick_Init();
   while(counter != 0);  // Delay
   // Continue the task
   while(1);
}
```

Interrupt Service Routine (*isr.s*)

```
        AREA ISR, CODE, READONLY
        IMPORT counter
        ENTRY

SysTick_Handler PROC
        EXPORT SysTick_Handler
        LDR r1, =counter
        LDR r0, [r1]     ; load counter
        SUB r0, r0, #1   ; counter--
        STR r0, [r1]     ; save counter
        BX  LR           ; exit
        ENDP
        END
```

Here compiler observes that, after counter is initialized to 10, the value of the counter Variable is not modified directly by **main()** or indirectly by any subroutine called from **main().**

If counter is not declared as volatile

```
__main PROC
       LDR r1, =counter
       MOV r0, #10
       STR r0, [r1]
       BL  SysTick_Init

wait   CMP r0, #0 ; r0 does not hold
                  ; latest counter value
       BNE wait   ; Thus, a dead loop
stop   B   stop
       ENDP
```

If counter is declared as volatile

```
__main PROC
       LDR r1, =counter
       MOV r0, #10
       STR r0, [r1]
       BL  SysTick_Init

wait   LDR r1, =counter
       LDR r0, [r1]
       CMP r0, #0
       BNE wait
stop   B   stop
       ENDP
```

If the counter is not declared as volatile, the while loop is a dead loop. SysTick_Handler periodically decrements the counter and stores its value in memory. However, the main memory repeatedly checks register r0, without reloading the latest value of the counter from memory. If the counter is declared as volatile, the dead loop problem is avoided.

3.12 Memory Mapped Registers:

A 'C' program should declare any variable that represents the data of a memory-mapped I/O register as volatile. Memory-mapped I/0 has been widely used to access peripheral devices.

Data and control registers of external devices are mapped to specific memory addresses, and a program can use memory pointers to access these hardware registers.

```
volatile uint32_t *p = (uint32_t *) 0x60002400;
```

In sum, a variable should be declared as **volatile** to prevent the compiler from optimizing it away when (1) this variable is updated by external memory-mapped hardware, or (2) this variable is global and is changed by interrupt handlers or by multiple threads.

4.1 Bitwise Operators:

- Bitwise operators work naturally with embedded hardware by permitting the test and modification of individual signals coming into or going out of the embedded processor.
- These operators are intended for work at the hardware level – that is, with the registers and input or output ports on a target machine.
- When working at the bit level, often all of the bits in a word are important because they generally represent the state of some signal in the hardware of the machine.

Operator		Meaning	Description
Shift			
	≫	Logical shift right	Operand shifted positions are filled with 0's
	≪	Logical shift left	Operand shifted positions are filled with 0's
Logical			
	&	Bitwise AND	
	|	Bitwise inclusive OR	
	^	Bitwise exclusive OR	
	~	Bitwise negation	

```
// we are working with byte sized pieces in this example

unsigned char a = 0xF3;      // a = 1111 0011 – note this is not a negative number
unsigned char b – 0x54;      // b – 0101 0100 – note this is not a positive number

unsigned char c = a & b;     // c gets a AND b
                             // a 1111 0011
                             // b 0101 0100
                             // c 0101 0000

unsigned char d = a | b;     // d gets a OR b
                             // a 1111 0011
                             // b 0101 0100
                             // d 1111 0111

unsigned char e = a ^ b;     // e gets a XOR b
                             // a 1111 0011
                             // b 0101 0100
                             // e 1010 0111

unsigned char f – ~a;        // f gets ~a
                             // a 1111 0011
                             // f  0000 1100
```

Examples:

i. Using the Bitwise Operators to test for a Pattern within a Data Word

```
unsigned char getPort(void);                  // port access function prototype

unsigned char testPattern0 = 0x1A;            // testPattern0 = 0001 1010
unsigned char mask = 0x1E;                    // mask = 0001 1110
unsigned char portData = 0x0;                 // working variable

// assume port holds 1101 1011
portData = getPort();                         // read the port

if !((portShadow & mask) ^ testPattern0)      // will give a zero result if pattern present
{
    printf( “pattern present \n”;
}
```

ii. Starting with the word 0x7C5E, determine if bit 7 is reset and bit 14 is set. If so, complement bits 6 and 12.

```
1  #include "stm32f4xx.h"
2
3  int main(void)
4  {
5  int a=0x7C5E;
6  if(((a&(1<<7))==0)&&((a&(1<<14))!=0))
7  {
8  a^=((1<<6)|(1<<12));
9  }
10 }
11
```

Output:

```
Name : a
    Details:27678
    Default:27678
    Decimal:27678
    Hex:0x6c1e
    Binary:110110000011110
```

iii. Starting with the word 0xF0A6, apply ‘C’ concepts to reset bit 3, set bit 6, set bit 8, and reset bit 13.

```c
#include "stm32f4xx.h"

//Starting with the word 0xF0A6, apply ‘C’ concepts
//to reset bit 3, set bit 6, set bit 8, and reset bit 13.

int main(void)
{
int a=0xF0A6;
a |=((1<<6)|(1<<8));
a &=~((1<<2)|(1<<13));
}
```

Output:

```
Name : a
    Details:53730
    Default:53730
    Decimal:53730
    Hex:0xd1e2
    Binary:1101000111100010
```

4.2 Logical Vs Boolean Operators:

```c
#include "stm32f4xx.h"

short a,b,c,d,e;
int main(void)
{
    a=0x0F0F; b=0x0;
  c = a&b;  /* logical result c will be 0x0000 */
  d = a&&b; /* Boolean result d will be 0 (False) */
  e = a||b; /* Boolean result e will be 1 (true) */
  return 1;
}
```

4.3 Binary relational operators:

The binary relational operators take two number inputs and give a single Boolean result.

operator	*meaning*	*example*	*result*
==	equal	100 == 200	0 (false)
!=	not equal	100 != 200	1 (true)
<	less than	100 < 200	1 (true)
<=	less than or equal	100 <= 200	1 (true)
>	greater than	100 > 200	0 (false)
>=	greater than or equal	100 >= 200	0 (false)

```
short a,b;
void program(void){
  if(a==0) subfunction(); /* execute subfunction if a is zero */
  if(b=0) subfunction();  /* set b to zero, never execute subfunction */
}
```

Why do we use an infinite loop in embedded system software?

Because most embedded systems are intended to run for ever. If you have a device with an embedded computer, there is rarely a point at which you say "I have finished with that device - I will never, ever, want to use it again". You may want to put it to sleep until you next need it, but you need it in a state that you can wake it up and have it resume it's duties.

So "loop until not needed" is the same as "loop for ever". The device exists to serve, and will loop serving infinitely.

The application in a **bare-metal** (without an OS) system must run forever, because the is nothing 'after' the application. The common way to do this is to use an infinite loop.

In bare metal programming without a RTOS a super loop coupled with interrupt service routines is the normal. Hence the use of a forever loop with varying priority of interrupts for time critical events.

```
while (1)
{

    for(i=0;i<1024;i++)
    {
        ADC1->CR2 |= (1<<30);  // start the conversion
        ADC1->SR = 0;         // clear the status register
        //ADC1->CR2 |= (1<<30);  // start the conversion
        while (!(ADC1->SR & (1<<1)));  // wait for EOC flag to se
        ADC_VAL[i]=ADC1->DR;
    }

    for(j=0;j<1024;j++)
    {
        value[j]=ADC_VAL[j];
    }

}
```

Return and busy waiting statement:

The return statement is used within a function when a useful value is to be returned to the caller.

As its name implies, it does exactly nothing. Why to have a statement that serves no purpose?

```
while (!(ADC1->SR & (1<<1)));  // wait for EOC flag to set
```

5. Modules:

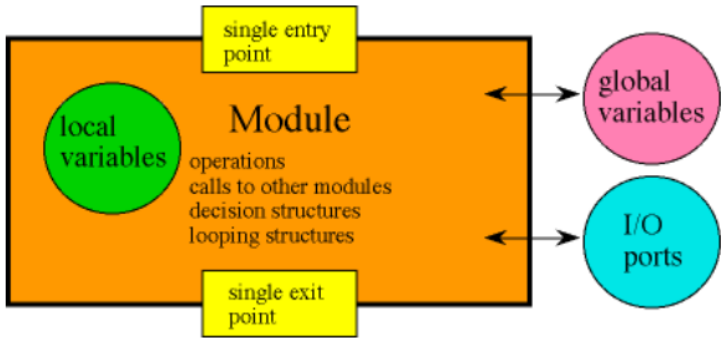
The key to effective software development is the appropriate division of a complex problem in modules. A module is a software task that takes inputs, operates in a well-defined way to create outputs. In C, functions are our way to create modules. A small module may be a single function. A medium-sized module may consist of a group of functions together with global data structures, collected in a single file. A large module may include multiple medium-sized modules.

Complex systems designed in a modular fashion are easier to debug because each module can be tested separately. Industry experts estimate that 50 to 90% of software development cost is spent. in maintenance. All five aspects of software maintenance:

- Correcting mistakes
- Adding new features
- Optimizing execution speed or program size
- Porting to new computers or operating systems

- Reconfiguring the software to solve a similar related problem

Above all are simplified by organizing the software system into modules, The approach is particularly useful when a task is large enough to require several programmers.



5.1 Function in C programming:

A mathematical function is a well-defined operation that translates a set of input values into a set of output values. In C, a function translates a set of input values into a single output value

Consider the function that converts temperature in degrees F into temperature in degrees C.

```
short FtoC(short TempF){
     short TempC;
     TempC=(5*(TempF-32))/9;    // conversion
return TempC;}
```

Similar to variables, C differentiates between a function declaration and a function definition
Similar to variables, C differentiates between a function declaration and a function definition
A declaration specifies the syntax (name and input/output parameters), whereas a function definition specifies the actual program to be executed when the function is called. A function must be declared (or defined) before it can be called.

5.2 Function declarations:

A function declaration begins with the type (format) of the return parameter. If there is no return parameter, then the type can be either specified as **void** or left blank. Next comes the function name, followed by the parameter list. In a function declaration we do not have to specify names for the input parameters, just their types. If there are no input parameters, then the type can be either specified as **void** or left blank.

```
//  declaration                   input            output
void Ritual(void);              // none             none
char InChar(void);              // none             8-bit
void OutChar(char);             // 8-bit            none
short InSDec(void);             // none             16-bit
void OutSDec(short);            // 16-bit           none
char Max(char,char);            // two 8-bit        8-bit
int EMax(int,int);              // two 32-bit       32-bit
void OutString(char*);          // pointer to 8-bit none
char *alloc(int);               // 32-bit           pointer to 8-bit
```

5.3 Function definitions:

Every function must be defined somewhere.

type Name(parameter list){
CompoundStatement
};

The **type** specifies the function return parameter. If there is no return parameter, we can use **void** or leave it blank. **Name** is the name of the function.

The **parameter list** is a list of zero or more names for the arguments that will be received by the function when it is called. Both the type and name of each input parameter is required. Compound statements may contain local declarations, simple statements, and other compound statements. It follows that functions may implement algorithms of any complexity and may be written in a structured style.

```
int add3(int z1, int z2, int z3){ int y;
    y=z1+z2+z3;
    return(y);}
```

5.4 Function calls:

A function is called by writing its name followed by a parenthesized list of argument expressions.

Name **(***parameter list***)**

Name is the name of the function to be called. The **parameter list** specifies the particular input parameters used in this call. Each input parameter may be as simple as a variable name or a constant, or it may be arbitrarily complex, including perhaps other function calls. The resulting value is pushed onto the stack where it is passed to the called function. C programs evaluate arguments from left to right, pushing them onto the stack in that order. On return, the return

parameter is located in **register R0**. The input parameters are removed from the stack at the end of the program.

```
y=add3(--counter,time+5,3);
```

5.5 Argument Passing:

With respect to the method by which arguments are passed, two types of subroutine calls are used in programming languages--*call by reference* and *call by value.* The *call by reference* method passes arguments in such a way that references (pointers) to the actual arguments are passed, instead of copies of the actual arguments themselves. Since C supports only one formal output parameter, we can implement additional output parameters using call by reference. The most important point to remember about passing arguments by value in C is that there is no connection between an actual argument and its source. Changes to the arguments made within a function, have no affect what so ever on the objects that might have supplied their values. C uses call by value that we can pass expressions, not just variables, as arguments.

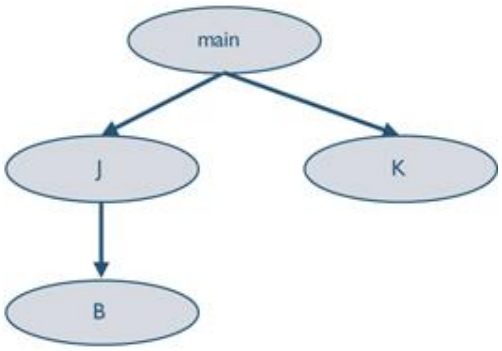
The function call graph is a diagram that shows possible function calls.

```
void B(void) {
    …
}

void J(void) {
    …
    B();
    …
}

void K(void) {
    …
}

void main(void) {
    …
    J();
    …
    K();
    …
}
```



Every C program must have a function called **main.** The main function for embedded systems never completes, unlike a program you might run on your personal computer or smart phone.

6. Structures:

A structure is a collection of variables that share a single name. With structures we specify the types and names of each of the elements or members of the structure. To access data stored in a structure, we must give both the name of the collection and the name of the element. The structure

must be declared before it can be used. The declaration specifies the tag name of the structure and the names and types of the individual members.

```
struct theport{
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;  // pointer to its address
   unsigned long volatile *ddr;}; // pointer to its direction reg
```

Declaration does not create any variables or allocate any space. To use a structure, we must define a global or local variable of this type. The above line defines the three variables and allocates 9 bytes for each of variable. Because the pointers will be 32-bit aligned the compiler will skip three bytes between **mask and addr**, so each object will occupy 12 bytes.

```
struct theport PortA,PortB,PortE;
```

To improve code reuse we can use **typedef** to actually create a new data type (called **port** in the example below) that behaves syntactically like **char, int, short** etc.

```
typedef struct {
   unsigned char mask;    // defines which bits are active
   unsigned long volatile *addr;        // address
   unsigned long volatile *ddr;}port_t; // direction reg
port_t PortA,PortB,PortE;
```

6.1 GPIO Port declaration of STM32F407:

```
654 typedef struct
655 {
656   __IO uint32_t MODER;    /*!< GPIO port mode register,
657   __IO uint32_t OTYPER;   /*!< GPIO port output type register,
658   __IO uint32_t OSPEEDR;  /*!< GPIO port output speed register,
659   __IO uint32_t PUPDR;    /*!< GPIO port pull-up/pull-down register,
660   __IO uint32_t IDR;      /*!< GPIO port input data register,
661   __IO uint32_t ODR;      /*!< GPIO port output data register,
662   __IO uint16_t BSRRL;    /*!< GPIO port bit set/reset low register,
663   __IO uint16_t BSRRH;    /*!< GPIO port bit set/reset high register,
664   __IO uint32_t LCKR;     /*!< GPIO port configuration lock register,
665   __IO uint32_t AFR[2];   /*!< GPIO alternate function registers,
666 } GPIO_TypeDef;
```

6.2 Accessing the members of the structure:

We need to specify both the structure name (name of the variable) and the member name when accessing information stored in a structure.

```
56              GPIOD->ODR ^= (0x1UL<<12);
```

Just like any variable, we can specify the initial value of a structure at the time of its definition.

```
port_t PortE={0x3F,
   (unsigned long volatile *)(0x400243FC),
   (unsigned long volatile *)(0x40024400)};
```

To place a structure in ROM, we define it as a global constant.

Example: Passing structures to functions

```
port_t PortE={0x3F,
   (unsigned long volatile *)(0x400243FC),
   (unsigned long volatile *)(0x40024400)};
port_t PortF={0x1F,
  (unsigned long volatile *)(0x400253FC),
  (unsigned long volatile *)(0x40025400)};
int MakeOutput(port_t *ppt){
  (*ppt->ddr) = ppt->mask; // make output
  return 1;}
int MakeInput(port_t *ppt){
  (*ppt->ddr )= 0x00; // make input
  return 1;}
unsigned char Input( port_t *ppt){
 return (*ppt->addr);}
void Output(port_t *ppt, unsigned char data){
 (*ppt->addr) = data;
}
int main(void){ unsigned char MyData;
  MakeInput(&PortE);
  MakeOutput(&PortF);
  Output(&PortF,0);
  MyData=Input(&PortE);
  return 1;}
```

7. Conclusion:

In this experiment, embedded ‘C’ programming concepts like variables, data types, operators, function, and structure are explored in greater detail with practice codes using STM32CubeIDE.

GPIO Programming: Interfacing Input and Output Devices

1. Aim:

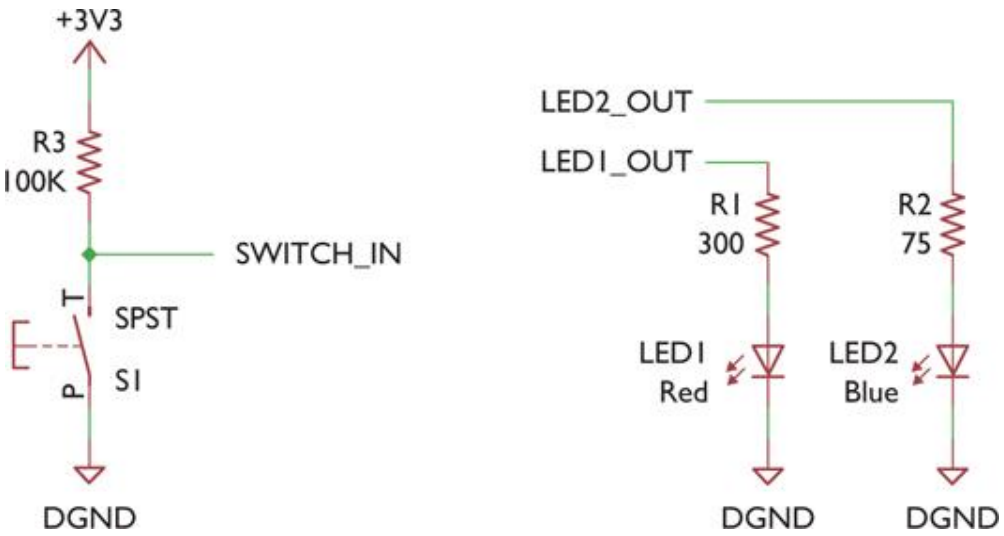
Write C programs to communicate with basic devices such as switches and light-emitting diodes (LEDs) using General Purpose Input /Output (GPIO) ports of STM32F407 microcontroller.

2. Introduction:

Embedded computers typically need to sense their environment and then control devices in response. Let's start with the basics by learning how to make an embedded computer flash light and read a switch by using a general purpose input/output (GPIO) port. A microcontroller unit (MCU) contains a central processing unit (CPU) that executes instructions. The CPU is surrounded by specialized hardware circuits called peripherals that interface with external devices or perform other functions for the MCU. Some of these peripherals are integrated into the MCU, whereas others may be added externally on the printed circuit board.

An input GPIO port bit lets us read a single bit digital value on an MCU pin. If we connect the pin to a switch (as with the signal SWITCH_IN), then the program can tell if the switch is open or closed. An output GPIO port bit enables the program to set an MCU pin to one of two voltage levels (e.g., either 3.3 V or 0 V). A program can light or extinguish an LED that is connected to this output pin (e.g., LED1_OUT or LED2_OUT).

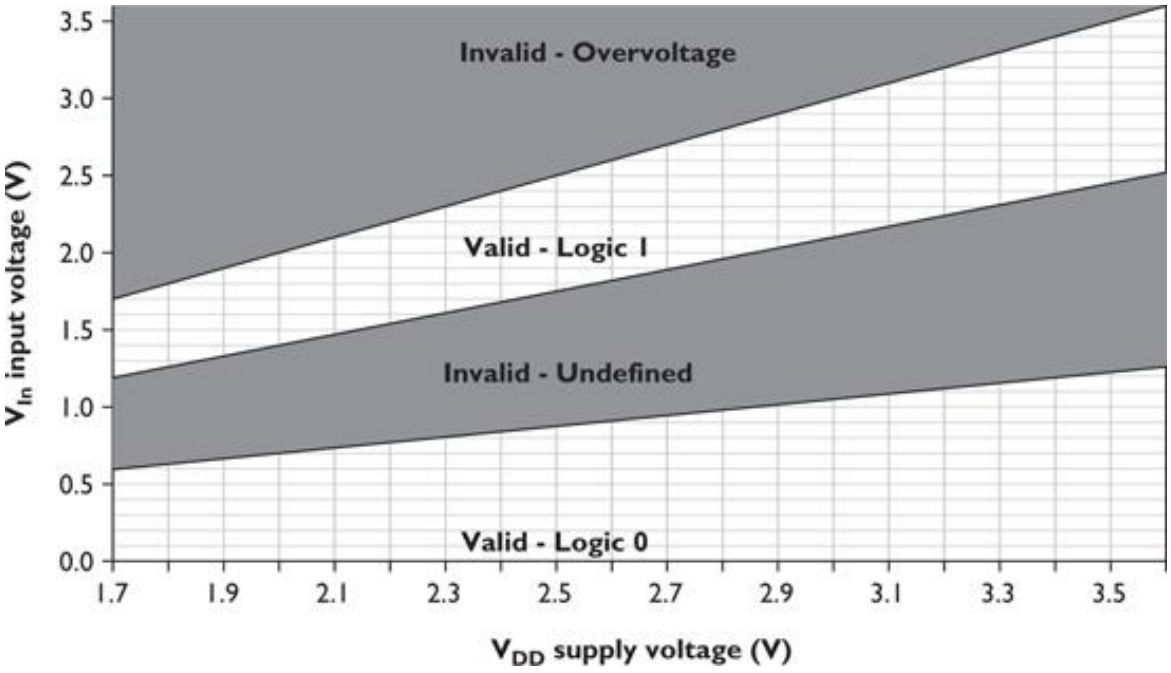
Basic digital input and output circuit:



An MCU is a digital computer, so it operates on ones and zeros. These values are represented by voltages within specific ranges relative to the MCU's power supply voltage, VDD. The actual voltages do not matter much to an embedded system developer when they are inside the MCU, but they do matter when we want to interface with external devices.

2.1 Input Signals:

Digital inputs are interpreted based on their voltage levels. The supply voltage VDD sets the thresholds for determining whether an input voltage will be considered a one or a zero. For example, the MCU's data sheet might specify that an input voltage between 0 V and 0.35 × VDD will be interpreted as a logic zero, whereas an input voltage between 0.7 × VDD and VDD will be interpreted as a logic one. Figure shows how input voltages are interpreted based on the supply voltage VDD.



For example, if VDD is 3.3 V, then input values between 0 and 1.155 V will be a logic zero and those between 2.31 and 3.3 V will be a logic one. An input voltage between 0.35 × VDD and 0.7 × VDD is undefined and may be read as a one or a zero. This is useless so we design external circuits to keep the voltage out of that range except when switching. Digital inputs do not draw much current from the signal source; they have high input impedance. Typically each GPIO pin will draw no more than 1 μA over the MCU's entire operating temperature range. At room temperature (25°C), the input current will be much smaller (e.g. no more than 25 nA).

2.2 Output Signals:

Output voltages for digital ports are typically specified as being within a certain voltage range from a supply rail. For common MCUs, a normal output generating a logic one will produce a voltage of VOH (out high), which is between VDD and VDD – 0.5 V. A logic zero will produce a voltage of V OL (out low) between 0 and 0.5 V.

This specification only holds true if a limited amount of current is drawn from the output. As that current increases, the output circuit's voltage drop increases, pulling the output voltage away from the supply rails (VDD and ground). If the output circuit draws enough current, it will overload and destroy the output transistor. So be careful not to exceed the specified ratings.

For many MCUs, the output current IOH or IOL must not exceed 5 mA. Some MCUs include output drivers with more powerful transistors, allowing greater output currents. "High drive" pads may be able to source or sink up more current (e.g. up to 18 mA) and still meet the output voltage specifications. The drive current capability also depends on the supply voltage VDD. As VDD falls, the transistors have a higher output resistance, increasing the voltage drop. So at lower operating voltages, the maximum output current available falls. Lowering VDD from 3.3 to 2.0 V might cut IOH and IOL from 5 to 1.5 mA.

3. On-board LED Blinking using Embedded C Program:

Each LED has its anode (positive end, base of triangle) connected to an MCU output signal (either LED1_OUT or LED2_OUT) through a resistor. The LED's cathode (negative end, bar) is connected to ground. An LED's brightness is roughly proportional to the current flowing through it. This current is nearly zero for low voltages, and then rises quickly as the voltage exceeds a threshold. The exact relationship between the voltage and current depends on the type of LED and its temperature. Applying 1.6 V may not light a red LED at all, but 2.0 V will make it bright (with 20 mA of current). Raising the voltage slightly to 2.3 V brightens it more and makes the current shoot up to 43 mA. Raising it further will cause the LED to overheat and fail.

The GPIO port outputs are digital and do not offer such fine-grain voltage control. They can provide two levels: and almost ground, and almost VDD (e.g. 3.3 V). The first will correctly keep the LED off, but the second will probably burn out the LED. We need to include resistor R to limit the current through the LED and MCU driver output to a safe value.

We can calculate the value R of a current-limiting resistor based on the supply voltage V_{DD}, the LED forward voltage V_F, and the desired LED current I_{LED}:

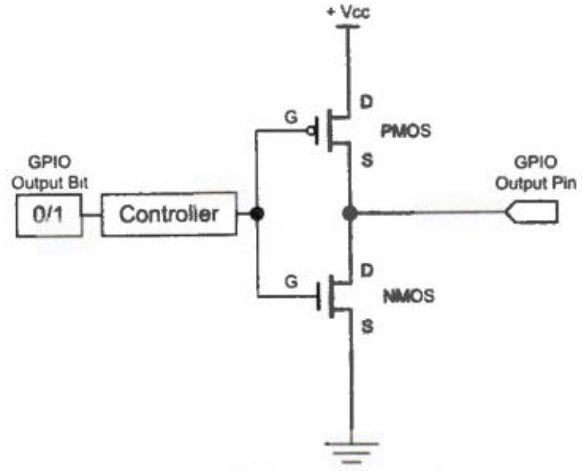
$$R = \frac{V_{DD} - V_F}{I_{LED}}$$

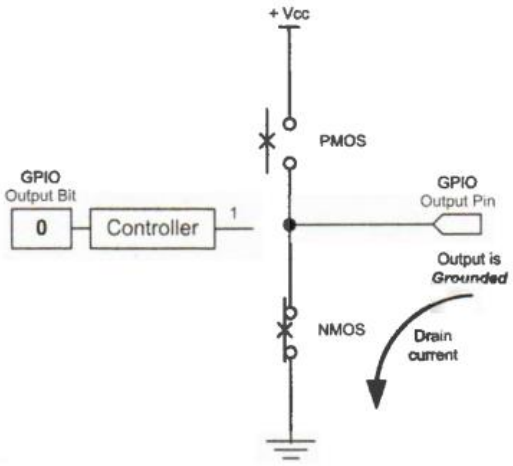
We start by picking a value of I_{LED} which is safe for both the LED and the MCU output. In this case let's set I_{LED} = 4 mA, and assume V_{DD} = 3.0 V. VF is 1.8 V for the red LED and 2.7 V for the blue LED. We can now solve for the resistor values. For the red LED, a 300 Ω resistor is needed. For the blue LED, a 75 Ω resistor is needed.

3.1 GPIO Push-Pull Output:

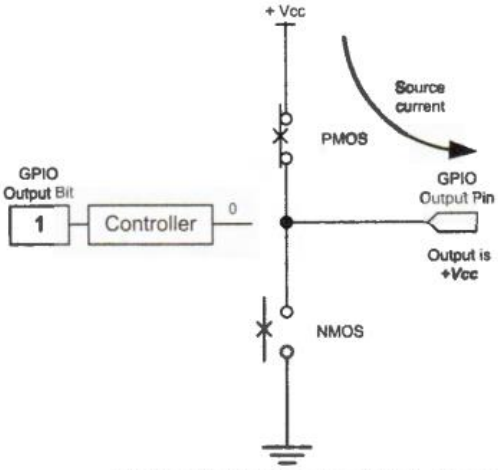
Software can configure a GPIO output pin as either push-pull or open-drain. Push-pull mode allows the pin to supply and absorb current. However, a GPIO pin in open-drain (also called collector) mode can only absorb current.

A push-pull output consists of a pair of complementary transistors, as shown in Figure and only one of them is turned on at any time.





If the digital output is 0, then the GPIO output pin is pulled down to the ground in a push-pull setting.

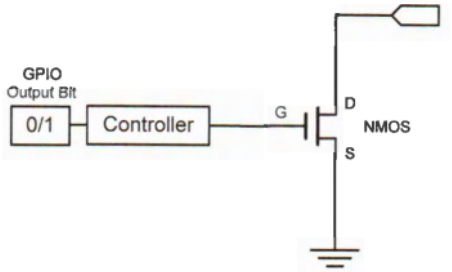


If the digital output is 1, then the GPIO output pin is pulled up to the Vcc in a push-pull setting.

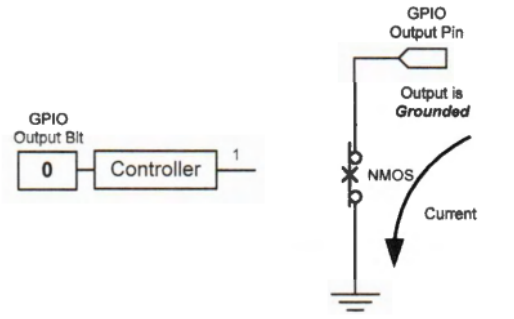
- When logic 0 is outputted, the transistor connected to the ground is turned on to sink an electric current from the external circuit, as shown in the figure above.
- When the pin outputs logic 1, the transistor connected to the power supply is turned on, and it provides an electric current to the external circuit connected to the output pin, as shown in the figure above.

3.2 GPIO Open Drain Output:

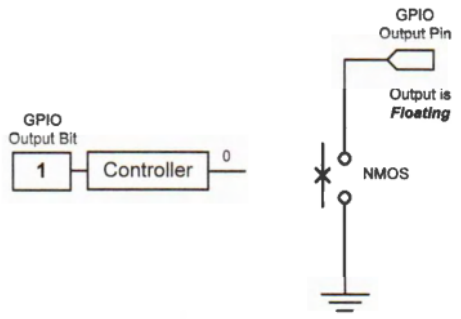
- When software outputs a logic 0, the open-drain circuit can sink an electric current from the external load connected to the GPIO pin.
- However, when software outputs a logic 1, it cannot supply any electric current to the external load because the output pin is floating, connected to neither the power supply nor the ground.



An open-drain GPIO digital output



If the digital output is 0, then the output pin is pushed to the ground in an open-drain setting (the scenario of *drain*).



If the digital output is 1, then the output pin is floating in an open-drain setting (the scenario of *open*).

One important usage of open-drain outputs is to directly connect several outputs together and implement wired logic AND (active high) or OR (active low) circuit in an easy way. If multiple open-drain output pins are connected and are pulled up via a shared resistor, any output pin can drive the output voltage low. The pin voltage is high if and only if all pins output a high voltage level.

- If a high voltage level represents logic state 1 (i.e., active high), it implements a wired-AND function. The final output is 1 (high) only if all outputs of connected pins are 1 (high).
- If a low voltage level represents logic state 1 (i.e., active low), it implements a wired-OR function. The final output is 1 (low) if the output of any pins is 1 (low).

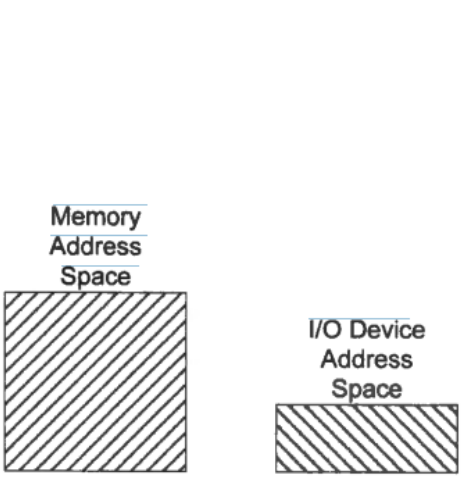
3.3 Memory Mapped I/O:

Typically, an on-chip peripheral device has a few registers, such as control registers, status registers, data input registers, and data output registers. A peripheral may also have data buffers, such as the display memory of the LCD controller. Input/output or I/O refers to data communication between the processor core and a peripheral device.

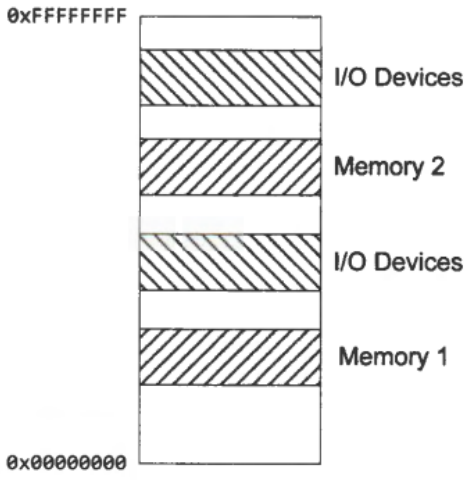
There are two complementary approaches to performing I/O operations: Port-mapped I/O, and memory-mapped I/O.

• Port-mapped I/O uses special machine instructions, which are designed specifically for I/O operations. The memory address space and the I/O device address space are independent of each other. Each device is assigned one or more unique port numbers. For example, Intel x86 processors use IN and OUT instructions to read from or write to a port.

• Memory-mapped I/O does not need any special instructions. The memory and the I/O devices share the same address space. Each peripheral register or data buffer is assigned to a memory address in the memory address space of the microprocessor. Memory-mapped I/O is performed by the native load and store instructions of the processor. Therefore, memory-mapped I/O is a more convenient way to interface I/O devices. The most significant disadvantage is that memory-mapped I/O has a more complex address decoding unit than port-mapped I/O.



Port-mapped I/O has two independent address spaces.



Memory-mapped I/O has a single shared address space.

ARM Cortex-M processors use memory-mapped I/O to access peripheral registers. All peripheral registers on STM32F4 are mapped to a small memory region starting at 0x40000000. This region includes the memory addresses of all on-chip peripherals, such as GPIO, timers, UART, SPI, and ADC. The memory address of each peripheral register is determined by chip manufacturers, and usually cannot be changed by software.

4. Inside the MCU: Control Registers and C code

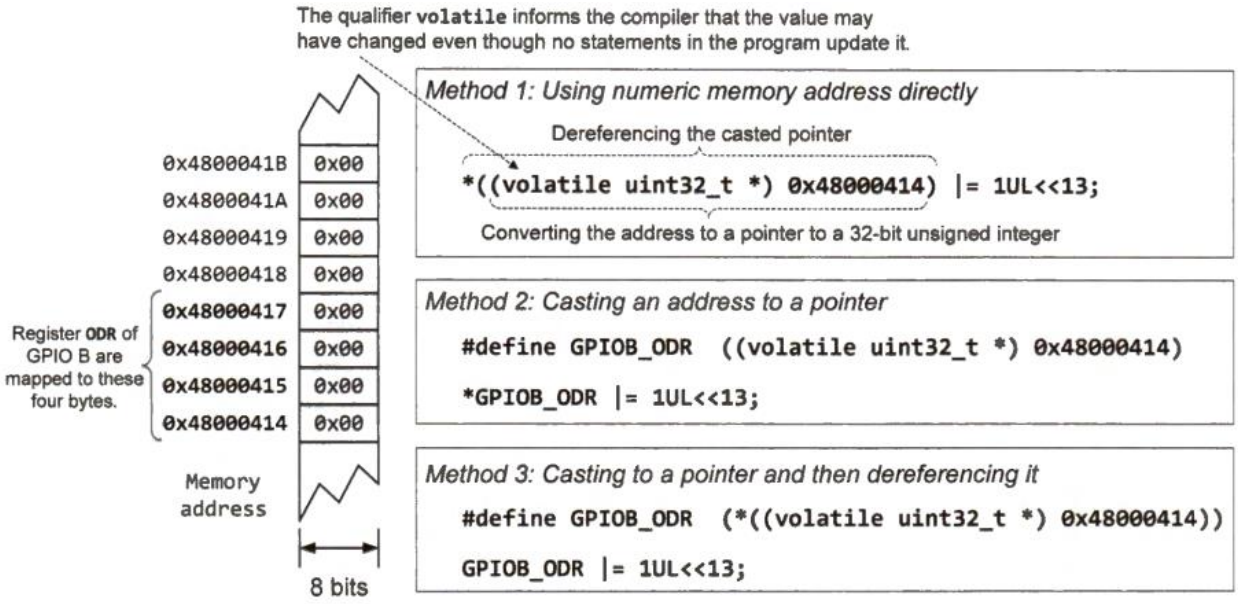
Peripherals are built with control registers to allow us to configure them, determine their status and transfer data. Software can program a GPIO pin as one of the following four different functions:

- Digital input that detects whether an external voltage signal is higher or lower than a predetermined threshold.
- Digital output that controls the voltage on the pin.
- Analog functions that perform digital-to-analog or analog-to-digital conversion.
- Other complex functions such as PWM output, LCD driver, timer-based input capture, external interrupt, and interface of USART, SPI, PC and USB communication.

A GPIO port consists of a group of GPIO pins, typically 8 or 16, which share the same data and control registers.

- When a GPIO pin i is set as a digital input, the binary data read from this pin of this GPIO group is saved at bit i in the input data register (IDR). Each bit in IDR holds the digital input of the corresponding pin.
- When a GPIO pin i is configured as a digital output, bit i in the output data register (ODR) holds the output of this pin. Therefore, when changing the output of a GPIO pin, the programmer should only alter the value of the corresponding bit of ODR, without affecting the other bits in ODR.
- All GPIO pins in a GPIO port can be configured as input or output independently.

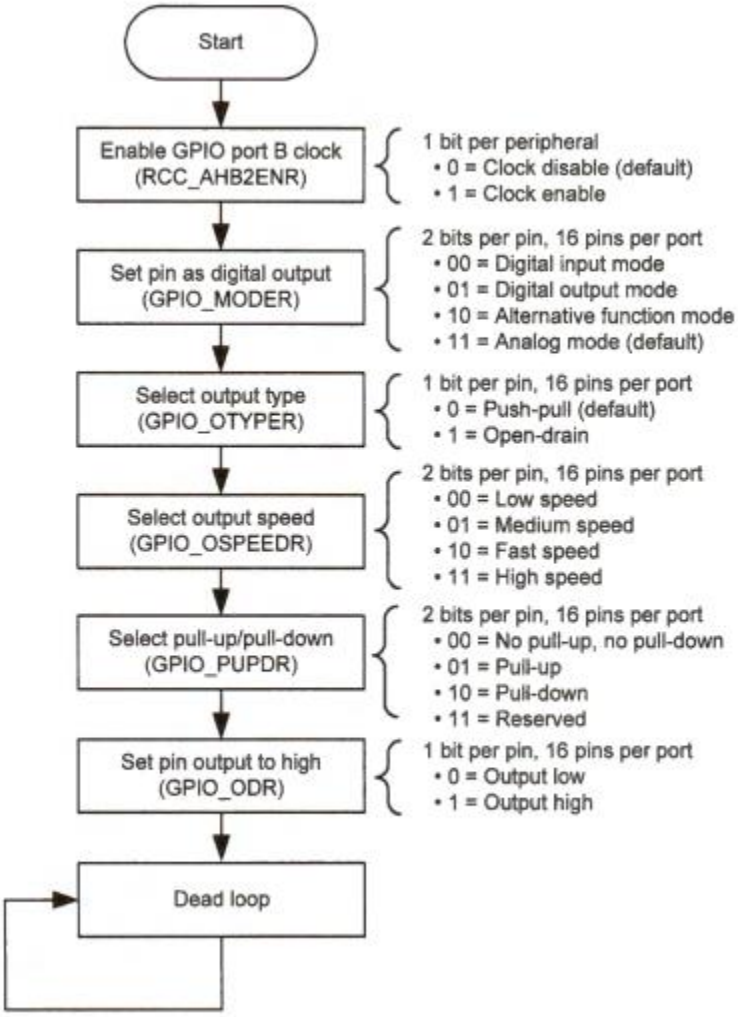
A peripheral register usually takes four bytes in memory. For example, the output data register (ODR) of Port B on STM32L4 is mapped to memory addresses 0x48000414 to 0x48000417, with the upper halfword being reserved. Note that values stored in peripheral registers are in the format of little-endian.



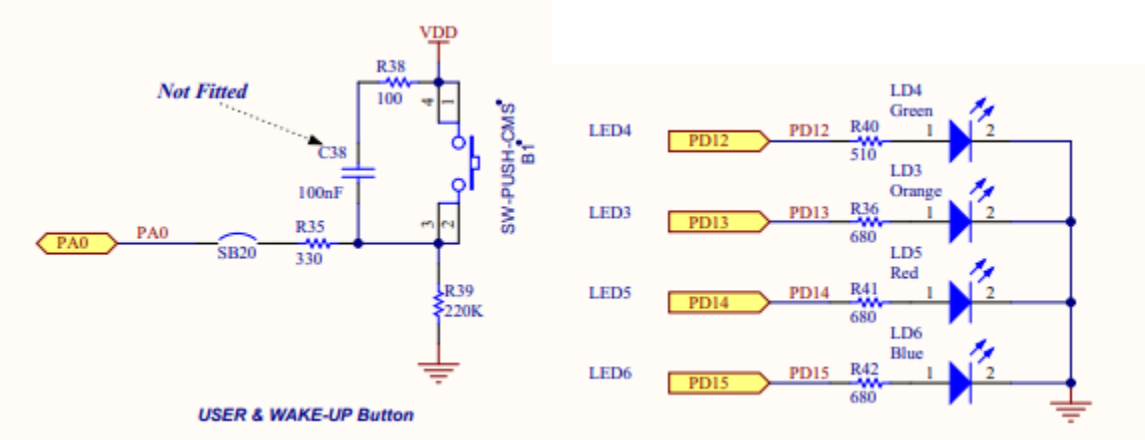
If we want to set the output of GPIO port B pin 6 to high, we can use the following C statement. lUL is an unsigned long integer with a value of 1. Note the pins are numbered 0 - 15, instead of 1 - 16.

