EMBEDDED SYSTEM
(15A04702)
LECTURE NOTES

B.TECH

IV YEAR – I SEM

REGULATION R15

Prepared by:

Dr. G. Elayaraja, Professor

Department of Electronics and Communication Engineering

VEMU INSTITUTE OF TECHNOLOGY

(Approved by AICTE, New Delhi and affiliated to JNTUA, Ananthapuramu)

NEAR PAKALA, P.KOTHAKOTA, CHITTOOR- TIRUPATHI HIGHWAY

CHITTOOR-517512 Andhra Pradesh

WEB SITE: www.Vemu.Org

AWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR

B. Tech IV-I Sem. (ECE) **L T P C**

3 1 0 3

15A04702 EMBEDDED SYSTEMS

Course Outcomes:

C412_1: Discuss the embedded system architecture and overview of design process

C412_2: Explain the architecture of ARM and TM4C microcontroller

C412_3: Explain concept for interfacing of processor, memory and I/O Devices and design metrics

C412_4: Explain the various modules of TM4C microcontroller

C412_5: Design the embedded networking and IOT applications

UNIT-I: Introduction to Embedded Systems

Embedded system introduction, host and target concept, embedded applications, features and architecture considerations for embedded systems- ROM, RAM, timers; data and address bus concept, Embedded Processor and their types, Memory types, overview of design process of embedded systems, programming languages and tools for embedded design

UNIT-II: Embedded processor architecture

CISC Vs RISC design philosophy, Von-Neumann Vs Harvard architecture. Introduction to ARM architecture and Cortex – M series, Introduction to the TM4C family viz. TM4C123x & TM4C129x and its targeted applications. TM4C block diagram, address space, on-chip peripherals (analog and digital) Register sets, Addressing modes and instruction set basics.

UNIT- III: Overview of Microcontroller and Embedded Systems

Embedded hardware and various building blocks, Processor Selection for an Embedded System , Interfacing Processor, Memories and I/O Devices, I/O Devices and I/O interfacing concepts, Timer and Counting Devices, Serial Communication and Advanced I/O, Buses between the Networked Multiple Devices. Embedded System Design and Co-design Issues in System Development Process, Design Cycle in the Development Phase for an Embedded System, Uses of Target System or its Emulator and In-Circuit Emulator (ICE), Use of Software Tools for Development of an Embedded System Design metrics of embedded systems - low power, high performance, engineering cost, time-to-market.

UNIT-IV: Microcontroller fundamentals for basic programming

I/O pin multiplexing, pull up/down registers, GPIO control, Memory Mapped Peripherals, programming System registers, Watchdog Timer, need of low power for embedded systems, System Clocks and control, Hibernation Module on TM4C, Active vs Standby current consumption. Introduction to Interrupts, Interrupt vector table, interrupt programming. Basic Timer, Real Time Clock (RTC), Motion Control Peripherals: PWM Module & Quadrature Encoder Interface (QEI).

UNIT-V: Embedded communications protocols and Internet of things

Synchronous/Asynchronous interfaces (like UART, SPI, I2C, USB), serial communication basics, baud rate concepts, Interfacing digital and analog external device, Implementing and programming UART, SPI and I2C, SPI interface using TM4C.Case Study: Tiva based embedded system application using the interface protocols for communication with external devices “Sensor Hub BoosterPack” Embedded Networking fundamentals, IoT overview and architecture, Overview of wireless sensor networks and design examples. Adding Wi-Fi capability to the Microcontroller, Embedded Wi-Fi, User APIs for Wireless and Networking applications Building IoT applications using CC3100 user API. Case Study: Tiva based Embedded Networking Application: “Smart Plug with Remote Disconnect and Wi-Fi Connectivity”

Text Books:

1. Embedded Systems: Real-Time Interfacing to ARM Cortex-M Microcontrollers, 2014, Create space publications ISBN: 978-1463590154.

2. Embedded Systems: Introduction to ARM Cortex - M Microcontrollers, 5th edition Jonathan W Valvano, Createspace publications ISBN-13: 978-1477508992

3. Embedded Systems 2E Raj Kamal, Tata McGraw-Hill Education, 2011 ISBN- 4. 0070667640, 9780070667648

References:

1. http://processors.wiki.ti.com/index.php/HandsOn_Training_for_TI_Embedded _Processors
2. http://processors.wiki.ti.com/index.php/MCU_Day_Internet_of_Things_2013_ Workshop
3. http://www.ti.com/ww/en/simplelink_embedded_wi-fi/home.html
4. CC3100/CC3200 SimpleLink™ Wi-Fi® Internet-on-a-Chip User Guide Texas Instruments Literature Number: SWRU368A April 2014–Revised August 2015.

UNIT-I

Introduction to Embedded Systems

Embedded System Introduction:

We live in an era where pervasive (spreading) computing exists everywhere, right from a small handheld device such as a mobile phone to the electronic control units within automobiles or avionics. Today, large volumes of information is getting processed and communicated over the Internet every microsecond. Buzz words such as Cloud Computing, Big Data Mining and Internet of Things are everywhere.

There are two broad classifications of computing systems - general purpose computing system and embedded computing systems. If we define these in simple words, general purpose computing systems are those used in desktop or laptop computers, which can process several different applications. An embedded system refers to any device that has some computational intelligence in it. It is generally used as a standalone system that repeatedly performs a specific task or as part of a large system to perform multiple tasks with the required hardware and software embedded within. Systems used in printers, washing machines, mp3 players; CT scan machines etc. are great examples of embedded systems.

An embedded system is a constrained system and its design goals vary from a general purpose system. The constraints are: high performance, low power consumption, small size and low cost of the system.

The basic components of an embedded system include hardware, software and some mechanical parts. Embedded hardware includes a processing unit, block of memory and I/O sub-unit which are called as the system resources. The embedded software can be thought of as the application software in a small computing system or both the system and the application software in case of a large complex system. The system software mentioned here is the real time operating system (RTOS) used to manage the usage of system resources by application software.

What is an Embedded System?

- An embedded system is a system that has software embedded into computer-hardware, which makes a system dedicated for an application or specific part of an application or product or part of a larger system.

- An embedded system is designed to do a specific/few task only.

- Examples:
  - A Washing machine can only wash clothes.
  - A Digital camera can only capture the images.
  - An Air conditioner can control the temperature in the room.

Host and Target concept:

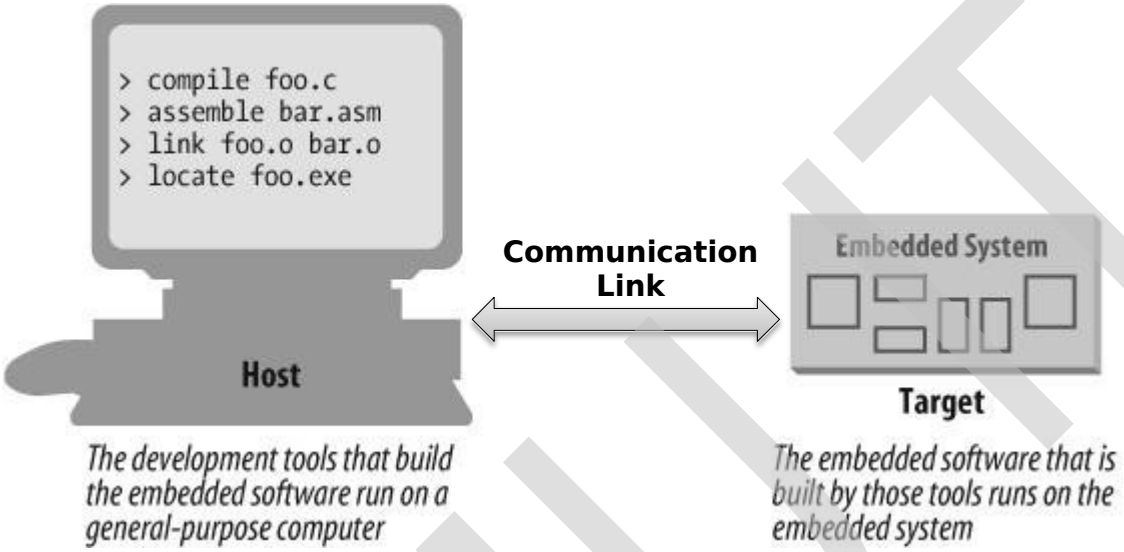


Figure: Embedded System using Host and Target Machine

Performance of Host machine:

The application program developed runs on the host computer. The host computer is also called as Development Platform. It is a general purpose computer. It has a higher capability processor and more memory. It has different input and output devices. The compiler, assembler, linker, and locator run on a host computer rather than on the embedded system itself. These tools are extremely popular with embedded software developers because they are freely available (even the source code is free) and support many of the most popular embedded processors. It contains many development tools to create the output binary image. Once a program has been written, compiled, assembled and linked, it is moved to the target platform.

