Internship Training Cum Project Report

Final

At

PHYTEC Embedded Pvt. Ltd.

For the period 04/02/2023 to 04/08/2023

Submitted
to
Electrical & Electronics Engineering

In partial fulfillment of the award of the degree of
Bachelor of Technology

In

Embedded Full Stack IoT Analyst

By

Deepak Kumar Beniya

1901227415

8 th Semester

Electrical & Electronics Engineering
C.V. Raman Global University, Bhubaneswar, Odisha

C.V. Raman Global University
Bhubaneswar, Odisha -752054

BONAFIDE CERTIFICATE

I hereby declare that work related to a Report titled "Internship at **PHYTEC INDIA**" submitted herein, has been carried out at **Skill Development Institute, BBSR**. The work is original and has not been submitted earlier as a whole or in part for the award of any degree at this or any other institution.

SIGNATURE	**SIGNATURE**
DR. Ashwani Kumar Sahoo	Gopalakrishna polimera
HEAD OF THE DEPARTMENT	**Technical Manager**
Electrical & Electronics Engineering	Embedded Full Stack IoT Analyst

C.V. Raman Global University
Bhubaneswar, Odisha -752054

CERTIFICATE OF APPROVAL

This is to certify that we have examined and approved the final Internship report for the 8th semester in **Electrical & Electronics Engineering** entitled "**Advance C Programming and Microcontroller**" submitted by **Deepak Kumar Beniya (1901227415)**.

We here by accord our approval of it as an Internship work carried out and presented in a manner required for its acceptance for the partial fulfillment of the award of the degree of Bachelor of Technology(B.Tech) in **Electrical & Electronics Engineering** (EEE) Department for which it has been submitted. This approval does not necessarily endorse or accept every statement made,opinion expressed, or conclusions drawn as recorded in this internship report, it only signifies the acceptance of the internship report for the purpose it has been submitted.

(Project Guide) **(External Examiner)** **(Internal Examiner)**

ACKNOWLEDGMENT

We would like to express our special thanks of gratitude to our Trainer **Gopalakrishna polimera and Chaman Kumar** as he gave us the excellent opportunity to do this wonderful Internship on the Module "**Advance C Programming and Microcontroller**", which also helped us in doinga lot of Research and we came to know about so many new things. I amthankful to him.
We express enough thanks to the college for their continued support and encouragement. We offer our sincere appreciation for the learning opportunities provided by our college.

Our completion of this Module could not have been accomplished without the support of our parents. Thanks to them as well. The countless times they kept us during our hectic schedules will not be forgotten. It was a great comfort and relief to know that they were willing to provide management of Ours SDI campus activities while I completed my work.

Their encouragement when the times got rough is much appreciated and noted. Our heartfelt thanks.

Thanks to the fellow interns as well for their continued support and comfort.
Thanks to everyone associated with the Module at every micro level.
Thanks to our college and all members associated with the college.

CONTENTS

1. INTRODUCTION

1.1 Advance C Programming

Advanced C programming is widely used in various domains, including system programming, embedded systems, game development, and scientific computing. It serves as the foundation for many other programming languages.

In This report is intended to expose the intermediate level C programmer to advanced concept. Concepts include File handling and makefile.

1.2 ARM MCU Processor

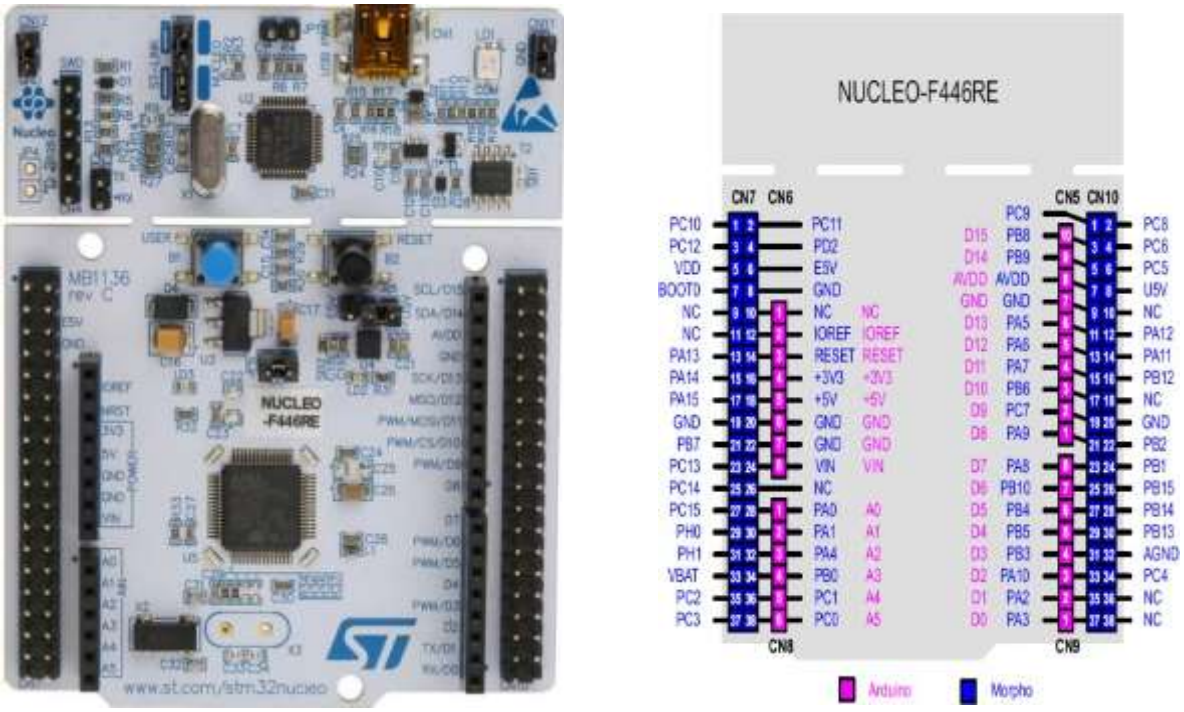
The Cortex-M4 processor is a low-power processor that features low gate count, low interrupt latency, and low-cost debug. The Cortex-M4 with FPU is a processor with the same capability as the Cortex-M4 processor and includes floating-point arithmetic functionality. Both processors are intended for deeply embedded applications that require fast interrupt response features.

The Cortex-M4 processor is a high performance 32-bit processor designed for the microcontroller market. It offers significant benefits to developers, including: • outstanding processing performance combined with fast interrupt handling • enhanced system debug with extensive breakpoint and trace capabilities • efficient processor core, system and memories • ultra-low power consumption with integrated sleep mode and an optional deep sleep mode • platform security robustness, with optional integrated Memory Protection Unit (MPU).

The Cortex-M4 processor is built on a high-performance processor core, with a 3-stage pipeline Harvard architecture, making it ideal for demanding embedded applications. The processor delivers exceptional power efficiency through an efficient instruction set and extensively optimized design, providing high-end processing hardware including optional IEEE754-compliant single-precision floating-point computation, a range of single-cycle and SIMD multiplication and multiply-with-accumulate capabilities, saturating arithmetic and dedicated hardware division. To facilitate the design of cost-sensitive devices, the Cortex-M4 processor implements tightly coupled system components that reduce processor area while significantly improving interrupt handling and system debug capabilities. The Cortex-M4 processor implements a version of the Thumb® instruction set based on Thumb-2 technology, ensuring high code density and reduced program memory requirements. The Cortex-M4 instruction set provides the exceptional performance expected of a modern 32-bit architecture, with the high code density of 8-bit and 16-bit microcontrollers. The Cortex-M4 processor closely integrates a configurable Nested Vectored Interrupt Controller (NVIC), to deliver industry-leading interrupt performance. The NVIC includes a Non Maskable Interrupt (NMI) that can provide up to 256 interrupt priority levels. The tight integration of the processor core and NVIC provides fast execution of

Interrupt Service Optional Embedded Trace Macrocell NVIC Optional Debug Access Port Optional Memory protection unit Optional WIC Optional Serial Wire viewer Bus matrix Code interface SRAM and peripheral interface Optional Data watchpoints Optional Flash patch Cortex-M4 processor Optional FPU Processor core Introduction ARM DUI 0553A Copyright © 2010 ARM. All rights reserved. 1-3 ID121610 Non-Confidential Routines (ISRs), dramatically reducing the interrupt latency. This is achieved through the hardware stacking of registers, and the ability to suspend load-multiple and store-multiple operations. Interrupt handlers do not require wrapping in assembler code, removing any code overhead from the ISRs. A tail-chain optimization also significantly reduces the overhead when switching from one ISR to another. To optimize low-power designs, the NVIC integrates with the sleep modes, that includes an optional deep sleep function. This enables the entire device to be rapidly powered down while still retaining program state.

Figure 1.1 STM32F446RE



System-level interface:-The Cortex-M4 processor provides multiple interfaces using AMBA® technology to provide high speed, low latency memory accesses. It supports unaligned data accesses and implements atomic bit manipulation that enables faster peripheral controls, system spinlocks and thread-safe Boolean data handling. The Cortex-M4 processor has an optional Memory Protection Unit (MPU) that permits control of individual regions in memory, enabling applications to utilize multiple privilege levels, separating and protecting code,

data and stack on a task-by-task basis. Such requirements are becoming critical in many embedded applications such as automotive.

Figure 1.2 Debugger

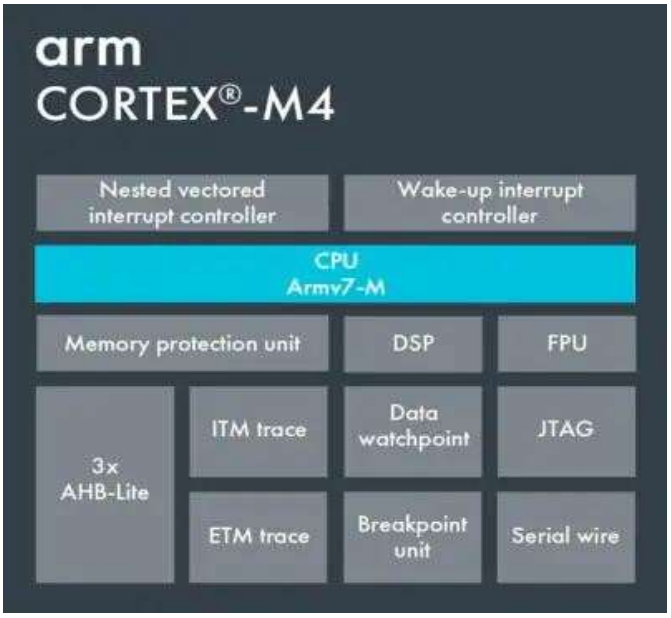
Optional integrated configurable debug:-The Cortex-M4 processor can implement a complete hardware debug solution. This provides high system visibility of the processor and memory through either a traditional JTAG port or a 2-pin Serial Wire Debug (SWD) port that is ideal for microcontrollers and other small package devices. For system trace the processor integrates an Instrumentation Trace Macrocell (ITM) alongside data watchpoints and a profiling unit. To enable simple and cost-effective profiling of the system events these generate, a Serial Wire Viewer (SWV) can export a stream of software-generated messages, data trace, and profiling information through a single pin. The optional Embedded Trace Macrocell™ (ETM) delivers unrivalled instruction trace capture in an area far smaller than traditional trace units, enabling many low cost MCUs to implement full instruction trace for the first time. The optional Flash Patch and Breakpoint Unit (FPB) provides up to eight hardware breakpoint comparators that debuggers can use. The comparators in the FPB also provide remap functions of up to eight words in the program code in the CODE memory region. This enables applications stored on a non-erasable, ROM-based microcontroller to be patched if a small programmable memory, for example flash, is available in the device. During initialization, the application in ROM detects, from the programmable memory, whether a patch is required. If a patch is required, the application programs the FPB to remap a number of addresses. When those addresses are accessed, the accesses are redirected to a remap table specified in the FPB configuration, which means the program in the non-modifiable ROM can be patched.