```
GPIOB->ODR |= 1UL<<6;     // Set bit 6
```

4.1 Program to light up a LED in STM32F407 Microcontroller:



4.2 Hardware Schematic Diagram:



4.3 LED Blinking Program:

```
#include "stm32f4xx.h"
void configureLED(void);
void msDelay(volatile int msTime);
int main(void)
{
	//Configure LED
	configureLED();
	//Define Delay function
	msDelay(1000);
	//GPIOD->ODR = (0x5UL<<12);
	while(1)
	{
		GPIOD->ODR ^= (0xFUL<<12);
		msDelay(250);
	}
}
void configureLED(void)
{
	RCC->AHB1ENR |=(1UL<<3);
	GPIOD->MODER &= ~(0xFFUL<<12*2);
	GPIOD->MODER |= (0x55UL<<12*2);
}
void msDelay(int msTime)
{
	//Assume for loop take 12 clock cycles and system clock is 16MHz
int Time=msTime*1333;
	for(int i=0;i<Time;i++);
}
```

4.4 User Push Button Interface Program:

```
#include "stm32f4xx.h"
volatile unsigned int PB_state;
void configureLED(void)
{
	RCC->AHB1ENR |=(1UL<<3);
	GPIOD->MODER &= ~(0xFFUL<<12*2);
	GPIOD->MODER |= (0x01UL<<12*2);
}


void configurePB(void)
{
	RCC->AHB1ENR |=(0x1UL<<0);
	GPIOA->MODER &= ~(0x3UL<<0);

}

void msDelay(int msTime)
{
	//Assume for loop take 12 clock cycles and system clock is 16MHz
int Time=msTime*1333;
	for(int i=0;i<Time;i++);

}


int main ()
{
	configureLED();
	configurePB();

	while(1)
	{
	PB_state=(GPIOA->IDR&(0x1UL<<0));
				if(PB_state==1)
				{
					GPIOD->ODR |=(0x1UL<<12);

				}
				else
				{
					GPIOD->ODR |= (0x0UL<<12);

				}
	 }

}
```

5. Conclusion:

In this experiment, input and output activities like LED blinking and push button interfacing were performed using the GPIO ports of STM32F4 microcontroller board.

Study of Polling and Interrupts using a Cortex-M Microcontroller

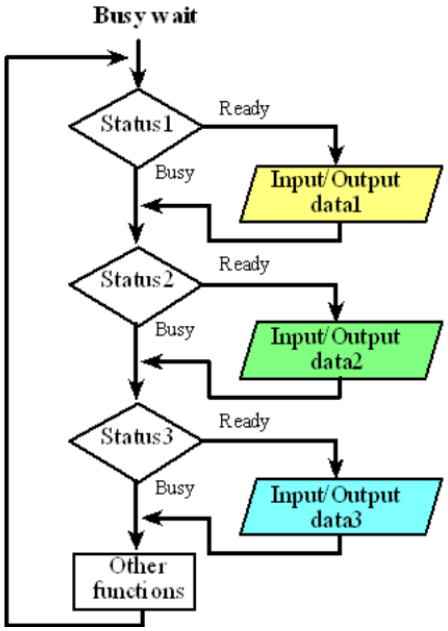
1. Aim:

i. To implement and compare the polling and interrupt-based service mechanisms for various events.
ii. To appreciate the need to perform multiple tasks concurrently.
iii. To understand perform measures of a real-time system such as bandwidth and latency.
iv. To learn how interrupts can be used to minimize latency.
v. To study the basics of interrupt programming: arm, enable, trigger, vector, priority, acknowledge.

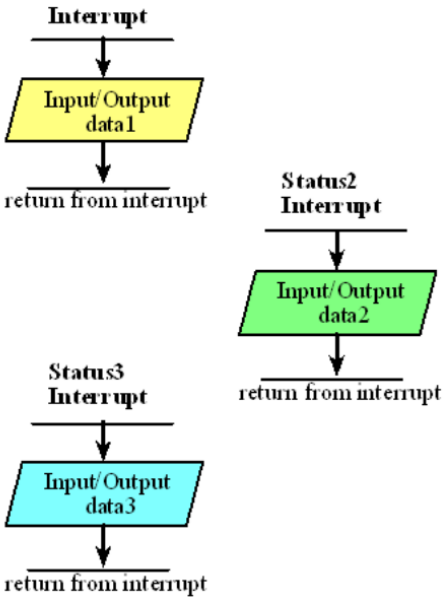
2. Introduction:

An **interrupt** is the automatic transfer of software execution in response to a hardware event that is asynchronous with the current software execution. This hardware event is called a **trigger**. The hardware event can either be a busy to ready transition in an external I/O device (like the UART input/output) or an internal event (like bus fault, memory fault, or a periodic timer). When the hardware needs service, signified by a busy to ready state transition, it will request an interrupt by setting its trigger flag. A **thread** is defined as the path of action of software as it executes. The execution of the interrupt service routine is called a background thread. This thread is created by the hardware interrupt request and is killed when the interrupt service routine returns from interrupt (e.g., by executing a **BX LR**). A new thread is created for each interrupt request. It is important to consider each individual request as a separate thread because local variables and registers used in the interrupt service routine are unique and separate from one interrupt event to the next interrupt.

2.1 Illustration of Busy Waiting or Polling:



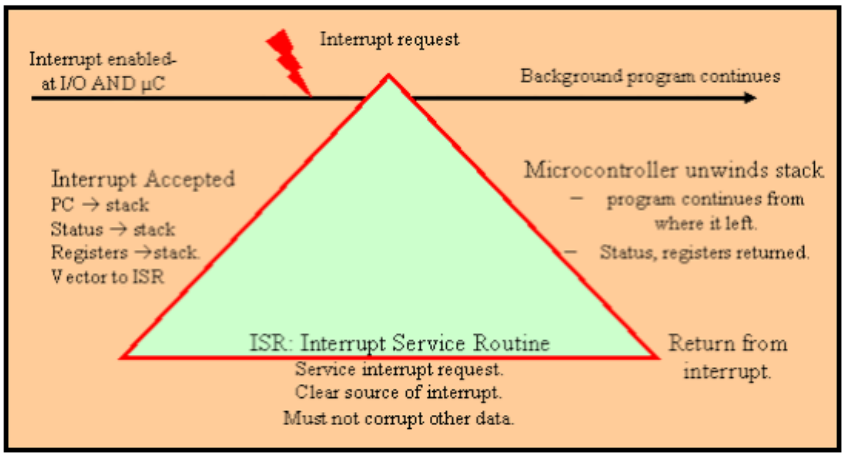
2.2 Interrupt Service Mechanism:



On the ARM Cortex-M processor there is one interrupt enable bit for the entire interrupt system. We disable interrupts if it is currently not convenient to accept interrupts. In particular, to disable interrupts we set the I bit in **PRIMASK**. In C, we enable and disable interrupts by calling the functions **EnableInterrupts()** and **DisableInterrupts()** respectively.

The software has dynamic control over some aspects of the interrupt request sequence. First, each potential interrupt trigger has a separate **arm** bit that the software can activate or deactivate. The software will set the arm bits for those devices from which it wishes to accept interrupts and will deactivate the arm bits within those devices from which interrupts are not to be allowed. Five conditions must be true for an interrupt to be generated: individual device Enable, NVIC enable, global enable, interrupt priority level must be higher than current level executing, and hardware event trigger.

2.3 Generic Response to Interrupts:



An interrupt causes the following sequence of five events.
1) Current instruction is finished,
2) Eight registers are pushed on the stack,
3) LR is set to 0xFFFFFFF9,
4) IPSR is set to the interrupt number,
5) PC is loaded with the interrupt vector.

If a trigger flag is set, but the interrupts are disabled (I=1), the interrupt level is not high enough, or the flag is disarmed, the request is not dismissed. Rather the request is held **pending**, postponed until a later time, when the system deems it convenient to handle the requests. In other words, once the trigger flag is set, under most cases it remains set until the software clears it. The five necessary events (device arm, NVIC enable, global enable, level, and trigger) can occur in any order. For example, the software can set the I bit to prevent interrupts, run some code that needs to run to completion, and then clear the I bit. A trigger occurring while running with I=1 is postponed until the time the I bit is cleared again.

Clearing a trigger flag is called **acknowledgement**, which occurs only by specific software action. Each trigger flag has a specific action software must perform to clear that flag. We will pay special

attention to these enable/disable software actions. The SysTick periodic interrupt will be the only example of an automatic acknowledgement. For SysTick, the periodic timer requests an interrupt, but the trigger flag will be automatically cleared when the ISR runs. For all the other trigger flags, the ISR must explicitly execute code that clears the flag.

The **interrupt service routine** (ISR) is the software module that is executed when the hardware requests an interrupt. There may be one large ISR that handles all requests (polled interrupts), or many small ISRs specific for each potential source of interrupt (vectored interrupts). The design of the interrupt service routine requires careful consideration of many factors. Except for the SysTick interrupt, the ISR software must explicitly clear the trigger flag that caused the interrupt (acknowledge). After the ISR provides the necessary service, it will execute **BX LR**. Because LR contains a special value (e.g., 0xFFFFFFF9), this instruction pops the 8 registers from the stack, which returns control to the main program. If the LR is 0xFFFFFFE9, then 26 registers (R0-R3, R12, LR, PC, PSW, and 18 floating point registers) will be popped by **BX LR**. There are two stack pointers: PSP and MSP. The software in this class will exclusively use the MSP. It is imperative that the ISR software balance the stack before exiting. Execution of the previous thread will then continue with the exact stack and register values that existed before the interrupt. Although interrupt handlers can create and use local variables, parameter passing between threads must be implemented using shared global memory variables. A private global variable can be used if an interrupt thread wishes to pass information to itself, e.g., from one interrupt instance to another. The execution of the main program is called the foreground thread, and the executions of the various interrupt service routines are called background threads.

An axiom with interrupt synchronization is that the ISR should execute as fast as possible. The interrupt should occur when it is time to perform a needed function, and the interrupt service routine should perform that function, and return right away. Placing backward branches (busy-wait loops, iterations) in the interrupt software should be avoided if possible. The percentage of time spent executing interrupt software should be small when compared to the time between interrupt triggers.

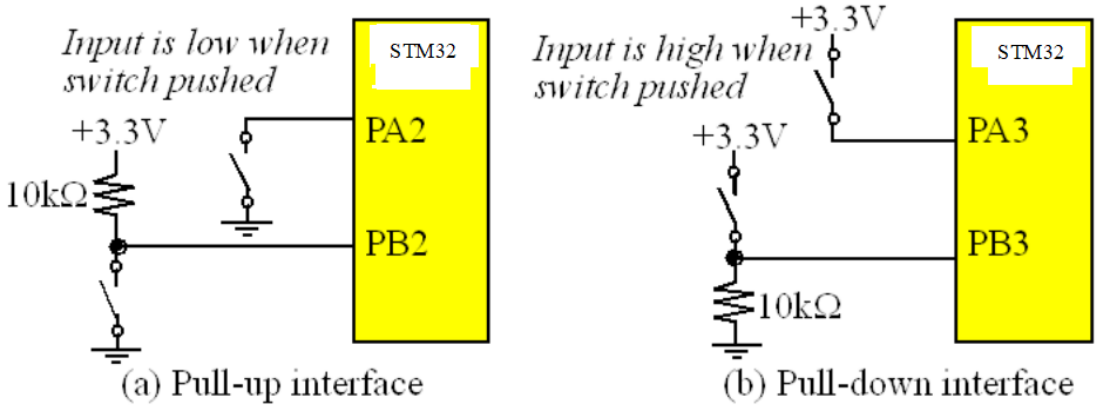
2.4 Edge Triggered Interrupts:

Synchronizing software to hardware events requires the software to recognize when the hardware changes states from busy to done. Many times, the busy to done state transition is signified by a rising (or falling) edge on a status signal in the hardware. For these situations, we connect this status signal to an input of the microcontroller, and we use edge-triggered interfacing to configure the interface to set a flag on the rising (or falling) edge of the input. Using edge-triggered interfacing allows the software to respond quickly to changes in the external world. If we are using

busy-wait synchronization, the software waits for the flag. If we are using interrupt synchronization, we configure the flag to request an interrupt when set.

For input signals we have the option of adding either a pull-up resistor or a pull-down resistor. If we set the corresponding **PUE** (Pull-Up Enable) bit on an input pin, the equivalent of a 13 kΩ to 30 kΩ resistor to +3.3 V power is internally connected to the pin. Similarly, if we set the corresponding **PDE** (Pull-Down Enable) bit on an input pin, the equivalent of a 13 kΩ to 35 kΩ resistor to ground is internally connected to the pin. We cannot have both pull-up and a pull-down resistor, so setting a bit in one register automatically clears the corresponding bit in the other register.

A typical application of pull-up and pull-down mode is the interface of simple switches. Using these modes eliminates the need for an external resistor when interfacing a switch. Compare the interfaces on Port A to the interfaces on Port B illustrated in Figure 12.4. The PA2 and PA3 interfaces will use software-configured internal resistors, while the PB2 and PB3 interfaces use actual resistors. The PA2 and PB2 interfaces in Figure a, implement negative logic switch inputs, and the PA3 and PB3 interfaces in Figure b) implement positive logic switch inputs.



3.1 Program for Monitoring User Push Button using ExternalInterrupt0:

```
#include "stm32f4xx.h"

void EXT_Init(void);
void configureLED(void);
void External_Interrupt_Enable(void);

void EXT_Init(void)
{

//1. Enable TIM@ and GPIO clock
		RCC->APB1ENR |= (1<<0);  // enable PORTA clock
		EXTI->RTSR |=(1<<0);
		EXTI->IMR |=(1<<0);
}
void configureLED(void)
{
		RCC->AHB1ENR |=(1UL<<3);  //Enable GPIOD clock
		GPIOD->MODER &= ~(0xFFUL<<12*2);
		GPIOD->MODER |= (0x55UL<<12*2);
}

void External_Interrupt_Enable(void)
{
		NVIC->ISER[0] |= 1<<6;

}


int main ()
{

		EXT_Init();
		configureLED();
		External_Interrupt_Enable();
		while(1)
		{
			//GPIOD->ODR = (0x0UL<<12);
}
}


void EXTI0_IRQHandler( )
{
		GPIOD->ODR ^= (0x1UL<<12);
		EXTI->PR |= (1<<0);

}
```

If two or more triggers share the same vector, these requests are called **polled interrupts**, and the ISR must determine which trigger generated the interrupt. If the requests have separate vectors, then these requests are called **vectored interrupts** and the ISR knows which trigger caused the interrupt.

One of the problems with switches is called **switch bounce**. Many inexpensive switches will mechanically oscillate for up to a few milliseconds when touched or released. It behaves like an under damped oscillator. These mechanical oscillations cause electrical oscillations such that a port pin will oscillate high/low during the bounce. In some cases this bounce should be removed. To remove switch bounce we can ignore changes in a switch that occur within 10 ms of each other. In other words, recognize a switch transition, disarm interrupts for 10ms, and then rearm after 10 ms. Alternatively, we could record the time of the switch transition. If the time between this transition and the previous transition is less than 10ms, ignore it. If the time is more than 10 ms, then accept and process the input as a real event.

3.2 Timer Interrupt with Multitasking Program:

```
#include "stm32f4xx.h"
#define ARM_MATH_CM4

void TIM2_Init(void);
void configureLED(void);
void TIM2_Interrupt_Enable(void);
void msDelay(int msTime);

void TIM2_Init (void)
{

//1. Enable TIM@ and GPIO clock
        RCC->APB1ENR |= (1<<0);  // enable TIM2 clock
                RCC->CFGR |= 0<<10; // set APB1 = 16 MHz

//2. Timer pre-scalar and period configuration
        TIM2->CR1 &= ~(0x0010);          //Set the mode to Count up
                TIM2->PSC = 16000-1;                                    //Set the
Prescalar
                TIM2->ARR = 200;
        //Set period (Auto reload) to 400
                TIM2->SR &= ~(0x0001);                                  //Clear Update
interrupt flag

}
void configureLED(void)
{
```

```
	RCC->AHB1ENR |=(1UL<<3);  //Enable GPIOD clock
	GPIOD->MODER &= ~(0xFFUL<<12*2);
	GPIOD->MODER |= (0x55UL<<12*2);
}

void TIM2_Interrupt_Enable(void)
{
	NVIC->ISER[0] |= 1<<28;
	TIM2->DIER |=(1<<0);
}

void configurePB(void)
{
	RCC->AHB1ENR |=(0x1UL<<0);

}

int main ()
{

	TIM2_Init ();
	configureLED();
	TIM2_Interrupt_Enable();
	configurePB();
		unsigned int PB_state;
	TIM2->CR1 |= 1UL;
while(1)
	{
	PB_state=(GPIOA->IDR&(0x1UL));
			if(PB_state==0x1)
			{
			GPIOD->ODR |= (1<<13);
				msDelay(100);
			}
			else
			{
				GPIOD->ODR &= (~(1<<13));
			}
}
}


void TIM2_IRQHandler( )
{
	GPIOD->ODR ^= (0x1UL<<12);
	TIM2->SR &= ~(0x0001);

}

void msDelay(int msTime)
{
```

```
	/* For loop takes 4 clock cycles to get executed. Clock frequency of stm32f407 by default is 16MHz
	So, 16MHz/4=4MHz. If we want 1000ms delay, 4MHz/1000=4000, so we have to multiply by 4000 to get a delay of 1s
	*/
	for(int i=0;i<msTime*4000;i++)
	{
		__NOP();
	}

}
```

3.3 Simultaneous Generation of Two Square Waveforms Program:

```
#include"stm32f4xx.h"
#define ARM_MATH_CM4
void TIM2_Init(void);
void TIM3_Init(void);
void configureLED12(void);
void TIM2_Interrupt_Enable(void);
void TIM3_Interrupt_Enable(void);
void TIM2_Init(void)
{
//1. Enable TIM2 and GPIO Clock
RCC->APB1ENR |= (1<<0); //Enable TIM2 clock
RCC->CFGR |= 0<<10; // Set APB1 = 16 MHz
//2. Timer pre-scalar and period configuration
TIM2->CR1 &= ~(0x0010); //Set mode to count up
TIM2->PSC = 16000-1; //Set pre-scalar
TIM2->ARR = 100; // Set period(auto reload) to 400/CHANNEL 1 5HZ
TIM2->SR &= ~(0x0001);
}
void TIM3_Init(void)
{
//1. Enable TIM3 and GPIO Clock
RCC->APB1ENR |= (1<<1); //Enable TIM3 clock
RCC->CFGR |= 0<<11; // Set APB1 = 16 MHz
//2. Timer pre-scalar and period configuration
TIM3->CR1 &= ~(0x0010); //Set mode to count up
TIM3->PSC = 16000-1; //Set pre-scalar
TIM3->ARR = 50; // Set period(auto reload) to 400 /CHANNEL 2,10HZ
TIM3->SR &=~ (0x0001);
}
void configureLED12(void)
{
RCC->AHB1ENR |= (1UL<<3); //ENABLE GPIO CLOCK
```

```
GPIOD->MODER &=~(0xFFUL<<12*2);
GPIOD->MODER |= (0x55UL<<12*2);
}
void TIM2_Interrupt_Enable(void)
{
NVIC->ISER[0] |= 1<<28;
TIM2->DIER |= (1<<0);
}
void TIM3_Interrupt_Enable(void)
{
NVIC->ISER[0] |= 1<<29;
TIM3->DIER |= (1<<0);
}
int main()
{
TIM2_Init();
TIM3_Init();
configureLED12();
TIM2_Interrupt_Enable();
TIM3_Interrupt_Enable();
TIM2->CR1 |= 1UL;
TIM3->CR1 |= 1UL;
while(1){ }
}
void TIM2_IRQHandler() {
GPIOD->ODR ^= (0x1UL<<12);
TIM2->SR &= ~(0x0001);
}
void TIM3_IRQHandler() {
GPIOD->ODR ^= (0x1UL<<13);
TIM3->SR &= ~(0x0001);
}
```

3.4 ADC with Interrupts Program:

```
#include "stm32f407xx.h"
void ADC_Config(void);
void interrupt_config(void);
int value;
void ADC_Config(void)
{
        RCC->AHB1ENR |= (1UL<<0);// Enable clock for Port A
        RCC->APB2ENR |= (1<<8);  // Enable ADC1 clock
        ADC->CCR |= (1UL<<16);  //Prescale clock to ADC by 4
        ADC1->CR1 |= (2UL<<24); //8 bit adc
        ADC1->CR2 |= (1UL<<1);  //Enable continuous conversion
```

```
		ADC1->CR2 |= (1UL<<10); //Enable EOC after every conversion
		ADC1->SMPR2 |= (7UL<<3);//Select sampling time as 480 cycles for channel 1
		GPIOA->MODER |= (3UL<<2); //Configure PA1 in analog mode
		ADC1->SQR3 |= (1<<0); //Select channel 1 for ADC conversion (corresponds to PA1)
}
void interrupt_config(void)
{
		NVIC_EnableIRQ(ADC_IRQn);
		ADC1->CR1 |= (1UL<<5);  //Enable interrupt
}
void ADC_IRQHandler(void)
{
		value=ADC1->DR;
		ADC1->SR = 0;
}
int main()
{
		ADC_Config();
		interrupt_config();
		ADC1->CR2 |= (1UL<<0); // Start the ADC
		ADC1->CR2 |= (1UL<<30);// Start the conversion
		ADC1->SR = 0;
		while(1)
		{

		}
}
```

4. Conclusion:

In this experiment, the concepts of polling and interrupt were studied with the sample programs using STM32CubeIDE.

PWM based Permanent Magnet DC Motor Speed Control

1. Aim:

To generate the Pulse Width Modulated (PWM) signal for different duty cycles to control the DC motor.

2. Introduction:

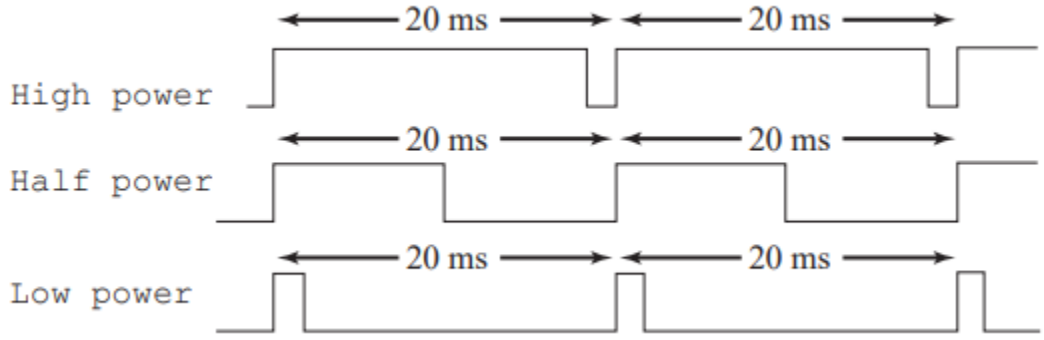
Pulse width modulation (PWM) is a simple digital technique to control the value of an analog variable. PWM uses a rectangular waveform to quickly switch a voltage source on and off to produce a desired average voltage output. Although the output is binary at any time instant, the average output over a time span can be any value between 0 and the maximum voltage.

Specifically, the percentage of time in the on state within one period is proportional to the mean value of the voltage output. Consequently, when software changes the duration of the on state, the output voltage is adjusted accordingly to emulate an analog signal.

PWM has been widely used in a variety of applications, such as motor speed and torque control, digital encoding in telecommunications, DC-to-DC power conversion, and audio amplification. We should select the PWM switching frequency carefully to avoid serious negative impacts on applications. For example, the PWM switching frequency of an LED light must be at least 120 Hz to prevent the flickering effects that humans can see.

2.1 DC Motor Control:

The objective of this exercise is to control the rotational speed of a DC motor. The 5 V DC motor has a resistance of 10Ω. Your control system will apply a variable power to the motor by varying the duty cycle of a 50 Hz signal, as shown below.

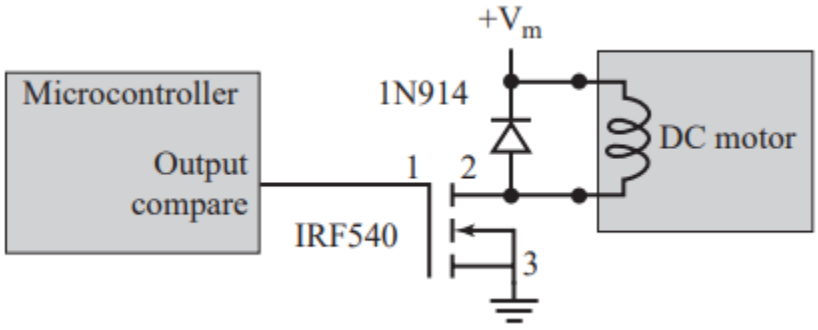


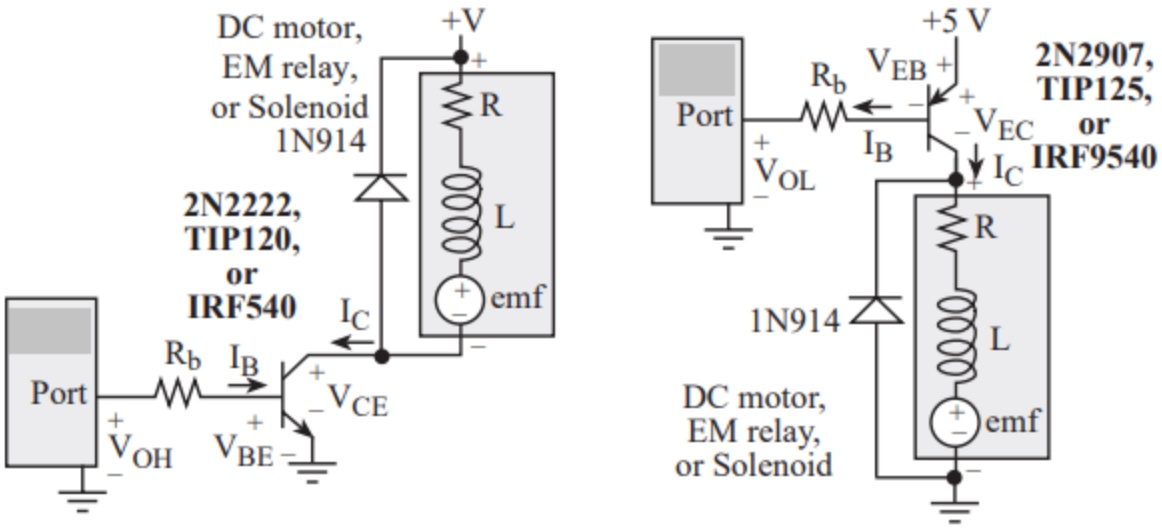
2.2 DC Motor Interface using a High-Current MOSFET:

The circuits in this section can be used to interface solenoids, EM relays, and DC motors because they all have electromagnets that behave like Figure. To interface an electromagnet, we consider voltage, current, and inductance. First, we need a power supply at the desired voltage requirement of the coil.

It is important to realize that many devices cannot be connected directly up to the microcontroller. In the specific case of motors, we need an interface that can handle the voltage and current required by the motor.

If the only available power supply is larger than the desired coil voltage, we use a voltage regulator (rather than a resistor divider) to create the desired voltage. We connect the power supply to the positive terminal of the coil, shown as V in Figure. We will use an NPN transistor to drive the negative side of the coil to ground. The other way to interface the coil is to use a PNP transistor to drive the positive side of the coil to V. The computer can turn the current on and off by controlling the base of the transistor.





3. Microcontroller Implementation:

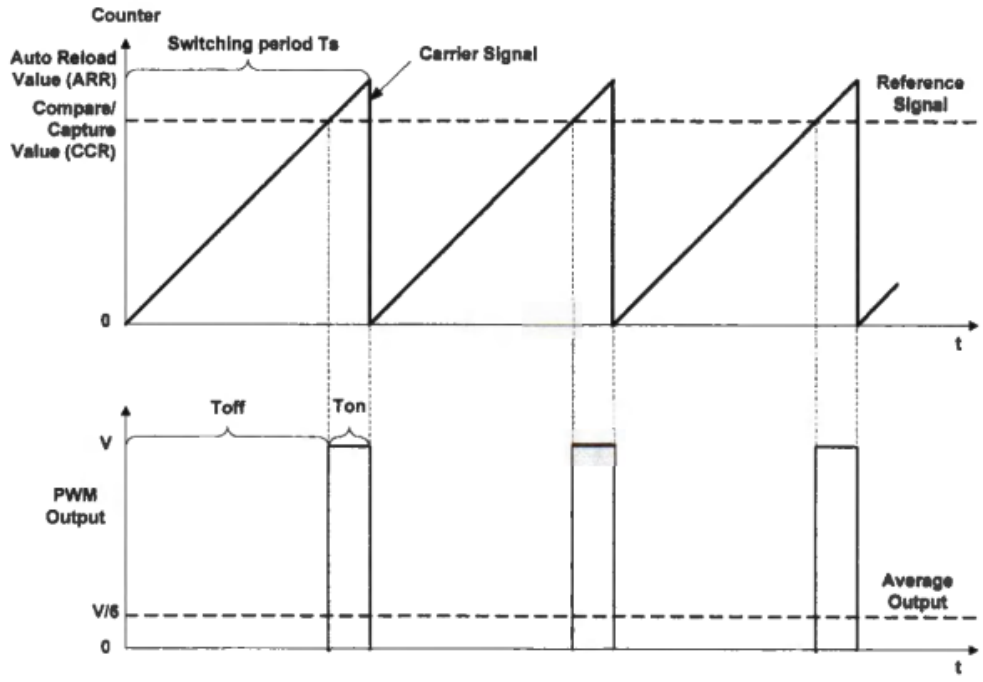
The average value of a simple PWM output based on a sawtooth carrier signal and a constant reference, as illustrated in Figure.

The duty cycle is defined as follows:

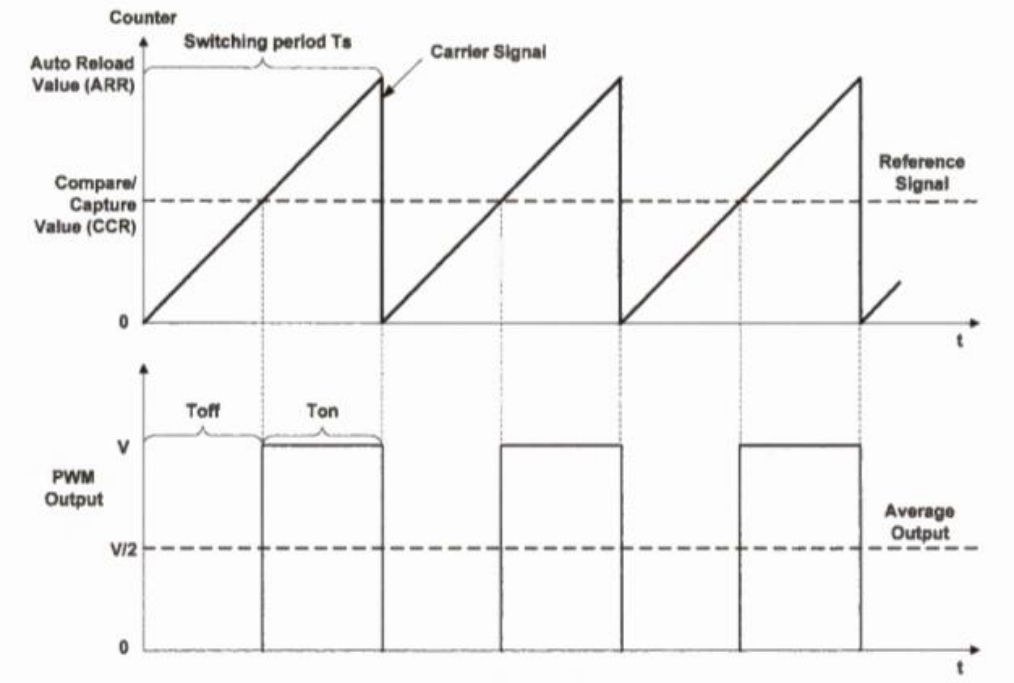
$$duty\ cycle = \frac{pulse\ on\ time\ (T_{on})}{pulse\ switching\ period\ (T_s)} \times 100\%$$
$$= \frac{T_{on}}{T_{on} + T_{off}} \times 100\%$$

where

$$pulse\ switching\ period = \frac{1}{PWM\ switching\ frequency}$$



Example of simple PWM when duty cycle is 1/6



Example of simple PWM when duty cycle is 1/2

The PWM output signal is determined by three factors: comparison between the timer counter (CNT) and the given reference valued stored in the compare and capture register (CCR), the PWM output mode, and the polarity bit.

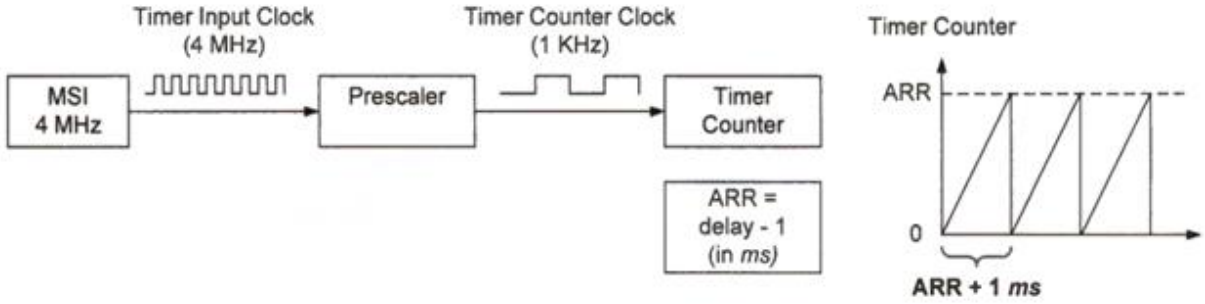
PWM Duty Cycle:

Depending on the PWM mode and the polarity bit, the duty cycle of the main output `OC` or the complementary output `OCN` in up-counting or down-counting is

$$Duty\ Cycle = \begin{cases} \dfrac{CCR}{ARR+1} \\ 1 - \dfrac{CCR}{ARR+1} \end{cases}$$

3.1 Determination of ARR and CCR Values:

Assume that one need to generate a PWM signal of 100Hz, with 60% duty cycle. If the given microcontroller is running at 4MHz, find the prescaler, ARR and CCR values.



In this example, the prescaler factor is set as 39, thus the frequency at which the counter increments is

$$f_{CK\_CNT} = \frac{f_{CL\_PSC}}{Prescaler + 1} = \frac{4MHz}{39 + 1} = 100\ KHz$$

For a given clock frequency, ARR determines the PWM period. Assume the counting mode is up-counting, and the PWM frequency is 100 Hz. Thus, the PWM period should be

$$PWM\ Period = \frac{1}{100\ Hz} = 0.01\ second$$

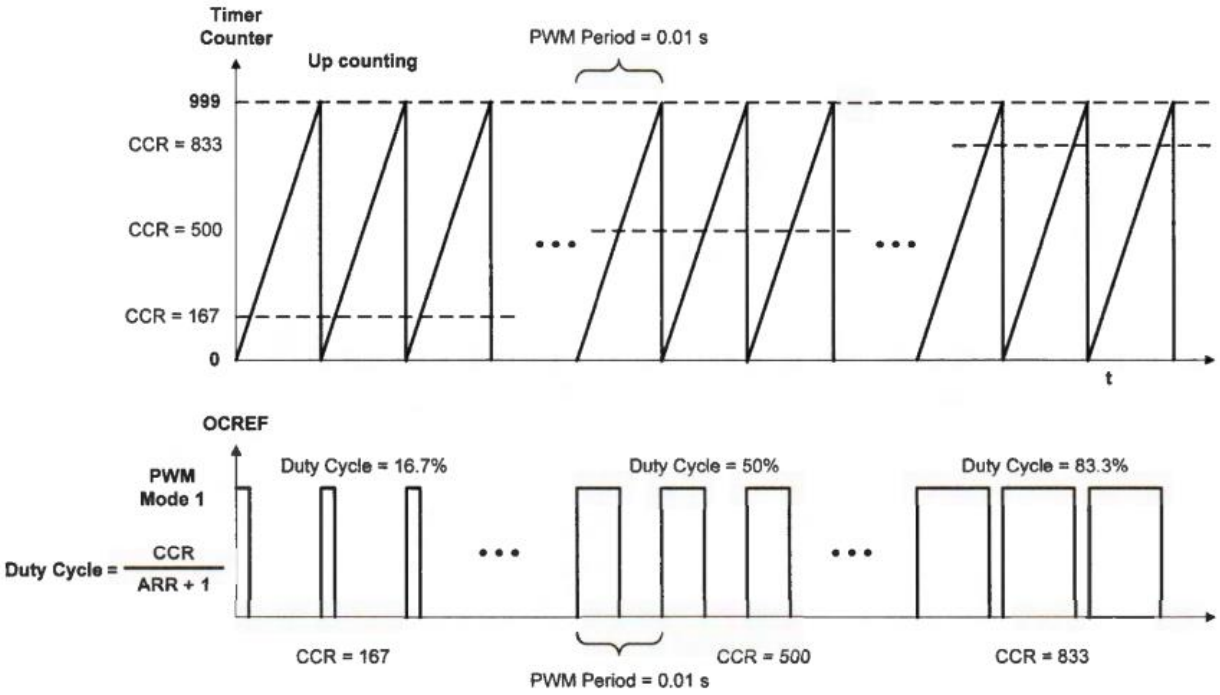
At the same time, we have

$$PWM\ Period = (1 + ARR) \times \frac{1}{f_{CK\_CNT}}$$

Thus, we have

$$ARR = PWM\ Period \times f_{CK_{CNT}} - 1 = 0.01 \times 100 \times 10^3 - 1 = 999$$

When ARR is kept at a fixed value, CCR controls the duty cycle. Assume timer 1 uses PWM mode 1, in which the reference output (OCRE F) is high for up-counting if the timer counter is less than CCR, and otherwise, OCREF is low. CCR controls the duty cycle of PWM output if ARR is fixed.



4.1 Program to Generate a PWM to Control the Brightness of a LED at 100Hz:

```
#include "stm32f4xx.h"


void TIM2_Init(void);
void configureLED(void);

void TIM4_Init (void)
{
```

```
//1. Enable TIM@ and GPIO clock
        RCC->APB1ENR |= (4UL<<0);  // enable TIM4 clock
                RCC->CFGR |= 0<<10; // set APB1 = 16 MHz

//2. Timer pre-scalar and period configuration
        TIM4->CR1 &= ~(0x0010);          //Set the mode to Count up
                TIM4->PSC = 160-1;                                          //Set the
Prescalar
                TIM4->ARR = 1000-1;
        //Set period (Auto reload) to 400
                //TIM2->SR &= ~(0x0001);                                    //Clear Update
interrupt flag

}
void configureLED(void)
{
        RCC->AHB1ENR |=(1UL<<3);  //Enable GPIOD clock
        GPIOD->MODER |= (2UL<<24); //Alternate function
        GPIOD->AFR[1] |= (2UL<<16); //Timer4_Channel1

}

void configurePWM(void)
{
        TIM4->CCMR1 |=(6UL<<4); //PWM mode
        //TIM4->CCMR1 |=(1UL<<3);
        TIM4->CCER |=(0x1<<0);
}

int main ()
{
int i,brightness,stepSize;
brightness=1;
stepSize=1;
        TIM4_Init ();
        configurePWM();
        configureLED();
        TIM4->CCR1=500;  //Initial duty cycle 50%
        TIM4->CR1 |= 1UL; //Start the timer
while(1)
        {
                        if(brightness>=999)
                        brightness=1;
                        brightness+=stepSize;
                        TIM4->CCR1=brightness;
                        for(i=0;i<5000;i++);
                        }
}
```

4.2 Program to Generate a Multichannel PWM Signals:

```
#include "stm32f4xx.h"

void GPIO_Config(void);
void TIM_Config(void);

int main(void)
{
	GPIO_Config();
	TIM_Config();
	//Start Timer2
	TIM4->CR1 |= 1UL;
	while(1)
	{
	}
}

void GPIO_Config(void)
{
	//GPIOD clock enable
	RCC->AHB1ENR |= (1UL << 3);
	//Set GPIOD pins 12 - 15 to alternate mode
	GPIOD->MODER |= (2UL << 12*2) | (2UL << 13*2) | (2UL << 14*2) | (2UL << 15*2);
	//Set alternate function to AF2 (TIM4)
	GPIOD->AFR[1] |= (2UL << 4*4) | (2UL << 5*4) | (2UL << 6*4) | (2UL << 7*4);
}
void TIM_Config(void)
{
	// Basic timer configuration
	RCC->APB1ENR |= (1UL << 2);		//Enable TIM4 clock
	TIM4->CR1 &= ~(0x0010);			//Set the mode to Count up
	TIM4->PSC = 160-1;                                                    //Set the Prescalar
	TIM4->ARR = 1000-1;                                                          //Set period
(Auto reload) 10ms duration of total pulse

	//Channel 1 config
	TIM4->CCMR1 |= (6UL << 4);    //Set OutputCompare mode to PWM
	TIM4->CCMR1 &= ~(3UL << 0);   //CC to output
	TIM4->CCER &= ~(1UL << 1);		//Output Compare polarity to active high
	TIM4->CCER |= (1UL << 0);		//Capture compare output enable
	//Set the Pulse width
	TIM4->CCR1 = 100;//10% duty cycle

	//Channel 2 config
	TIM4->CCMR1 |= (6UL << (4+8));    //Set OutputCompare mode to PWM
	TIM4->CCMR1 &= ~(3UL << 8);   //CC to output
	TIM4->CCER &= ~(1UL << (1+4));		//Output Compare polarity to active high
```

```
        TIM4->CCER |= (1UL << 4);           //Capture compare output enable
        //Set the Pulse width
        TIM4->CCR2 = 200;//20% duty cycle

        //Channel 3 config
        TIM4->CCMR2 |= (6UL << 4);    //Set OutputCompare mode to PWM
        TIM4->CCMR2 &= ~(3UL << 0);   //CC to output
        TIM4->CCER &= ~(1UL << 9);          //Output Compare polarity to active high
        TIM4->CCER |= (1UL << 8);           //Capture compare output enable
        //Set the Pulse width
        TIM4->CCR3 = 300;//30% duty cycle