Performance of Target machine:

The output binary image is executed on the target hardware platform. It consists of two entities - the target hardware (processor) and runtime environment (OS). It is needed only for final output. It is different from the development platform and it does not contain any development tools.

Embedded Applications:

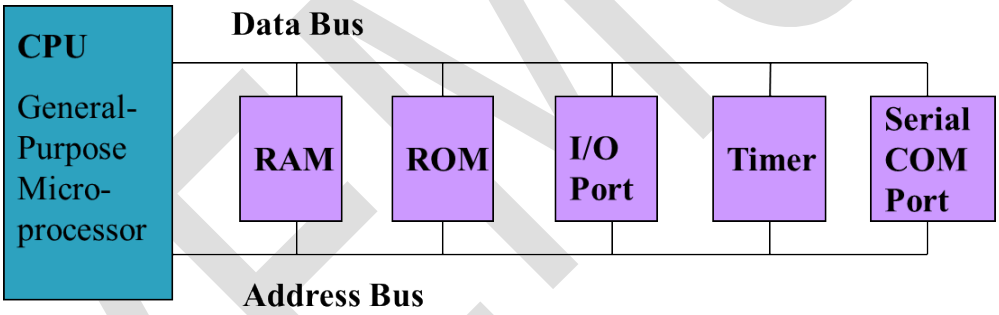
Where Embedded Systems?

- Avionics: Navigation systems, Inertial Guidance Systems, GPS Receivers etc

- Automotives: Automatic Breaking System, Air Bag System, Ignition Systems etc.

- Consumer Electronics: MP3 Players, Mobile Phones, Video Game Consoles, Digital Cameras, Set top Boxes, Camcorders etc.
- Telecommunication: Cell Phones, Telephone Switches etc.
- Medical Systems: ECG, EEG, MRI, CT scan, BP Monitors, etc.
- Military Applications
- Industries
- Home Appliances: AC, TV, DVD Players, Washing machine, Refrigerators, Microwave Oven, etc.
- Computer Peripherals: Printers, Scanners, Fax Machines.
- Computer Networking Systems: Network Routers, Switches, Hubs etc.
- Security Systems: Intruder Detection Alarms, CC Cameras, Fire Alarms
- Measurement & Instrumentation: Digital Multimeters, Digital CRO‘s, Logic Analyzers, PLC Systems.
- Banking and Retail: ATM, Currency Counters.
- Card readers: Barcode, Smart Card Readers, Hand Held Devices.
- And Many More….

Features and Architecture Considerations for Embedded Systems:



CPU/Processor: It act as a brain to the system, it is used to execute the program and to sending and receiving signals. The processing unit could be a microprocessor, a microcontroller, embedded processor, DSP, ASIC or FPGA selected for an embedded system based on the application requirements.

ROM for the program: Nonvolatile (read-only memory, ROM); meaning that it retains its contents even when power is off. It also called Program memory. In embedded systems, the application program after being compiled is saved in the ROM. The processing unit accesses the ROM to fetch instructions sequentially and executes them within the CPU. There are different categories of ROM such as: programmable read only memory (PROM), erasable programmable read only memory (EPROM), electrically erasable programmable read only memory (EEPROM) etc. There is also flash memory which is the updated version of EEPROM and extensively used in embedded systems.

RAM for data: Random access memory (RAM) is volatile i.e. it does not retain the contents after the power goes off. It is used as the data memory in an embedded system. It holds the variables declared in the program, the stack and intermediate data or results during program run time. The Processing unit accesses the RAM for instruction execution to save or retrieve data. There are different variations of RAM such as: static RAM (SRAM), dynamic RAM (DRAM), pseudo static RAM (PSRAM), non-volatile RAM (NVRAM), synchronous DRAM, (SDRAM) etc.

I/O PORTS: To provide digital communication with the outside world. These Ports are two types Parallel Port and Serial Port, Parallel are used to communicate with Parallel I/O devices like LEDs, LCDs, 7-Segment Displays, Keypads etc., Serial Port is used to communicate with Serial devices like Bluetooth module, Wi-Fi, Zigbee, GSM Modules, PCs etc.

Address and Data buses: To link these subsystems to transfer data and instructions. The system bus consists of three different bus systems: address bus, data bus and control bus. Processor sends the address of the destination through the address bus. So address bus is unidirectional from processor to the external end. Data can be sent or received from any unit to any other unit in the diagram. So data bus is bidirectional. Control bus is basically a group of control signals from the processing unit to the external units

Timers: Timers are a fundamental concept in embedded systems and they have many use cases such as creating accurate delays, executing a periodic task, implementing a PWM (Pulse With Modulation) output or capturing the elapsed time between two events, to vary the

speed of data transfer rate i.e. Baud rate (bps-bits per second) in case serial communication etc.

Clock: To keep the whole system synchronized. It may be generated internally or obtained from a crystal or external source; modern MCUs offer considerable choice of clocks.

Watchdog timer: This is a safety feature, which resets the processor if the program becomes stuck in an infinite loop/hang.

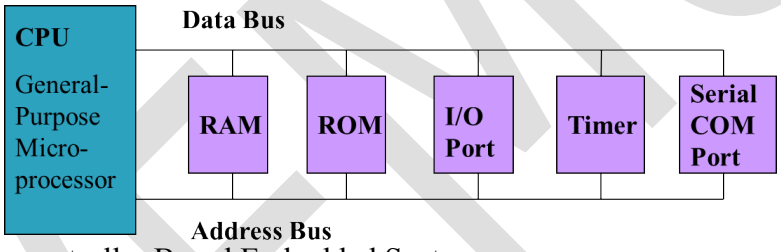
Communication interfaces: A wide choice of interfaces is available to exchange information with another IC or System. They include Universal Asynchronous Receiver/Transmitter (UART), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I²C or IIC), Universal Serial Bus (USB), Controller Area Network (CAN), Ethernet, and many others.

Non-volatile Memory for data: This is used to store data whose value must be retained even when power is removed. Serial numbers for identification and network addresses are two obvious candidates.