1.3 Cortex-M4 Core Peripherals

Nested Vectored Interrupt Controller: - The NVIC is an embedded interrupt

controller that supports low latency interrupt processing.
System Control Block: -The System Control Block (SCB) is the programmer's model interface to the processor. It provides system implementation information and system control, including configuration, control, and reporting of system exceptions.
System timer: - The system timer, SysTick, is a 24-bit count-down timer. Use this as a Real Time Operating System (RTOS) tick timer or as a simple counter.

Figure1.3 Cortex-M4 Core Peripherals



Memory Protection Unit: - The Memory Protection Unit (MPU) improves system reliability by defining the memory attributes for different memory regions. It provides up to eight different regions, and an optional predefined background region. Floating-point Unit the Floating-Point Unit (FPU) provides IEEE754-compliant operations on single-precision, 32-bit, floating-point values.

2 Advance C Programming

2.1 File Handling

2.1.1 Introduction

File handling is a fundamental aspect of advanced C programming that allows developers to read from and write to files, enabling data persistence and manipulation. This report provides an in-depth analysis of file handling in advanced C programming, discussing its purpose, techniques, usage, and benefits.

2.1.2 Purpose Of File Handling

File handling serves several crucial purposes in advanced C programming:

a. Data Persistence: File handling enables the storage and retrieval of data beyond the program's runtime. It allows the preservation of information across multiple program executions, facilitating data sharing and long-term storage.

b. Data Manipulation: File handling provides mechanisms for manipulating data within files. Developers can read from and write to files, perform search operations, modify content, and organize data structures efficiently.

c. Data Exchange: File handling enables seamless data exchange between programs and systems. It allows data to be transferred between different platforms, facilitating interoperability and integration of software components.

2.1.3 Techniques For File Handling

Advanced C programming offers various techniques for file handling:

a. File Pointers: File pointers are used to access and manipulate files. They provide the means to perform read and write operations at specific positions within a file. File pointers keep track of the current file position, allowing sequential or random access to data.

b. File Opening and Closing: Before accessing a file, it must be opened using the fopen() function, which returns a file pointer. After completing the operations, the file should be closed using the fclose() function to release system resources and ensure data integrity.

c. File Reading and Writing: File handling functions such as fread() and fwrite() are used to read and write data to files, respectively. These functions enable

efficient reading and writing of multiple data structures, ensuring accuracy and consistency.

d. File Positioning: File positioning functions like fseek() and ftell() allow developers to navigate within a file. They enable seeking to a specific position in the file or obtaining the current position, providing flexibility in data access and manipulation.

2.1.4 Usage of File Handling

To perform file handling operations in advanced C programming, developers typically follow these steps:

a. Open the File: Use the fopen() function to open the file and obtain a file pointer. Specify the file mode, which can be "r" for reading, "w" for writing, "a" for appending, or combinations such as "r+" for reading and writing.

b. Perform Operations: Use file handling functions such as fread(), fwrite(), fseek(), and others to perform desired operations on the file. Read data, write data, navigate within the file, or modify its content as needed.

C. Close the File: After completing the operations, close the file using the fclose() function. This ensures proper release of system resources and prevents data corruption.

2.1.5 Benefits Of File Handling In Advanced C Programming

File handling in advanced C programming offers several benefits:

Data Persistence: File handling allows developers to store data persistently, ensuring that information is retained across program executions.

b. Data Manipulation: File handling provides efficient techniques for reading, writing, and manipulating data within files. It enables complex data operations and organization.

c. Portability: File handling operations are platform-independent, allowing seamless data exchange between different systems and environments.

e. Integration: File handling facilitates the integration of software components by enabling data exchange between different programs and systems.

2.2 MAKEFILE

2.2.1 Introduction

Makefiles are an essential tool in advanced C programming for automating the compilation and building process of a project. They allow developers to define a set of rules and dependencies, making it easier to manage large projects with multiple source files. This report will provide an overview of Makefiles in advanced C programming, including their purpose, structure, and usage.

2.2.2 Purpose Of Make Files

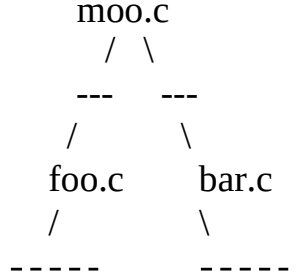
Makefiles serve the purpose of automating the build process by specifying rules and dependencies between files. They determine which files need to be recompiled based on the changes made in the source code, ensuring that only the necessary parts of the project are rebuilt. Makefiles are especially useful when dealing with large projects with numerous source files, libraries, and dependencies.

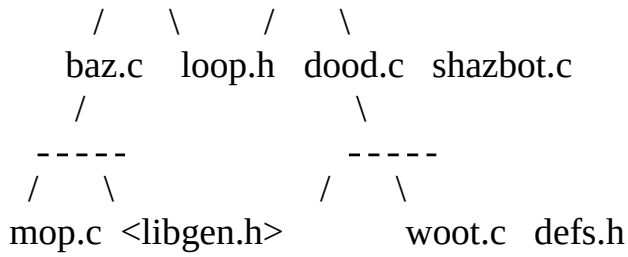
2.2.3 Structure Of Makefiles

Makefiles consist of rules, variables, and directives. Let's look at each of these components:

Rules: Rules define the dependencies and actions required to build a specific target. A rule typically consists of a target, prerequisites (dependencies), and commands. The target is usually the name of a file to be generated, while prerequisites are the files required to build the target. Commands are shell commands that are executed to build the target.
Variables: Variables are used to store values that can be referenced throughout the Makefile. They are helpful for defining compiler flags, source file lists, and other project-specific settings. Variables can be defined using the variable_name = value syntax and referenced using the $(variable_name) syntax.

Let's use a different example. The hypothetical source tree:





2.2.4 Makefile Example

Stack contains StackImplementation.c and the following Makefile:

```
export: StackImplementation.o
StackImplementation.o: StackImplementation.c \
                       /Include/StackTypes.h \
                       /Include/StackInterface.h
                       gcc -I/Include -c StackImplementation.c
#substitute a print command of your choice for lpr below    print:
lpr StackImplementation.c clean:
                                  rm -f *.o
```

2.2.5 The make Utility

Programs consisting of many modules are nearly impossible to maintain manually.

This can be addressed by using the make utility.

```
    # Makefile for the sample
    sample:  sample.o  my_stat.o
                cc –o sample sample.o my_stat.o
    sample.o: sample.c  my_stat.h
                cc –c sample.c
    my_stat.o: my_stat.c my_stat.h
                cc –c my_stat.c
  clean:
    rm sample *.o core
```

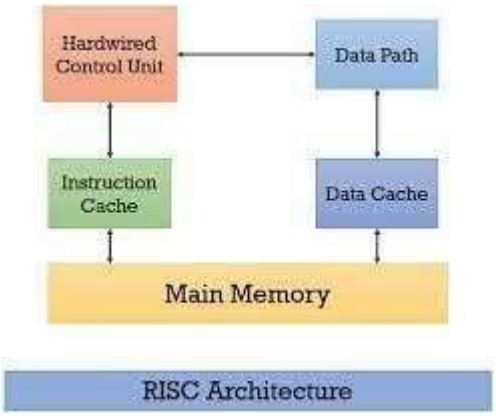
3. ARM MCU Programming

3.1 ARM Processor

The ARM processor is a type of microprocessor architecture developed by ARM Holdings. It stands for "Advanced RISC Machines" and is known for its energy efficiency and widespread use in mobile devices, embedded systems, and other applications.

ARM processors are based on a Reduced Instruction Set Computing (RISC) architecture, which simplifies the instruction set and allows for faster execution of instructions. They are designed to consume less power while delivering high performance.

Figure 3.1 RISC Architecture



ARM processors are commonly found in smartphones, tablets, smartwatches, and other mobile devices due to their power efficiency. They are also used in a wide range of other applications, including automotive systems, industrial control systems, medical devices, and Internet of Things (IoT) devices.

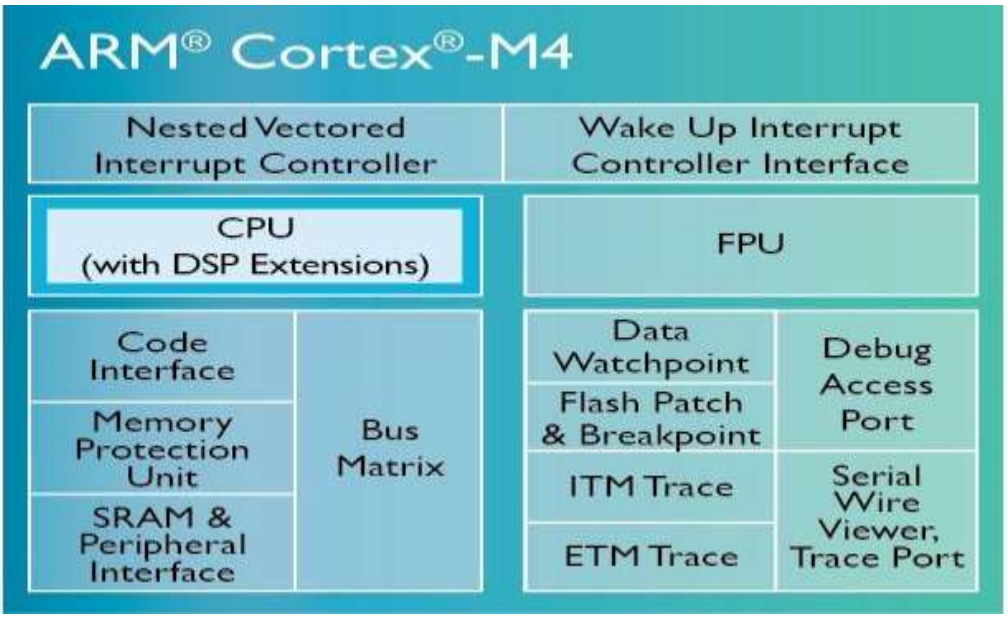
Figure 3.2 Internet Of Things

ARM-based processors are licensed to various companies, which often customize them to meet their specific requirements. Some popular ARM processor families include Cortex-A (application processors), Cortex-M (microcontrollers), and Cortex-R (real-time processors).