        //Channel 4 config
        TIM4->CCMR2 |= (6UL << (4+8));    //Set OutputCompare mode to PWM
        TIM4->CCMR2 &= ~(3UL << 8);   //CC to output
        TIM4->CCER &= ~(1UL << 13);         //Output Compare polarity to active high
        TIM4->CCER |= (1UL << 12);          //Capture compare output enable
        //Set the Pulse width
        TIM4->CCR4 = 400;//40% duty cycle
}
```

5. Conclusion:

In this experiment, generation of PWM signals were learnt, which can be applied to various applications, like control of DC motor.

Current Measurement using ADC in STM32F407 Microcontroller

1. Aim:

Measure the AC/DC current in a power circuit using a LEM LA-55P current sensor and a STM32F407 Discovery Board.

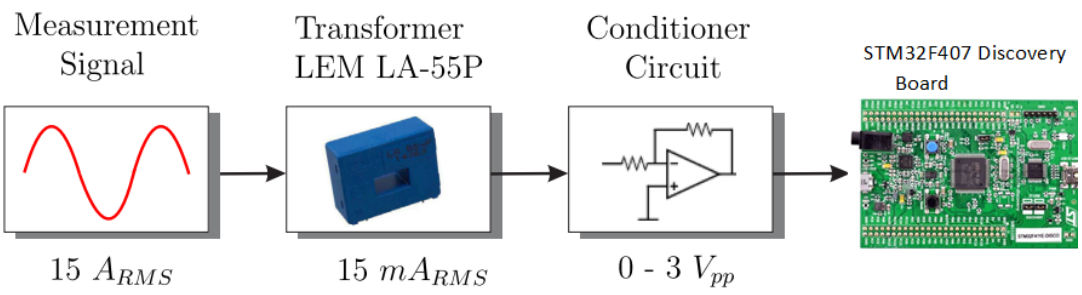
2. Introduction:

An ADC (Analog-To-Digital) converter is an electronic circuit that takes in an analog voltage as input and converts it into digital data, a value that represents the voltage level in binary code. The ADC samples the analog input whenever you trigger it to start conversion. And it performs a process called quantization so as to decide on the voltage level and its binary code that gets pushed in the output register.

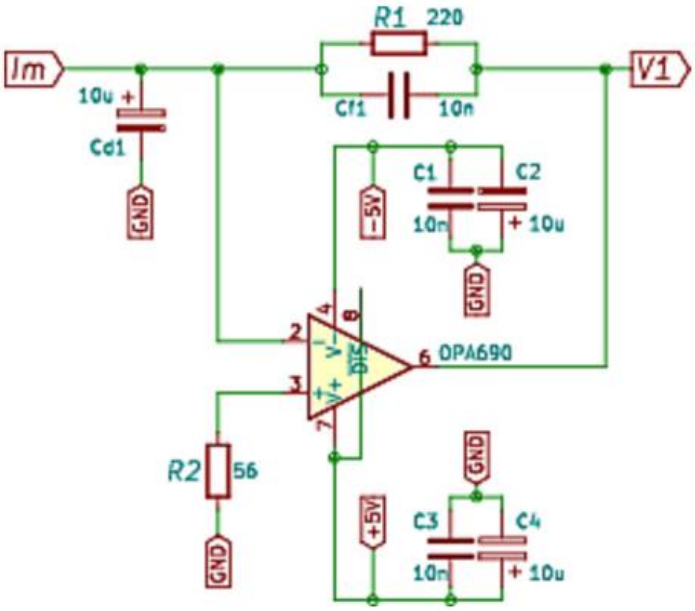
The ADC is one of the most expensive electronic components especially when it does have a high sampling rate and high resolution. Therefore, it's a valuable resource in microcontrollers and different manufacturers provide us (the firmware engineers) with various features so as to make the best use of it. And the flexibility also to make a lot of decisions like sacrificing resolution in exchange for a higher resolution or having the ADC to trigger on an internal timer signal to periodically sample the analog channels, and much more as we'll see in this tutorial.

3. Hardware Circuit:

The current measurement scheme and conditioning circuit consist of three stages, as shown in Figure. The first stage is responsible for obtaining a voltage of 3 V_{pp} proportional to the output of the current sensor LEM LA 55-P. For this reason, a trans-impedance amplifier is used, as shown in the figure below.



Current Measurement System



Trans-impedance Amplifier Stage

Resistor R1 sets the gain for this stage.

$$V_1 = I_m R_1$$

Subsequently, the next stage generates a reference of 1.5 V, used for an offset voltage. Finally, V_1 is added with 1.5V to provide the offset. The final output voltage of signal conditioning circuit is 0-3V.

The firmware has been implemented on the proposed experiment using the following flowchart.

To measure the current, collect the samples for 1second or 8000 samples. Using those samples, calculate the V_{rms} value as given in the equation.

$$V_{rms} = \sqrt{\frac{1}{N}\sum v^2(n)}$$

Finally, to convert it into a current value I_{rms} use the following equation.

$I_{rms} = V_{rms}$ x Scaling Factor

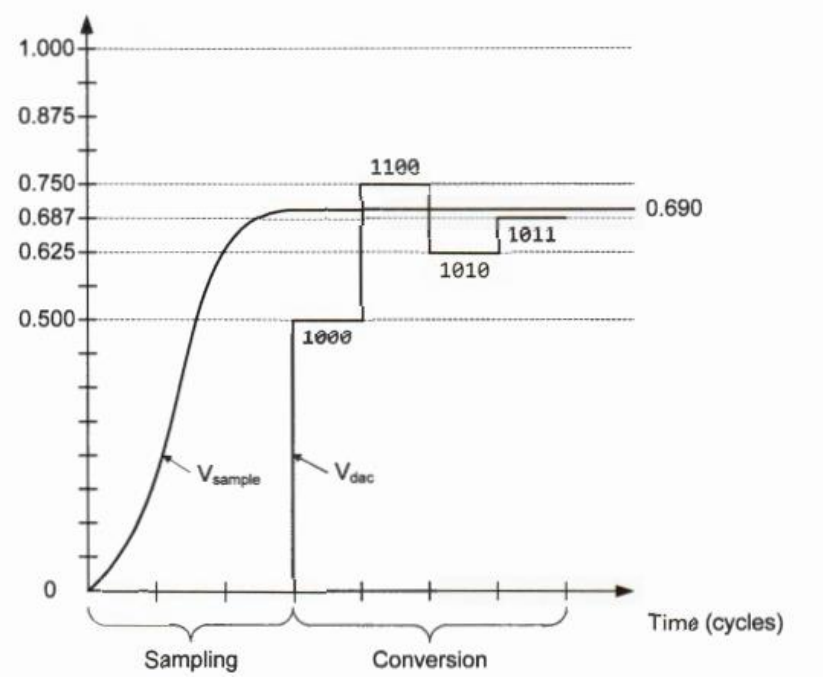
4. ADC Registers Configuration:

An analog-to-digital converter (ADC) produces a finite-precision signed or unsigned digital number to represent approximately the size of an analog voltage relative to a reference voltage. The reference voltage is a fixed voltage provided by the internal circuit of the microprocessor or by an external circuit connected to a pin of the microprocessor. It does not convert a voltage larger than the reference voltage.

There are three most popular ADC architectures: sigma-delta ADC for low-speed applications, successive-approximation (SAR) ADC for low-power applications, and pipelined ADC for high-speed applications.

Sigma-delta ADCs are mostly used in applications requiring low sampling rates, but high resolution, typically less than 100 kilo samples per second and 12 to 24- bit resolution, such as voice band and audio applications. They have been widely used in modern cell phones.

SAR ADCs are suitable for applications with low-power data acquisitions and moderate sampling rates, typically less than 5 million samples per second (MSPS). If the ADC has a resolution of n bits, the successive-approximation conversion takes n steps to complete. It plays a tradeoff between the resolution and sampling rate. A higher resolution usually reduces the ADC conversion rate. An example of four-bit successive-approximation (SAR) ADC. Suppose the input voltage is between 0 and 1 V.



The figure above shows an example of converting an input voltage of 0.690V into a 4-bit binary value by using SAR ADC. Suppose the range of the input voltage is between 0V and 1V.

After the input voltage is sampled, the SAR control logic starts the conversion by setting V_{dac} as O.5V (i.e., the ADC output is 0b1000) and comparing it with V_{sample}.

Since V_{sample} is larger than V_{dac}, the control logic then sets V_{dac} as 0.750V (i.e., the ADC output is 0b1100).

Because V_{sample} is smaller than V_{dac}, the control logic then sets V_{dac} as 0.625V.

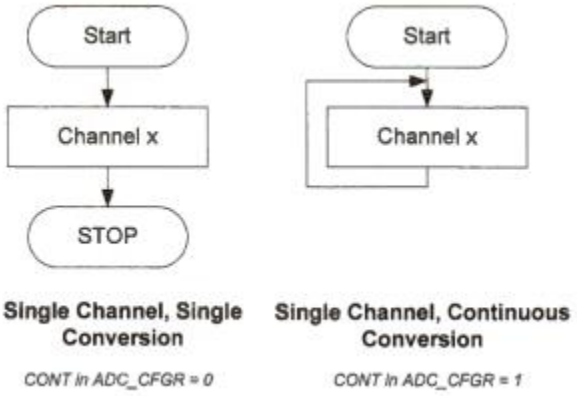
The above process repeats, and the final output of the ADC is 0b1011. In this example, the conversion process takes four cycles.

In the single-end mode, if the ADC result has n bits, we can calculate the conversion result of an input voltage V as follows:

$$V = \frac{Digital\ Value}{2^n} \times V_{REF}$$

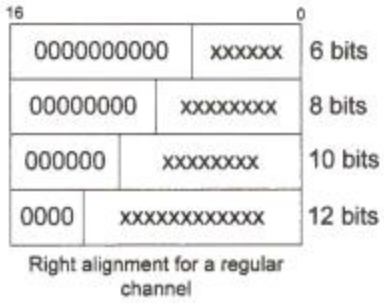
4.1 ADC Conversion:

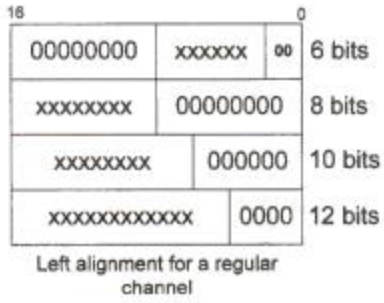
The conversion operation can be set up to perform only once or repeatedly, depending on bit CONT bit in the ADC_CFGR register. After each conversion, ADC result should be copied to a user buffer because ADC may overwrite the ADC data registers. An interrupt request or a DMA request can be triggered at the end of each conversion if enabled. Thus, to reduce the software overhead, we often use the ADC interrupt handler or the DMA controller to copy the ADC results to a user buffer.



4.2 ADC Data Alignment:

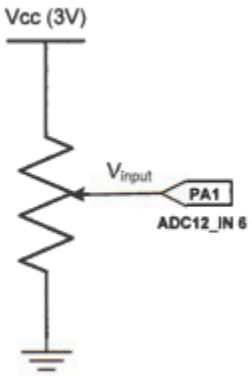
Software can change the ADC resolution. The resolution can be either 12, 10, 8 or 6 bits, determined by the RES[1:0] bits in the ADC configuration register ADC_CFGR. However, each ADC data register (DR) has 16 bits. Because ADC results have fewer bits than ADC data registers, alignment must be considered when a data result is stored in a data register. Figure 20-8 shows different data alignment formats. ADC output data registers can be either right-aligned or left-aligned.





4.3 Measuring the Input Voltage:

A potentiometer, informally a pot, is a three-terminal variable resistor. It uses a sliding contact and works as an adjustable voltage divider. When two outer terminals are connected to V_{CC} and the ground respectively, the center terminal generates a voltage that varies from 0 to V_{CC} depending on the position of the sliding contact. In the following sections, we use the internal voltage reference, which is 3V. In this example, we measure the input voltage adjusted by a potentiometer. If the input voltage V_{input} is larger than 1/2 of V_{CC}, then we turn on an LED. An interesting application is that we use the potentiometer to control the brightness of an LED dynamically if a PWM controls the LED. The V_{input} is used to adjust the duty cycle of the PWM output signal.



Suppose the ADC result has 12 bits and ADC is configured as single-ended. Then, we have the following conversion result:

$$ADC\ Result = \frac{V_{input}}{V_{REF}} \times 4096$$

Therefore, we have

$$V_{input} = \frac{ADC\ Result}{4096} \times V_{REF}$$

5. Program for Current Measurement using ADC:

```
#include "stm32f4xx.h"

void ADC_Init(void);
void ADC_Enable(void);
void ADC_Start(int);
float32_t value[8500],value1[8500],sum,RMS,out,currentRMS;
uint8_t ADC_VAL[8500];
uint32_t len=8500;
float32_t step=1.5/128;
void ADC_Init (void)
{
		/*************** STEPS TO FOLLOW ******************
		1. Enable ADC and GPIO clock
		2. Set the prescalar in the Common Control Register (CCR)
		3. Set the Resolution in the Control Register 1 (CR1)
4. Set the Continuous Conversion, EOC, and Data Alignment in Control Reg 2 (CR2)
		5. Set the Sampling Time for the channels in ADC_SMPRx
		6. Set the Regular channel sequence length in ADC_SQR1
		7. Set the Respective GPIO PINs in the Analog Mode
		**********************************************/

//1. Enable ADC and GPIO clock
		RCC->APB2ENR |= (1<<8);  // enable ADC1 clock
		RCC->AHB1ENR |= (1<<0);  // enable GPIOA clock
		RCC->CFGR |= 6<<13; // set APB2 = 2 MHz (16/8)

//2. Set the pre-scalar in the Common Control Register (CCR)
		ADC->CCR |= 0<<16;			// PCLK2 divide by 2

//3. Set the Resolution in the Control Register 1 (CR1)
		//ADC1->CR1 &= ~(1<<8);    // SCAN mode disabled, enable for multichannel use
		ADC1->CR1 |= (2<<24);   // 8 bit RES

//4. Set the Continuous Conversion, EOC, and Data Alignment in Control Reg 2 (CR2)
			//ADC1->CR2 |= (1<<1);     // enable continuous conversion mode
			ADC1->CR2 |= (1<<10);    // EOC after each conversion
			ADC1->CR2 &= ~(1<<11);   // Data Alignment RIGHT
```

```
//5. Set the Sampling Time for the channels
		//ADC1->SMPR2 &= ~(1<<0);  // Sampling time of 112 cycles for channel 0
		ADC1->SMPR2|=(5<<0);
//6. Set the Regular channel sequence length in ADC_SQR1
		//ADC1->SQR1 &= ~(1<<20);   // SQR1_L =0 for 1 conversion

//7. Set the Respective GPIO PIN in the Analog Mode
		GPIOA->MODER |= (3<<0);  // analog mode for PA 0 (channel 0)

}
void msDelay(uint32_t msTime)
{
		/* For loop takes 4 clock cycles to get executed. Clock frequency of stm32f407 by default is
16MHz
		So, 16MHz/4=4MHz. If we want 1000ms delay, 4MHz/1000=4000, so we have to multiply by
4000 to get a delay of 1s
		*/
		for(uint32_t i=0;i<msTime*3000;i++)
		{
			__NOP();
		}

}

int main ()
{
		ADC_Init ();
		ADC1->CR2 |= 1<<0;   // ADON =1 enable ADC1
		uint32_t delay = 10000;
		while (delay--);//Wait sometime for ADC to start

		//ADC1->CR2 |= (1<<30);  // start the conversion
		while (1)
		{
			for(int i=0;i<8500;i++)
			{
				ADC1->CR2 |= (1<<30);  // start the conversion
				ADC1->SR = 0;       // clear the status register
				//ADC1->CR2 |= (1<<30);  // start the conversion
				while (!(ADC1->SR & (1<<1)));  // wait for EOC flag to set
				ADC_VAL[i]=ADC1->DR;
			}
//((float)value[i]-128)*step
//float step=1.5/128
			for(int j=0;j<8500;j++)
                                {
```

```
                                        value[j]=((float)ADC_VAL[j]);
                                        value1[j]=(((float32_t)ADC_VAL[j]*0.0118)-1.506);

                                        }

                                arm_rms_f32(value1,len,&RMS);
                        //sum=sum/8500;
                        //arm_sqrt_f32(sum,&out);
currentRMS=RMS*1.658;
                }

}
```

6. Conclusion:

In conclusion, current from the current sensor has been measured using the ADC in STM32F407.

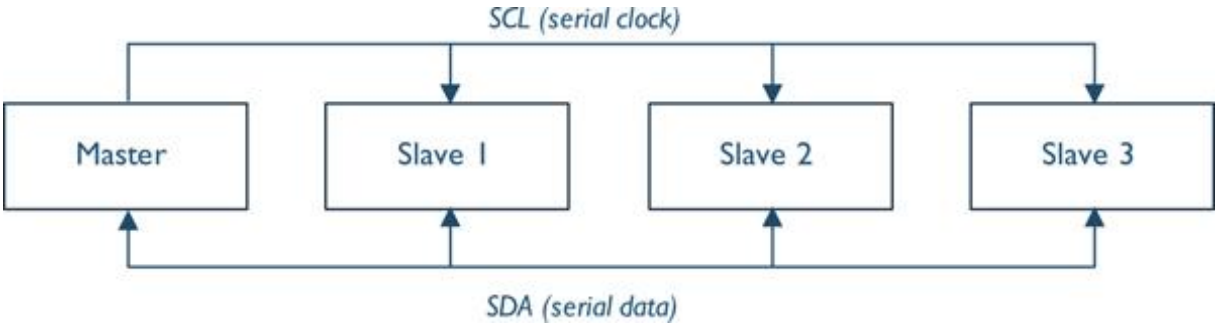
I2C Communication based LCD Interface using an 8-bit I/O Expander

1. Aim:

To display the string data using PCF8574A LCD and 8-bit I/O expander with help of 2-wire I2C interface.

2. Introduction:

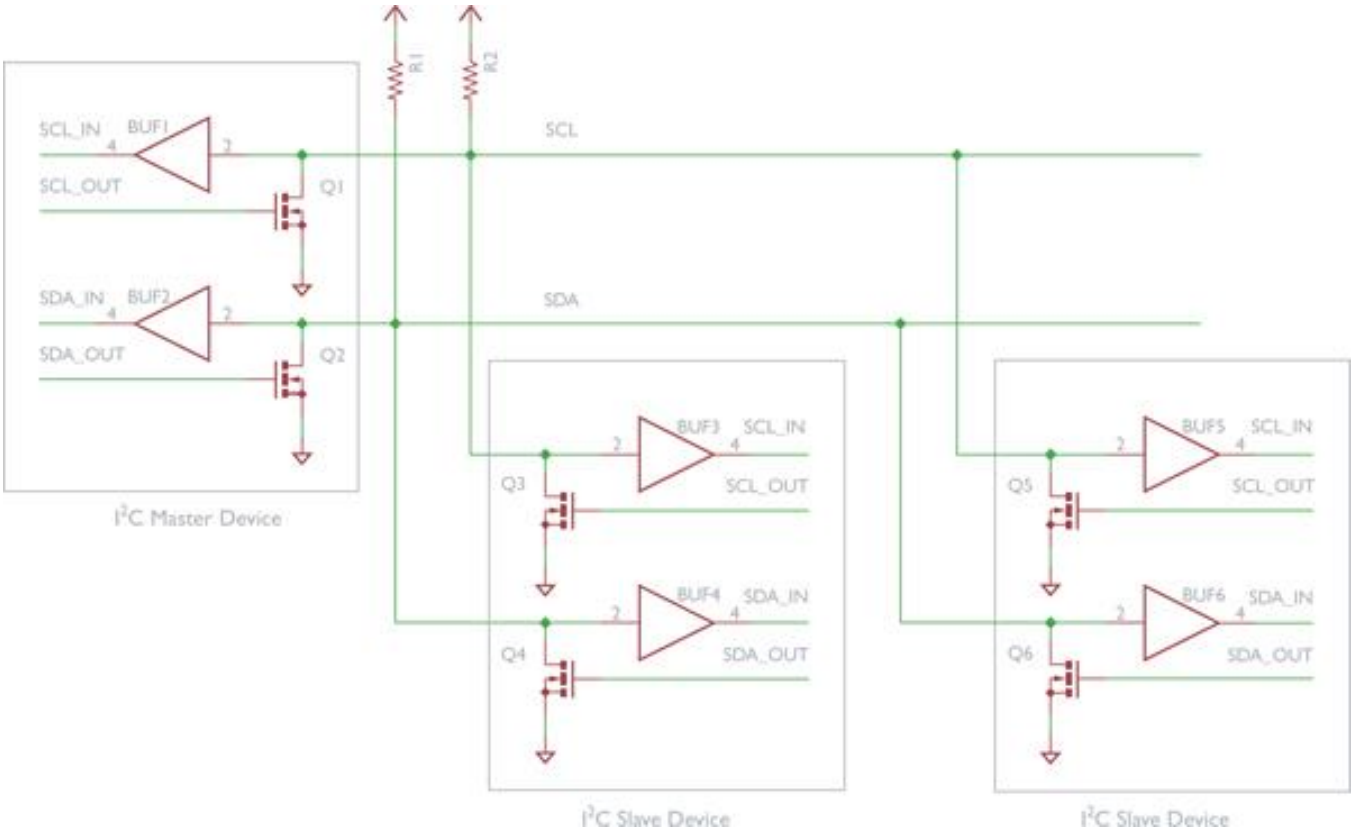
I2C is a synchronous protocol that uses a serial data (SDA) signal and serial clock (SCL) signal. I2C is a master/slave protocol. The master initiates all communications, and slave devices transmit only when the master allows them to. A major feature of the protocol is device addressing: each message includes device addressing information, and each device on the bus has a unique address. Only the addressed device will respond to a message. This allows a system to be built that shares the SDA/SCL bus as shown in the figure below without using additional control signals (such as slave selects for SPI).



The circuits driving the SDA and SCL signals are shown in Figure and are designed so that multiple devices can drive the signal simultaneously without damage. Each device's drive circuit consists of a transistor connected between the I2C bus signal (SDA or SCL) and ground. A separate pull-up resistor is connected between the signal and VDD. For the master to send a 0 on SDA, the transistor Q2 is turned on, pulling SDA to ground. To send a one, the transistor Q2 is turned off, allowing the resistor R1 to pull SDA up to VDD. If two devices attempt to transmit different data simultaneously, the SDA signal will be a zero.

I2C supports a range of communication speeds: 100 kbit/s (standard), 400 kbit/s (full speed), 1 Mbit/s (fast), 3.2 Mbit/s (high speed). The maximum communication speed for a given system implementation is limited by how quickly the pull-up resistor can pull SCL or SDA up to VCC.

This depends on the capacitance of the SCL or SDA signal, which is affected by the number of devices and the bus length. The maximum speed for a given device will be listed in its data sheet or reference manual.



3.1 Message Format:

There are several types of I2C message, with the basic format shown in Figure. Each message contains several fields:

•The start condition indicates the start of a message.
•The slave device address identifies the target of the communication.
•The read/write bit indicates whether the following data is to be read from the slave or written to it.
•The acknowledgment bit has two uses: to indicate if the addressed slave is present, and whether more data will be read. These are explained later in this section.
•One or more data bytes. For some I2C slave devices, the first data byte will be interpreted as a register address.
•A stop condition indicates the end of a message.
•An optional repeated start condition is used in some types of messages.

	Slave Device Address									Data Byte									
Start	AD7	AD6	AD5	AD4	AD3	AD2	AD1	R/W	ACK	D7	D6	D5	D4	D3	D2	D1	D0	ACK	Stop

Communication operations are structured as sequences of conditions (e.g. start, stop) and data transfers (one byte and one acknowledgment bit).

3.2 Device Addressing:

The master uses the device address to select a particular slave device on the bus. There are two addressing modes, one with 7-bit addresses and another with 10-bit addresses. For simplicity we will just discuss the 7-bit mode.

The first byte in the message has two parts, as shown in Figure. The device address is held in the upper seven bits, and the R/W bit is the LSB. This bit indicates whether the master will read from the slave (one) or write to it (zero). In practice, this byte is formed by shifting the slave address left by one bit and then adding the R/W bit.

i. Sequence for a master to write two data bytes to a slave device

Text indicates transmitting device. Blank indicates listening device.

Master	Start	Slave Dev. Add.	W(0)		Data		Data		Stop
Slave				ACK (0)		ACK(0)		ACK(0)	

•The master first sends the start condition, the slave device address, and a write command (zero). If the addressed slave is present, it will assert the ACK bit. If the ACK bit is not asserted, then the master will terminate the message with a stop condition.

•The master sends the first byte of data.

•The slave sends an ACK to indicate it has been received.

•The master sends the second byte of data.

•The slave sends an ACK to indicate it has been received.

•The master sends the stop condition, indicating to all slaves that the message has completed.

ii. Sequence for a master to read two data bytes from a slave device

Master	Start	Slave Dev. Add.	R(1)			ACK(0)		NACK(1)	Stop
Slave				ACK(0)	Data		Data		

- The master first sends the start condition, the slave device address, and a read command (one). If the addressed slave is present, it will assert the ACK bit. If the ACK bit is not asserted, then the master will terminate the message with a stop condition.

•The master clocks the first byte of data out of the slave.
•The master sends an ACK to indicate it will read more data.
•The master clocks the second byte of data out of the slave.
•The master sends a NACK (one) to indicate that it does not want to read any more data in this message.
•The master sends the stop condition, indicating to all slaves that the message has completed.

3.3 Register Addressing:

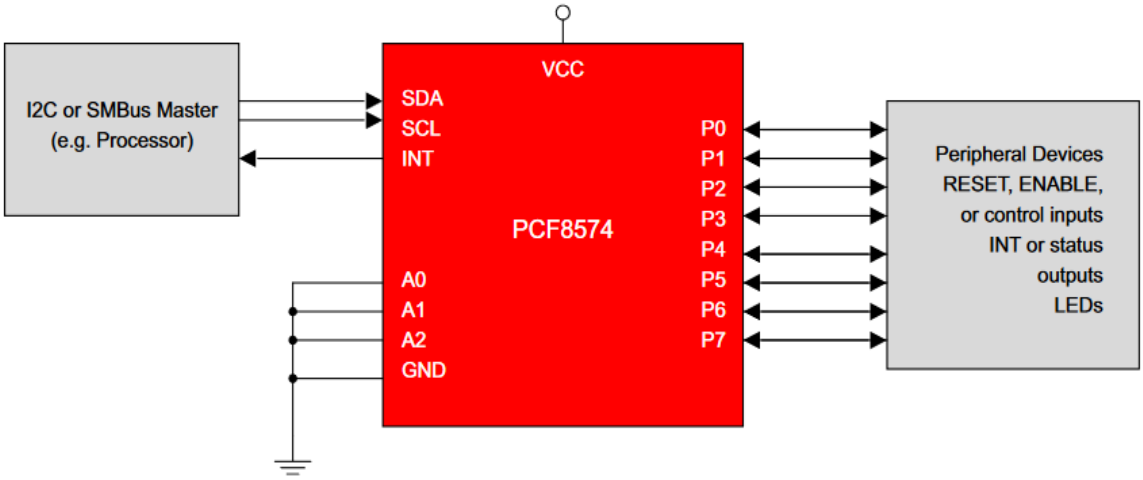
I2C also supports register addressing, in which each device is structured as a series of addressable registers that can be read or written. This standardizes information organization and simplifies system development. The first data byte is interpreted by the slave device as a register address.

Sequences of writing and reading to a register in a device.

Master	Start	Slave Dev. Add.	W(0)		Slave Reg. Add.		Data		Stop
Slave				ACK(0)		ACK(0)		ACK(0)	

Master	Start	Slave Dev. Add.	W(0)		Slave Reg. Add.		Start	Slave Dev. Add.	R(0)			NACK(1)	Stop
Slave				ACK(0)		ACK(0)				ACK(0)	Data		

3.4 I/O Expander Addressing:

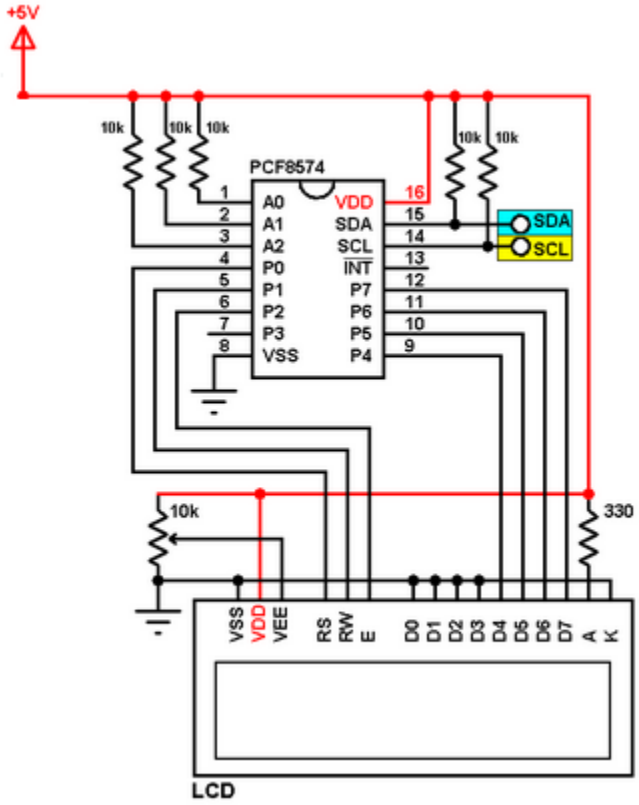


As shown in the figure below, based on the A0, A1, and A2 values device address is determined. In this experiment.

INPUTS			I²C BUS SLAVE 8-BIT READ ADDRESS	I²C BUS SLAVE 8-BIT WRITE ADDRESS
A2	A1	A0		
L	L	L	65 (decimal), 41 (hexadecimal)	64 (decimal), 40 (hexadecimal)
L	L	H	67 (decimal), 43 (hexadecimal)	66 (decimal), 42 (hexadecimal)
L	H	L	69 (decimal), 45 (hexadecimal)	68 (decimal), 44 (hexadecimal)
L	H	H	71 (decimal), 47 (hexadecimal)	70 (decimal), 46 (hexadecimal)
H	L	L	73 (decimal), 49 (hexadecimal)	72 (decimal), 48 (hexadecimal)
H	L	H	75 (decimal), 4B (hexadecimal)	74 (decimal), 4A (hexadecimal)
H	H	L	77 (decimal), 4D (hexadecimal)	76 (decimal), 4C (hexadecimal)
H	H	H	79 (decimal), 4F (hexadecimal)	78 (decimal), 4E (hexadecimal)

The address used for this experiment is 0x4E for Write, and 0x4F for Read. Since the LCD display is used in the experiment, only the write operation used throughout the experiment.

4. Hardware Schematic Diagram:



5. Program to Send a String “VIT” from a Microcontroller to PCF8574A LCD and 8-bit I/O Expander:

```
#include "stm32f4xx.h"                    // Include header file for the board

unsigned int temp;                  // Temporary variable to clear the bit

char Mes[10]="VIT";
// Pre initializing the functions
void configure_GPIOB(void);
void LCD_init(void);
void Transmit(char*);
void msDelay(uint32_t msTime);
void configure_I2C(void);
void I2C_start(void);
void I2C_send(uint8_t data);
void I2C_address(void);
```

```
void I2C_stop(void);
void I2C_write4(uint8_t data);
void I2C_write8(uint8_t data);
void I2C_write4C(uint8_t data);
void I2C_write8C(uint8_t data);


int main ()
{
configure_GPIOB();                    // To configure the GPIO pins PB6 and PB7
configure_I2C();                      // To configure the I2C1 register
msDelay(1);
LCD_init();
                                                        //lower nibble
   Transmit(Mes);



            while (1)
                        {}
            }

void configure_GPIOB(void){                             //to configure the GPIO pins PB6 and PB7
            //SDA - PB7, SCL - PB6
            RCC -> AHB1ENR |=(1UL<<1);                          //Enable clock for port B
            GPIOB -> MODER |=(2UL<<(6*2));                      //PB6 to alternate function pin
"10"
            GPIOB -> MODER |=(2UL<<(7*2));                      //PB7 to alternate function pin
"10"
            GPIOB -> PUPDR |=(0x5UL<<(6*2));        //set PB6 and PB7 as pull up pins "01"
            GPIOB -> OTYPER |=(0x3UL<<6);                       //Set PB6 and PB7 as open drain
"1"
            GPIOB -> OSPEEDR |=(0xAUL<<(6*2));                  //Set PB6 and PB7 as high speed
"10"
            GPIOB -> AFR[0] |= (0x44<<(6*4));       //Set PB6 and PB7 to alternate function 4
"0100"
}

void msDelay(uint32_t msTime){                                  //function for software delay
            // "For loop" takes 4 clock cycles to get executed. Clock frequency is 16MHz
            // 16MHz/4=4MHz. If we want 1000ms (1second) delay, 4MHz/1000=4000, so we have to
multiply by 4000 to get a delay of 1s
            for (uint32_t i=0;i<msTime*4000;i++){
                        __NOP();
            }
}
void configure_I2C(void){                               // To configure the I2C registers
            RCC -> APB1ENR |=(1UL<<21);                         // Enable I2C clock
            I2C1 -> CR1 |= (1UL<<15);                   // Reset I2C
            I2C1 -> CR1 &= ~(1UL<<15);                        // set I2C
            I2C1 -> CR2 |=(16UL<<0);                    // Set peripheral clock at 16MHz
```

```
        I2C1 -> OAR1 |=(1UL<<14);                   // Should be set high
        I2C1 -> CCR |=(0x50UL<<0);                  // Set SCL as 100KHz
        I2C1 -> TRISE |=(17UL<<0);                  // Configure maximum rise time
        I2C1 -> CR1 |= (1UL<<0);                    // Enable I2C
}
void I2C_start(void){                               // To send the I2C start bit
        I2C1 -> CR1 |= (1<<10);                     // Enable the ACK Bit
        I2C1 -> CR1 |= (1<<8);                      // Send the start bit
        while (!(I2C1->SR1 & (1<<0)));              // Wait for SB bit to set
}
void I2C_address(void){                             // To send the I2C address
        I2C1 -> DR = 0x4E;                          // Send the slave address     // change
according to the device
        while (!(I2C1->SR1 & (1<<1)));              // Wait for ADDR bit to set
        temp = ((I2C1->SR1) | (I2C1->SR2));         // Read SR1 and SR2 to clear the ADDR bit
}
void I2C_send(uint8_t data){                        // To send data to the I2C device
        while (!(I2C1->SR1 & (1<<7)));              // Wait till TX buffer is empty
        I2C1 -> DR = data;                          // Write data to I2C slave
        while (!(I2C1->SR1 & (1<<2)));              // Wait till Byte transfer is completed
}
void I2C_stop(void){                                // To send the I2C stop bit
        I2C1 -> CR1 |= (1<<9);                      // Stop I2C
}
void I2C_write4(uint8_t data){                      // To send upper four bit for commands
        uint8_t d=0;                                // Reset value for d
        d |= (data & 0xF0);                         // To select only the upper nibble
        d |= 1<<2;                                  // To set the enable bit
        d |= 1<<3;                                  // To set the backlight
        I2C_start();                                // Send the start bit
        I2C_address();                              // Send the I2C address
        I2C_send(d);                                // Send the data
        msDelay(1);                                 // Wait for 1ms
        d &= ~(1<<2);                               // Set the enable bit low
        I2C_send(d);                                // Send the same data but without enable bit to
latch the data
        I2C_stop();                                 // Send the stop bit
}

void I2C_write8(uint8_t data){                      // To send lower four bit for commands
        uint8_t d=0;
        d |= ((data & 0x0F)<<4);                    // To select only the lower nibble
        d |= 1<<2;
        d |= 1<<3;
        I2C_start();
        I2C_address();
        I2C_send(d);
        msDelay(1);
        d &= ~(1<<2);
        I2C_send(d);
        I2C_stop();
```