Microprocessor vs. Microcontroller

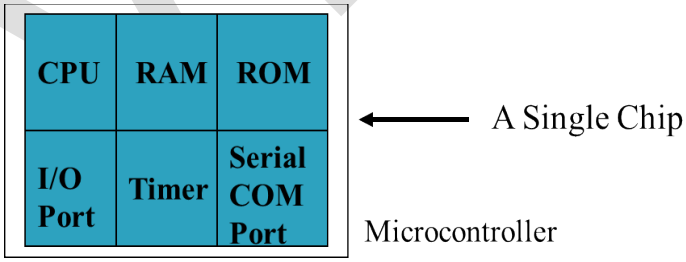
Microprocessor Based Embedded System:

- CPU for Computers
- No RAM, ROM, I/O on CPU chip itself
- Example: Intel‘s 8085,8086, Motorola‘s 680xx



Microcontroller Based Embedded System:

- A Small Computer or System on Chip (SoC)
- On-chip RAM, ROM, I/O ports...
- Example: TI‘s MSP430, Motorola‘s 6811, Intel‘s 8051, Zilog‘s Z8 and PIC 16X



S.No.	Microprocessor	Microcontroller
1	Contains Only CPU (μP); RAM, ROM, I/O Ports, Serial Port, Timers are Separately Interfaced	CPU (μP), RAM, ROM, I/O Ports, Serial Port, Timers are all on a Single Chip.
2	It has Many Instructions to Move data between Memory and μP	It has One or Two Instructions to Move data between Memory and μC
3	It has Few Bit Manipulation Instructions	It has Many Bit Manipulation Instructions
4	It has Less Number of Multifunctional Pins	It has More Number of Multifunctional Pins
5	General Purpose	Single (Specific) Purpose
6	High Speed	Low Speed
7	High Power Consumption	Low Power Consumption
8	μP Based System Requires More Hardware & High Cost	μC Based System Requires Less Hardware & Less Cost
9	Designer can decide on the amount of ROM, RAM and I/O ports.	Fixed amount of on-chip ROM, RAM, I/O ports

Embedded Processor and their types:

Microprocessor

Microprocessor is a programmable digital device which has high computational capability to run a number of applications in general purpose systems. It does not have memory or I/O ports built within its architecture. So, these devices need to be added externally to make a system functional. In embedded systems, the design is constrained with limited memory and I/O features. So microprocessors are used where system capability needs to be expanded by adding external memory and I/O.

Microcontroller

A microcontroller has a specific amount of program and data memory, as well as I/O ports built within the architecture along with the CPU core, making it a complete system. As a result, most embedded systems are microcontroller based, where are used to run one or limited number of applications.

Embedded Processor

Embedded processors are specifically designed for embedded systems to meet design constraints. They have the potential to handle multitasking applications. The performance and power efficiency requirements of embedded systems are satisfied by the use of embedded processors.

DSP

Digital signal processors (DSP) are used for signal processing applications such as voice or video compression, data acquisition, image processing or noise and echo cancellation.

ASIC

Application specific integrated circuit (ASIC) is basically a proprietary device designed and used by a company for a specific line of products (for example Samsung cell phones or Cisco routers etc.). It is specifically an algorithm called intellectual property core implemented on a chip.

FPGA

Field programmable gate arrays (FPGA) have programmable macro cells and their interconnects are configured based on the design. They are used in embedded systems when it is required to enhance

Memory Types:

The semiconductor memory can be classified into two types

- Volatile Memory
- Non-volatile memory

Volatile memory

- RAM
  - Static RAM
  - Dynamic RAM

Non-volatile memory

- ROM
  - Masked ROM
  - OTPROM
  - EPROM
  - EEPROM
  - FLASH
    - NOR FLASH
    - NAND FLASH
- NVRAM (Battery Backup RAM)

Volatile Memory:

RAM (Random Access Memory):

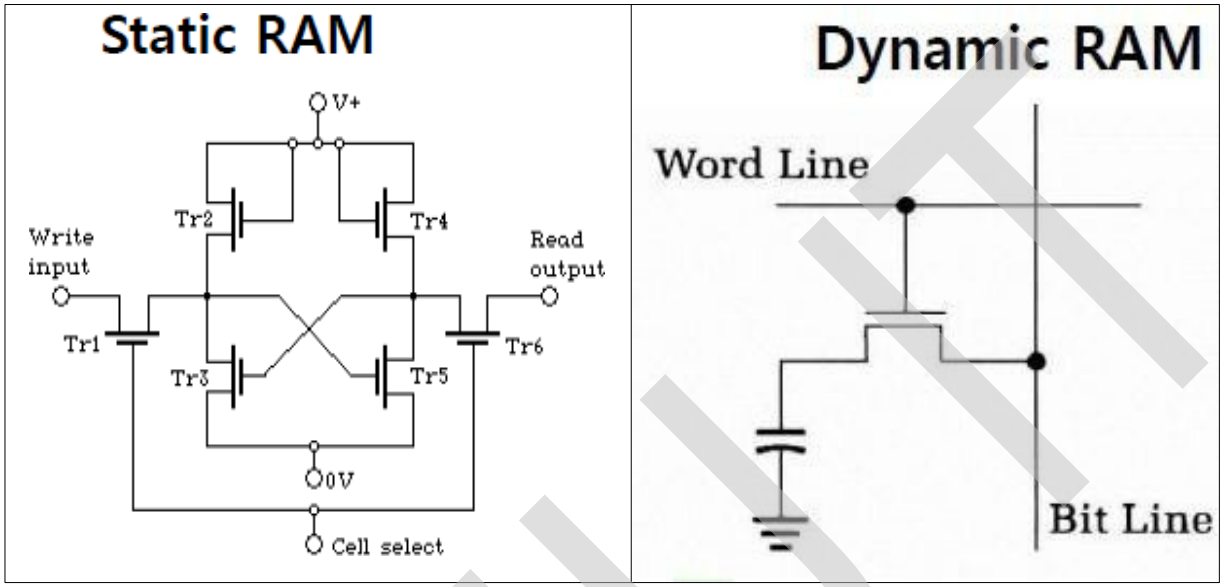
Loses its contents when power is removed, it is usually called Random-Access Memory-RAM. The vital feature is that data can be read or written with equal ease. It has ability to access any memory cell directly. RAM is much faster than ROM. It used to write and read data values while program running. Local variables, pointers, functions, recursive functions results in using large amounts of RAM. Volatile memory is used for data, and small microcontrollers often have very little RAM.

Static RAM:

Means that it retains its data even if the clock is stopped (provided that power is maintained, of course). A single cell of static RAM needs six transistors. RAM therefore takes up a large area of silicon, which makes it expensive.

Dynamic RAM:

This needs only one transistor per cell but must be refreshed regularly to maintain its contents, so it is not used in small microcontrollers. Most memory in a desktop computer is dynamic RAM.



Static RAM Vs Dynamic RAM:

Static RAM (SRAM)	Dynamic RAM (DRAM)
Made from Flip-Flops.	Made from Capacitors
High cost (per bit)	Low cost (per bit)
High using Power	Low using Power
Fast	Slow
Used in Cache Memory	Used in Main Memory
Large in Size	Low in Size
Will Retain State Forever	Automatically Discharges after sometime, Need Refreshing

Nonvolatile Memory:

ROM (Read-Only Memory):

Retains its contents when power is removed and is therefore used for the program and constant data. It is usually called Read-Only Memory-ROM. It is used as Program Memory in Microcontroller. It can't be written or modified at run time. There are many types of nonvolatile memory in use:

Masked ROM:

The data are encoded into one of the masks used for photolithography and written into the IC during manufacture. This memory really is read-only. It is used for the high-volume production of stable products, because any change to the data requires a new mask to be produced at great expense.

OTP ROM (One-Time Programmable ROM):

This is just PROM (Programmable ROM) in a normal package without a window, which means that it cannot be erased. It can be programmed one time only. Used when the firmware is stable and the product is shipping in bulk to customers. Devices with OTP ROM are still widely used and the first family of the MSP430 used this technology.

EPROM (Erasable Programmable ROM):

As its name implies, it can be programmed electrically but not erased. Devices must be exposed to ultraviolet (UV) light for about ten to twenty minutes to erase them. Erasable devices need special packages with quartz windows, which are expensive.

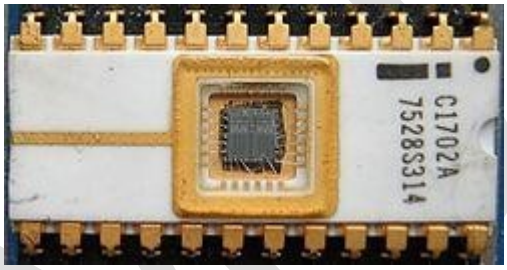


Figure: Example of EPROM IC with quartz window

EEPROM:

EEPROM (also E^2PROM) stands for Electrically Erasable Programmable ROM. EEPROMs can be programmed and erased in-circuit i.e. without removing from hardware kit, by applying electrical signals. The contents of this memory may be changed during run time (similar to RAM), but remains permanently saved even if the power supply is off (similar to ROM). EEPROM is often used to read and store values, created during operation, which must be permanently saved.