3.1.1 Basic ARM Cortex Arch

- Architecture-Arm.
- Version-7.
- Core-cortexM4.
- 512 kb of flash memory(ROM).
- 128 kb of SRAM.
- Maximum CPU Frequency 180 mhz.
- The STM32F446RE devices are based on high performance ARM cortex M4 32 bit RISC core operating at a frequency up to 180 mhz.
- Here we use bus matrix with AHB and APB bus .APB1 has maximum frequency 45 MHZ while APB2 has maximum frequency 90 MHZ.and all the peripheral are executed by s-bus.
- The cortex-M4 processor is build on high-performance processor core, with 3-stage pipeline Harvard architecture, making it ideal for demanding embedded application.

Figure 3.3 ARM Cortex-M4



VON NEUMANN ARCHITECTURE	HARVARD ARCHITECTURE
It is ancient computer architecture based on stored program computer concept.	It is modern computer architecture based on Harvard Mark I relay based model.
Same physical memory address is used for instructions and data.	Same physical memory address is used for instructions and data.
There is common bus for data and instruction transfer.	Separate buses are used for transferring data and instruction.
Two clock cycles are required to execute single instruction.	An instruction is executed in a single cycle.
It is cheaper in cost.	It is costly than Von Neumann Architecture.
CPU cannot access instructions and read/write at the same time.	CPU can access instructions and read/write at the same time.
It is used in personal computers and small computers	It is used in micro controllers and signal processing.

3.1.2 Features Of Cortex M0/M3/M4 Processor

- Operational mode of the processor
- The different access levels of the processor.
- The register set of process.
- Exception and Execution handling.
- Interrupt handling.
- Bus Interfaces & bus matrix.
- Memory architecture of the processor ,bit banding, memory map.
- Aligned and unaligned data transfer.

3.2 ARM Core

3.2.1 Operational Mode

processor gives 2 operational mode:-

- ❑ thread mode
- ❑ handler mode

thread mode:-all our application code will execute under thread mode of the processor. This is called user mode.

handler mode:-all the execution handler or interrupt handlers will run under handler mode of the processor.

Access levels of the processor:-

It is of two type:-

- ❑ Privileged mode
- ❑ Non privileged mode

Privileged mode:-if the code is run with PAL,then ours code is full accessed to process specific recourses register.

Non privileged mode:-if our code is running with NPAL,then ours code don't have access to some of the register of the processor.

- By default it will run in privileged mode.
- Once we move out of the pal to npal being in thread mode then its not possible to comeback to pal unless your code change the processor from handler mode to thread mode.
- Handler mode code execution with pal.

3.2.2 Non-memory Mapped

There are register located with in the microcontroller. These register are different as processor register or core register. These are non mapped register and they do not appear in the processor map of the microprocessor because these register do not have any addresses associated with them.

3.2.3 Memory Mapped

Register external to that core are memory mapped register, The register are outside of the processor core and are found in the processor map of the microcontroller also these register have unique address associate can be identified .

3.3 ARM GCC Inline Assembly Code

- ➢ Inline assembly code is used to write pure assembly code inside a c program.
- MOV R0,R1
- Inline assembly statement
- __asm volatile("MOV R0,R1)
- Syntax:-

__asm volatile(::::);

__asm volatile(code:output operand:input operand:cobbler list);

Arg 1:-assembly code and defined as string argument.

Arg 2:-A list of output operand separated by colon.

Arg 3:-A list of input operand separated by colon.

Arg 4:-cobbler list is mainly used to tell the compiler about modification done by the assembler code.

RESET SEQUENCE of cortex-M4:-

- After reset, PC is loaded with the address 0x00000000
- Processor reads the value from the address location 0x00000000 into MSP.
- The processor reads the address of reset handler from the location 0x00000004
- Set the vector table entries with the exceptions ISR address
- Branches to the main in the C library.

Memory mapped of the processor:-

- Memory mapped explains mapping of different peripheral registers and memory in the processor memorable location range.
- The processor addressable memory location range depends upon the size of address bus.
- The mapping of the different region in the addressable memory location range is called memory mapped.
- Processor has the fixed default memory map that provides up to 4GB of addressable memory.
- The processor has a fix default memory map that provides up to 4Gb of addressable memory .

3.3.1 Code Block

Code Blocks is an amazing free, open-source, cross-platform Integrated Development Environment or IDE. It's a powerful tool that's made even more useful when used with plugins which further increase its functionality. To put it in simple terms it's like WordPress in a way since even that platform is made more useful when additional plugins are installed. This is the best way of doing it I think because the plugins allow for a lot of new and interesting stuff to be produced even by the community and that's important. At the moment Code Blocks has primarily focused on C/C++ or Fortran so if these interest you then I definitively suggest you look at this platform.

3.3.2 SRAM

The fastest volatile memory, SRAM, is fast enough to operate close to the processor speed. It also requires less power than DRAM, but it is also more expensive. Engineers use it in more limited ways in embedded systems. Timo Aarnipuro, Senior Software Engineer for Qt, points out that many SoCs and MCUs have a small amount of internal RAM on the chip. "This is usually a combination of SRAM and CPU cache on SoCs. Some chip families have a selection of variants with different on-chip memory configurations. This allows some flexibility on selecting the variant that has the most suitable amount of fast memory."

3.3.3 Peripheral Block

- Peripheral memory region also have size of 512 mb.it is used mostly for unchip peripherals,SRAM region is intented either onchip or offchip memory.
- We can execute code in this region and this region is intended for external device or shared memory.
- This region include ENVIC system timer and system control block.

3.3.4 Bus Protocol And Bus Interfaces

- In cortex M4 processor bus interface are based on AMBS(Advanced microcontroller Bus Architecture specification).
- ABBS is a specification designed by ARM which standard for un chip communication inside the system on chip.
- AMBS specification supports some several Bus protocols like -AHB,APB.

AHB	APB
It is mainly used for main bus interface	It is mainly used for APB access and some on chip peripheral .
Majorly used for High speed comn.	Mostly used for low speed

3.3.5 Stack Memory

- Stack is a part of RAM which is used for temporary storage of data registers.
- It is mainly used during function, interrupts/exception handling.
- Stack can be accessed by using push and pop instruction or using any memory manipulation instructions.
- The stack is traced by using static pointer register.
- Local variable are stored in stable mode
- Push and pop operation can be done.
- In Architecture temporary data storage which is equivalent to stack.

Stack operation are:-Full stack Ascending, Full stack descending, Empty Ascending order, Empty Descending order.

Stack memory uses:-

- Temporary storage of processor register value.

- Temporary storage of local variable of the function
- During system exception or interrupt stack memory will be uses to save the context of general purpose register process. And it have status register and return address of current execution port.

3.3.6 Linker Script

- It decided the starting and end address of stack ,heap, and other
- Estack is the symbol which is populated with stack memory location. Stack(20020000)—starting address

3.4 MCU Programming On ARM7 /Cortex-M4

3.4.1 GPIO Programming

Each general-purpose I/O port has four 32-bit configuration registers (GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR and GPIOx_PUPDR), two 32-bit data registers (GPIOx_IDR and GPIOx_ODR), a 32-bit set/reset register (GPIOx_BSRR), a 32-bit locking register (GPIOx_LCKR) and two 32-bit alternate function selection register (GPIOx_AFRH and GPIOx_AFRL).

3.4.2 GPIO Main Features

- Up to 16 I/Os under control
- Output states: push-pull or open drain + pull-up/down
- Output data from output data register (GPIOx_ODR) or peripheral (alternate function output)
- Speed selection for each I/O
- Input states: floating, pull-up/down, analo
- nput data to input data register (GPIOx_IDR) or peripheral (alternate function input)
- it set and reset register (GPIOx_BSRR) for bitwise write access to GPIOx_ODR
- Locking mechanism (GPIOx_LCKR) provided to freeze the I/O configuration

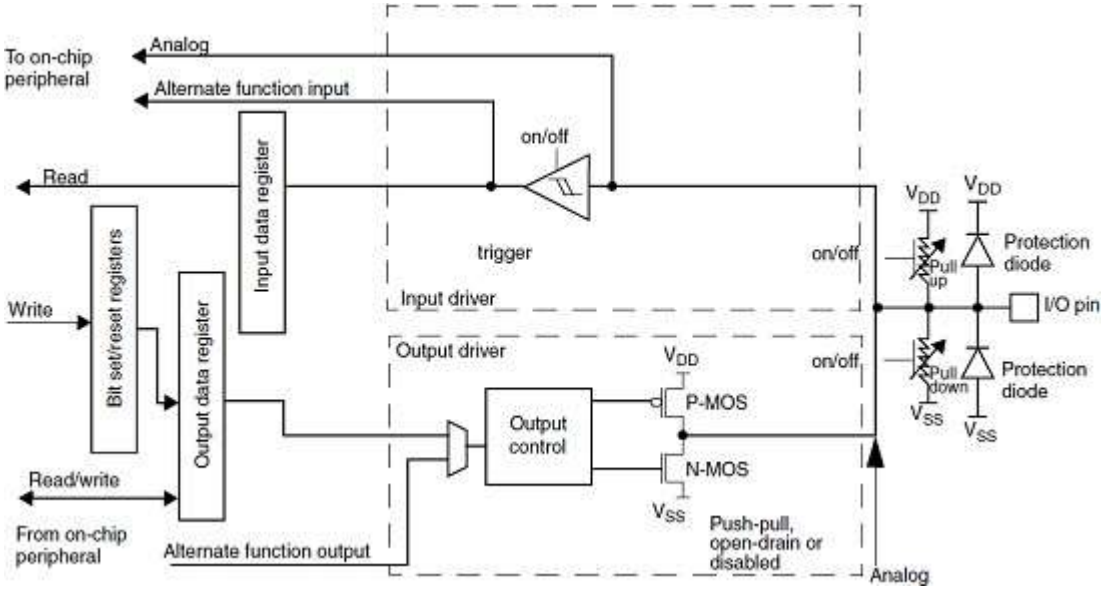
3.4.3 Output Configuration

- The output buffer is enabled: – Open drain mode: A “0” in the Output register activates the N-MOS whereas a “1” in

the Output register leaves the port in Hi-Z (the P-MOS is never activated) –Push-pull mode: A “0” in the Output register activates the N-MOS whereas a “1” in the Output register activates the P-MOS| The Schmitt trigger input is activated

- The weak pull-up and pull-down resistors are activated or not depending on the value in the GPIOx_PUPDR register
- The data present on the I/O pin are sampled into the input data register every AHB1 clock cycle
- A read access to the input data register gets the I/O state
- A read access to the output data register gets the last written value