```
}

void I2C_write4C(uint8_t d){  // To send upper four bits for data

        d |= 1<<2;
        d |= 1<<0;                      // To select the data register not the command
        d |= 1<<3;
        I2C_start();
        I2C_address();
        I2C_send(d);
        msDelay(1);
        d &= ~(1<<2);
        I2C_send(d);
        I2C_stop();
}


void LCD_init(void)
{
        // Initial commands to set up the LCD in 4 bit mode
                I2C_write4(0x30); //to clear the screen
                msDelay(1);
                I2C_write4(0x30); //to clear the screen
                msDelay(1);
                I2C_write4(0x30); //to clear the screen
                msDelay(1);
                I2C_write4(0x20); //to turn on the screen
                msDelay(1);

                // Set LCD to 4 bit mode, 2 lines, 5x8 font
                I2C_write4(0x28);    //upper nibble
                msDelay(1);
                I2C_write8(0x28);    //lower nibble
                msDelay(1);


                // Turn the display on with blinking cursor
                I2C_write4(0x0C);       //upper nibble
                msDelay(1);
                I2C_write8(0x0C);   //lower nibble
                msDelay(1);

                // Clear the contents on the screen
                I2C_write4(0x01);    //upper nibble
                msDelay(37);
                I2C_write8(0x01);    //lower nibble
                msDelay(1);

                // Set LCD to write from left to right
                I2C_write4(0x06);      //upper nibble
                msDelay(1);
```

```
                        I2C_write8(0x06);
}
void Transmit(char * add)
{
        char data,d_H=0,d_L=0;
        for(int i=0;i<3;i++)
        {
        data=add[i];
        d_H= (data & 0xF0);
        I2C_write4C(d_H);
        d_L= ((data & 0x0F)<<4);
        I2C_write4C(d_L);               //lower nibble
        msDelay(1);
                }
}
```

6. Conclusion:

In conclusion, the I2C communication protocol has been properly used to send strings of words to the LCD display from the microcontroller.

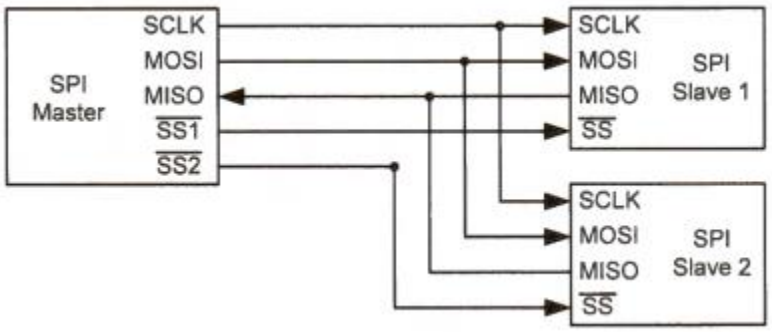
Interfacing a 3-axis MEMS Accelerometer using SPI Communication Protocol in ARM Cortex-M microcontroller

1. Aim:

i. Write embedded C program to study about Serial Peripheral Interface (SPI) communication protocol in STM32F407 microcontroller.
ii. Understand ADXL345 datasheet and interface it with microcontroller to capture the 3-axis data.

2. Introduction:

Serial peripheral interface (SPI) is a synchronous serial communication interface widely used to exchange data between a microprocessor and peripheral devices using four wires. For example, a digital camera often uses SPI to control its lens and save photos to a MMC or SD media. SPI is simple, has low power requirements, and supports high throughput. Disadvantages of SPI include that it does not support multiple masters, and slaves cannot start the communication or control data transfer speed. The master initiates and controls all communications. A SPI interface consists of four lines: a master-in-slave-out data line (MISO), a master-out-slave-in data line (MOSI), a serial clock line (SCLK), and an active-low slave select line (SS), as shown in the figure below.



SPI only supports a single master communicating with multiple slave devices. As shown in Figure, when the master wishes to exchange data with a slave, it pulls down the corresponding select line (SSN). The master then generates clock pulses to coordinate the data transmission on the MOSI and MISO lines.

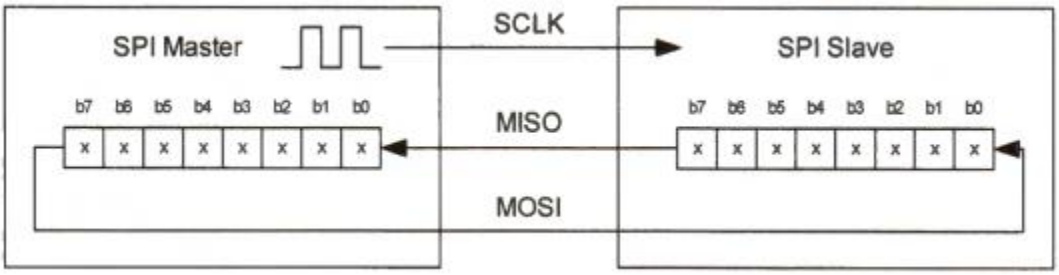
Data exchange can take place in both directions simultaneously, and this two-way serial channel is often called full duplex. Data bits are transmitted on both the MOSI line and the MISO line

synchronously, with the flow directions opposite to each other. Note the SCLK line has only one direction, and only the master can generate the clock signal. The slave devices cannot control the clock line. When there are multiple slave devices, the master decides which slave device it wants to communicate. There is a dedicated Slave Select (SS) line for each slave device. The master selects the target slave device by pulling the corresponding SS line to a low voltage prior to data transfer. The selected slave device then listens for the clock and MOSI signals. When there is only one slave device, the SS line can be directly connected to ground physically, or the program can make the slave continuously selected.

3.1 Data Exchange:

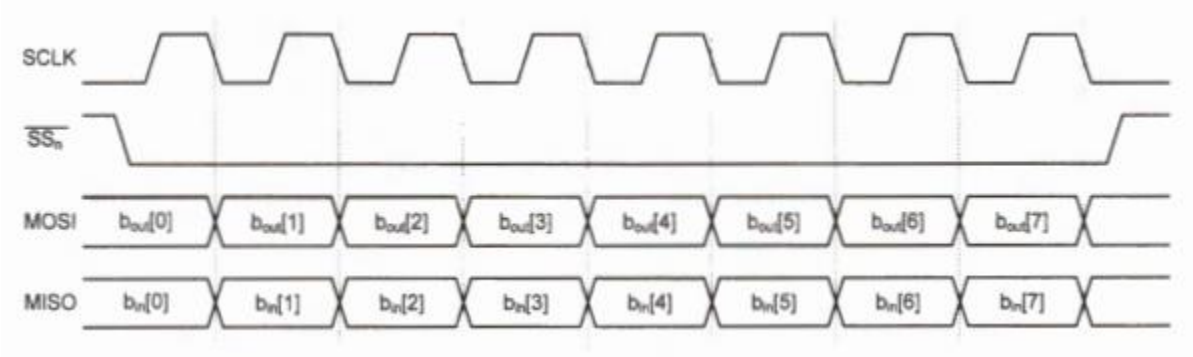
SPI is a synchronous protocol, and the slave devices must send and receive data based on the clock provided by the master. It differs from an asynchronous protocol in which no clock signal is provided physically. SPI devices must exchange data at the same speed. The master and a slave perform data exchange at synchronized time steps based on the clock signal generated by the master.

When a bit is shifted out on the MISO line from the slave's data register during a clock period, a new data bit is shifted into this register from the MOSI line in the same clock period, as shown in Figure. When one device writes a bit to the data line at the rising or falling edge of the clock, the other device then reads the bit at the opposite edge of the same clock period. The data transfer size is usually a byte or halfword (16 bits).



Communication from the master to a slave and communication from a slave to the master are always taking place concurrently. In each communication link (either MISO or MOSI), each device sends out a data item and at the same time receives a new data item. No devices can just be a transmitter or a receiver. Therefore, when a slave wants to send data to the master via the MISO line, the slave must wait for the clock signal. At the same time, the master must send some dummy data out via the MOSI line to generate the clock signal to initiate the data transfer.

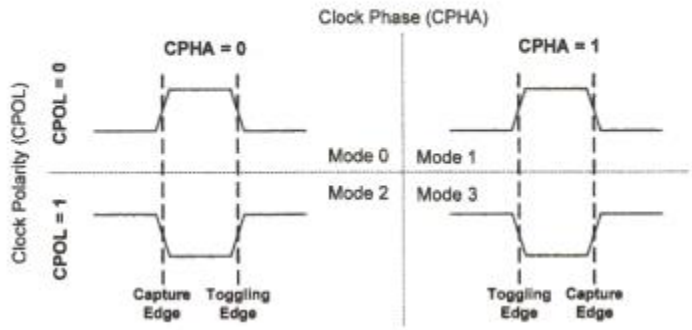
When a master exchanges data with slave n, the master must set SS_n low to select slave n, as shown in Figure. During the communication, the most significant bit of both data registers is sent out first.



3.2 Clock Configuration:

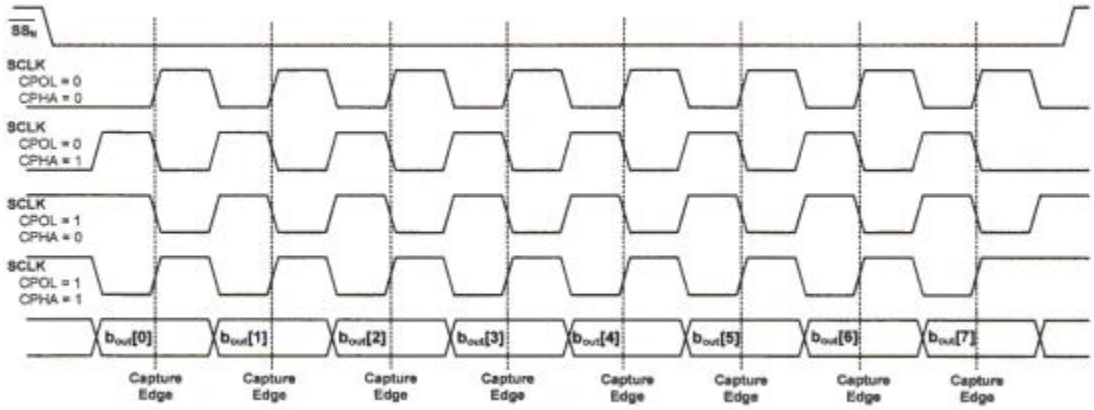
The clock speed determines the data transfer rate. The data rate ranges from 1 to 20 megabits per second. The master can change the clock speed by programming the clock prescaler register. For STM32L processors, the baud rate control factor is stored in the BR[2:0] bits of the SPI control register (CR1). The SCLK clock frequency is programmed by setting the baud rate control factor.

$$f_{SCLK} = \frac{f_{SYSCLK}}{2^{1+BR[2:0]}}$$



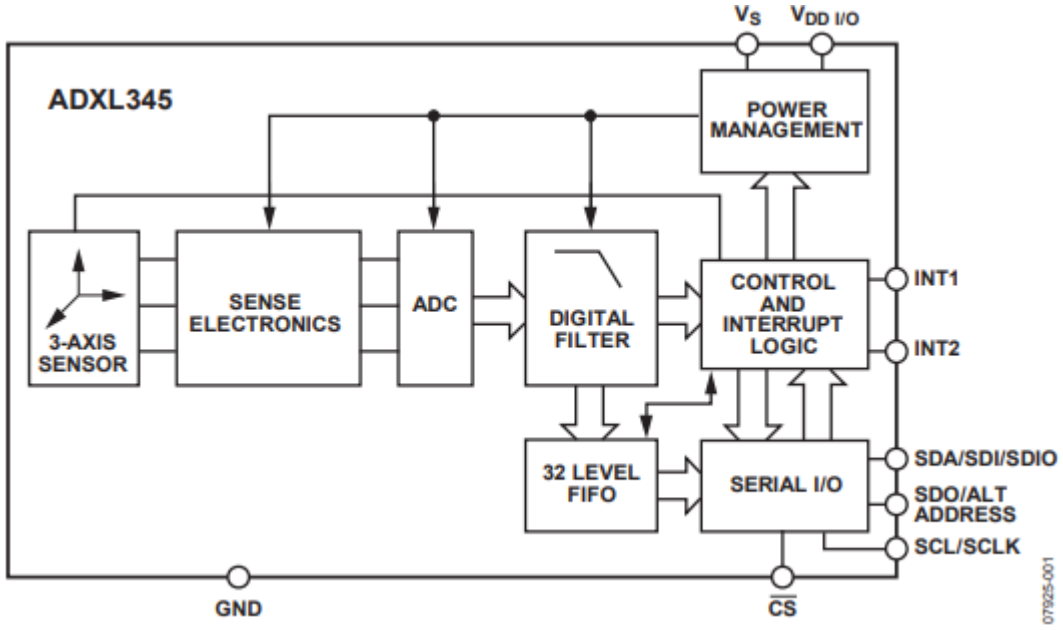
Four possible clock modes are available to program the clock edge used for data sampling and data toggling, as shown in Figure. The clock modes depend on two parameters: clock phase (CPHA) and clock polarity (CPOL). When CPOL is 0, the SCLK line is pulled low during idle time. When CPOL is 1, the SCLK line is pulled high during idle. When CPHA is 0, the first clock transition (either rising or falling) is the first data capture edge. When CPHA is 1, the second clock transition is the first capture edge. The combination of CPOL and CPHA selects the clock edge for

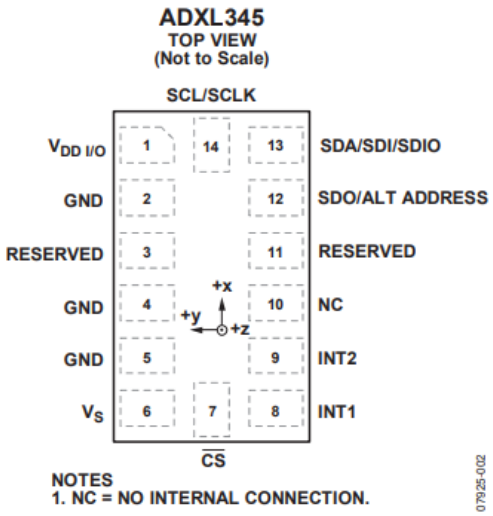
transmitting and capturing data. The capture of the first bit is delayed a half cycle in mode 0 and 2.



4. ADXL345 General Description:

The ADXL345 is a small, thin, ultralow power, 3-axis accelerometer with high resolution (13-bit) measurement at up to ±16 g. Digital output data is formatted as 16-bit twos complement and is accessible through either a SPI (3- or 4-wire) or I2C digital interface. The ADXL345 is well suited for mobile device applications. It measures the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion or shock. Its high resolution (3.9 mg/LSB) enables measurement of inclination changes less than 1.0°. The ADXL345 is supplied in a small, thin, 3 mm × 5 mm × 1 mm, 14-lead, plastic package. The functional block diagram is shown below.



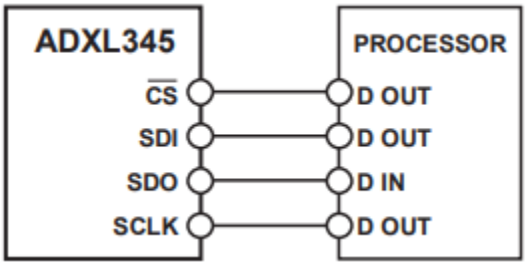


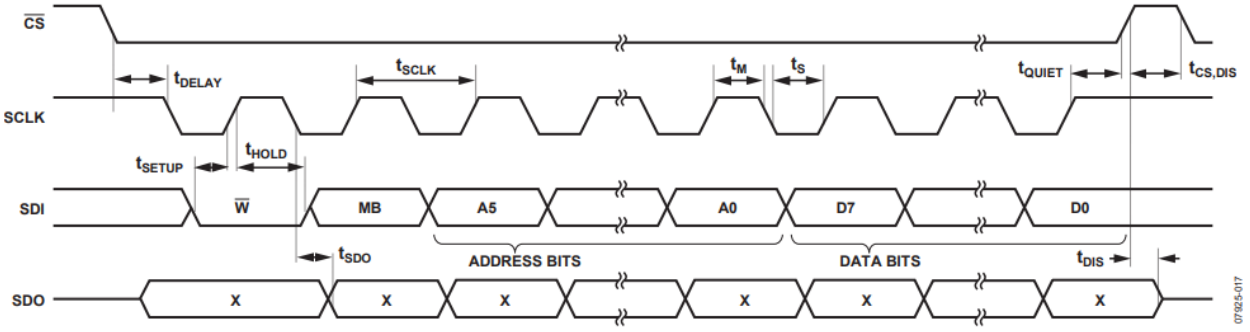
Pin No.	Mnemonic	Description
1	$V_{DD\ I/O}$	Digital Interface Supply Voltage.
2	GND	This pin must be connected to ground.
3	RESERVED	Reserved. This pin must be connected to V_S or left open.
4	GND	This pin must be connected to ground.
5	GND	This pin must be connected to ground.
6	V_S	Supply Voltage.
7	$\overline{CS}$	Chip Select.
8	INT1	Interrupt 1 Output.
9	INT2	Interrupt 2 Output.
10	NC	Not Internally Connected.
11	RESERVED	Reserved. This pin must be connected to ground or left open.
12	SDO/ALT ADDRESS	Serial Data Output (SPI 4-Wire)/Alternate I^2C Address Select (I^2C).
13	SDA/SDI/SDIO	Serial Data (I^2C)/Serial Data Input (SPI 4-Wire)/Serial Data Input and Output (SPI 3-Wire).
14	SCL/SCLK	Serial Communications Clock. SCL is the clock for I^2C, and SCLK is the clock for SPI.

The ADXL345 automatically modulates its power consumption in proportion to its output data rate. If additional power savings is desired, a lower power mode is available. In this mode, the internal sampling rate is reduced, allowing for power savings in the 12.5 Hz to 400 Hz data rate range at the expense of slightly greater noise. To enter low power mode, set the LOW_POWER bit (Bit 4) in the BW_RATE register (Address 0x2C).

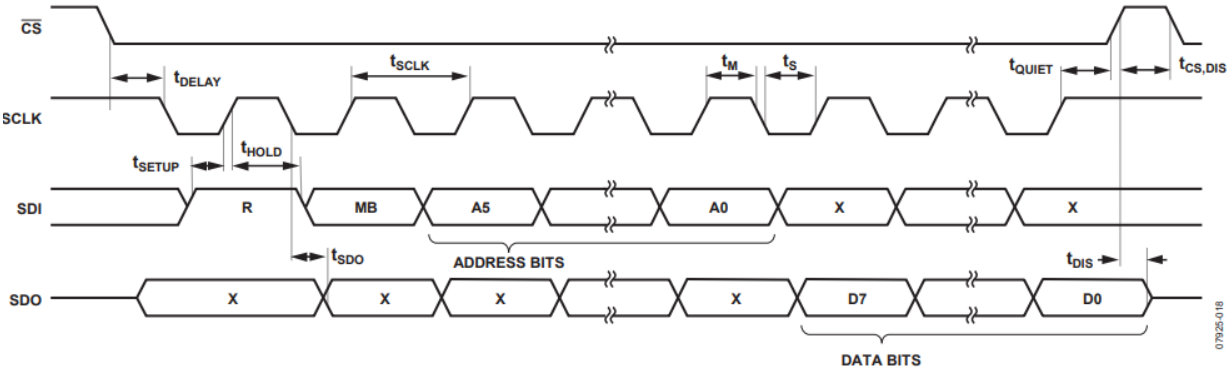
For SPI, either 3- or 4-wire configuration is possible, as shown in the connection diagrams. Clearing the SPI bit (Bit D6) in the DATA_FORMAT register (Address 0x31) selects 4-wire mode, whereas setting the SPI bit selects 3-wire mode. The maximum SPI clock speed is 5 MHz with 100 pF maximum loading, and the timing scheme follows clock polarity (CPOL) = 1 and clock phase (CPHA) = 1. If power is applied to the ADXL345 before the clock polarity and phase of the host processor are configured, the CS pin should be brought high before changing

the clock polarity and phase. When using 3-wire SPI, it is recommended that the SDO pin be either pulled up to VDD I/O or pulled down to GND via a 10 kΩ resistor.





SPI 4-Wire Write



SPI 4-Wire Read

- According to the timing diagram, in order to read/write multiple bits in same operation, the 6th bit should be set. That's why in the code, the register addresses are OR'd with 0x40, which sets the 6th bit high.
- Also, according to the timing diagram, the MSB defines whether it is a read or write operation. During write operation, the MSB should be 0 and for read operation, it should be HIGH. That's why for read operations, the register addresses are OR'd with 0x80, setting the MSB.
- Now, the appropriate registers in the sensor should be configured, so that it can be used in data transfer. First, the device should be activated. So, the measure bit (D3) should be set to high in order to start the measurement. So, 0x08 should be written into register 0x2D to start the device.

Register 0x2D—POWER_CTL (Read/Write)

D7	D6	D5	D4	D3	D2	D1	D0
0	0	Link	AUTO_SLEEP	Measure	Sleep	Wakeup	

- Next, range of measurement has to be selected. Writing 01 in the LSB sets it to +/-4g. Also, the interrupt configuration by default is active high. So, setting the bit D5 makes it active low. So, if interrupt is needed in active low, 0x21 has to be written in register 0x31.

Register 0x31—DATA_FORMAT (Read/Write)

D7	D6	D5	D4	D3	D2	D1	D0
SELF_TEST	SPI	INT_INVERT	0	FULL_RES	Justify	Range	

- The register corresponding to sampling rate is 0x2C. Bits D0 to D3 correspond to sampling rate. So, from the below table taken from the datasheet, we can set the lower nibble so as to get the required sampling rate.

Register 0x2C—BW_RATE (Read/Write)

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	LOW_POWER	Rate			

LOW_POWER Bit

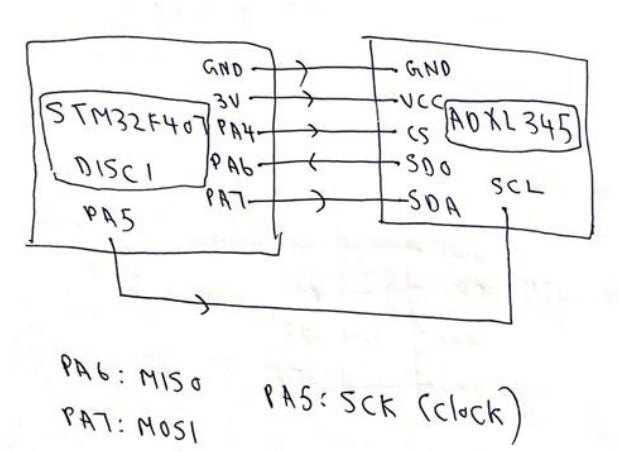
A setting of 0 in the LOW_POWER bit selects normal operation, and a setting of 1 selects reduced power operation, which has somewhat higher noise (see the Power Modes section for details).

Rate Bits

These bits select the device bandwidth and output data rate (see Table 7 and Table 8 for details). The default value is 0x0A, which translates to a 100 Hz output data rate. An output data rate should be selected that is appropriate for the communication protocol and frequency selected. Selecting too high of an output data rate with a low communication speed results in samples being discarded.

Output Data Rate (Hz)	Bandwidth (Hz)	Rate Code	I_{DD} (μA)
3200	1600	1111	140
1600	800	1110	90
800	400	1101	140
400	200	1100	140
200	100	1011	140
100	50	1010	140
50	25	1001	90
25	12.5	1000	60
12.5	6.25	0111	50
6.25	3.13	0110	45
3.13	1.56	0101	40
1.56	0.78	0100	34
0.78	0.39	0011	23
0.39	0.20	0010	23
0.20	0.10	0001	23
0.10	0.05	0000	23

5. Hardware Schematic Diagram:



6. Program for ADXL345 Interfacing:

```
#include "stm32f4xx.h"
#define add 0x32
int8_t array[6],buf;
int16_t x,y,z;
float xacc,yacc,zacc;

//ADXL345  Analog Devices
void GPIO_Config(void)
{
		//PA5,6,7-CLK,MISO,MOSI
		RCC->AHB1ENR |=(1UL<<0);//Enable Port A clock
		GPIOA->MODER |=(2UL<<10);//Set PA5,6,7 to AF
		GPIOA->MODER |=(2UL<<12);
		GPIOA->MODER |=(2UL<<14);
		GPIOA->AFR[0] |=(5UL<<20); //enable SPI CLK to PA5
		GPIOA->AFR[0] |=(5UL<<24); //enable MISO to PA6
		GPIOA->AFR[0] |=(5UL<<28); //enable MOSI to PA7
		//PA4-CS
		GPIOA->MODER |=(1UL<<8);//PA4 as output
}
void SPI_Config(void)
{
		RCC->APB2ENR |=(1UL<<12);//Enable SPI clock
		SPI1->CR1 |=(2UL<<3);//Set baud rate as 2Mbit/s
		SPI1->CR1 |=(1UL<<2);//Set as Master mode
```

```
	SPI1->CR1 |=(1UL<<1);//Set clock polarity as HIGH
	SPI1->CR1 |=(1UL<<0);//Set clock phase as HIGH
	SPI1->CR1 |=(3UL<<8);//Should be set HIGH
	SPI1->CR1 |=(1UL<<6);//Start SPI
	SPI1->CR2 = 0x0000;//Motorola format
}
int8_t SPI_Send (uint8_t byte)
{
	SPI1->DR = byte;
	while ((SPI1->SR) & (1<<7)); /* Wait for send to finish */
	buf=SPI1->DR;
	return buf;
}
void Accel_Write(uint8_t address,uint8_t val)
{
	GPIOA->ODR |=(1UL<<21);//Select accelerometer
	SPI_Send(address);
	SPI_Send(val);
	GPIOA->ODR |=(1UL<<5);//Disconnect accelerometer
}
void ADXL_Config(void)
{
	Accel_Write(0x2D,0x00);// Reset before configuring power register
	Accel_Write(0x2D,0x08);//Configure the power register, and turn on the device
	// Configuring the data format register
	//The 5th bit corresponds to setting interrupt to active low if set
	Accel_Write(0x31,0x01);//Lower nibble:Selecting +/- 4g range and 4 wire SPI mode
	//Configuring sampling rate to 100hz
	Accel_Write(0x2C,0x0A);
}
void Accel_Read(uint8_t address)
{
	//X0
	GPIOA->ODR |=(1UL<<21);//Select accelerometer
	address |=0x80;//Read operation
	SPI_Send(address);
	array[0]=SPI_Send(0);
	GPIOA->ODR |=(1UL<<5);//Disconnect accelerometer
	//X1
	GPIOA->ODR |=(1UL<<21);//Select accelerometer
	++address;
	SPI_Send(address);
	array[1]=SPI_Send(0);
	GPIOA->ODR |=(1UL<<5);//Disconnect accelerometer
	//Y0
	GPIOA->ODR |=(1UL<<21);//Select accelerometer
	++address;
	SPI_Send(address);
	array[2]=SPI_Send(0);
	GPIOA->ODR |=(1UL<<5);//Disconnect accelerometer
	//y1
```