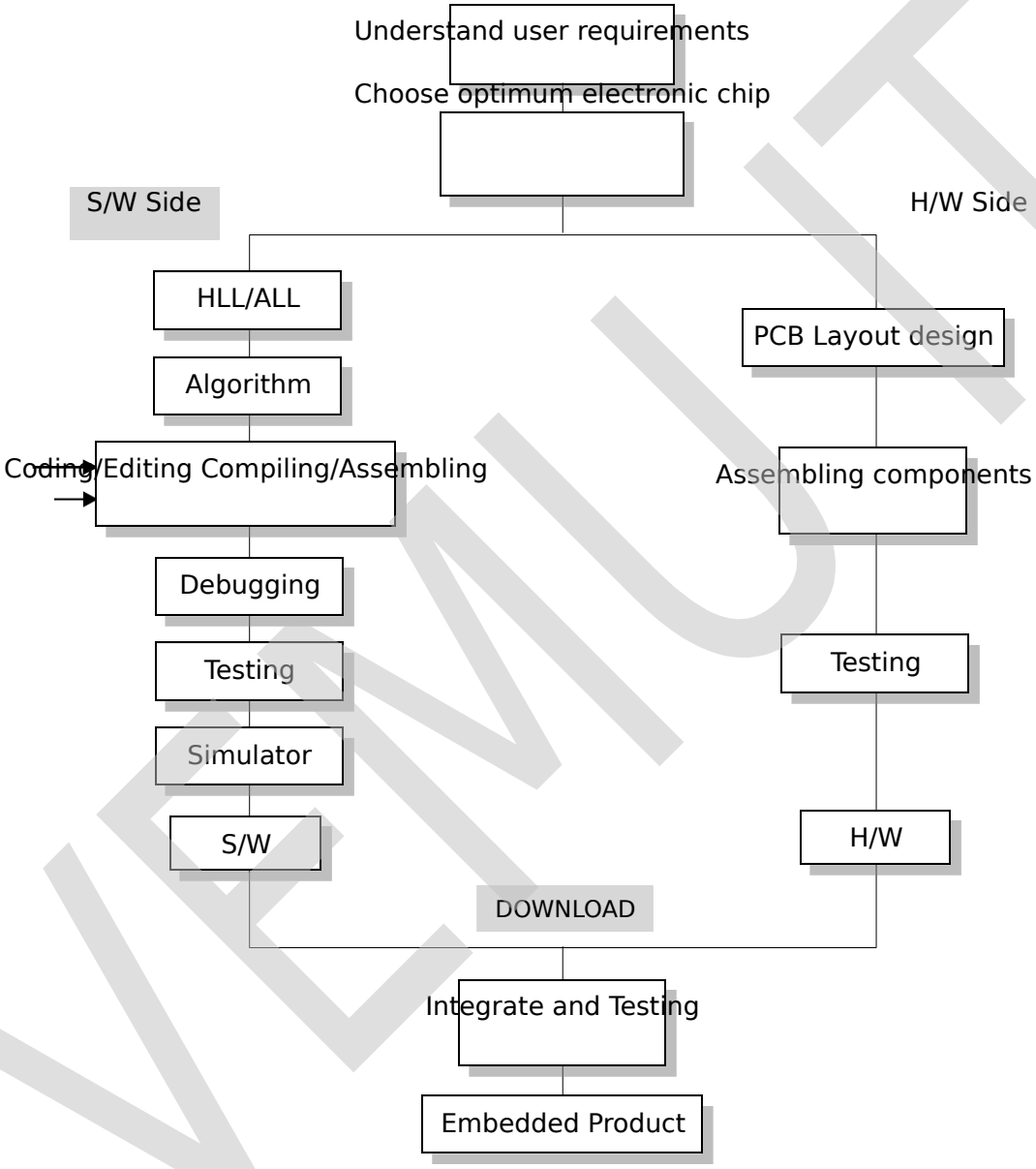
FLASH Memory:

This can be both programmed and erased electrically and is now by far the most common type of memory. Flash Memory is designed for high speed and high density, at the expense of large erase blocks (typically 512 bytes or larger). The practical difference is that individual bytes of EEPROM can be erased but flash can be erased only in blocks. Most MSP430 devices use flash memory, shown by an F in the part number. Microcontrollers use NOR flash, which is slower to write but permits random access. NAND flash is used in bulk storage devices and can be accessed only serially in rows.

Many microcontrollers include both: Flash Memory for the firmware (embedded program), and a small EEPROM for parameters and history.

Overview of Design Process of Embedded Systems:

The below figure shows the Design Process of Embedded Systems, In the first stage, according to user requirements the designer chooses the electronic chip which is suitable. In the next stage the designer will work on S/W side and H/W side separately. After that they will integrate the both S/W and H/W to design the Embedded System.



An example of an embedded system:

Here is a simple application to introduce a small embedded system – a stepper motor controller for a robotics system. The stepper motor mentioned here is an electromechanical device that rotates in discrete step angles when electrical pulses are applied to it. Suppose in an industrial environment, a robot arm is employed to pick-up components from one container and deposit to another container. The robot arm is operated by three stepper motors,

one to move the arm from one container to the other and other two to make a grip to tightly hold the component. To control these stepper motors, a small embedded system can be designed as shown in below figure. The hardware components are a microcontroller, three stepper motors and a robot arm. To make this system functional, three program modules are required.

- Module1 for the stepper motor1 to rotate counter clockwise with definite angle so that robot arm can move from container1 to container 2 also to rotate clockwise to move back to container1 when the component is picked.
- Module 2 for the stepper motors 2 & 3 to make the motor 2 to rotate in counter clockwise and motor 3 to rotate in clockwise both simultaneously so that the robot arm can pick and make a grip on the component.

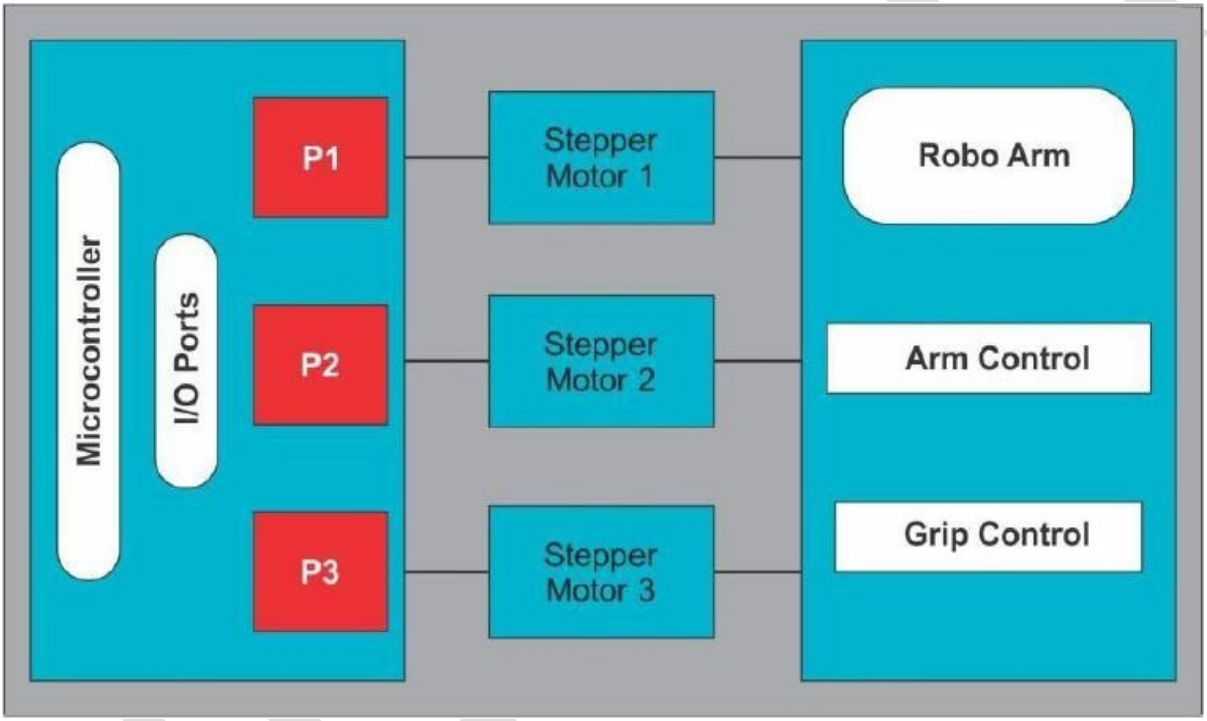


Figure: Stepper Motor Control Embedded System

Programming Languages and Tools for Embedded Design:

Programming Languages:

- Code is typically written in C or C++, but various high-level programming languages, such as Python and JavaScript, are now also in common use to target microcontrollers and embedded systems. Ada is used in some military and aviation projects.