Figure 3.4 Output Configuration

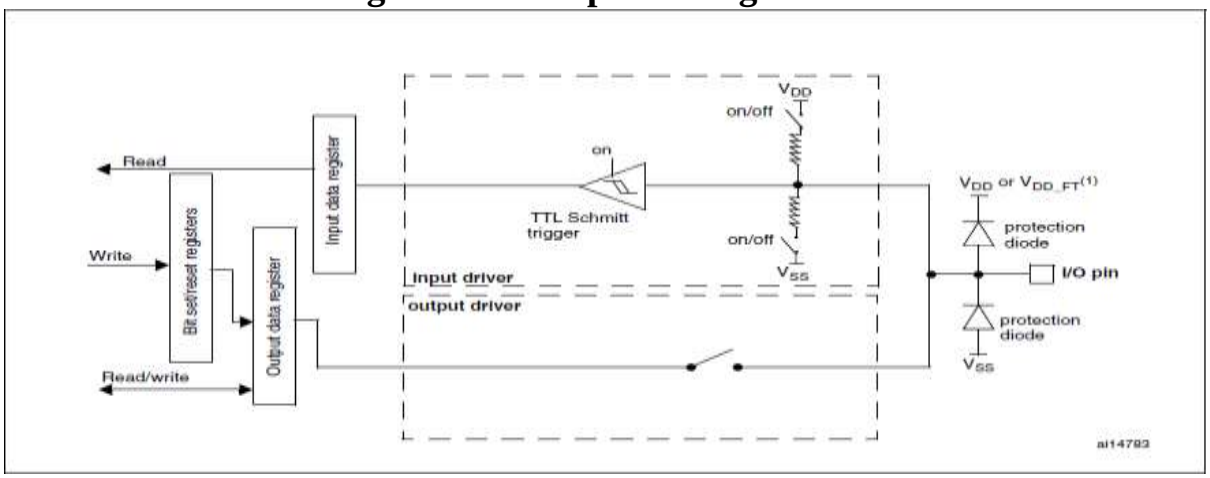


3.4.4 Input Configuration

- the output buffer is disabled
- the Schmitt trigger input is activated
- the pull-up and pull-down resistors are activated depending on the value in the GPIOx_PUPDR register

- The data present on the I/O pin are sampled into the input data register every AHB1 clock cycle
- A read access to the input data register provides the I/O State

Figure 3.5 Input Configuration



3.4.5 GPIO Register

This section gives a detailed description of the GPIO registers. For a summary of register bits, register address offsets and reset values. The GPIO registers can be accessed by byte (8 bits), half-words (16 bits) or words (32 bits).

3.4.6 GPIO Port Mode Register (GPIOx_MODER) (x = A..H)

Address offset: 0x00
Reset values:
0xA800 0000 for port A
0x0000 0280 for port B
0x0000 0000 for other port

Figure 3.6 GPIO Port Mode Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 MODERy[1:0]: Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)

01: General purpose output mode
10: Alternate function mode
11: Analog mode

3.4.7 GPIO Port Output Type Register (GPIOx_OTYPER) (x = A..H)

Address offset: 0x04
Reset value: 0x0000 0000

Figure 3.7 GPIO Port Output Type Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

These bits are written by software to configure the output type of the I/O port.
0: Output push-pull (reset state)
1: Output open-drain

3.4.8 GPIO Port Output Speed Register (GPIOx_OSPEEDR) (x = A.H)

Address offset: 0x08
Reset values:
0x0000 00C0 for port B
0x0000 0000 for other ports

Figure 3.8 GPIO Port Output Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15 [1:0]		OSPEEDR14 [1:0]		OSPEEDR13 [1:0]		OSPEEDR12 [1:0]		OSPEEDR11 [1:0]		OSPEEDR10 [1:0]		OSPEEDR9 [1:0]		OSPEEDR8 [1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1 [1:0]		OSPEEDR0 1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

00: Low speed
01: Medium speed
10: Fast speed
11: High speed

3.4.9 GPIO Port Pull-up/Pull-down Register (GPIOx_PUPDR) (x = A.H)

Address offset: 0x0C
Reset values:
0x6400 0000 for port A
0x0000 0100 for port B
0x0000 0000 for other ports

Figure 3.9 GPIO Port Pull-up/Pull-down Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

These bits are written by software to configure the I/O pull-up or pull-down
00: No pull-up, pull-down
01: Pull-up
10: Pull-down
11: Reserved

3.4.10 GPIO Port Input Data Register (GPIOx_IDR) (x = A..H)

Address offset: 0x10
Reset value: 0x0000 XXXX (where X means undefined

Figure 3.10 GPIO Port Input Data Register

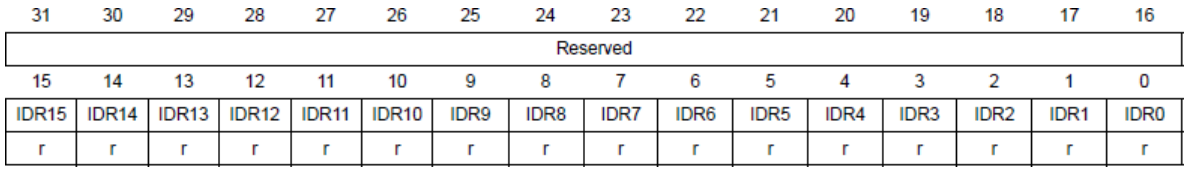
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

These bits are read-only and can be accessed in word mode only. They contain the input value of the corresponding I/O port.

3.4.11 GPIO Port Output Data Register (GPIOx_ODR) (x = A..H)

Reset value: 0x0000 0000

Figure 3.11 GPIO Port Output Data Register



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

ODRy: Port output data (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..H)

3.4.12 GPIO Port Bit Set/Reset Register (GPIOx_BSRR) (x = A..H)

Address offset: 0x18

Reset value: 0x0000 0000

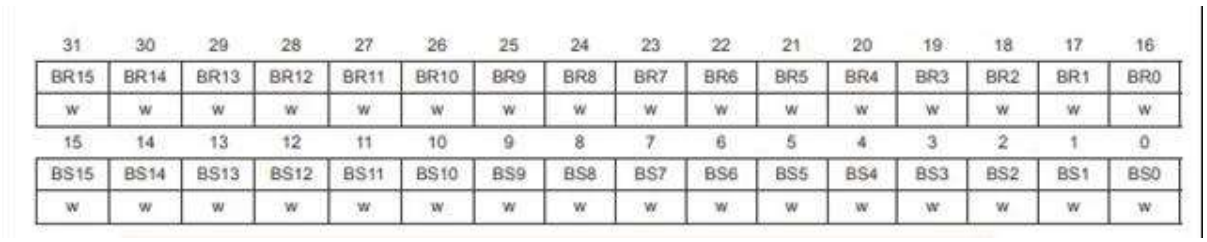
Port x reset bit y (y = 0..15)

These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Resets the corresponding ODRx bit

Figure 3.12 GPIO Port Bit Set/Reset Register



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Port x set bit y (y= 0..15)

These bits are write-only and can be accessed in word, half-word or byte mode.
A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Sets the corresponding ODRx bit

3.4.13 System Configuration Controller (SYSCFG)

The system configuration controller is mainly used to remap the memory accessible in the code area and to manage the external interrupt line connection to the GPIOs.

3.4.14 I/O Compensation Cell

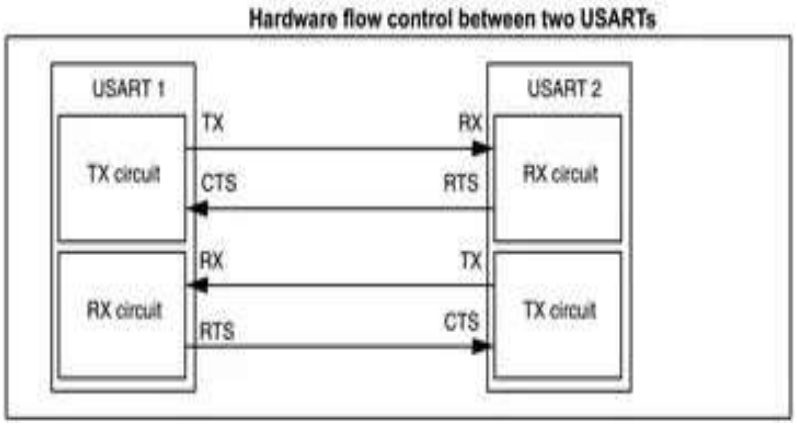
By default the I/O compensation cell is not used. However when the I/O output buffer speed is configured in 50 MHz or 100 MHz mode, it is recommended to use the compensation cell for slew rate control on I/O tf(IO)out)/tr(IO)out commutation to reduce the I/O noise on power supply.
When the compensation cell is enabled, a READY flag is set to indicate that the compensation cell is ready and can be used. The I/O compensation cell can be used only when the supply voltage ranges from 2.4 to 3.6 V.

3.5 USART And UART

3.5.1 USART Introduction

- The universal synchronous asynchronous receiver transmitter (USART) offers a flexible means of full-duplex data exchange with external equipment requiring an industry standard NRZ asynchronous serial data format. The USART offers a very wide range of baud rates using a fractional baud rate generator.

Figure 3.13 Hardware Flow Control Between Two USARTs



- It supports synchronous one-way communication and half-duplex single wire communication. It also supports the LIN (local interconnection network), Smartcard Protocol and IrDA (infrared data association) SIR ENDEC specifications, and modem operations (CTS/RTS). It allows multiprocessor communication.
- High speed data communication is possible by using the DMA for multibuffered configuration.
- USART main features.
- Full duplex, asynchronous communications
- Configurable oversampling method by 16 or by 8 to give flexibility between speed and clock tolerance.
- Fractional baud rate generator systems – Common programmable transmit and receive baud rate (refer to the datasheets for the value of the baud rate at the maximum APB frequency.
- Programmable data word length (8 or 9 bits)
- Configurable stop bits - support for 1 or 2 stop bits.