```
		GPIOA->ODR |=(1UL<<21);//Select accelerometer
		++address;
		SPI_Send(address);
		array[3]=SPI_Send(0);
		GPIOA->ODR |=(1UL<<5);//Disconnect accelerometer
		//Z0
		GPIOA->ODR |=(1UL<<21);//Select accelerometer
		++address;
		SPI_Send(address);
		array[4]=SPI_Send(0);
		GPIOA->ODR |=(1UL<<5);//Disconnect accelerometer
		//Z1
		GPIOA->ODR |=(1UL<<21);//Select accelerometer
		++address;
		SPI_Send(address);
		array[5]=SPI_Send(0);
		GPIOA->ODR |=(1UL<<5);//Disconnect accelerometer
		x=((array[1]<<8)|array[0]);
		y=((array[3]<<8)|array[2]);
		z=((array[5]<<8)|array[4]);
		xacc=x*0.0078;
		yacc=y*0.0078;
		zacc=z*0.0078;
}
int main()
{
		GPIO_Config();
		SPI_Config();
		ADXL_Config();
		while(1)
		{
			Accel_Read(add);
		}
}
```

Conclusion:

SPI communication protocol was used to read data from ADXL345 sensor.

Implementation of Controller Area Network (CAN) using Cortex-M Microcontroller

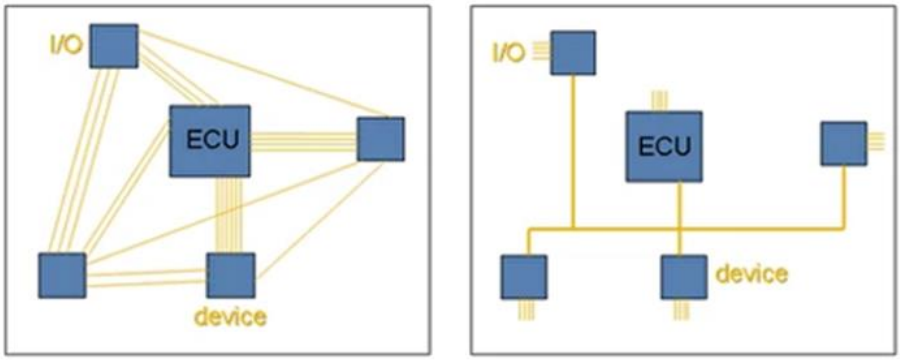
1. Aim:

Establish the CAN communication between two microcontrollers for the given bit rate.

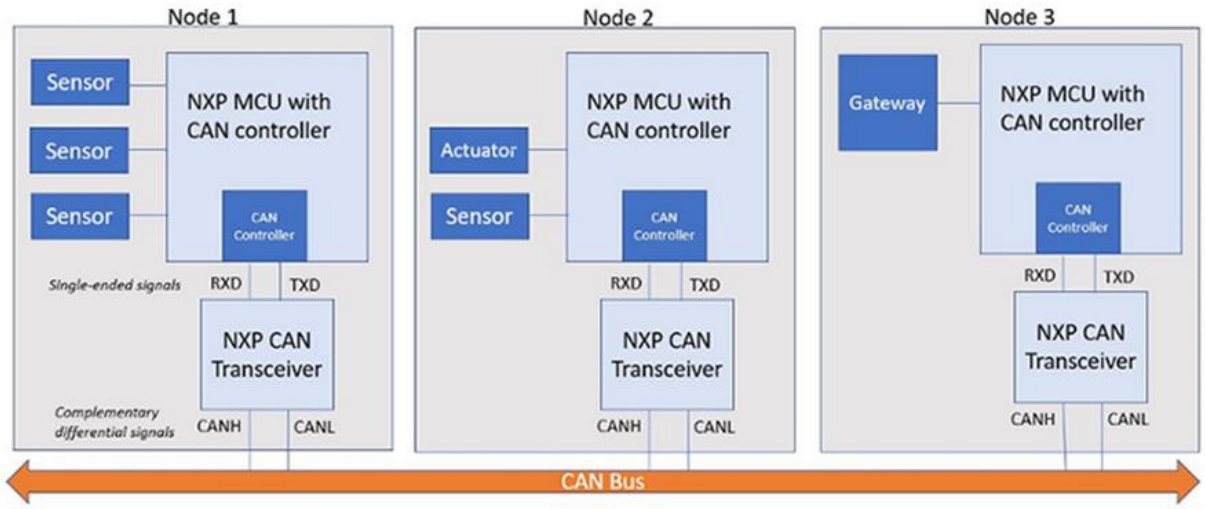
2. Introduction:

CAN stands for Controller Area Network. It was first created by the German Automotive Company Robert Bosch in the mid-1980s for automobile applications. CAN is a message-based protocol, which means that communication does not happen based on addresses but rather the CAN message itself contains the priority and the data that needs to be transmitted. Every node on the bus receives this message, and then each node decides whether that information is useful or needs to be discarded. A single message can be destined for a particular receiver or for multiple nodes that are present on the bus. Another important feature of the CAN protocol is its ability to request information from other nodes which is known as "Remote Transmission Request" or "RTR". Since CAN is a message-based protocol a node can be added to the system without the need to reprogram the other nodes for acknowledging the new addition.

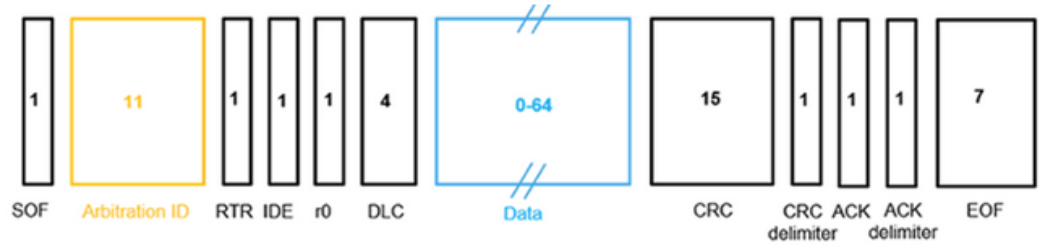
There are four types of CAN frames in the CAN protocol. The most common type is the "Data Frame", which is used when a node transmits information to other nodes. The second one is the "Remote Frame" which is a Data frame with the RTR bit set to signify a Remote Transmission Request. The other two frames are namely the Error Frame and the Overload Frame which are used for handling errors. Error Frames are generated by any one of the nodes that detect any one of the many protocol errors defined in CAN. Overload Frames are generated by nodes that require more time to process the data that has already been received.



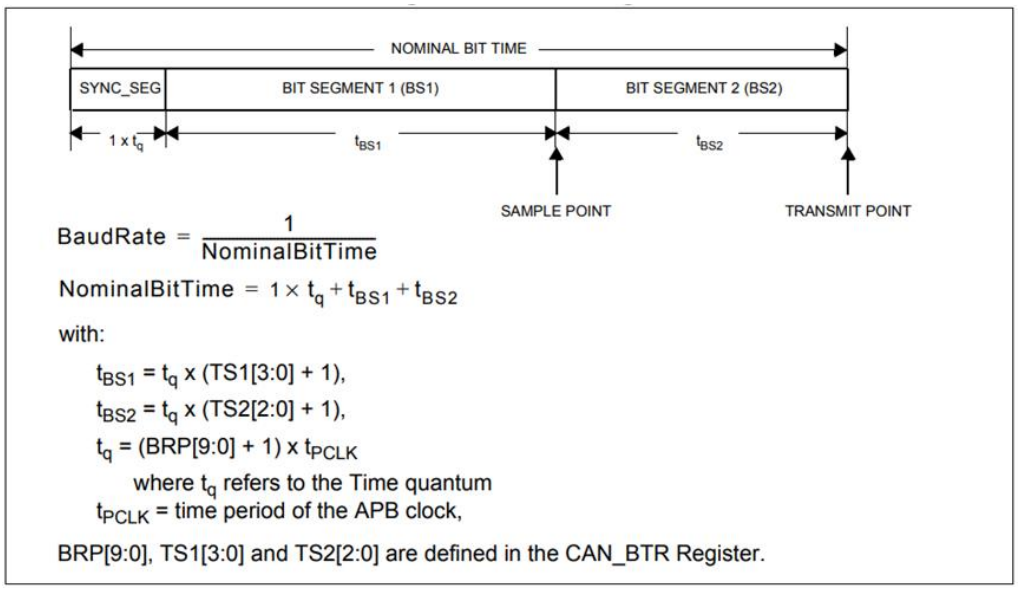
3.1 CAN Network Topology:



3.2 CAN Normal and Extended Data Frames:



3.3 CAN Timing Configuration for the Given Bit Rate:



3.4 Illustration:

From the formulas given in the above figure, we will see how our settings result in a baud rate of **125 Kbps**. Or `APB1` clock is running at `16MHz` so,

$$t_{PCLK} = 1/16MHz = 62.5ns = tp$$

So,

$$t_q = (BRP[9:0] + 1) * (t_p)$$

$$t_q = (15 + 1) * (62.5ns) = 1000ns = tq$$

And,

$$t_{BS1} = tq * (TS1[3:0] + 1)$$

$$t_{BS1} = 1000 * 3 = 3000ns$$

$$t_{BS2} = tq * (TS2[2:0] + 1)$$

$$t_{BS2} = 1000 * 4 = 4000ns$$

Therefore,

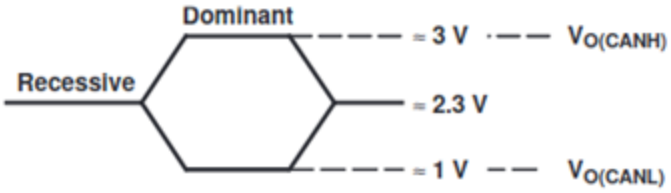
$$NominalBitTime = tq + tBS1 + tBS2$$

$$NominalBitTime = 1000 + 3000 + 4000 = 8000ns$$

$$BaudRate = 1/NominalBitTime = 1/8000ns = 125000bits/s$$

3.5 CAN Signaling:

Logic 1 is a recessive state. To transmit 1 on CAN bus, both CAN high and CAN low should be applied with 2.5V. Logic 0 is a dominant state. To transmit 0 on CAN bus, CAN high should be applied at 3.5V and CAN low should be applied at 1.5V. The ideal state of the bus is recessive. If the node reaches the dominant state, it cannot move back to the recessive state by any other node



4. Components Required:

i. 2 Cortex-M4 STM32F407 Discovery boards
ii. CAN bus setup
iii. 2 CAN transceivers
iv. Analog Discovery or logic analyzer for CAN frame visualization
v. Connecting wires

5. Embedded C Codes:

5.1 Problem Statement:

i. Push button connected in PA0 is monitored continuously.
ii. Whenever the user presses the button, corresponding count is transmitted.
iii. Receiver turns ON the count number of LEDs connected in PD12, P13, PD14, and PD15.
iv. Count variable is reset when it reaches 4.

5.2 CAN Transmit Code:

```
#include <stdint.h>

#include <stm32f4xx.h>



uint8_t value;
uint8_t arr[5] = {0x0,0x1,0x3,0x7,0xF};

void Set_Pin(void)
{
   RCC->AHB1ENR |= (1<<1)|(1<<3);  // GPIOB, GPIOD are to be set
```

```
    GPIOD->MODER |= (0x55 << 24);   // PD12, PD13, PD14, PD15 will be turned on

    GPIOB->MODER |= (0xA << 16);          // Alternate functionality mode

    GPIOB->AFR[1] |= (0x9 | (0x9<<4));    // AF9 for PB8 , PB9 pins
}

uint8_t CAN_Rx(void)
{
    while(!(CAN1->RF0R & 3));   // waiting for atleast one message

    uint8_t data = (CAN1->sFIFOMailBox[0].RDLR) & 0xFF; // only 1 byte

    CAN1->RF0R |= (1<<5);  // Release FIFO MailBox

    return data;
}

void CAN_Init(void)
{
    RCC->APB1ENR |= 1<<25;   // CAN1 Initialize

    CAN1->MCR |= (1<<0);     // Enter into Initialization mode
    while(!(CAN1->MSR & 0x1));   // wait until INAK bit is set

    /* Exit Sleep Mode */
            CAN1->MCR &= ~(1<<1);
            while(CAN1->MSR & 0X2);     // SLAK Sleep ACK

    CAN1->BTR &= ~(3<<24);      // SWJ 1 Time Quantum

    CAN1->BTR &= ~(0x7F << 16);

    CAN1->BTR |= (12 << 16);    // Time Segment 1
    CAN1->BTR |= (1 << 20);     // Time Segment 2

    CAN1->BTR |= (7 << 0);      // Baud-Rate Prescaler

    /* Exit Initialization mode */
    CAN1->MCR &= ~(1<<0);
    while(CAN1->MSR & 0x1);



    /* Filter Configuration */

    CAN1->FMR |= 1<<0;   // Initialise filter mode

    CAN1->FMR |= 14<<8;  // they define start bank for CAN2

    CAN1->FS1R |= 1<<13;  // 32 bit scale configuration Filter
```

```
    // Filtering it with respect to the ID

    CAN1->FM1R |= (1<<13);   // Identifier List-Mode

    CAN1->sFilterRegister[13].FR1 = 254 << 21;   // STD ID

    CAN1->FA1R |= (1<<13);

    CAN1->FMR &= ~(0x1);

}

void delay(uint32_t d)
{
    for(uint32_t i=0;i<d*1000;i++);
}

int main(void)
{
    /* Loop forever */
            Set_Pin();
            CAN_Init();
            while(1)
            {
                    value = CAN_Rx();
                    GPIOD->ODR &= ~(0xF << 12);
                    GPIOD->ODR |= arr[value] << 12;
                    delay(100);
            }

}
```

5.3 CAN Receive Code:

```
#include <stdint.h>

#include <stm32f4xx.h>



uint8_t value,count=0;
//PB8 is CRX, PB9 is CTX
void Set_Pin(void)
{
    RCC->AHB1ENR |= (1<<1)|(1<<0);  // GPIOA, GPIOB
```

```
    GPIOA->MODER &= ~(3<<0);        // Input

    GPIOB->MODER |= (0xA << 16);          // Alternate functionality mode

    GPIOB->AFR[1] |= (0x9 | (0x9<<4));    // AF9 for PB8 , PB9 pins
}

void CAN_Tx(void)
{

    CAN1->sTxMailBox[0].TDLR = count;

    /* Request for transmission */
    CAN1->sTxMailBox[0].TIR |= 1;
}



void CAN_Init(void)
{
    RCC->APB1ENR |= 1<<25;   // CAN1 Initialize

    CAN1->MCR |= (1<<0);     // Enter into Initialization mode
    while(!(CAN1->MSR & 0x1));   // wait until INAK bit is set

    /* Exit Sleep Mode */
            CAN1->MCR &= ~(1<<1);
            while(CAN1->MSR & 0X2);     // SLAK Sleep ACK

    CAN1->BTR &= ~(3<<24);      // SWJ 1 Time Quantum

    CAN1->BTR &= ~(0x7F << 16);

    CAN1->BTR |= (12 << 16);    // Time Segment 1
    CAN1->BTR |= (1 << 20);     // Time Segment 2

    CAN1->BTR |= (7 << 0);      // Baud-Rate Prescaler

    /* Exit Initialization mode */
    CAN1->MCR &= ~(1<<0);
    while(CAN1->MSR & 0x1);

    /* Setting Up Transmission */

    CAN1->sTxMailBox[0].TIR = 0;

    CAN1->sTxMailBox[0].TIR |= (254 << 21);    // STD ID : 254

    CAN1->sTxMailBox[0].TDHR = 0;     // Data byte 4, 5, 6, 7
    CAN1->sTxMailBox[0].TDLR = 0;
```

```
    CAN1->sTxMailBox[0].TDTR = 1;    // Sending only 4 bytes of data

    /* Filter Configuration */

    CAN1->FMR |= 1<<0;   // Initialise filter mode

    CAN1->FMR |= 14<<8;  // they define start bank for CAN2

    CAN1->FS1R |= 1<<13;  // 32 bit scale configuration Filter

    // Filtering it with respect to the ID

    CAN1->FM1R |= (1<<13);   // Identifier List-Mode

    CAN1->sFilterRegister[13].FR1 = 138 << 21;   // STD ID

    CAN1->FA1R |= (1<<13);

    CAN1->FMR &= ~(0x1);

}

void delay(uint32_t d)
{
   for(uint32_t i=0;i<d*1000;i++);
}

int main(void)
{
   /* Loop forever */
            Set_Pin();
            CAN_Init();

            while(1)
            {
                        if(GPIOA->IDR & (1<<0))
                        {
                                    count++;
                                    CAN_Tx();

                                    if(count == 4)
                                                count = 0;

                        }

                        delay(200);
            }

}
```

6. Logic Analyzer Output:

Insert images from logic analyzer here

7. Conclusion:

In this experiment, CAN message is successfully transmitted and received by using two STM32F407 Discovery boards. The CAN message is visualized using the logic analyzer.

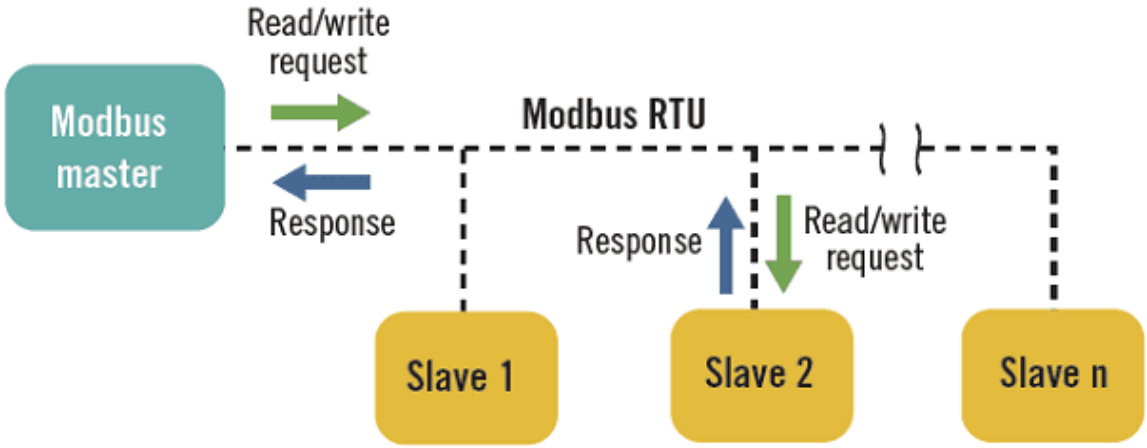
Modbus Communication between Commercial Meter and Cortex M Microcontroller

1. Aim:

To establish Modbus communication between a commercial single phase meter and Cortex M microcontroller and fetch value such as voltage and frequency.

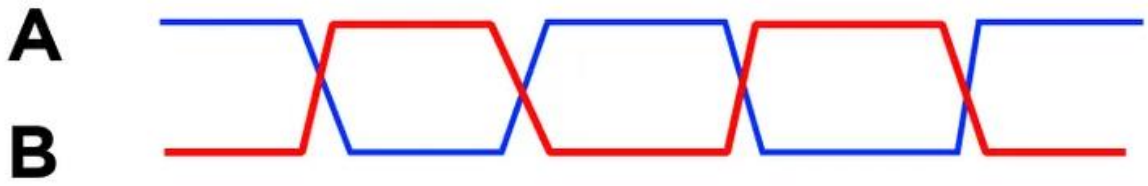
2. Introduction:

Modbus is a widely adopted serial communication protocol used in industrial automation systems. It enables the exchange of data between devices such as sensors, actuators, and controllers. Modbus supports both master-slave and client-server architectures, where a master or client initiates requests, and slaves or servers respond to those requests. Modbus supports various physical communication interfaces, including RS-232, RS-485, and Ethernet (TCP/IP). In this experiment, RS-485 is utilized with the MAX 485 module to establish communication between the Cortex-M processor and the power meter.



2.1 Signaling:

The RS-485 interface allows for long-distance communication and multi-drop connections, making it suitable for industrial applications. It uses a differential signal to transmit data, which ensures noise immunity and reliable communication in harsh environments.



3. Modbus Protocol:

In Modbus communication, devices are assigned unique addresses to differentiate them. The master or client device initiates requests by sending Modbus frames to the addressed slaves or servers. The frame consists of several fields, including the slave address, function code, data, and error checking.

3.1 Function Codes in Modbus Communication:

Function codes define the type of operation to be performed in Modbus communication. Some commonly used function codes include:

Read Holding Registers (Function Code: 0x03):

This function code is used to retrieve data from holding registers. Holding registers are typically used to store measurement values, configuration parameters, or control settings. The Cortex-M processor can send a request with this function code to the power meter to obtain specific data.

Read Input Registers (Function Code: 0x04):

The Read Input Registers function code allows the Cortex-M processor to access data from input registers. Input registers are used to store data received from external devices or sensors. The power meter can make specific measurements or status information available through these registers.

Write Single Register (Function Code: 0x06):

With the Write Single Register function code, the Cortex-M processor can write a single value to a specific register in the power meter. This function is often used to update control parameters or setpoints.

Write Multiple Registers (Function Code: 0x10):

The Write Multiple Registers function code enables the Cortex-M processor to write multiple values to consecutive registers. This function is commonly used to update multiple parameters or send a series of commands to the power meter.

3.2 Steps Followed in Modbus Communication:

To retrieve data from input and holding registers using Modbus communication, the following steps are typically followed:

1. **Establish Modbus Communication**

The Cortex-M processor initializes the communication by configuring the serial port and establishing a connection with the power meter. The MAX 485 module is used to interface the processor with the RS-485 network.

2. **Frame the Modbus Request**

The Cortex-M processor constructs a Modbus request packet, specifying the function code and the starting register address. Additional parameters, such as the number of registers to read or write, can also be included in the request.

3. **Send Modbus Request:**

The Cortex-M processor transmits the Modbus request packet to the power meter via the RS-485 network. The request is sent to the address of the specific slave device.

4. **Receive Modbus Response:**

The power meter receives the Modbus request, processes it, and generates a response packet. The response packet contains the requested data from the input or holding registers. It is transmitted back to the Cortex-M processor over the RS-485 network.

5. **Extract Data from Modbus Response:**

The Cortex-M processor receives the Modbus response packet and extracts the data from the appropriate register or registers. The extracted data can then be processed, displayed, or further used for analysis or control purposes.

3.3 Modbus Message Format:

The Modbus protocol defines a specific message format for both requests and responses. The structure of a Modbus message consists of several fields, each serving a specific purpose. Let's explore the structure of Modbus messages for both requests and responses in the context of the above experiment.

3.3.1 Modbus Request Message Format:

A Modbus request message consists of the following fields:

Slave Address:

The Slave Address field indicates the address of the slave device (in this case, the power meter) that the request is intended for. It is a single byte value ranging from 1 to 247.

Function Code:

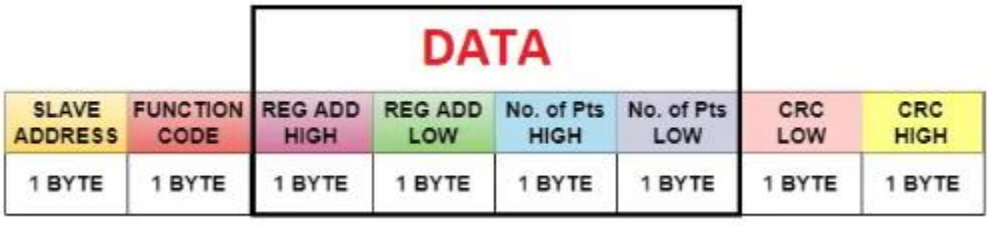
The Function Code field specifies the operation to be performed by the slave device. Different function codes represent various actions such as reading or writing data. For example, Function Code 0x03 is used for reading holding registers.

Data:

The Data field contains additional parameters required for the specific function code. It may include the starting register address, the number of registers to read or write, and any other necessary information.

Error Checking:

The Error Checking field is used to ensure the integrity of the message. It typically consists of a CRC (Cyclic Redundancy Check) or LRC (Longitudinal Redundancy Check) value calculated over the preceding fields.



Example Modbus Request Message (Read Holding Registers)

Let's assume we want to read two holding registers starting from address 0x100 in the power meter.

Slave Address: 0x01

Function Code: 0x03

Data: Starting Register Address: 0x0100, Number of Registers: 0x0002

Error Checking: CRC or LRC value

The Modbus request message would look like this:

| Slave Address | Function Code | Starting Register Address | Number of Registers | Error Checking |

| 0x01 | 0x03 | 0x01 |0x00| 0x00| 0x02 | CRC or LRC |

3.3.2 Modbus Response Message Format:

A Modbus response message also consists of specific fields:

Slave Address:

The Slave Address field mirrors the address specified in the corresponding request message, indicating the source of the response.

Function Code:

The Function Code field matches the function code of the corresponding request message, indicating the operation performed by the slave device.

Number of Bytes:

This contains number of bytes.

Data:

The Data field contains the actual data requested or processed by the slave device. In the case of a read operation, it will include the values from the requested registers.

Error Checking:

Similar to the request message, the Error Checking field ensures the integrity of the response message by employing CRC or LRC values.

Example Modbus Response Message (Read Holding Registers):

Following the previous example, let's assume the power meter responds with the values 0x1234 and 0xABCD for the two holding registers.

Slave Address: 0x01

Function Code: 0x03

Data: Register 1: 0x1234, Register 2: 0xABCD

Error Checking: CRC or LRC value

The Modbus response message would look like this:

| Slave Address | Function Code |number of bytes | Data (Register 1) | Data (Register 2) | Error Checking |

| 0x01 | 0x03 |0x02 | 0x12 | 0x34 | CRC or LRC |

| 0x01 | 0x03 | 0x02| 0xAB | 0xCD | CRC or LRC |

In this example, the power meter responds with the requested data from the holding registers, which is 0x1234 and 0xABCD. The response message reflects the slave address, function code, data values, and error checking.

4. Circuit Connection:

4.1 Power Connections:

- Connect the VCC pin of the MAX 485 module to the 5V power supply on the STM32F407 Discovery board.
- Connect the GND pin of the MAX 485 module to the ground (GND) of the board.

4.2 UART Connections:

- Connect the TX (Transmit) pin of the STM32F407 microcontroller (PD5) to the RO (Receiver Output) pin of the MAX 485 module.
- Connect the RX (Receive) pin of the STM32F407 microcontroller (PD6) to the DI (Driver Input) pin of the MAX 485 module.
- Connect the DE (Driver Enable) pin of the MAX 485 module to any available GPIO pin on the STM32F407 microcontroller (GPIO Output Pin PD12).

- Connect the RE (Receiver Enable) pin of the MAX 485 module to any available GPIO pin on the STM32F407 microcontroller (GPIO Output Pin PD312).

4.3 RS-485 Connections:

- Connect the A and B differential signal lines of the MAX 485 module to the corresponding A and B terminals of the RS-485 network.
- Connect the A and B terminals of the RS-485 network to the corresponding terminals of the SELEC single-phase power meter.

5. Working Principle:

Before transmitting the request, set the DE (Driver Enable) pin of the MAX 485 module to logic HIGH, enabling the driver. Set the RE (Receiver Enable) pin of the MAX 485 module to logic LOW, disabling the receiver. Transmit the Modbus request message from the STM32F407 microcontroller to the MAX 485 module through the UART interface.

After transmitting the request, wait for the response from the power meter. Set the DE (Driver Enable) pin of the MAX 485 module to logic LOW, disabling the driver. Set the RE (Receiver Enable) pin of the MAX 485 module to logic HIGH, enabling the receiver. Receive the Modbus response message on the STM32F407 microcontroller through the UART interface.

6. Program:

In this program we are fetching voltage value from the meter which is located at 0x0 location of input register area. It is located in 2 consecutive registers in mid little-endian format in float value (IEEE 754 format).

Data format: Mid Little Endian
If Total Active Energy = 1234.12kWh
Start Address : 30090, No. Of register : 02
Hexadecimal Equivalent of 1234.12 is 0x449A43D7

	C	D
Data stored at 30090 is LSB :	43	D7

	A	B
Data Stored at 30091 is MSB :	44	9A

Data Format to be followed is C-D-A-B

```c
#include <stdint.h>
#include "stm32f4xx.h"
#include <stdio.h>
#include <math.h>

#define ARM_MATH_CM4
uint8_t arr[100];
uint8_t msg_r[100];
void USART2_init(void);
void configurePin(void);
void GPIO_init(void);
void USART_write(uint8_t *buffer);
void USART_read (uint8_t *buffer);
void USART_write_array(uint8_t buffer[],uint32_t nBytes);
void USART_read_array(uint8_t buffer[],uint32_t nBytes);
void msDelay(int msTime);
float ieee754_to_float(int value);
uint16_t crc16(uint8_t *buffer, uint16_t buffer_length);
```

```c
float value;
int main(void)
{
    GPIO_init();
    USART2_init();
    configurePin();
    uint32_t nBytes = 8;
    uint8_t msg[nBytes];
    msg[0] = 0x01;  //slave id
    msg[1] = 0x04;  //function code
    msg[2] = 0x00;
    msg[3] = 0x00;  //reg address
    msg[4] = 0x00;
    msg[5] = 0x02;  //number of reg
    uint16_t crc = crc16(msg, 6);
    uint8_t crcLow  = crc&0xFF;   // CRC LOW
    uint8_t crcHigh = (crc>>8)&0xFF;  // CRC HIGH
    msg[6] = crcLow;
    msg[7] = crcHigh;
    USART_write_array(msg,nBytes);
    uint32_t number = msg[5]*2 + 5;
    USART_read_array(arr,100);
    int i =0;
```

```
    while(i<100){
        if(arr[i] == msg[0]){
            for(int j=0;j<number;j++){
                msg_r[j] = arr[i+j];
            }
            break;
        }
        i = i+1;
    }
     //little endian format
    uint32_t val = (msg_r[5]<<24)|(msg_r[6]<<16)|(msg_r[3]<<8)|(msg_r[4]);
    value = ieee754_to_float(val);
    printf("the voltage is %f\n",value);

}
```

```
void USART2_init(void){
USART2->CR1 &= ~(1<<13); //USART DISABLE
USART2->CR1 &= ~(1<<12); //SETTING 8 BIT DATA TRANSMISSION
USART2->CR2 |= (0<<12); //SETTING ONE STOP BIT
USART2->CR1 &= ~(1<<10); //SET NO PARITY
USART2->CR1 &= ~(1<<15); //OVERSAMPLING BY 16
USART2->BRR = 0x683; //SETTING BAUD RATE TO 9600

//ENABLE TRANSMSSION AND RECEPTION
USART2->CR1 |= (3<<2);

USART2->SR &= ~(1<<7); //CLEARING TXE FLAG
USART2->SR &= ~(1<<6); //CLEARING TC FLAG
USART2->SR &= ~(1<<5); //CLEARING RXE FLAG
//ENABLE USART
USART2->CR1 |= (1<<13);
}
```

```c
void USART_write(uint8_t *buffer){
        GPIOD->ODR |= (1<<12);
        while(!(USART2->SR & (1<<7))); //WAIT UNTIL TXE FLAG IS SET
        USART2->DR = (*buffer)&0x1FF; //WRITING TO DR FLAG CLEARS TXE FLAG
        while(!(USART2->SR & (1<<6))); //WAIT UNTIL TC FLAG IS SET
        USART2->SR &= ~(1<<6); //CLEARING TC FLAG

}

void USART_read (uint8_t *buffer) {

        GPIOD->ODR &= ~(1<<12);
        while (!(USART2->SR & (1<<5))); // Wait until hardware sets RXNE

        *buffer = USART2->DR&(0x1FF); // Reading DR clears RXNE
        //printf("the data received is %d\n",*buffer);

}
```

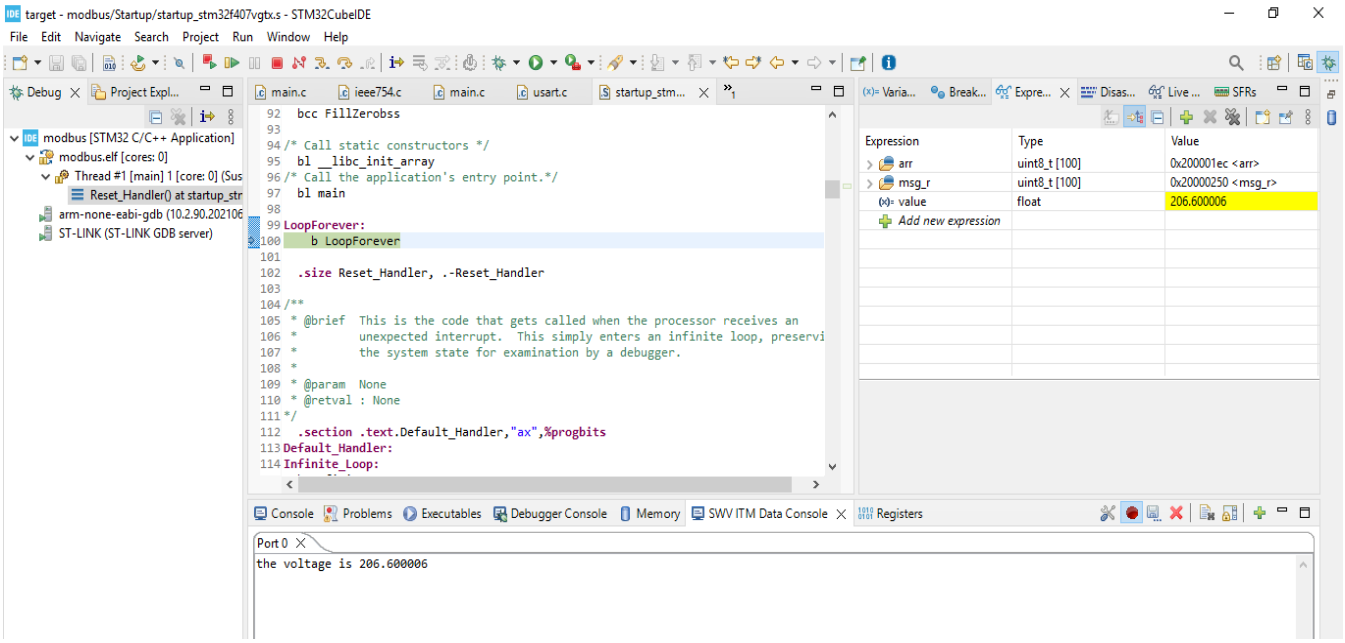
```c
void USART_write_array(uint8_t buffer[],uint32_t nBytes){
        for (int i=0;i<nBytes;i++){
            USART_write(&buffer[i]);
            //msDelay(sec);
        }
}

void USART_read_array(uint8_t buffer[],uint32_t nBytes){
        for (int i=0;i<nBytes;i++){
            USART_read(&buffer[i]);
            //msDelay(sec);
        }