Machine code: The binary data that the processor itself understands. Each instruction has a binary value called an opcode. It is unrecognizable to humans, unless you spent a very long time on low-level debugging. Some very early computers had to be programmed in machine code, but that was long ago, thank goodness. You will see it, however, because the contents of memory are shown in the debugger and machine code is included in the –disassembly‖.

Assembly language: Little more than machine code translated into English. The instructions are written as words called mnemonics rather than binary values and a program called an assembler translates the mnemonics into machine code. It does a little more than direct translation, but not a lot, nothing like a compiler for a high-level language.

A major disadvantage of assembly language is that it is intimately tied to a processor and is therefore different for each architecture. Even worse, the detailed usage varies between development environments for the same processor. Most programming of small microcontrollers was done in assembly language until recently, despite these problems, mainly because compilers for C produced less-efficient code. Now the compilers are better and modern processors are designed with compilers in mind, so assembly language has been pushed to the fringes. A few operations, notably bitwise rotations, cannot be written directly in C, and for these assembly language may be much more efficient. However, the main argument for learning assembly language is for debugging. There is no escape if you need to check the operation of the processor, one instruction at a time. Disassembly is the opposite process to assembly, the translation of machine code to assembly language.

C: The most common choice for small microcontrollers nowadays. A compiler translates C into machine code that the CPU can process. This brings all the power of a high-level language—data structures, functions, type checking and so on—but C can usually be compiled into efficient code. Compilation used to go through assembly language but this is now less common and the compiler produces machine code directly. A disassembler must then be used if you wish to review the assembly language.

C++: An object-oriented language that is widely used for larger devices. A restricted set can be used for small microcontrollers but some features of C++ are notorious for producing highly inefficient code. Embedded C++ is a subset of the language intended for embedded systems. Java is another object-oriented language, but it is interpreted rather than compiled and needs a much more powerful processor.

BASIC: Available for a few processors, of which the Parallax Stamp is a well-known example. The usual BASIC language is extended with special instructions to drive the peripherals. This enables programs to be developed very rapidly, without detailed understanding of the peripherals. Disadvantages are that the code often runs very slowly and the hardware is expensive if it includes an interpreter.

Programming Tools:

The microcontrollers can be programmed using various Programming/Development tools like Embedded Software from Mentor Graphics, IAR Systems, Keil MicroVision from ARM Ltd., and Code Composer Studio (CCS) from Texas Instruments.

Introduction to Code Composer Studio:

At this juncture of the book, it is important to concentrate on a single development tool for programming the hardware. In this book, Code Composer Studio is used because it is

free and is supported by a large TI community commonly known as E2E. This forum is a congregation of engineers and hobbyist working on TI products across the world.

Code Composer Studio (CCS):

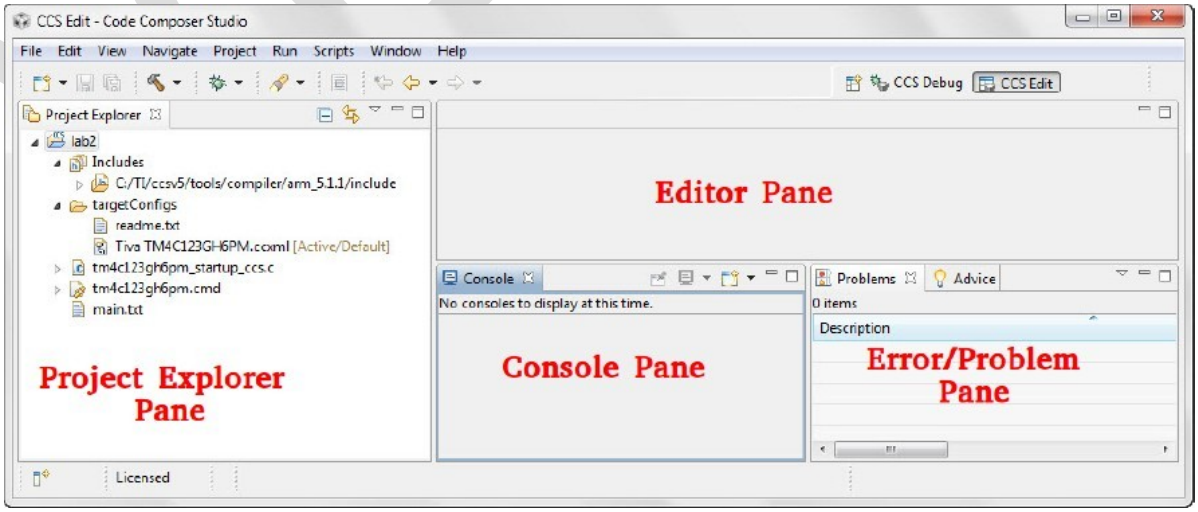
CCS is a TI proprietary cross platform IDE used to convert C and assembly language code to executable for TI processors, involving Digital Signal Processors, ARM, and other microcontrollers. CCS combines editing, debugging and analysis tools into a single IDE based on Eclipse open source development tool.

Compilation (Build Process):

CCS uses a Code Generation Tool (often called CGT) for compilation process also called as build process. It is used to generate executables for a target device. This process takes source codes written in C, C++ and/or Assembly Language and produces executable at the output called binaries. This process undergoes many intermediated stages. A normal development flow of a compilation process takes the source files written in C/C++ and compiles them to create assembly language files using a compiler. For projects where source code is written in assembly language, this process is bypassed. In many projects, some of the logic functions are written in assembly languages along with other C/C++ source codes. After the assembly language files are generated by the compiler, an assembler converts them to re-locatable object files. These object files are linked to the runtime libraries included in the project by a linker. The linker produces executables from these files using protocols mentioned in the command file. A command file typically consists of a list of all memories of the part and the type of software compatible to them.

CCS uses C/C++ compiler and other compilation tools which are developed by engineers at TI over the years and packaged into TI‟s code generation tools. They are commonly amalgamation (combination) of various tools used in compilation processes, namely, Compiler, Assembler, Linker, Optimizer, Code Generator, Parser, Linear Assembler, Archiver, Disassembler and many more.

Review the CCS Editing GUI – Figure below shows various panes of CCS which appear after project is created. Understanding their nomenclature and functionality with help in process of application development



Debugger:

The CCS debugger depends on a configuration file and a general extension language (GEL) file. The debugger initializes and loads the software on a target device using information provided by these files. A target configuration file specifies

(i) Connection type to the target device,

(ii) Target device, and

(iii) About a startup script

Table: Programming/Development Tools for Tiva C Series

Product	License	Compiler	IDE	Debugger	JTAG
Embedded Software, Mentor Graphics	30-day full function	GNU C/C++	Gdb	Eclipse	
KEIL MicroVision, ARM Ltd.	Full function. Onboard emulation limited	Real View C/C++	µVision	µVision	U-Link, 199 USD
CCS, Texas Instruments	32KB code size limited. Upgradeable	TI C/C++	Eclipse	CCS	XDS100 79 USD
IAR Systems	32KB code size limited. Upgradeable	IAR C/C++	Embedded Workbench	C-SPY	J-Link, 299 USD

In CCS, startup scripts are specified to setup the memory map for debugger. It is also used to setup any initial target state that is necessary for connection to the debugger using memory or register writes. These scripts are known as GEL script file. „OnStartup()" function in the GEL file runs when the debugger is launched. After the target is connected,

„OnTargetConnect()"defined in the GEL file is executed.