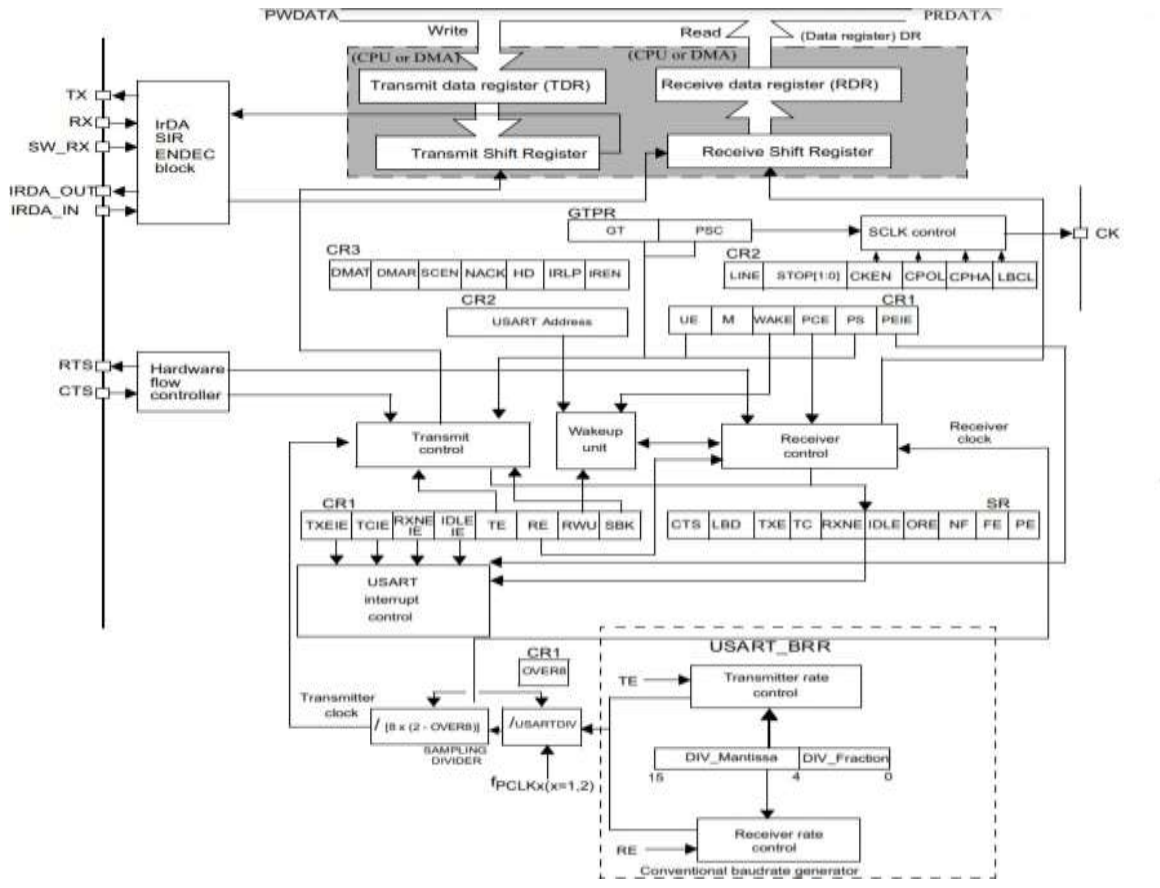
3.5.2 USART Functional Description

- The interface is externally connected to another device by three pins . Any USART bidirectional communication requires a minimum of two pins: Receive Data In (RX) and Transmit Data Out (TX):
- RX: Receive Data Input is the serial data input. Oversampling techniques are used for data recovery by discriminating between valid incoming data and noise.
- TX: Transmit Data Output. When the transmitter is disabled, the output pin returns to its I/O port configuration. When the transmitter is enabled and

nothing is to be transmitted, the TX pin is at high level. In single-wire and smartcard modes, this I/O is used to transmit and receive the data (at USART level, data are then received on SW_RX.

- Through these pins, serial data is transmitted and received in normal USART mode as frames comprising:
- An Idle Line prior to transmission or reception A start bit.
- A data word (8 or 9 bits) least significant bit first .
- 0.5,1, 1.5, 2 Stop bits indicating that the frame is complete .
- This interface uses a fractional baud rate generator - with a 12-bit mantissa and 4-bit fraction .
- A status register (USART_SR) .
- Data Register (USART_DR) .
- A baud rate register (USART_BRR) - 12-bit mantissa and 4-bit fraction.
- A Guardtime Register (USART_GTPR) in case of Smartcard mode.

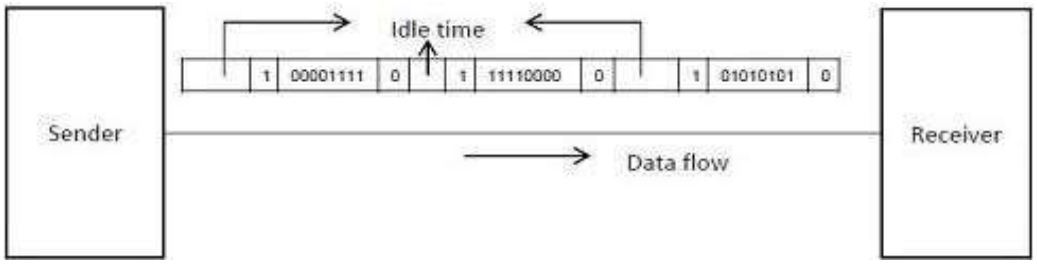
Figure 3.14 USART Function Description



3.5.3 USART Synchronous Mode

- The synchronous mode is selected by writing the CLKEN bit in the USART_CR2 register to 1. In synchronous mode, the following bits must be kept cleared.
- LINEN bit in the USART_CR2 register,
- SCEN, HDSEL and IREN bits in the USART_CR3 register.
- The USART allows the user to control a bidirectional synchronous serial communications in master mode. The SCLK pin is the output of the USART transmitter clock. No clock pulses are sent to the SCLK pin during start bit and stop bit. Depending on the state of the LBCL bit in the USART_CR2 register clock pulses will or will not be generated during the last valid data bit (address mark). The CPOL bit in the USART_CR2 register allows the user to select the clock polarity, and the CPHA bit in the USART_CR2 register allows the user to select the phase of the external clock.
- In this mode the USART receiver works in a different manner compared to the asynchronous mode. If RE=1, the data is sampled on SCLK (rising or falling edge, depending on CPOL and CPHA), without any oversampling. A setup and a hold time (that depends on the baud rate: 1/16 bit time) must be respected.

Figure 3.15 USRAT Synchronization



3.5.4 Single-wire Half-duplex Communication

- The single-wire half-duplex mode is selected by setting the HDSEL bit in the USART_CR3 register. In this mode, the following bits must be kept cleared.
- LINEN and CLKEN bits in the USART_CR2 register.

- SCEN and IREN bits in the USART_CR3 register.
- The USART can be configured to follow a single-wire half-duplex protocol where the TX and RX lines are internally connected. The selection between half- and full-duplex communication is made with a control bit ‘HALF DUPLEX SEL’ (HDSEL in USART_CR3).

3.5.5 Continuous Communication Using DMA

- The USART is capable of continuous communication using the DMA. The DMA requests for Rx buffer and Tx buffer are generated independently.
- DMA mode can be enabled for transmission by setting DMAT bit in the USART_CR3 register. Data is loaded from a SRAM area configured using the DMA peripheral (refer to the DMA specification) to the USART_DR register whenever the TXE bit is set. To map a DMA channel for USART transmission, use the following procedure (x denotes the channel number):

3.5.6 USART Interrupts

- The USART interrupt events are connected to the same interrupt vector.
- During transmission: Transmission Complete, Clear to Send or Transmit Data Register empty interrupt.
- During transmission: Transmission Complete, Clear to Send or Transmit Data Register empty interrupt.
- While receiving: Idle Line detection, Overrun error, Receive Data register not empty, Parity error, LIN break detection, Noise Flag (only in multi buffer communication) and Framing Error (only in multi buffer communication).

3.5.7 Status Register (USART_SR)

Address offset: 0x00
Reset value: 0x00C0 0000

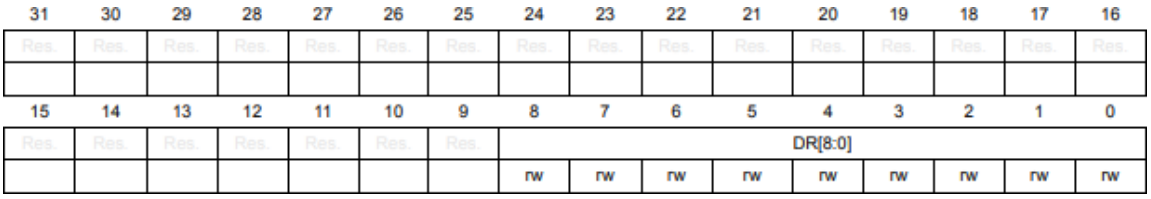
Figure 3.16 Status Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CTS	LBD	TXE	TC	RXNE	IDLE	ORE	NF	FE	PE
						rc_w0	rc_w0	r	rc_w0	rc_w0	r	r	r	r	r

3.5.8 Data Register (USART_DR)

Address offset: 0x04
Reset value: 0x0000 0000

Figure 3.17 Status Register

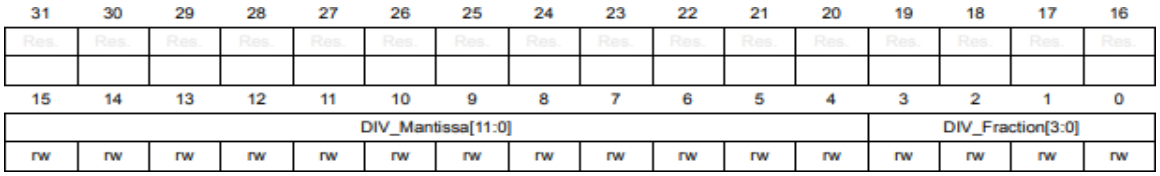


Data value:- Contains the Received or Transmitted data character, depending on whether it is read from or written to.
The Data register performs a double function (read and write) since it is composed of two registers, one for transmission (TDR) and one for reception (RDR)
The RDR register provides the parallel interface between the input shift register and the internal bus
When transmitting with the parity enabled (PCE bit set to 1 in the USART_CR1 register), the value written in the MSB (bit 7 or bit 8 depending on the data length) has no effect because it is replaced by the parity.

3.5.9 Baud Rate Register (USART_BRR)

Address offset: 0x08
Reset value: 0x0000 0000

Figure 3.18 Baud Register



3.5.10 Control Register 1 (USART_CR1)

Address offset: 0x0C
Reset value: 0x0000 0000

Figure 3.19 Control Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Res.	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

4 Interrupts And Events

4.1 Introduction

In computer science and embedded systems, an interrupt is a signal that a device or program sends to the operating system or another program indicating that an event has occurred. An interrupt can be generated by hardware, such as a mouse or keyboard, or software, such as a program that needs to communicate with the operating system. Interrupts allow multiple programs or devices to share the resources of a computer or embedded system and ensure that important events are handled immediately.

In this report, we will discuss the concept of interrupts and events, their types, and how they are used in computer systems and embedded systems.

4.1.1 Types Of Interrupts

There are two types of interrupts: hardware interrupts and software interrupts

1. Hardware Interrupts: Hardware interrupts are generated by hardware devices, such as a keyboard or a mouse. These interrupts are triggered by a change in the state of a device, such as a key being pressed or released. Hardware interrupts are asynchronous and occur independently of the processor's execution.
2. Software interrupts are a mechanism used by computer programs to request services from the operating system. They are typically triggered by software instructions, allowing programs to communicate with the underlying system and perform tasks that require privileged access or specialized services..

4.1.2 Types Of Events

Events are generated by software programs to indicate that a specific condition has occurred or that a specific task needs to be performed. There are two types of events: synchronous events and asynchronous events.

1. Synchronous Events: Synchronous events are generated by a program and are handled by the same thread of execution. These events are typically used to communicate between different parts of a program or to synchronize the execution of different tasks.
2. Asynchronous Events: Asynchronous events are generated by external sources, such as hardware devices or other programs. These events are typically handled by separate threads of execution and occur independently of the program's execution.

4.1.3 Uses of Interrupts and Events

Interrupts and events are used in a variety of computer systems and embedded systems to ensure that important tasks are performed immediately and that resources are shared efficiently.