}
```

7. Result:

In the expression we get the value in the voltage:



Data Acquisition System Design and FFT analysis using ARM Cortex-M4 Microcontroller

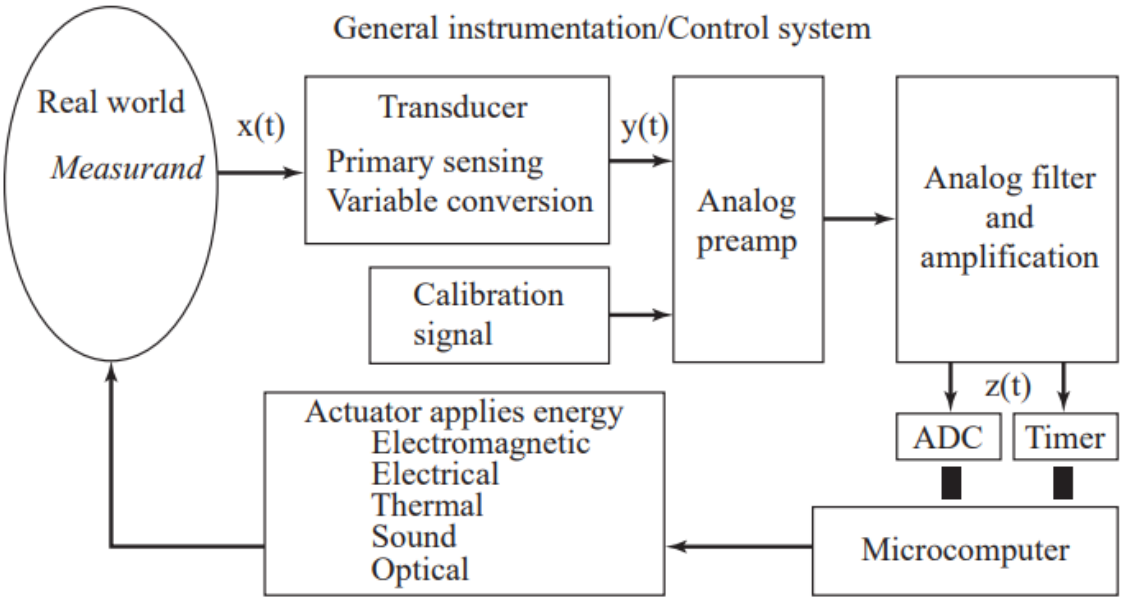
1. Aim:

To perform Fast Fourier Transform (FFT) on a current signal obtained from ADC.

2. Introduction:

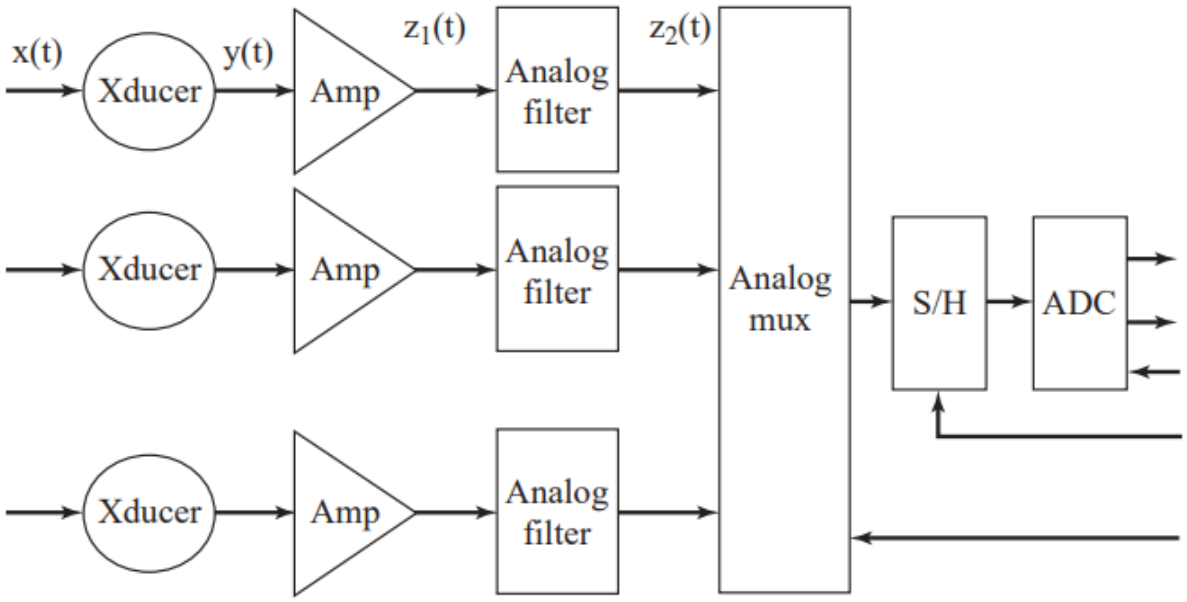
A system that collects information is called a data acquisition system (DAS). In this experiment, we will use the two terms DAS and instrument interchangeably. Sometimes the acquisition of data is the fundamental purpose of the system, such as with a voltmeter, a tachometer, a multi-meter, an audio recorder, or a camera. At other times, the acquisition of data is an integral part of a larger system such as a control system or communication system.

The measurand is the physical quantity, property, or condition that the instrument measures. The measurand can be inherent to the object (like position, size, mass, or color), located on the surface of the object (like the human EKG or surface temperature), located within the object (e.g., fluid pressure or internal temperature), or separated from the object (like emitted radiation). A typical DAS data flow graph is given in the figure below.



Data flow graph of a DAS

The transducer converts the physical signal into an electric signal. The amplifier converts the weak transducer electric signal into the range of the ADC (e.g., 0 to 3 V). The analog filter removes unwanted frequency components within the signal. The analog filter is required to remove aliasing error caused by the ADC sampling. The analog multiplexer is used to select one signal from many sources. The S/H is an analog latch used to keep the ADC input voltage constant during the ADC conversion. The clock is used to control the sampling process. Inherent in digital signal processing is the requirement that the ADC be sampled on a fixed time basis. The computer is used to save and process the digital data. A digital filter may be used to amplify or reject certain frequency components of the digitized signal. Block diagram of a multichannel ADC is given in the figure below.



Multichannel ADC

3. Determination of Sampling Rate:

There are two errors introduced by the sampling process. Voltage quantizing is caused by the finite word size of the ADC. The precision is determined by the number of bits in the ADC. If the ADC has n bits, then the number of distinguishable alternatives is

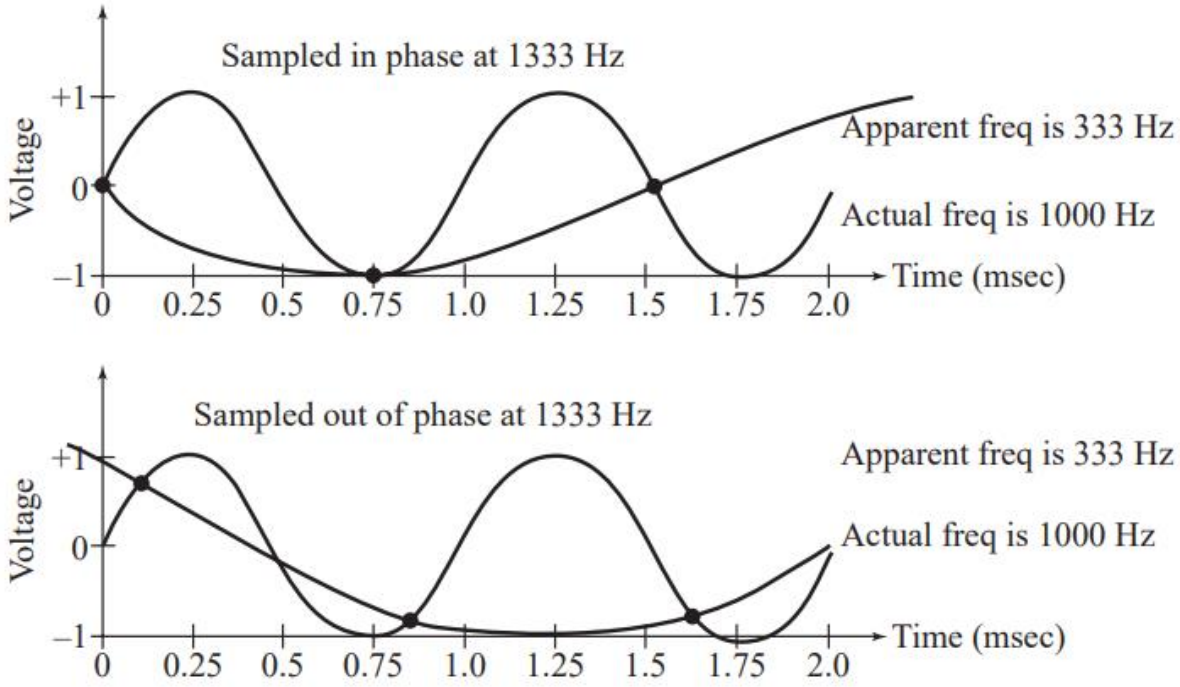
$$Levels = 2^n$$

Time quantizing is caused by the finite discrete sampling interval. Nyquist theory states that if the signal is sampled at Fs, then the digital samples contain frequency components from only 0 to 0.5 Fs. Conversely, if the analog signal does contain frequency components larger than 0.5 Fs, then there will be an aliasing error. Aliasing is when the digital signal appears to have a different frequency than the original analog signal. Simply put, if one samples a sine wave at a sampling rate of Fs,

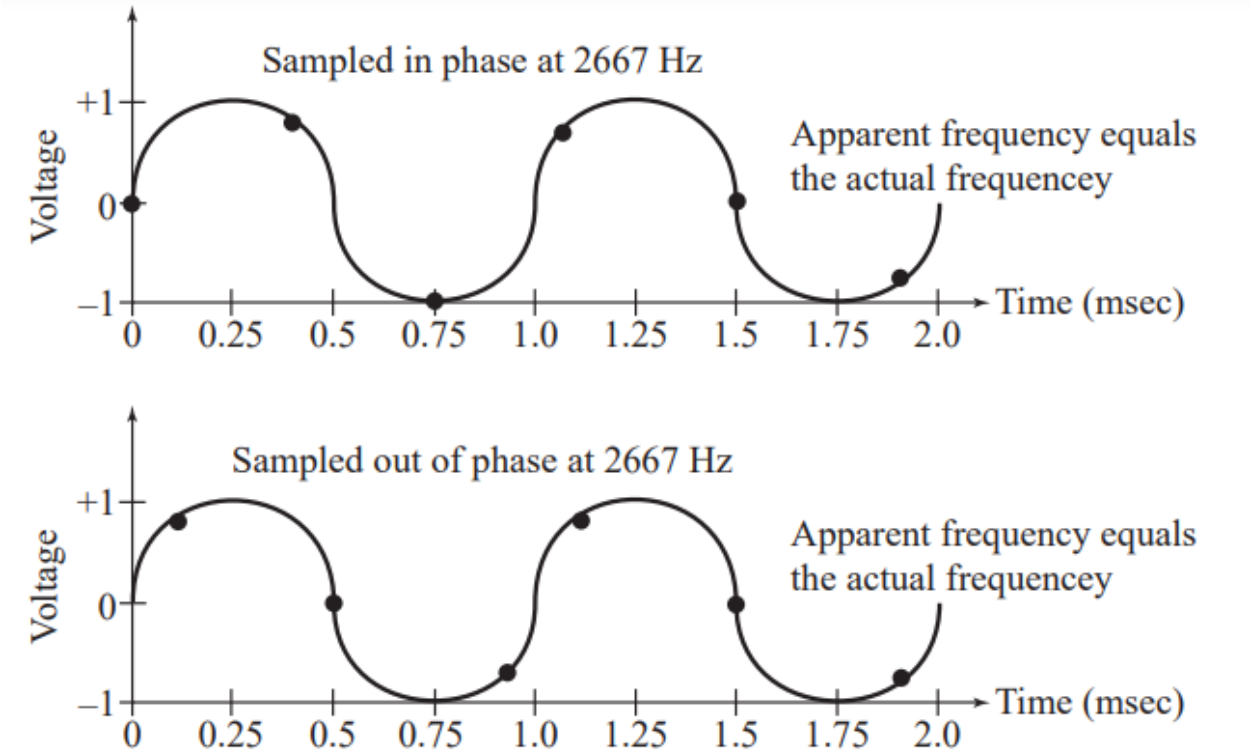
$$V(t) = Asin(2\pi ft + \emptyset)$$

is it possible to determine A, f, and Φ from the digital samples? Nyquist theory says that if Fs is strictly greater than twice f, then one can determine A, f, and Φ from the digital samples. In other words, the entire analog signal can be reconstructed from the digital samples. But if Fs is less than or equal to twice f, then one cannot determine A, f, and Φ. In this case, the apparent frequency, as predicted by analyzing the digital samples, will be shifted to a frequency between 0 and 0.5 Fs.

Example illustration:



Aliasing due to improper Fs



Sampling without aliasing

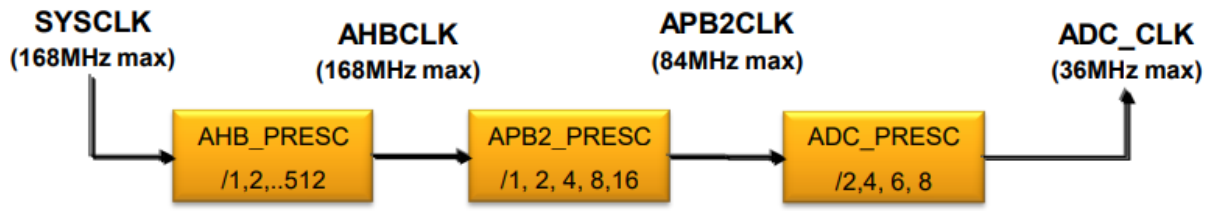
4. STM32F407 ADC Features:

The 12-bit ADC is a successive approximation analog-to-digital converter. It has up to 19 multiplexed channels allowing it to measure signals from 16 external sources, two internal sources, and the VBAT channel. The A/D conversion of the channels can be performed in single, continuous, scan or discontinuous mode. The result of the ADC is stored into a left or right-aligned 16-bit data register.

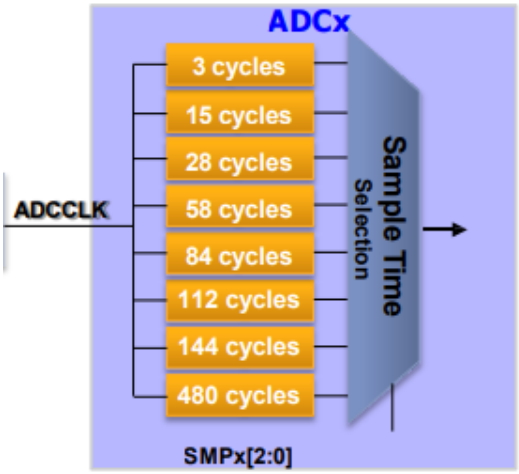
- 12-bit, 10-bit, 8-bit or 6-bit configurable resolution
- Interrupt generation at the end of conversion
- Single and continuous conversion modes
- Channel-wise programmable sampling time
- ADC input range: $V_{REF-} \leq V_{IN} \leq V_{REF+}$

5. Procedure for 8000Hz Sampling Rate Configuration of ADC:

Assume that we have configured our ADC as 8-bit. The ADC input clock is generated from the PCLK2 clock divided by a prescaler.



Total Conversion Time = $T_{sampling}$ + $T_{conversion}$



Resolution	$T_{Conversion}$
12 bits	12 Cycles
10 bits	10 Cycles
8 bits	8 Cycles
6 bits	6 Cycles

For 8000 sampling rate,

Since we have configured it as 8-bit, $T_{conversion}$ is 8 cycles. We can choose T sampling time as 112 cycles. Here total conversion time is 120cycles. Assume that ADC_CLK is 1MHz with suitable prescaler values. Hence,

Sampling rate =1000000/120=8333Hz.

5.1 Program for Current Signal Acquisition using ADC with 8000Hz Sampling Frequency:

```
#include "stm32f4xx.h"

void ADC_Init(void);
void ADC_Enable(void);
void ADC_Start(int);

float value[1024];
uint32_t i,j;
uint8_t ADC_VAL[1024];
void ADC_Init (void)
{
		/*************** STEPS TO FOLLOW ******************
		1. Enable ADC and GPIO clock
		2. Set the prescalar in the Common Control Register (CCR)
		3. Set the Resolution in the Control Register 1 (CR1)
		4. Set the Continuous Conversion, EOC, and Data Alignment in Control Reg 2 (CR2)
		5. Set the Sampling Time for the channels in ADC_SMPRx
		6. Set the Regular channel sequence length in ADC_SQR1
		7. Set the Respective GPIO PINs in the Analog Mode
		**********************************************/

//1. Enable ADC and GPIO clock
		RCC->APB2ENR |= (1<<8);  // enable ADC1 clock
		RCC->AHB1ENR |= (1<<0);  // enable GPIOA clock
		RCC->CFGR |= 6<<13; // set APB2 = 2 MHz

//2. Set the pre-scalar in the Common Control Register (CCR)
		ADC->CCR |= 0<<16;			// PCLK2 divide by 4, works at 4MHz

//3. Set the Resolution in the Control Register 1 (CR1)
		//ADC1->CR1 &= ~(1<<8);    // SCAN mode disabled, enable for multichannel use
		ADC1->CR1 |= (2<<24);   // 8 bit RES

//4. Set the Continuous Conversion, EOC, and Data Alignment in Control Reg 2 (CR2)
			//ADC1->CR2 |= (1<<1);     // enable continuous conversion mode
			ADC1->CR2 |= (1<<10);    // EOC after each conversion
			ADC1->CR2 &= ~(1<<11);   // Data Alignment RIGHT


//5. Set the Sampling Time for the channels
		//ADC1->SMPR2 &= ~(1<<0);  // Sampling time of 3 cycles for channel 0
		ADC1->SMPR2|=(7<<0);
//6. Set the Regular channel sequence length in ADC_SQR1
		//ADC1->SQR1 &= ~(1<<20);   // SQR1_L =0 for 1 conversion

//7. Set the Respective GPIO PIN in the Analog Mode
```

```
	GPIOA->MODER |= (3<<0);  // analog mode for PA 0 (channel 0)

}
void msDelay(uint32_t msTime)
{
	/* For loop takes 4 clock cycles to get executed. Clock frequency of stm32f407 by default is 16MHz
	So, 16MHz/4=4MHz. If we want 1000ms delay, 4MHz/1000=4000, so we have to multiply by 4000 to get a delay of 1s
	*/
	for(uint32_t i=0;i<msTime*3000;i++)
	{
		__NOP();
	}

}

int main ()
{

	ADC_Init ();
	ADC1->CR2 |= 1<<0;   // ADON =1 enable ADC1
	uint32_t delay = 10000;
	while (delay--);//Wait sometime for ADC to start

	//ADC1->CR2 |= (1<<30);  // start the conversion
	while (1)
	{

		for(i=0;i<1024;i++)
		{
			ADC1->CR2 |= (1<<30);  // start the conversion
			ADC1->SR = 0;		// clear the status register
			//ADC1->CR2 |= (1<<30);  // start the conversion
			while (!(ADC1->SR & (1<<1)));  // wait for EOC flag to set
			ADC_VAL[i]=ADC1->DR;
		}

		for(j=0;j<1024;j++)
		{
			value[j]=ADC_VAL[j];
		}

	}

}
```

5.2 Program for Current Signal Acquisition using ADC with 8000Hz Sampling Frequency and real-time FFT Computation:

Fast Fourier Transform:

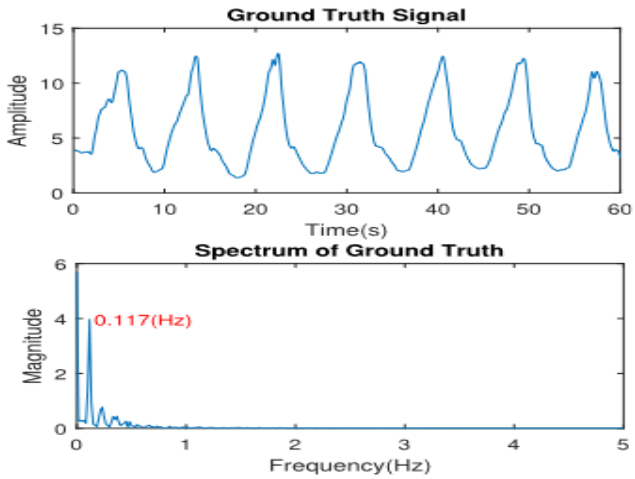
FFT equation is employed to extract the dominant frequency of the given signal:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_n^{kn} \quad for \;\; k = 0 \ldots . N-1$$

FFT resolution is computed to obtain the analog frequency of corresponding dominant peak:

$$\Delta F = \frac{F_S}{N}$$

where Fs is the sampling rate of resampled signal, N is the size FFT, and ΔF is the FFT resolution. Real-time computation of FFT for a given breathe signal is illustrated below.



To implement real-time FFT in ARM Cortex-M4 microcontroller, APIs from CMSIS-DSP library is used. This user manual describes the CMSIS DSP software library, a suite of common signal processing functions for use on Cortex-M and Cortex-A processor-based devices.

The library is divided into a number of functions each covering a specific category:

- Basic math functions
- Fast math functions
- Complex math functions
- Filtering functions
- Matrix functions
- Transform functions
- Motor control functions

- Statistical functions
- Support functions
- Interpolation functions
- Support Vector Machine functions (SVM)
- Bayes classifier functions
- Distance functions

The library has generally separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integer and 32-bit floating-point values. The library functions are declared in the public file `arm_math.h` which is placed in the `Include` folder. Simply include this file. If you don't want to include everything, you can also rely on headers in Include/dsp folder and use only what you need.

```
void arm_rfft_fast_f32 ( const arm_rfft_fast_instance_f32 * S,
                         float32_t *                        p,
                         float32_t *                        pOut,
                         uint8_t                            ifftFlag
                       )
```

Parameters

[in] **S**	points to an **arm_rfft_fast_instance_f32** structure	
[in] **p**	points to input buffer (Source buffer is modified by this function.)	
[in] **pOut**	points to output buffer	
[in] **ifftFlag**	• value = 0: RFFT • value = 1: RIFFT	

Returns

none

```
#include "stm32f4xx.h"
#define ARM_MATH_CM4
#include "arm_math.h"
#include "arm_const_structs.h"
void ADC_Init(void);
void ADC_Enable(void);
void ADC_Start(int);
float32_t testOutput2[1024];
float32_t testMag2[512],value[1024],value1[1024],Frequency,maxValue;
float32_t delF=2051/1024;//Fs/N
uint32_t fftSize = 1024,maxIndex;
uint32_t ifftFlag = 0,i,j;
uint32_t doBitReverse = 1;
uint8_t ADC_VAL[1024];
void ADC_Init (void)
{
        /************** STEPS TO FOLLOW *****************
        1. Enable ADC and GPIO clock
        2. Set the prescalar in the Common Control Register (CCR)
        3. Set the Resolution in the Control Register 1 (CR1)
```

```
		4. Set the Continuous Conversion, EOC, and Data Alignment in Control Reg 2 (CR2)
		5. Set the Sampling Time for the channels in ADC_SMPRx
		6. Set the Regular channel sequence length in ADC_SQR1
		7. Set the Respective GPIO PINs in the Analog Mode
		**********************************************/
//High speed internal clock is 16MHz
//1. Enable ADC and GPIO clock
		RCC->APB2ENR |= (1<<8);  // enable ADC1 clock
		RCC->AHB1ENR |= (1<<0);  // enable GPIOA clock
		RCC->CFGR |= 6<<13; //  AHB clock divided by 8; set APB2 = 2 MHz

//2. Set the pre-scalar in the Common Control Register (CCR)
		ADC->CCR |= 0<<16;			//00: PCLK2 divided by 2, works at 1MHz

//3. Set the Resolution in the Control Register 1 (CR1)
		//ADC1->CR1 &= ~(1<<8);    // SCAN mode disabled, enable for multichannel use
		ADC1->CR1 |= (2<<24);   // 8 bit RES

//4. Set the Continuous Conversion, EOC, and Data Alignment in Control Reg 2 (CR2)
			//ADC1->CR2 |= (1<<1);     // enable continuous conversion mode
			ADC1->CR2 |= (1<<10);    // EOC after each conversion
			ADC1->CR2 &= ~(1<<11);   // Data Alignment RIGHT


//5. Set the Sampling Time for the channels
		//ADC1->SMPR2 &= ~(1<<0);  // Sampling time of 3 cycles for channel 0
		ADC1->SMPR2|=(7<<0);// 480+8cycles (number of bits)=488 cycles
//6. Set the Regular channel sequence length in ADC_SQR1
		//ADC1->SQR1 &= ~(1<<20);   // SQR1_L =0 for 1 conversion

//7. Set the Respective GPIO PIN in the Analog Mode
		GPIOA->MODER |= (3<<0);  // analog mode for PA 0 (channel 0)

}
void msDelay(uint32_t msTime)
{
		/* For loop takes 4 clock cycles to get executed. Clock frequency of stm32f407 by default is
16MHz
		So, 16MHz/4=4MHz. If we want 1000ms delay, 4MHz/1000=4000, so we have to multiply by
4000 to get a delay of 1s
		*/
		for(uint32_t i=0;i<msTime*3000;i++)
		{
			__NOP();
		}

}

int main ()
{
		//FPU enable
```

```
		SCB->CPACR|=(0xF<<20); //Co-processor (core peripheral) access control register
	ADC_Init ();
	ADC1->CR2 |= 1<<0;   // ADON =1 enable ADC1
	uint32_t delay = 10000;
	while (delay--);//Wait sometime for ADC to start
	arm_rfft_fast_instance_f32 S2;
	arm_rfft_fast_init_f32(&S2,1024);

	while (1)
	{

		for(i=0;i<1024;i++)
		{
			ADC1->CR2 |= (1<<30);  // start the conversion
			ADC1->SR = 0;        // clear the status register
			//ADC1->CR2 |= (1<<30);  // start the conversion
			while (!(ADC1->SR & (1<<1)));  // wait for EOC flag to set
			ADC_VAL[i]=ADC1->DR;
		}
		for(j=0;j<1024;j++)
				{
					value1[j]=ADC_VAL[j];
				}

		for(j=0;j<1024;j++)
		{
			value[j]=ADC_VAL[j];
		}
		arm_rfft_fast_f32     (&S2,value,testOutput2,ifftFlag);
		arm_cmplx_mag_f32(testOutput2, testMag2, fftSize);
		arm_max_f32(testMag2+1, 511, &maxValue, &maxIndex);
		Frequency=delF*(maxIndex+1);
	}