Organization and Building a CCS Project:

In CCS, designs are organized in workspaces and projects which are merely folders in the file system. When CCS is launched, it prompts the users to provide a folder path to the workspace. This folder consists of individual project folders and a folder named as

_.metadata'. The .metadata folder consists of CCS settings and preferences for the particular workspace. Along with source files, header files and library files, each project folder contains the build and the tool settings for the project. It also contains the target configuration file and the command file required by the debugger. CCS has two predefined build configurations, namely, debug and release. It also provides custom build configurations.

Debug – It is generally used when it is required to operate in debugging mode. It includes the symbol tables and executes compilation without any optimization.

Release – Building project in this mode is suited when the user requires performance. It discards all symbol tables and implements the full code optimization. It is therefore a noted convention to use this mode only when the final version of a project is to be deployed on the hardware. For all other intermediate versions, debug mode is rather preferred.

Custom Configuration – CCS also provide its users to add custom build configurations for a particular project. It can be done by going to processor options under _properties -> build -> ARM Compilers".

UNIT-II

Embedded Processor Architecture

CISC Vs RISC Design Philosophy:

Instruction Set Architecture:

Instruction set can be defined as the communication interface between the processor and the programmer. Every processor has its own instruction set implemented in the hardware to execute instructions such as move, add or multiply data in a definite way. Programmers can either use any high level language such as C, C ++, Java etc. or assembly language to write the program. Accordingly, a compiler or assembler can be used to translate the program into machine understandable language following the processor instruction set. There are two classic architectures of instruction set implementation, the Complex Instruction Set Computer (CISC) and the Reduced Instruction Set Computer (RISC). Each has its own advantages and disadvantages. The CISC architecture has more complexity in the hardware itself while RISC architecture offers more complexity to the software. The features of each architecture are summarized as below.

S.No.	CISC	RISC
1	Most of the instructions are complex in type	Most of the instructions are simple in nature.
2	"LOAD“ and "STORE“ incorporated in Instructions Ex: MOV	"LOAD" and "STORE" are independent instructions Ex: LOAD, STORE
3	It has more addressing modes	It has less addressing modes
4	Instruction sizes are different	Instruction sizes are same
5	Number of machine cycles for instruction are different	Number of machine cycles for instruction are same
6	Fewer working registers and more frequent memory access.	Higher number of working registers so less frequent memory access.
7	Pipeline implementation is difficult.	Pipeline implementation is easier.
8	Programming is easy	Programming is complex
9	Hardware is complex	Hardware is simple
10	Small code size	Large code size
11	More complexity is given to the hardware design.	More complexity is offered to the compiler design.

12	Ex: Intel, IBM, Atmel 805x	Ex: MSP430, AMD, ARM, Atmel AVR

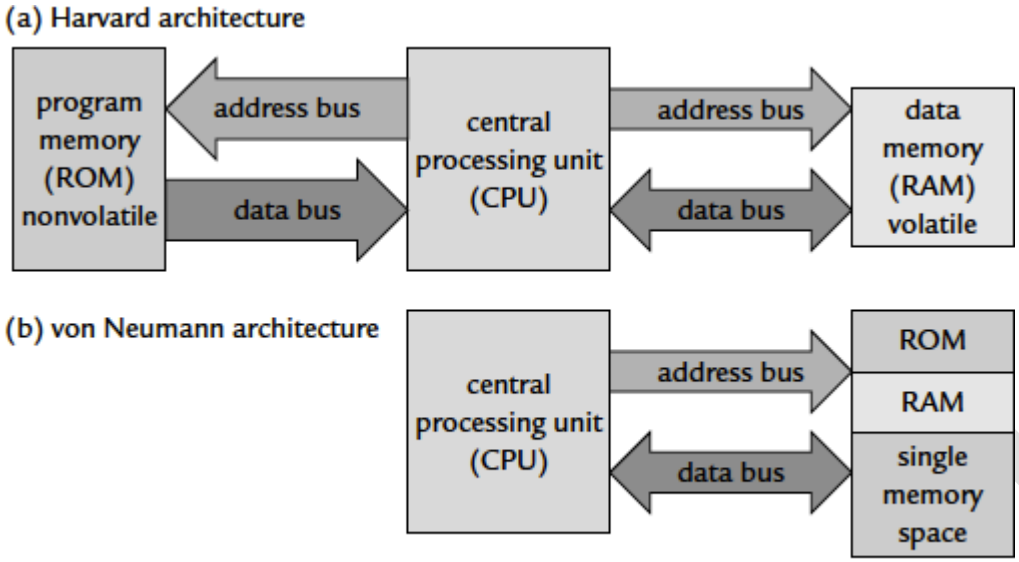
Von-Neumann Vs Harvard architecture:

Memory Block:

The memory block consists of program and data memory. ROM is used as the program memory and RAM is used as the data memory. There are two memory architectures: Harvard and Von-Neumann.

In *Harvard architecture*, the program and data memories are segregated with separate address and data bus drawn to each. So there can be parallel access to both and performance of the system can be improved at the cost of hardware complexity. On the other-hand, the *Von-Neumann architecture* has one unified memory used for both program and data. The system is comparatively slower, but the design implementation is simple and cost effective for an embedded system. The Von-Neumann also called as Princeton Architecture.

S.No.	Von-Neumann	Harvard
1	It has single data/address bus to fetch both program and data	It has separate data/address bus to fetch both program and data
2	Here, both program and data are stored in a same (single) memory.	Here separate memory for program and data.
3	Slower execution	Faster execution.
4	It first fetches an instruction, and then it fetches the data to process the instruction.	Here, both instruction and data are fetches simultaneously
5	More instruction cycles are required	Less instruction cycles are required
6	Program can be easily modified by itself because it is stored in read-write memory	It is difficult for program contents to be modified by the program itself
7	Widely used in microprocessors	Widely used in microcontrollers
8	Ex: Intel 8086, MSP430, ARM7	Ex: Intel 8051, PIC, ARM9



Harvard and von Neumann architectures for memory.

I/O mapped I/O vs. Memory mapped I/O

There are two techniques for addressing an I/O device by CPU:

- I/O mapped I/O (Isolated I/O or Port mapped I/O)
- Memory mapped I/O

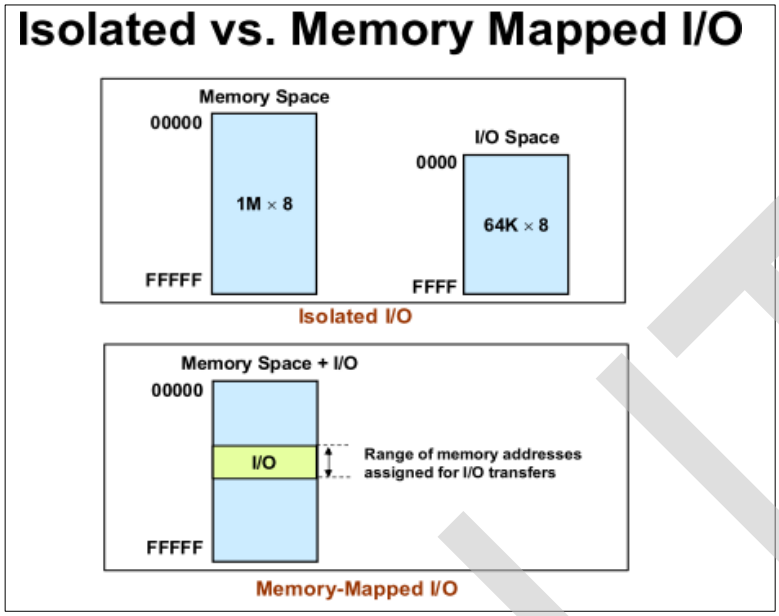
I/O mapped I/O:

- Here two separate address spaces are used - one for memory and other for I/O devices.
- The I/O devices are addressed with dedicated address space.
- Hence there are two separate control lines for memory and I/O transfer.
  - I/O read and I/O write lines for I/O transfer
  - Memory Write and Memory Read for memory transfer
- Hence IN and OUT instruction deals with I/O transfer and MOV instruction deals with memory transfer.
- Ex: Intel's 8085, 8086.