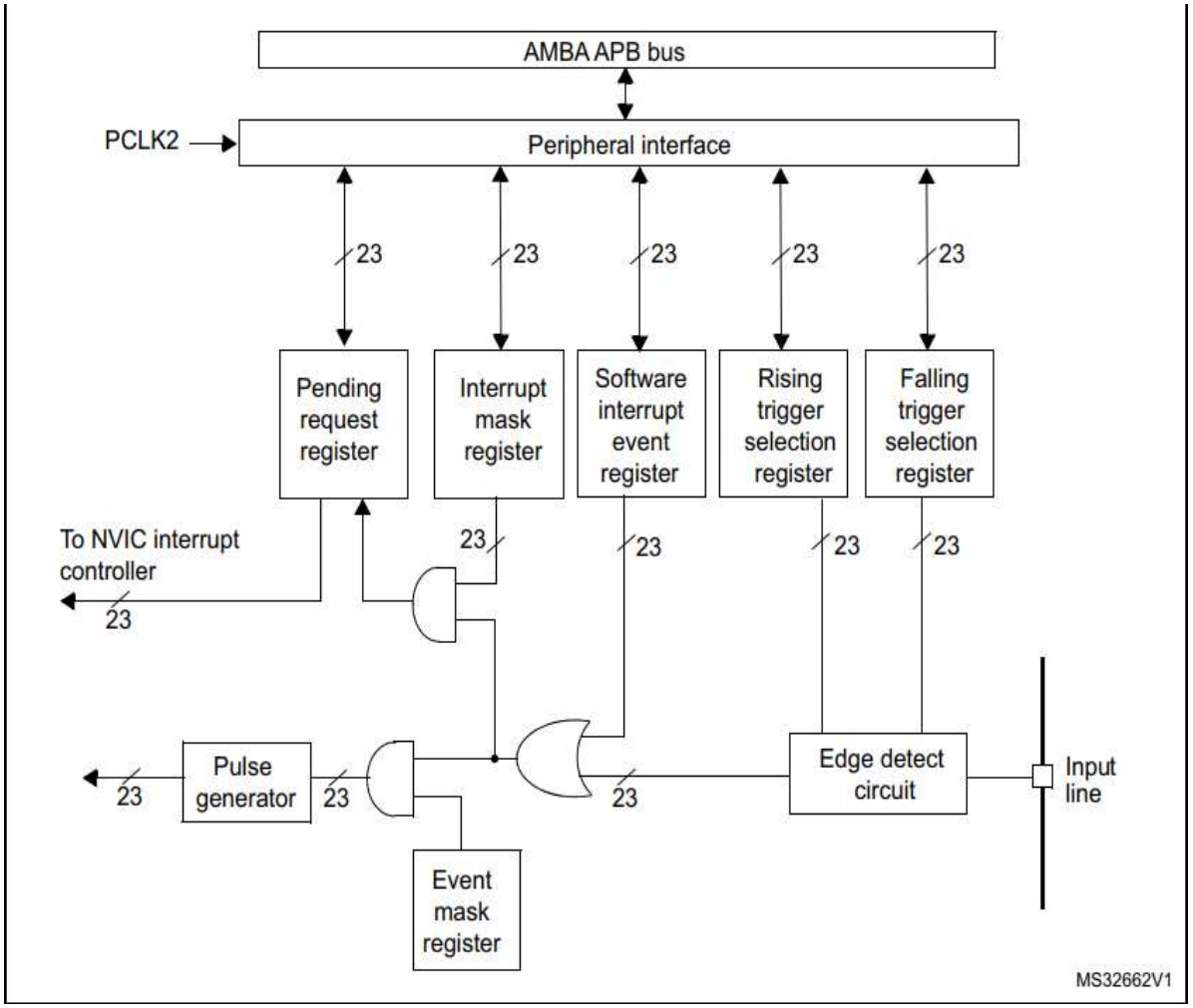
In computer systems, interrupts are used to handle user input from devices such as keyboards and mice. Interrupts are also used to communicate with hardware devices such as disk drives and network adapters. Software interrupts are used to request services from the operating system or to communicate with other software programs.

In embedded systems, interrupts are used to handle real-time events such as sensor input and motor control. Interrupts are also used to communicate with external devices such as displays and communication modules. Events are used to signal the completion of tasks and to synchronize the execution of different parts of the system.

4.1.4 EXTI block diagram

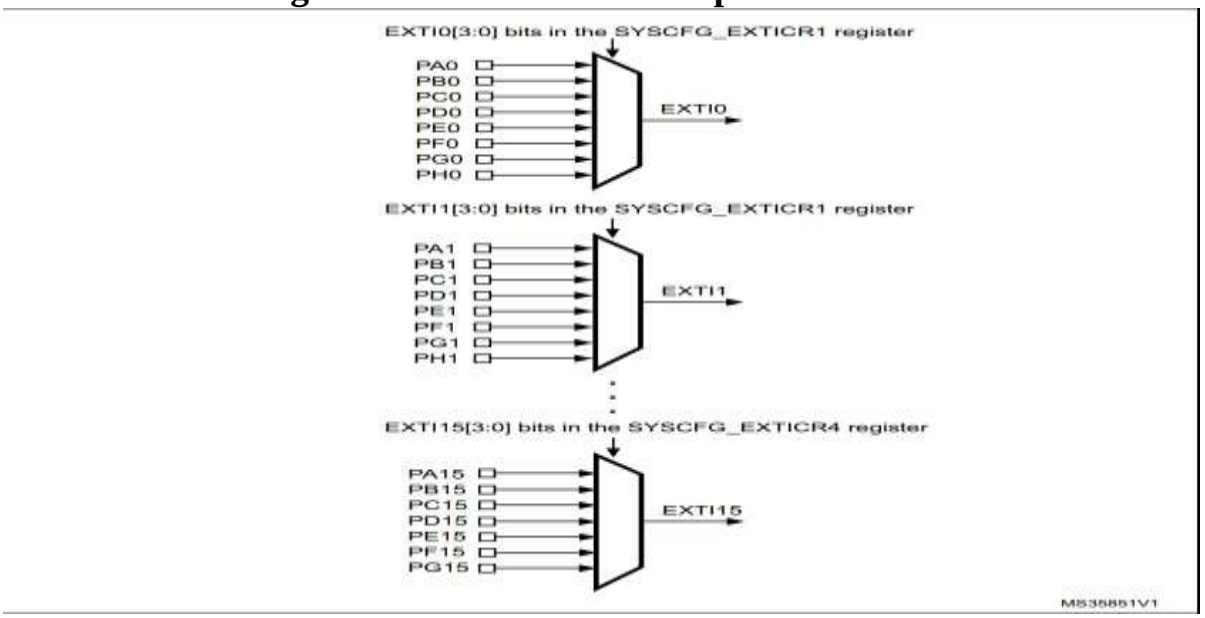
External Interrupt/Event Controller Block Diagram

Figure 4.1 External Interrupt/Event Controller Block Diagram



4.1.5 External Interrupt/Event GPIO Mapping

Figure 4.2 External Interrupt/Event GPIO



4.1.6 EXTI Registers

EXTI (External Interrupt) registers are a set of registers that are used to configure and control the external interrupt lines of a microcontroller. These registers are specific to the ARM Cortex-M family of microcontrollers and are used to enable or disable external interrupts, configure their triggering modes, and determine which interrupt line caused the interrupt.

The EXTI registers are located in the peripheral register space of the microcontroller and are accessed through memory-mapped I/O. There are a total of six EXTI registers in the Cortex-M family of microcontrollers, each of which is used for a specific purpose. These registers are:

1. EXTI_IMR (Interrupt Mask Register): This register is used to enable or disable the external interrupt lines. Each bit in this register corresponds to an interrupt line and can be set or cleared to enable or disable that interrupt line.

Figure 4.3 EXTI_IMR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	MR22	MR21	MR20	MR19	MR18	MR17	MR16
									rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

2. EXTI_RTSR (Rising Edge Trigger Selection Register): This register is used to configure the external interrupt lines to trigger on a rising edge. Each bit in this register corresponds to an interrupt line and can be set or cleared to enable or disable the rising edge trigger for that interrupt line.

Figure 4.4 EXTI_RTSR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	TR20	Res.	TR18	TR17	TR16
									rw	rw	rw		rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

3. EXTI_FTSR (Falling Edge Trigger Selection Register): This register is used to configure the external interrupt lines to trigger on a falling edge. Each bit in this register corresponds to an interrupt line and can be set or cleared to enable or disable the falling edge trigger for that interrupt line.

Figure 4.5 EXTI_FTSR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	TR22	TR21	TR20	Res.	TR18	TR17	TR16
									rw	rw	rw		rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

4. EXTI_SWIER (Software Interrupt Event Register): This register is used to generate a software interrupt event on the external

interrupt lines. Each bit in this register corresponds to an interrupt line and can be set or cleared to generate a software interrupt event for that interrupt line.

Figure 4.6 EXTI_SWIER

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	SWIER 22	SWIER 21	SWIER 20	SWIER 19	SWIER 18	SWIER 17	SWIER 16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWIER 15	SWIER 14	SWIER 13	SWIER 12	SWIER 11	SWIER 10	SWIER 9	SWIER 8	SWIER 7	SWIER 6	SWIER 5	SWIER 4	SWIER 3	SWIER 2	SWIER 1	SWIER 0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

5. EXTI_PR (Pending Register): This register is used to determine which interrupt line caused the interrupt. Each bit in this register corresponds to an interrupt line and is set when that interrupt line is triggered. This register is cleared by writing a 1 to the corresponding bit.

Figure 4.7 EXIT_PR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	PR22	PR21	PR20	PR19	PR18	PR17	PR16
									rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1	rc_w1

6. EXTI_EMR (Event Mask Register): This register is used to enable or disable event detection on the external interrupt lines. Each bit in this register corresponds to an interrupt line and can be set or cleared to enable or disable event detection for that interrupt line.

Figure 4.8 EXTI_EMR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	MR22	MR21	MR20	MR19	MR18	MR17	MR16
									rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR5	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

5.Timer And Counters

5.1 Introduction

Timers and counters are essential components of microcontrollers used in a wide variety of applications. They are used to measure time intervals, generate precise time delays, and count external events. In this report, we will discuss the basics of timers and counters, their types, and their applications in microcontrollers.

5.1.1 What Are Timers And Counters

Timers and counters are hardware components that are used to measure time intervals and count external events. They are based on a clock source and can generate precise time delays. They are widely used in microcontroller-based applications, including embedded systems, industrial automation, and consumer electronics.

5.1.2 Timers

Timers are used to measure time intervals and generate precise time delays. They are based on a clock source and consist of a counter and a prescaler. The counter is used to count the clock pulses, while the prescaler divides the clock frequency by a specified factor.

There are two types of timers: free-running timers and periodic timers. Free-running timers continuously count the clock pulses until they reach their maximum value, while periodic timers generate an interrupt or an output signal when the timer value reaches a specified value.

5.1.3 Counters

Counters are used to count external events such as pulses, edges, or transitions. They are based on a clock source and consist of a counter and a prescaler. The counter is used to count the external events, while the prescaler divides the clock frequency by a specified factor.