}
```

6. Conclusion:

The current signal obtained from the ADC has been transformed to Frequency Domain by performing FFT on it. Also, in addition to this, by analyzing the spectrum of the current signal, the most dominant signal frequency has been found.

7. References:

1. ARM CMSIS Library: https://www.keil.com/pack/doc/CMSIS/General/html/index.html

Implementation of Motor Position and Speed Measurement using Cortex-M Microcontroller

1. Aim:

Encoder based measurement of position and speed of DC servomotor.

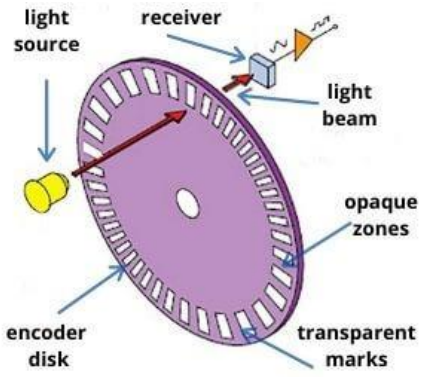
2. Introduction:

Motors are commonly used in several applications such as antenna positioning systems, metal cutting and forming machines, robotic joints etc. Precise position and speed control is necessary for such critical applications. To achieve this precision, several control mechanisms such as PID are applied using a microcontroller which requires position and/or speed measurement as feedback signals. These feedback signals are obtained through various sensor based and sensor less techniques some of which use encoders, hall effect sensors, potentiometers, tachometers etc.

Encoders are a popular type of sensors and are classified into various categories based on the motion translation technology or into linear and rotary, absolute and incremental. Some of the technologies used in encoders to translate motion related data are magnetic, optical, resistive, and mechanical. An optical encoder uses light beams to measure parameters. A rotary encoder, or a shaft encoder, collects data and provides feedback based on the rotation of an object such as a motor. An incremental rotary encoder can measure parameters such as distance, speed, and position whereas absolute encoders measure angular position.

To measure the position and speed of a motor through an **optical, rotary, incremental** encoder is the objective of this experiment.

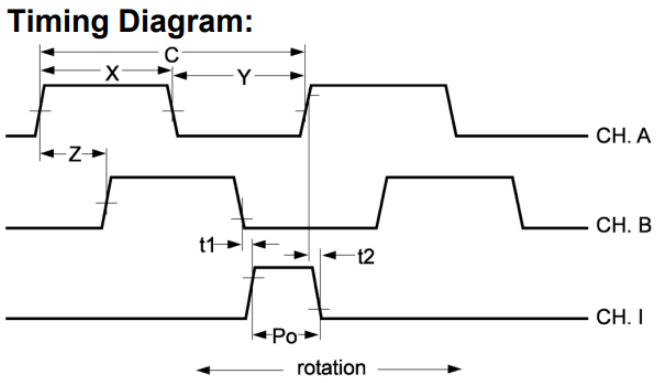
3. Working Principle of Optical Encoders:



Optical encoder working principle

Above is the image of an optical encoder disk. The disk contains slots through which light passes and is received by a receiver on the other side of the disk. During rotation, the alternating opaque and transparent zones cause light to be transmitted, not continuously, but as pulses. Therefore, the term PPR or Pulses Per Revolution is widely used to describe optical encoders. PPR is equal to the number of slots on the disk and describes how many pulses can pass through the encoder disk to the receiver in one full rotation. The maximum possible amount of encoder pulses determines its resolution. These details are available in the datasheet of the encoder.

The receiver captures all incoming light pulses and generates a digital or analog output signal, sending it to the microcontroller. One of the most used angular sensors are quadrature encoders. They send the signal with results as square waves with usually three channels, A, B and I or Z.

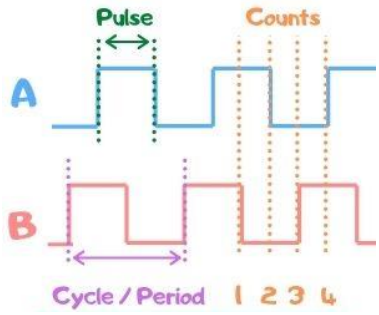


Channel A determines the number of pulses.

Channel B is identical to Channel A but is used to determine the direction of rotation based on the phase shift of 90 degrees between the channels.

Channel Z is needed for notification of the passage of a certain point or zero point. It is also called encoder Z pulse (zero or index pulse). It is necessary when you need a point that defines a certain displacement. For this experiment, only Channels A and B are required.

Another useful term is CPR or Cycles Per Revolution. Here, cycle denotes a complete square wave electrical cycle. In a quadrature encoder, for every cycle on one channel, 4 pulses are generated. This is illustrated below. The definition of CPR varies by manufacturer, hence only PPR is used in this experiment for generality.



Knowing PPR and obtaining both the channels A and B signals means the position and speed can be measured.

For a time period of 1 second,

Position in degrees is,

$$Degrees = \frac{360 * PulseCount}{PPR * 1.0}$$

Speed in rps is,

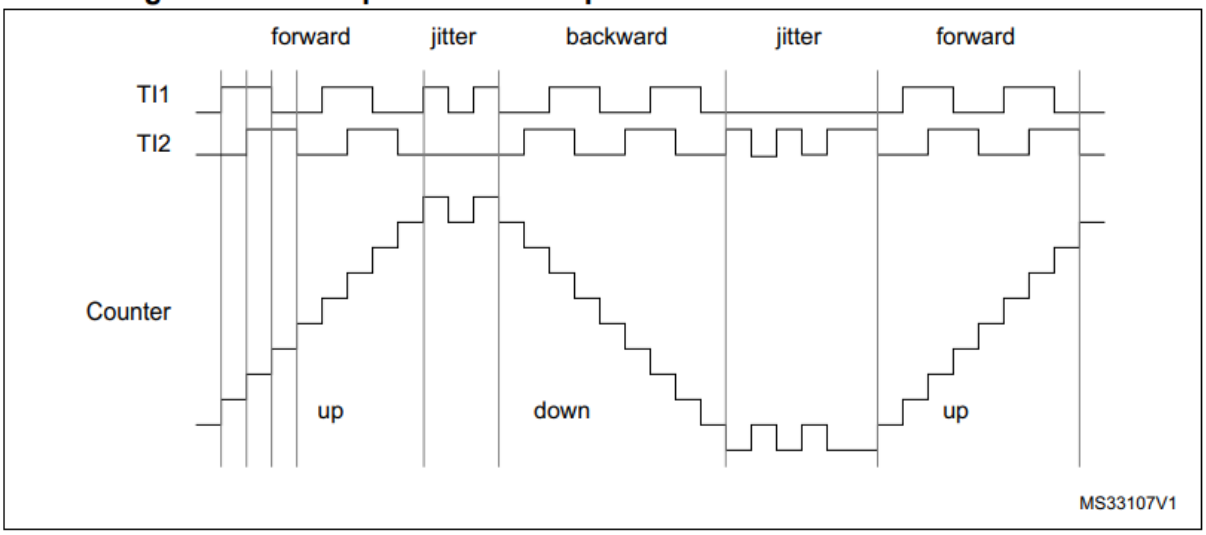
$$RPS = \frac{PulseCount}{PPR * 1.0}$$

4.1 Input Capture:

The input capture method allows us to configure the timer as a recipient to an input which it timestamps. There is a specific “Encoder Interface Mode” in which the timer acts simply as an external clock with direction selection. This is because the square wave signals from the encoder act as an external clock.

For input capture, both channels A and B or either channel can be input with filtering and/or pre-scaling. To decide how many channels are taken as input, SMS or Slave Mode Selection bits in the SMCR or Slave Mode Control Register are configured. Filtering and Prescaling can be configured in CCMR or Capture Compare Mode Registers by manipulating the relevant bits (ICxPSC for prescaling and ICxF for filtering.)

Figure 127. Example of counter operation in encoder interface mode.



15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ETP	ECE	ETPS[1:0]		ETF[3:0]				MSM	TS[2:0]			Res.	SMS[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	Res.	rw	rw	rw

SMCR

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2 CE	OC2M[2:0]			OC2 PE	OC2 FE	CC2S[1:0]		OC1 CE	OC1M[2:0]			OC1 PE	OC1 FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

CCMR

4.2 External Interrupt:

In this method, for finding the position and speed only one channel input has been taken. However, you may extend the code to configure another GPIO pin and take the other channel input too and by adding flags to the ISR, you can find the direction.

Interrupts are managed by the NVIC or Nested Vector Interrupt Controller. This peripheral needs to be activated using the CPACR i.e., Co-Processor Access Control Register in the SCB or System Control Block of the processor. **Detailed information is found in the "*Cortex™ -M4 Devices Generic User Guide*" published by ARM.** To execute the ISR, in the NVIC, the ISER or Interrupt Set Enable register needs to be configured by setting the correct bit which corresponds to the peripheral and desired interrupt. This information is found in Table 61 of RM0090 Reference Manual that contains the Interrupt Vector Table.

For example, if an external interrupt needs to be configured, the following excerpt from Table 61 provides the necessary information:

5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C

EXTI0 has a position of 6 and a priority of 13. Therefore, the 6th bit of the ISER needs to be set.

4.3 Extra Timer:

With either of the above methods, a different timer must be configured to generate an interrupt whose ISR performs the calculations of position/speed at regular intervals. After calculation, the value of the pulses needs to be reset to 0 so that a fresh count can begin before the next ISR.

5. Components Required:

i. 1 Cortex-M4 STM32F407 Discovery board
ii. Motor with Encoder
iii. Analog Discovery or logic analyzer for encoder signal visualization
iv. Connecting wires

6. Embedded C Codes:

6.1 Input Capture:

6.1.1 Problem Statement:

i. Encoder channels A and B are connected to GPIO PE9 and PE11.
ii. TIM1 is configured in encoder interface mode.
iii. TIM2 is configured for interrupting every 1 second to compute position and speed.
iv. TIM2 ISR resets the count value to take a fresh reading in the next iteration.
v. Please read the comments and accordingly comment out the portion of the code depending on whether you want both channels to be used or the rotation is clockwise or anticlockwise.

6.1.2 Input Capture Code:

```
#include "stm32f4xx.h"
#include <stdio.h>
#include<math.h>
#define ARM_MATH_CM4
#define PPR 1024    //Modify this for different encoders based on value given in encoder datasheet

float speedRad;     // in rad/sec
uint16_t ticks;


/*Enabling Clocks*/
void clocksEnable(void)
{
        RCC->AHB1ENR |= RCC_AHB1ENR_GPIOEEN;        //GPIO Port E for PE9, PE11
        RCC->APB2ENR |= RCC_APB2ENR_TIM1EN;      //TIM1
        RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;      //TIM2
}

/*Configuring GPIO for input capture*/
void configGPIO(void)
{
        GPIOE->MODER |= ((2<<18)|(2<<22));      //AF for PE9, PE11
        GPIOE->AFR[1] |= ((1<<4)|(1<<12));      //AF1 for TIM1_CH1, TIM1_CH2
}


/*Configuring TIM1 in Encoder Interface Mode*/
void configTIM1(void)
{
        TIM1->PSC = 0x0000;              //Prescalar = 0
        TIM1->ARR = 0xFFFF;               //Period = 65536

        //Decide this based on how many channels are taken as input
        //TIM1->SMCR |= (3<<0);        //Encoder mode 3, both TI1 and TI2 i.e. channel A and B are
captured
        TIM1->SMCR |= (1<<0);         //Encoder mode 1, only TI1 i.e. channel A is captured

        /*
         * IC1F=IC2F = 0b0000, IC1PSC=IC2PSC = 0b00, CC1S=CC2S = 0b01
         * TIM1->CCMR1 = 0b0000000100000001;
         */

        TIM1->CCMR1 |= ((0x1<<0)|(0x1<<8));          //CC1 and CC2 are input. CC1 = TI1, CC2 = TI2
        TIM1->CCMR1 &= ~((0x3<<2)|(0x3<<10));     //Prescalars are 0 for IC1PSC and IC2PSC
```

```
	TIM1->CCMR1 &= ~((0xF<<4)|(0xF<<12));	//Filters are 0 for IC1F and IC2F
	TIM1->CCER |= ((0x1<<0)|(0x1<<4));  //CC1E AND CC2E ARE ENABLED
}


/*Configuring TIM2 for an interrupt every 1s*/
void configTIM2(void)
{
	TIM2->CR1 &= ~(0x0010);			//Set the mode to Count up
	TIM2->PSC = 16000-1;			//Set the Prescalar
	TIM2->ARR = 1000-1;			//Set period (Auto reload) to 1000
	TIM2->SR &= ~(0x1<<0);				//Clear Update interrupt flag

}

int main(void)
{
	SCB->CPACR|=(0xF<<20);
	clocksEnable();
	configGPIO();
	configTIM1();
	configTIM2();

	NVIC->ISER[0] |= 1<<28;
	TIM2->DIER |=(1<<0);

	TIM2->CR1 |= (1<<0);
	TIM1->CR1 |= (1<<0);

	while(1)
	{
		//Nothing here
	}
}

void TIM2_IRQHandler()
{
//Select which code to execute based on direction of rotation and no. of inputs.

	#if 0
	  //For 2 channel input,
	  ticks = TIM1->CNT;
	  //Comment out speedRAD formula based on whether it's clockwise or anti-clockwise
	  //speedRad = (2*3.14*(65535-ticks))/(4*PPR*1.0);	//rad/s for anti-clockwise as seen from
ceiling
```

```
            //speedRad = (2*3.14*(ticks))/(4*PPR*1.0);              //rad/s for clockwise as seen from
ceiling
            TIM1->CNT=0;
            TIM2->SR &= ~(0x1<<0);
          #endif

          #if 0
            //For 1 channel input,
            ticks = (TIM1->CNT)*2;
            //Comment out speedRAD formula based on whether it's clockwise or anti-clockwise
            //speedRad = (2*3.14*(65535-ticks))/(4*PPR*1.0);      //rad/s for anti-clockwise as seen from
ceiling
            //speedRad = (2*3.14*(ticks))/(4*PPR*1.0);              //rad/s for clockwise as seen from
ceiling
            TIM1->CNT=0;
            TIM2->SR &= ~(0x1<<0);
          #endif

  }
```

6.2 External Interrupt:

6.2.1 Problem Statement:

i. Encoder single channel signal is given to GPIO PA0.
ii. TIM2 is configured for interrupting every 1 second to compute position and speed.
iii. TIM2 ISR resets the count value to take a fresh reading in the next iteration.

6.2.2 External Interrupt Code:

```
/*
 * This code computes speed of the motor using just channel A of the encoder.
 * Hence, for every cycle of signal A, there are 2 pulses generated. PPR = CPR = 1024.
 * Using external interrupts, the number of cycles is determined as pulseCount.
 * Every 1s, pulseCount is read, speed is computed and pulseCount is reset by TIM2's ISR.
 * Give input from encoder to PA0
 */

#include "stm32f4xx.h"

#define ARM_MATH_CM4
```

```
#define PPR 1024
#define PI 3.14

void EXT_Init(void);
void External_Interrupt_Enable(void);

unsigned int pulseCount = 0;
float positionDEG;
float positionRAD;
float speedRPS;
float speedRADS;

void EXT_Init(void)
{
	RCC->APB1ENR |= (1<<0);  // enable PORTA clock
	EXTI->RTSR |=(1<<0);
	EXTI->IMR |=(1<<0);
}


/*Configuring TIM2 for an interrupt every 1s*/
void configTIM2(void)
{
	RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;			//TIM2 clock enable
	TIM2->CR1 &= ~(0x0010);				//Set the mode to Count up
	TIM2->PSC = 16000-1;				//Set the Prescalar
	TIM2->ARR = 1000-1;				//Set period (Auto reload)
	TIM2->SR &= ~(0x0001);				//Clear Update interrupt flag
	NVIC->ISER[0] |= 1<<28;
	TIM2->DIER |=(1<<0);
}

void External_Interrupt_Enable(void)
{
	NVIC->ISER[0] |= 1<<6;
}

int main ()
{
	SCB->CPACR|=(0xF<<20); //Co-processor (core peripheral) access control register
	EXT_Init();

	External_Interrupt_Enable();
	configTIM2();
	TIM2->CR1 |= (1<<0);			//Enable TIM2
	while(1)
```

```
        {

        }
}

void EXTI0_IRQHandler( )
{
        pulseCount++;
        EXTI->PR |= (1<<0);
}

void TIM2_IRQHandler( )
{
        positionDEG  = 360*pulseCount/(PPR*1.0);
        positionRAD  = 2*PI*pulseCount/(PPR*1.0);
        speedRPS  = pulseCount/(PPR*1.0);
        speedRADS = 2*PI*pulseCount/(PPR*1.0);

        pulseCount = 0;
        TIM2->SR &= ~(0x0001);
}
```

7. Logic Analyzer Output:

Insert images from logic analyzer here

8. Conclusion:

In this experiment, the position and speed of a motor have been measured using an encoder through both input capture and external interrupt methods.

Pre-Emptive Task Scheduling using RTOS Kernel for Multitasking Applications

1. Aim:

To implement the creation of tasks, scheduling of tasks using FreeRTOS APIs and ARM Cortex-M microcontroller.

2. Introduction:

A real-time system is one in which tasks have deadlines. If the software and hardware are not sufficiently responsive, then the task will not complete before its deadline, leading to a system failure. Real-time scheduling analysis gives us the mathematical methods to calculate the worst-case response time for each task in such a software system. We can compare these response times to our system's deadlines in order to verify whether the system is schedulable (will always meet its deadlines).

If the system is not schedulable, then we have several options to make it schedulable. We could change the hardware (use a faster processor) if the customer budget allows it. We could improve the application software by speeding up the code or by reducing the amount of processing needed. We could also improve the scheduling approach by changing the balance of work performed in ISRs versus deferred activities, adding or changing task priorities, or adding preemption.

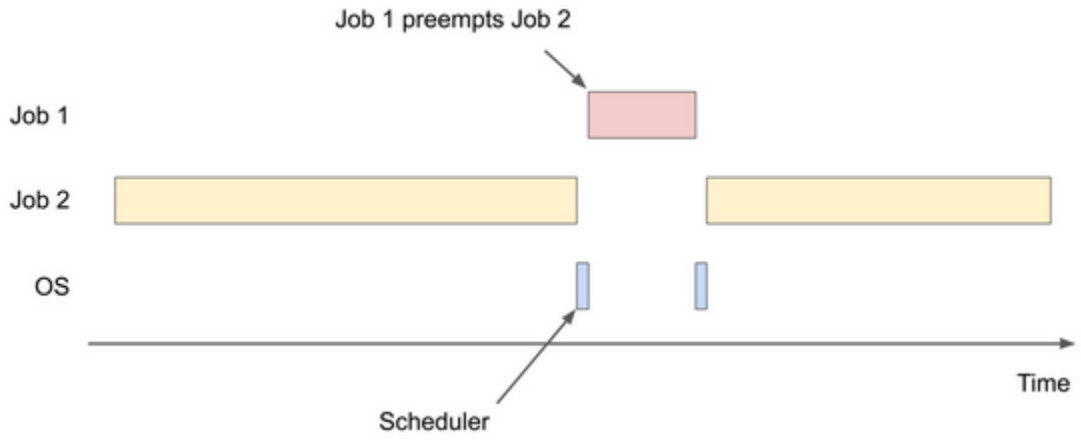
A real-time kernel (RTK) or a real-time operating system (RTOS) is designed to make it easier to create real-time systems. Preemptive scheduling is typically used to provide short response times. Prioritized task scheduling also reduces response times. The kernel is designed and built to execute with consistent and predictable timing, rather than with widely varying behavior.

Example:

The focus changes in most microcontroller applications. Often, meeting strict timing deadlines is critically important. Think about a motor controller or an automatic braking system in an electric vehicle. In these cases, if timing is off by even a few milliseconds, human lives could be in danger. As a result, an RTOS would be your best bet to manage several jobs running concurrently.

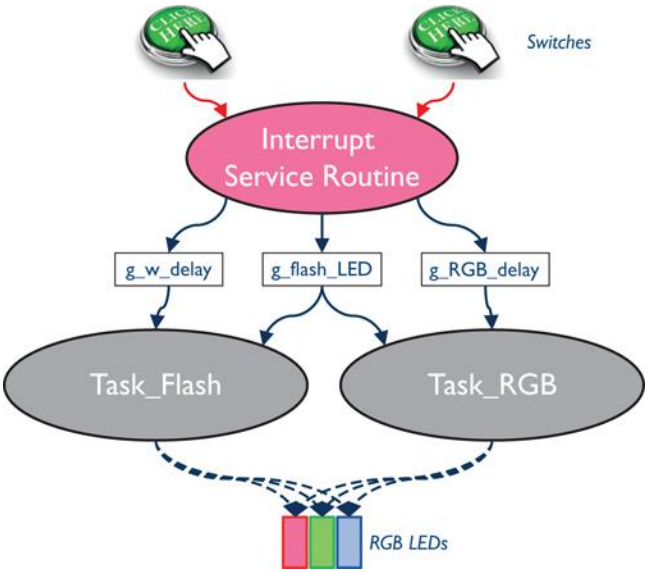
In this second example, a vehicle's electronic control unit (ECU) might be in charge of controlling and assisting with braking. Job 2 monitors the driver's input and helps apply the brakes and turns

on the taillights. However, let's say that our ECU gets notification that the car's sensors have detected an impending crash. As a result, job 1 will preempt job 2 to control the brakes. This all assumes, of course, that this is a vehicle equipped with automatic braking assist.



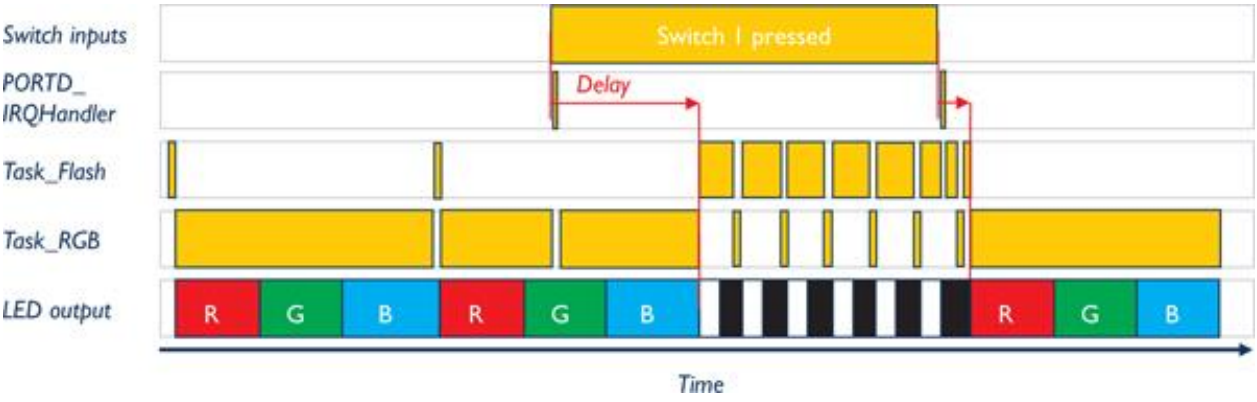
3. Problem Statement:

Consider a system with an MCU, two switches, and an RGB (red, green, blue) LED. When switch 1 is not pressed, the system displays a repeating sequence of colors (red, then green, then blue). When switch 1 is pressed, the system makes the LED flash white (all LEDs on) and off (all LEDs off) until the switch is released. Assume that the IRQ handler can tell the scheduler to change which task to run, and that tasks can preempt each other. IRQHandler starts executing as soon as the switch changes from pressed to released or from released to pressed.



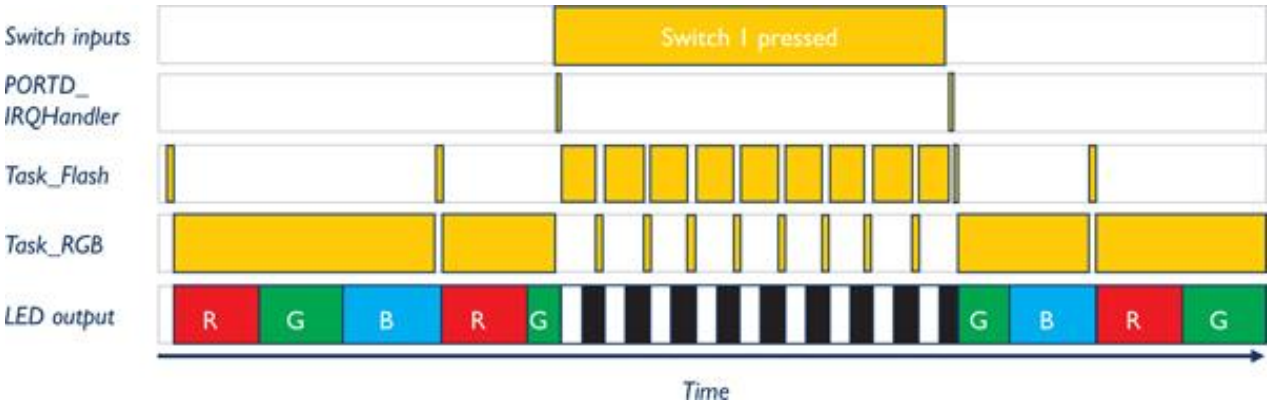
The cooperative multitasking approach we have been examining does not switch to running a different task until the currently running task yields the processor. This delays the system's response. A preemptive multitasking approach uses task preemption so that an urgent task is not delayed by a currently running task of lower urgency. Consider that task A is already running, and task C becomes ready to run, perhaps due to an ISR executing and saving some deferred work for C. A preemptive scheduler can temporarily halt the processing of task A, run task C, and then resume the processing of task A.

The LED flasher with two tasks and an interrupt handler is shown in Figure. Pressing or releasing the switch triggers an interrupt event. The interrupt support hardware forces the CPU to preempt the currently running Task_RGB, execute the code of PORTD_IRQHandler (the ISR), and then resume Task_RGB where it left off. However, we still have to wait for Task_RGB to complete.



An LED flasher with two tasks and an interrupt handler is still delayed by Task_RGB after interrupt.

With task preemption in Figure, we can use kernel features so that Task_Flash (not Task_RGB) runs after PORTD_IRQHandler. Once Task_Flash completes its work and waits, then Task_RGB can run.



An LED flasher with task preemption. Task_Flash prempts Task_RGB in the green cycle when the switch is pressed.

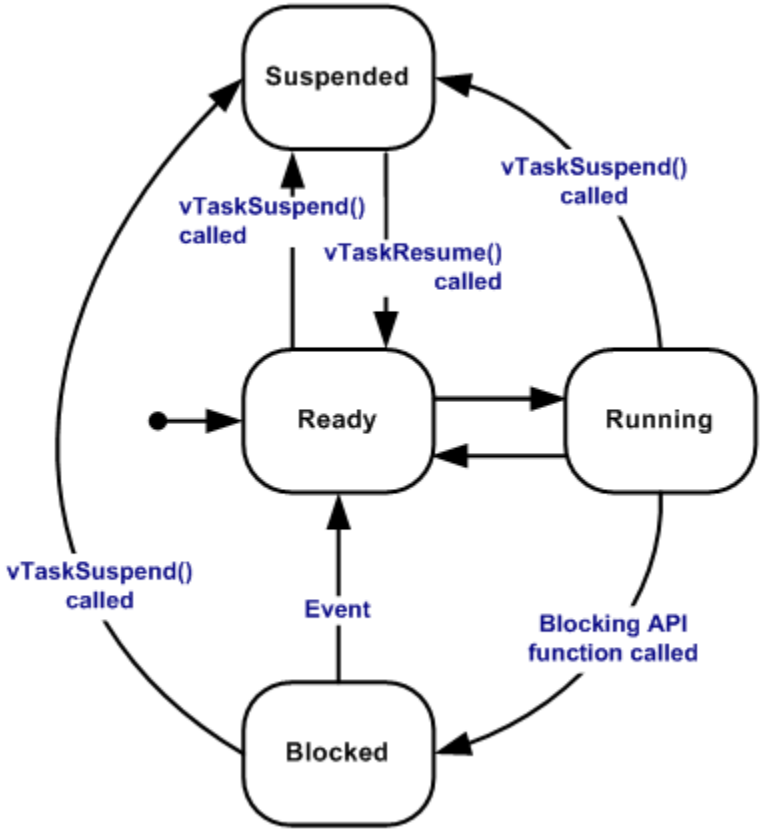
4. FreeRTOS:

FreeRTOS is a market-leading real-time operating system (RTOS) for microcontrollers and small microprocessors. Distributed freely under the MIT open source license, FreeRTOS includes a kernel and a growing set of IoT libraries suitable for use across all industry sectors. FreeRTOS is built with an emphasis on reliability and ease of use.

4.1 Features:

- Has a minimal ROM, RAM and processing overhead. Typically, an RTOS kernel binary image will be in the region of 6K to 12K bytes.
- Very simple - the core of the RTOS kernel is contained in only 3 C files. The majority of the many files included in the .zip file download relate only to the numerous demonstration applications.
- Truly free for use in commercial applications .

4.2 FreeRTOS Task State Diagram:



i. Creating a Task

```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           configSTACK_DEPTH_TYPE usStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                       );
```

ii. Implementing a Task

A task should have the following structure:

```
void vATaskFunction( void *pvParameters )
{
    for( ;; )
    {
        -- Task application code here. --
    }

    /* Tasks must not attempt to return from their implementing
    function or otherwise exit.  In newer FreeRTOS port
    attempting to do so will result in an configASSERT() being
    called if it is defined.  If it is necessary for a task to
    exit then have the task call vTaskDelete( NULL ) to ensure
    its exit is clean. */
    vTaskDelete( NULL );
}
```

4.3 FreeRTOS vs. CMSIS-RTOS:

It's important to understand how STM32CubeIDE has bundled FreeRTOS. While FreeRTOS is an underlying software framework that allows for switching tasks, scheduling, etc., we won't be making calls to FreeRTOS directly. ARM has created the CMSIS-RTOS library, which allows us to make calls to an underlying RTOS, thus improving the portability of code among various ARM processors.

5. Program Code:

Procedure:

i. Create two tasks called blink01 and blink02.
ii. Assign the required stack and priority.
iii. Implement the tasks blink01 and blink02 as per the application requirement.
iv. Schedule the tasks.
v. Trace the task execution.
vi. Change the priorities of the tasks and analyze the behavior.

```
/* Includes ------------------------------------------------------------------*/

#include "main.h"
#include "cmsis_os.h"
/* Definitions for blink01 */
osThreadId_t blink01Handle;
const osThreadAttr_t blink01_attributes = {
  .name = "blink01",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityBelowNormal,
};
/* Definitions for blink02 */
osThreadId_t blink02Handle;
const osThreadAttr_t blink02_attributes = {
  .name = "blink02",
  .stack_size = 128 * 4,
  .priority = (osPriority_t) osPriorityNormal,
};

void StartBlink01(void *argument);
void StartBlink02(void *argument);

//GPIO Initialization
void configureLED(void)
{
		RCC->AHB1ENR |=(1UL<<3);
		GPIOD->MODER &= ~(0xFFUL<<12*2);
		GPIOD->MODER |= (0x55UL<<12*2);
}
void msDelay(int msTime)
{
		//Assume for loop take 12 clock cycles and system clock is 16MHz
int Time=msTime*1333;
		for(int i=0;i<Time;i++);
}
int main(void)
{

  HAL_Init();
  configureLED();

  osKernelInitialize();
  blink01Handle = osThreadNew(StartBlink01, NULL, &blink01_attributes);

  /* creation of blink02 */
  blink02Handle = osThreadNew(StartBlink02, NULL, &blink02_attributes);

  /* Start scheduler */
  osKernelStart();
```

```
  while (1)
  {
  }
}



void StartBlink01(void *argument)
{

                for(;;)
                 {
                          GPIOD->ODR ^= (0x1UL<<12);
                          msDelay(2000);
                   osDelay(2000);
                 }

}

/* USER CODE END Header_StartBlink02 */
void StartBlink02(void *argument)
{

  for(;;)
  {
                GPIOD->ODR ^= (0x2UL<<12);;
                osDelay(200);
  }

}
```

6. Conclusion:

Studied in detail about RTOS and implemented a simple RTOS system via embedded C code.

7. References:

1. FreeRTOS APIs: https://www.freertos.org/a00106.html