Memory mapped I/O:

- The technique in which CPU addresses an I/O device just like a memory i.e., address space is same for both I/O and Memory is called memory mapped I/O scheme.
- In this scheme only one address space is used by CPU. Some addresses of the address space are assigned to memory location and other are assigned to I/O devices.
- There is only one set of read and write lines. Hence there is no separate IN, OUT instructions. MOVE instruction can be used to accomplish both the transfer.
- The instructions used to manipulate the memory can be used for I/O devices.

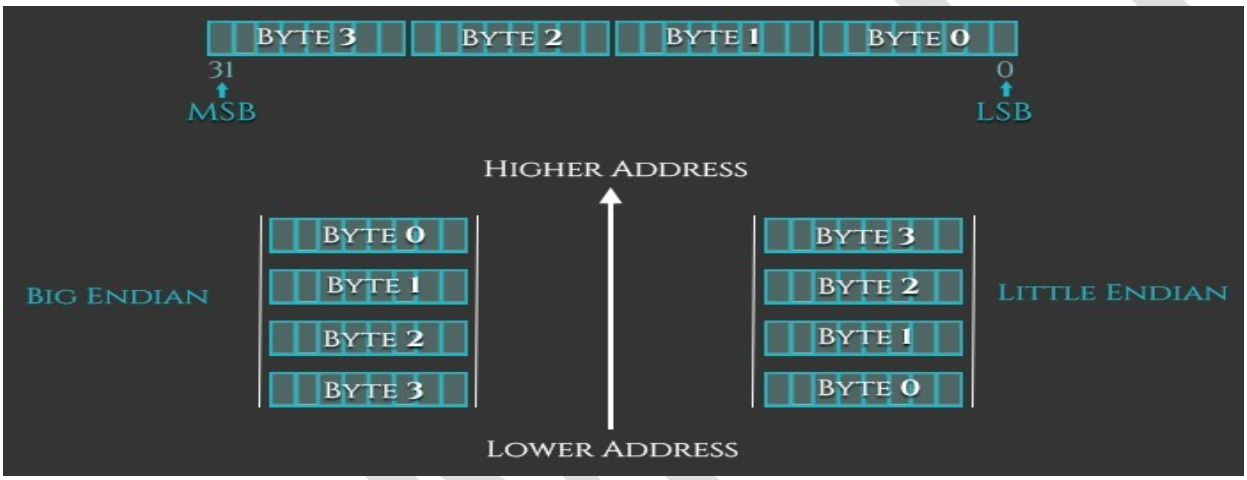
- Ex: Intel's 8051, TI's MSP430



Little-Endian vs. Big-Endian

This is just a way of saying the order of storing bytes in memory. This order is categorized into two i.e., little-endian and big-endian.

- The little-endian machines like Intel x86, the low-order byte is stored at the lower address and the high-order byte at the higher address.
  - Ex: Intel's 8086, 80x86, TI's MSP430
- The big-endian machines like Motorola MC680xx, the high-order byte is stored at the lower address and the low-order byte at the higher address.
  - Ex: Motorola MC680x0, Freescale HCS08



Introduction to ARM Architecture and Cortex – M series:

ARM Architecture:

ARM cores are designed specifically for embedded systems. The needs of embedded systems can be satisfied only if features of RISC and CISC are considered together for processor design. So ARM architecture is not a pure RISC architecture. It has a blend of both RISC and CISC features.

S.No.	Features	Benefits to Embedded System
1	High Performance	Ensures the system has a fast response
2	Low power consumption	Makes the system more energy efficient
3	Low silicon area	Reduces the size and also consumes less power
4	High Code density	Helps embedded system to have less memory footprint
5	Load/store architecture	Used to load data from the memory to the ARM CPU register or store data from the CPU register to the memory; enables the memory access when required
6	Register bank with large number of working registers	Required to perform most of the operations within the CPU and provides faster context switch in a multitasking applications

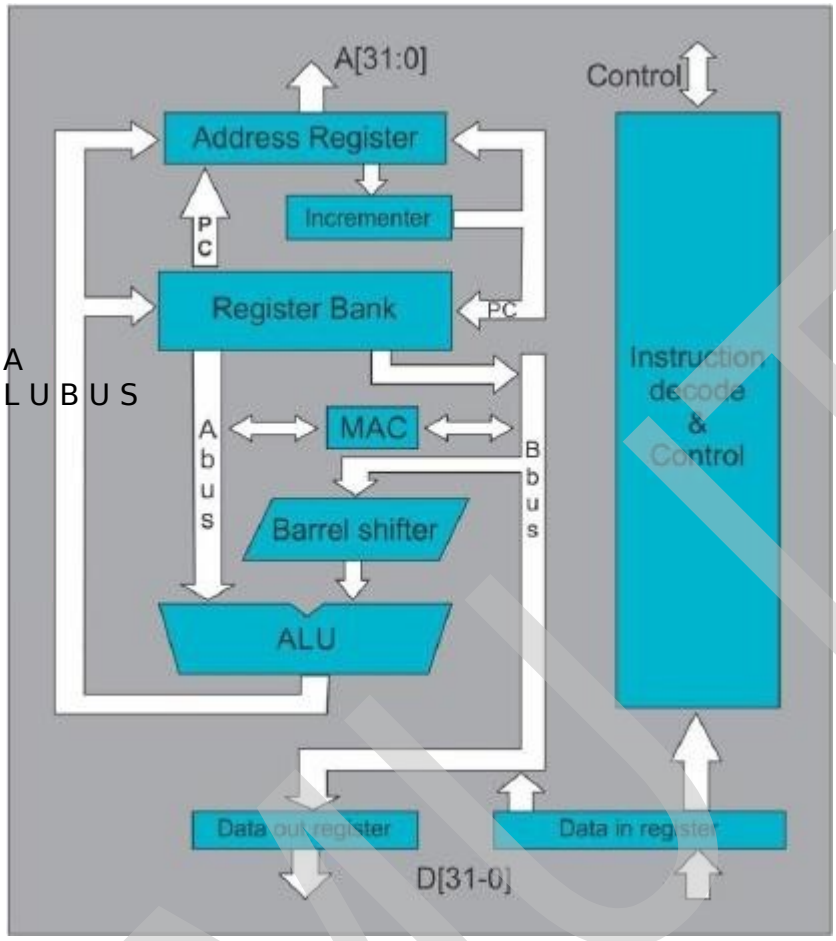


Figure: ARM7 Architecture

A Basic architecture of the ARM7 core:

ARM7, the basic architecture of ARM series of cores, is introduced here in this section. A brief introduction about each functional block of the architecture of ARM7 core shown in above Figure is presented below.

The Register Bank has sixteen general purpose registers (R0-R15) and a current program status register (CPSR) which are accessible by user applications. In addition to that, it has twenty numbers of banked registers specifically used for different operating modes of ARM core. These are invisible to user applications. The register bank has two read ports to read operand1 and operand2 and one write port to write back the result of operation to the any register specified in the instruction. It has an additional bidirectional port to update the program counter with address register and incrementer. Address register content is incremented at every sequential byte access by the incrementer but the program counter is incremented by four in ARM state of the core or is incremented by 2 in Thumb state of the core at every instruction access. Address register is directly connected to the address bus.

- The barrel shifter can shift or rotate operand 2 by specified number of bits prior to arithmetic or logic operations.
- The 32 bit ALU performs the arithmetic and logic functions.
- The data in and data out registers hold the input and output data from and to the memory.

- The instruction decoder and associated control logic generates appropriate control signals for the data path after decoding the fetched instruction.
- The MAC unit is to multiply two register operands and accumulate with another register holding the partial sum of the products.