There are two types of counters: up counters and down counters. Up counters count up from zero to a maximum value, while down counters count down from a maximum value to zero.

5.1.4 Applications Of Timers And Counters

Timers and counters are widely used in microcontroller-based applications, including embedded systems, industrial automation, and consumer electronics. Some of the applications of timers and counters are:

1. Pulse-width modulation (PWM): Timers are used to generate a periodic waveform with a variable duty cycle, which is used in motor control, lighting control, and other applications.
2. Real-time clock (RTC): Timers are used to keep track of time in applications such as digital watches, alarm clocks, and other time-sensitive devices.
3. Frequency measurement: Counters are used to measure the frequency of external signals such as oscillators, sensors, and other devices.
4. Event counting: Counters are used to count the number of external events such as pulses, edges, or transitions.
5. Timing and synchronization: Timers and counters are used to synchronize multiple devices or events and to measure time intervals between events.

6. Real-Time Clock

6.1 Introduction

Real-Time Clock (RTC) is an essential component of the STM32 microcontroller that is used to keep track of time and date. In this report, we will discuss the STM32 RTC module, its features, and its applications.

6.1.1 STM32 RTC Module

The STM32 RTC module is a low-power, high-precision clock that is used to keep track of time and date. It is based on a 32-bit counter and a prescaler that divides the clock frequency by a specified factor. The RTC module also includes a set of registers that are used to configure and control the RTC functions.

Figure 6.1 STM32 RTC Module

6.1.2 Features Of STM32 RTC Module

The STM32 RTC module has several features that make it an ideal choice for a wide range of applications. Some of the features of the STM32 RTC module are:

1. High precision: The STM32 RTC module provides high precision clock that can be accurate up to a few microseconds.

2. Low power: The STM32 RTC module operates on a low power mode

which makes it ideal for battery-powered applications.

3. Alarm function: The RTC module includes an alarm function that can be used to generate an interrupt or an output signal at a specific time or date.

4. Calibration: The RTC module includes a calibration function that can be used to adjust the clock frequency to compensate for temperature variations.

5. Time and date functions: The RTC module includes functions for setting and reading the time and date.

6.1.3 Applications Of STM32 RTC Module

The STM32 RTC module is used in a wide range of applications, including:

1. Battery-powered devices: The low power consumption of the RTC module makes it ideal for use in battery-powered devices, such as handheld devices, sensors, and other mobile applications.

2. Timing and scheduling: The RTC module is used in applications that require accurate timing and scheduling, such as industrial automation, power management, and smart buildings.

3. Data logging: The RTC module is used in data logging applications to timestamp data and keep track of when data was acquired.

4. Security: The RTC module is used in security applications to provide a tamper-proof timestamp for events, such as access control, authentication, and encryption.

7 ADC

7.1 Introduction

The STM32 microcontroller series is a popular choice for embedded systems applications that require analog-to-digital conversion (ADC) capabilities. In this report, we will discuss the ADC module of the STM32 microcontroller, its features, and its applications.

7.1.1 STM32 ADC Module

The STM32 ADC module is a low-power, high-precision ADC that is used to convert analog signals into digital signals. It is based on a successive approximation register (SAR) architecture and includes a set of registers that are used to configure and control the ADC functions.

7.1.2 Features Of STM32 ADC Module

The STM32 ADC module has several features that make it an ideal choice for a wide range of applications. Some of the features of the STM32 ADC module are:

1. High resolution: The STM32 ADC module provides high resolution up to 12 or 16 bits, depending on the specific STM32 microcontroller model.

2. Sampling rates: The STM32 ADC module can operate at high sampling rates of up to several MSPS (Mega samples per second) for faster data acquisition.

3. Multiple channels: The STM32 ADC module includes multiple channels that can be configured to sample different input signals.

4. DMA support: The STM32 ADC module includes DMA (Direct Memory Access) support, which enables the ADC to transfer data directly to memory without CPU intervention.

5. Conversion trigger sources: The STM32 ADC module includes several conversion trigger sources, such as software trigger, timer trigger, and external trigger.

7.1.3 Applications Of STM32 ADC Module

The STM32 ADC module is used in a wide range of applications, including:

1. Industrial automation: The STM32 ADC module is used in industrial automation applications to measure and control process variables, such as temperature, pressure, and flow rate.

2. Medical devices: The STM32 ADC module is used in medical devices, such as blood glucose meters, ECG machines, and pulse oximeters, to measure vital signs and other physiological parameters.

3. Audio processing: The STM32 ADC module is used in audio processing applications, such as audio recording and playback, and voice recognition.

4. Power management: The STM32 ADC module is used in power management applications to monitor and control the power consumption of electronic devices.

5. Automotive: The STM32 ADC module is used in automotive applications, such as engine control and fuel injection systems.

I2C (Inter-Integrated Circuit) is a popular communication protocol used in microcontrollers, including the STM32 microcontroller series. In this report, we will discuss the basics of the I2C protocol, its features, and its applications in STM32 microcontrollers.

8 PWM

8.1 Introduction

- Definition and Purpose of PWM (Pulse Width Modulation)
- Importance and Applications in Embedded Systems

8.1.1 PWM Basics

- Concept and Operation of PWM
- Duty Cycle and Frequency
- PWM Resolution

8.1.2 Hardware Implementation

- Microcontrollers and PWM Capabilities
- Timer/Counter Modules
- Output Pins and Signal Generation

8.1.3 Software Implementation

- Programming Languages and Development Tools
- PWM Libraries and APIs
- Configuration and Initialization

8.1.4 PWM Parameters

- Duty Cycle Control
- Frequency Selection

9 I2C PROTOCOL:

9.1 Introduction

The I2C protocol is a serial communication protocol that is used to transfer data between devices. It uses two wires, SDA (Serial Data) and SCL (Serial Clock), to transfer data between devices. The I2C protocol supports multiple devices on the same bus and enables devices to communicate with each other using a simple master-slave architecture.

9.1.1 Features Of I2C Protocol

The I2C protocol has several features that make it an ideal choice for a wide range of applications. Some of the features of the I2C protocol are:

1. Two-wire interface: The I2C protocol uses only two wires, SDA and SCL, to transfer data between devices, which reduces the number of wires required for communication.

2. Multi-master support: The I2C protocol supports multiple masters on the same bus, which enables multiple devices to communicate with each other.

3. Clock synchronization: The I2C protocol uses a clock signal to synchronize data transfer between devices, which ensures accurate data transfer.

4. Addressing: The I2C protocol uses 7-bit or 10-bit addressing to identify each device on the bus.

5. Error detection: The I2C protocol includes error detection mechanisms, such as ACK (acknowledge) and NACK (not acknowledge), to ensure reliable data transfer.

9.1.2 Applications Of I2C Protocol

The I2C protocol is used in a wide range of applications, including:
1. Sensor interface: The I2C protocol is used to interface with sensors, such as temperature sensors, pressure sensors, and humidity sensors, to transfer data to the microcontroller.

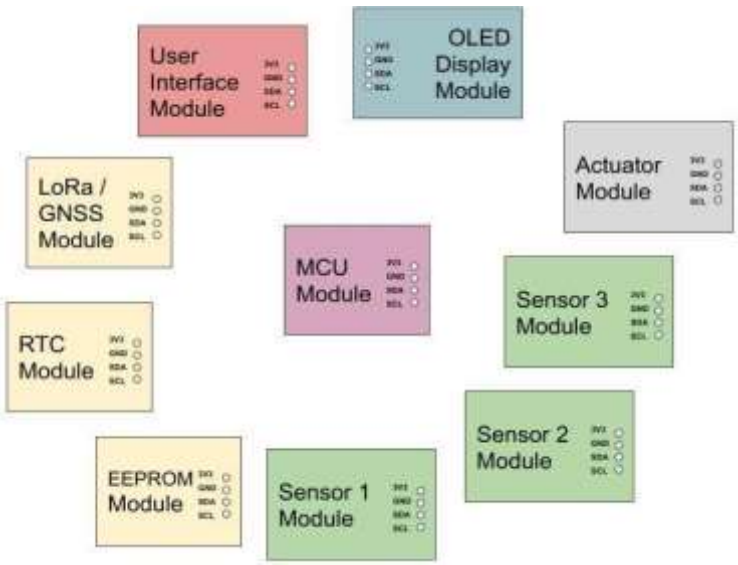
2. EEPROM interface: The I2C protocol is used to interface with EEPROM (Electrically Erasable Programmable Read-Only Memory) to store and retrieve data.

3. Real-time clock: The I2C protocol is used to interface with real-time clock (RTC) modules to keep track of time.

4. Display interface: The I2C protocol is used to interface with LCD (Liquid Crystal Display) and OLED (Organic Light Emitting Diode) displays to display data.

5. Audio interface: The I2C protocol is used to interface with audio codecs to transfer audio data.

Figure 9.1 Applications Of I2C Protocl



9.1.3 I2C On STM32

The STM32 microcontroller series supports the I2C protocol, and the I2C interface is implemented using dedicated hardware modules. The I2C hardware module in STM32 supports standard (100 Kbps) and fast (400 Kbps) modes of operation.

The I2C interface in STM32 supports both master and slave modes of operation, and the microcontroller can act as either a master or a slave device. In master mode, the microcontroller initiates the data transfer by sending the start condition and generates the clock signal. In slave mode, the microcontroller responds to the start condition and clock signal generated by the master device.

The STM32 microcontroller series also supports various advanced features of the I2C protocol, such as SMBus (System Management Bus) and PEC (Packet Error Checking).
Serial peripheral interface

10. SPI Protocol

10.1 Introduction

The SPI/I²S interface can be used to communicate with external devices using the SPI protocol or the I2S audio protocol. SPI or I2S mode is selectable by software. SPI mode is selected by default after a device reset.

The serial peripheral interface (SPI) protocol supports half-duplex, full-duplex and simplex synchronous, serial communication with external devices. The interface can be configured as master and in this case it provides the communication clock (SCK) to the external slave device. The interface is also capable of operating in multimaster configuration.

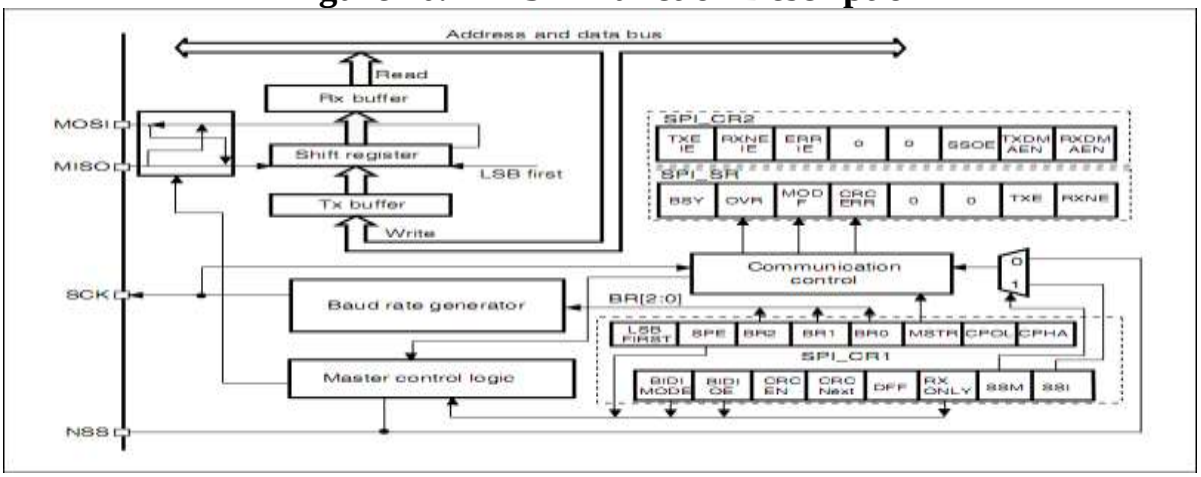
10.1.1 SPI Main Features

Master or slave operation Full-duplex synchronous transfers on three lines
Half-duplex synchronous transfer on two lines (with bidirectional data line)
Simplex synchronous transfers on two lines (with unidirectional data line)
8-bit or 16-bit transfer frame format selection
Multimeter mode capability
NSS management by hardware or software for both master and slave: dynamic change of master/slave operations.

10.1.2 SPI Functional Description

The SPI allows synchronous, serial communication between the MCU and external devices. Application software can manage the communication by polling the status flag or using dedicated SPI interrupt. The main elements of SPI and their interactions are shown in the following block diagram.

Figure 10.1 SPI Function Description



Four I/O pins are dedicated to SPI communication with external devices.

MISO: Master In / Slave Out data. In the general case, this pin is used to transmit data in slave mode and receive data in master mode.

MOSI: Master Out / Slave In data. In the general case, this pin is used to transmit data in master mode and receive data in slave mode.

SCK: Serial Clock output pin for SPI masters and input pin for SPI slaves.

NSS: Slave select pin. Depending on the SPI and NSS settings, this pin can be used to either:

select an individual slave device for communication

– synchronize the data frame or

– detect a conflict between multiple masters

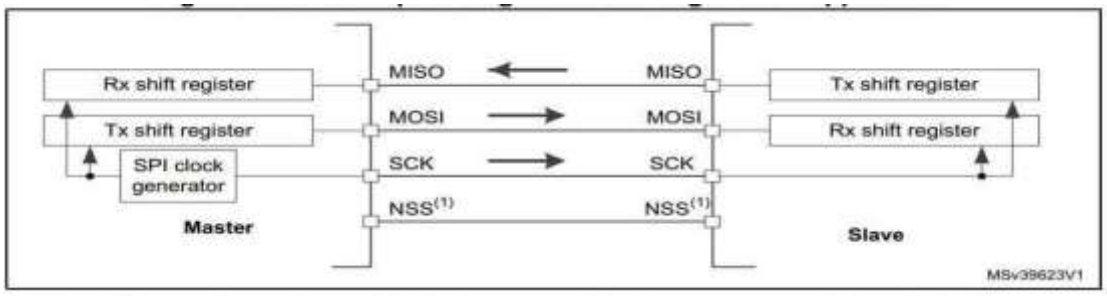
Communications between one master and one slave

The SPI allows the MCU to communicate using different configurations, depending on the device targeted and the application requirements. These configurations use 2 or 3 wires (with software NSS management) or 3 or 4 wires (with hardware NSS management). Communication is always initiated by the master

10.1.3 Full-duplex Communication

By default, the SPI is configured for full-duplex communication. In this configuration, the shift registers of the master and slave are linked using two unidirectional lines between the MOSI and the MISO pins. During SPI communication, data is shifted synchronously on the SCK clock edges provided by the master. The master transmits the data to be sent to the slave via the MOSI line and receives data from the slave via the MISO line. When the data frame transfer is complete (all the bits are shifted) the information between the master and slave is exchanged.

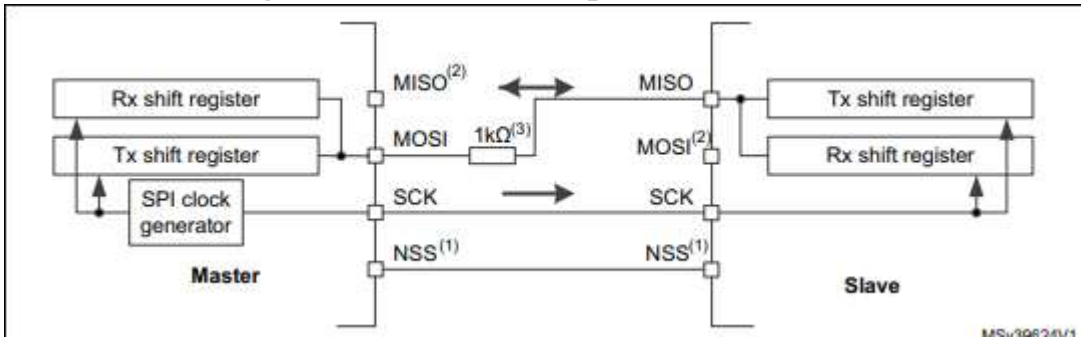
Figure 10.2 Full-duplex Communication



10.1.4 Half-duplex Communication

The SPI can communicate in half-duplex mode by setting the BIDIMODE bit in the SPIx_CR1 register. In this configuration, one single cross connection line is used to link the shift registers of the master and slave together. During this communication, the data is synchronously shifted between the shift registers on the SCK clock edge in the transfer direction selected reciprocally by both master and slave with the BDIOE bit in their SPIx_CR1 registers. In this configuration, the master's MISO pin and the slave's MOSI pin are free for other application uses and act as GPIOs.

Figure 10.3 Half-duplex Communication



10.1.5 Simplest Single Master Slave Application

The SPI can communicate in simplex mode by setting the SPI in transmit-only or in receiveonly using the RXONLY bit in the SPIx_CR2 register. In this configuration, only one line is used for the transfer between the shift registers of the master and slave. The remaining MISO and MOSI pins pair is not used for communication and can be used as standard GPIOs.

Transmit-only mode (RXONLY=0): The configuration settings are the same as for full duplex. The application has to ignore the information captured on the unused input pin. This pin can be used as a standard GPIO.

Receive-only mode (RXONLY=1): The application can disable the SPI output function by setting the RXONLY bit. In slave configuration, the MISO output is disabled and the pin can be used as a GPIO.

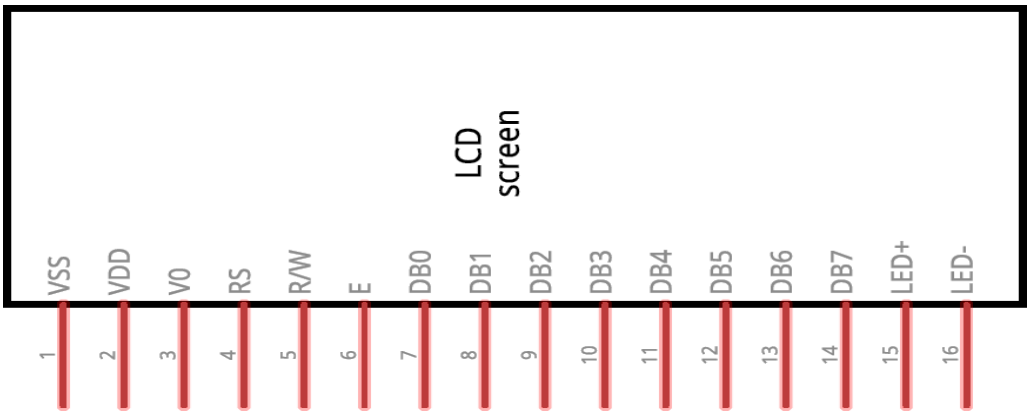
11 Device Interfacing

11.1 LCD

LCD interface is a link between the flat panel display module and the multimedia processor. Therefore, the interface can be separated or incorporated as part of the structure on the chip. Additionally, the application produces an image, and then the screen displays it using an LCD interface for the user.

11.1.1 LCD Pinout Diagram

Figure 11.1 LCD Pinout Diagram



11.1.2 Code

```
#include<stdio.h>
#include<string.h>
void print(unsigned char);
void lcd_data(unsigned char);
void lcd_cmd(unsigned char);
void lcd_display(unsigned char *,unsigned int);
void print(unsigned char dat)
{
        if((dat & 0x01) == 0x01)
        {
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, 1);
        }
        else
        {
                HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, 0);
        }
        if((dat & 0x02) == 0x02)
        {
                HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, 1);
        }
```

```
	else
	{
		HAL_GPIO_WritePin(GPIOB, GPIO_PIN_3, 0);
	}

	if((dat & 0x04) == 0x04)
		{
			HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, 1);
		}
		else
		{
			HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, 0);
		}

	if((dat & 0x08) == 0x08)
		{
			HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, 1);
		}
		else
		{
			HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, 0);
		}

	if((dat & 0x10) == 0x10)
		{
			HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, 1);
		}
		else
		{
			HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, 0);
		}

	if((dat & 0x20) == 0x20)
		{
			HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, 1);
		}
		else
		{
			HAL_GPIO_WritePin(GPIOA, GPIO_PIN_8, 0);
		}

	if((dat & 0x40) == 0x40)
		{
			HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9, 1);
		}
		else
		{
			HAL_GPIO_WritePin(GPIOA, GPIO_PIN_9, 0);
		}

	if((dat & 0x80) == 0x80)
		{
			HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, 1);
		}
		else
		{
			HAL_GPIO_WritePin(GPIOC, GPIO_PIN_7, 0);
		}
}

void lcd_cmd(unsigned char cmd)
{
	print(cmd);
	HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, 0);
```

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7, 0);
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, 1);
HAL_Delay(2);
```

```
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, 0);
}
void lcd_data(unsigned char mydat)
{
   print(mydat);
   HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, 1);
   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_7, 0);
   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, 1);
   HAL_Delay(2);
   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, 0);
}

void lcd_init()
{
        lcd_cmd(0x38);
        lcd_cmd(0x06);
        lcd_cmd(0x01);
        lcd_cmd(0x0c);
}
void lcd_display(unsigned char * str,unsigned int length)
{
  for(int i=0;i<=length;i++)
  {
          lcd_data(str[i]);
  }
}
```

11.1.3 Conclusion

The liquid crystal display (LCD) is a viable option for displaying the parameters used in an Arduino program or a project. The liquid crystal display comes in multiple sizes but mostly the 16×2 size is preferred as it is easy to use and easy to interface it with Arduino. In this write-up the 16×2 LCD is explained briefly which will make it easy for the reader to interface it with any device**.**

Training Schedule

Training date	Time	Trainer name
04-02-2023-	9.00 Am – 7.00 PM	P Gopakrishna

This is 6 Month training duration we learn embedded course. This program helped us in narrowing the gap between our college and professional life. It helped us in clearing all the concepts of embedded programming.

Evaluation schedule

- Here a daily exam is conducted, and it is mandatory to give exam.
- In every Saturday we must give ppt presentation.