The encoded instruction byte of the program saved in the code memory is fetched through the data bus and first enters into the data-in register of the ARM architecture from where it is delivered to the instruction decoder. After the instruction is decoded, appropriate control signals are generated for the data path. The required registers are activated in the register bank and the operands flow out from two read ports of register bank to the ALU: operand1 through A-bus and operand2 through B-bus after preprocessing at barrel shifter. The result of operation at ALU is written back to the result register through a write port at register bank. For Load/Store instructions, after decoding the instruction, the data memory address is first calculated at ALU as specified in the instruction and the pointer register is updated at the register bank. The address in the pointer register is given to the address register to access the memory and transfer data. If it is a load multiple or store multiple instruction, the core does not halt before completing the required number of data transfers unless it is a reset exception.

Migration to Cortex Series:

In the path of architectural evolution, ARM has contributed many versions of IP cores to the embedded computing world. ARM pioneered embedded products are excelling in every visible spectrum. Since its inception, ARM has migrated over a long meaningful road map starting from v4T ARM7TDMI to v7 Cortex series of architectures achieving many strong milestones in between. It is currently the new era of feature rich ARM Cortex series architectures truly empowering the embedded computing world.

ARM Architecture Evolution:

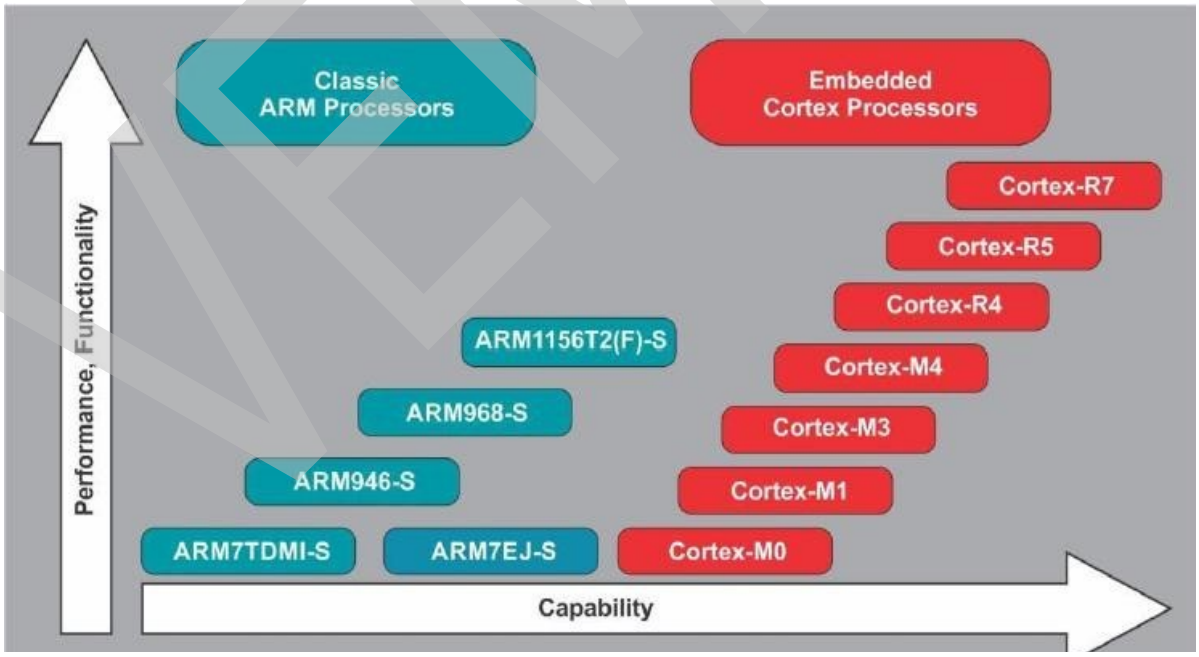


Figure: Performance and capability graph of Classic ARM and Cortex embedded processors.

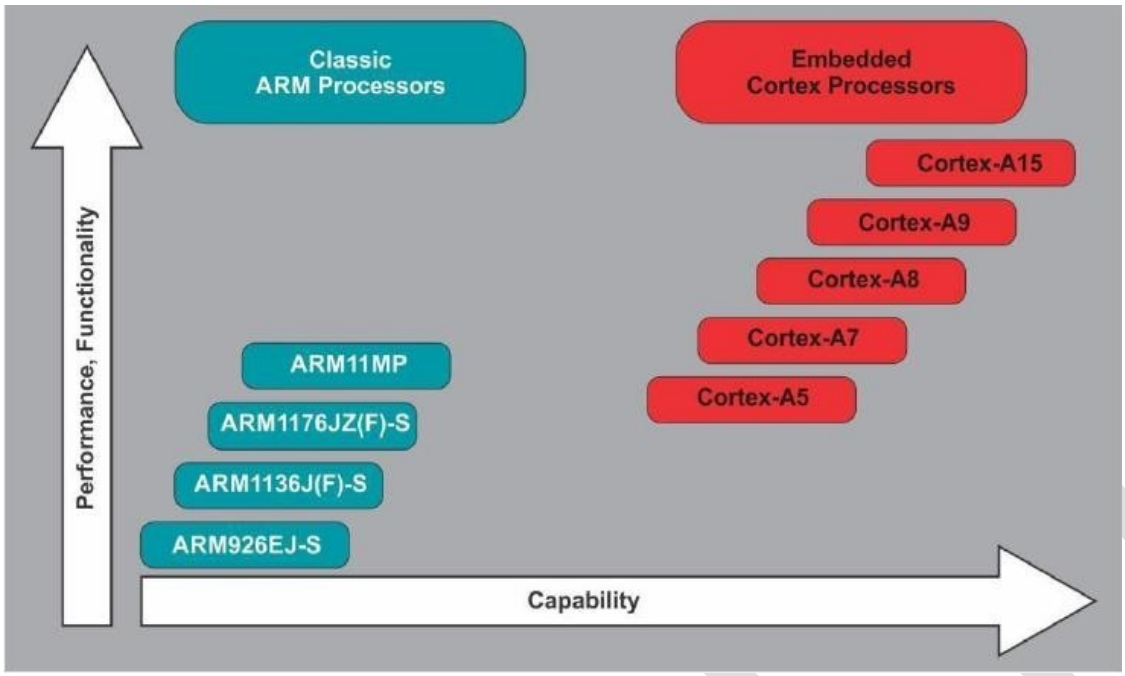


Figure: Performance and capability graph of Classic ARM and Cortex application processors.

ARM architecture has been improved a lot in the road map from classic ARM to ARM Cortex. The above figure depicts the performance and capability comparison of classic ARM with embedded cortex and application cortex series of processors. Even though ARM had earlier versions of products i.e. v1, v2, v3 and v4, the classic group of ARM starts with v4T. The classic group is divided into four basic families called ARM7, ARM9, ARM10 and ARM11.

- ARM7 has three-stage (fetch, decode, execute) pipeline, Von-Neumann architecture where both address and data use the same bus. It executes v4T instruction set. T stands for Thumb.
- ARM9 has five-stage (fetch, decode, execute, memory, write) pipeline with higher performance, Harvard architecture with separate instruction and data bus. ARM9 executes v4T and v5TE instruction sets. E stands for enhanced instructions.
- ARM10 has six-stage (fetch, issue, decode, execute, memory, write) pipeline with optional vector floating point unit and delivers high floating point performance. ARM10 executes v5TE instruction sets.
- ARM11 has eight-stage pipeline, high performance and power efficiency and it executes v6 instructions set. With the addition of vector floating point unit, it performs fast floating point operations

Nomenclature:

ARM processor implementation is described by the product nomenclature as given below

ARM [x][y][z][T][D][M][I][E][J][F][-S]

x - Family

y - Memory Management.

z - Cache size

T- Thumb state

D - JTAG Debug option
M - Fast multiplier
I - Embedded ICE Macrocell

E - Enhanced instructions

J - Jazelle state (Java)

F - Vector floating point unit

S - Synthesizable version

Referring to the nomenclature, ARM7TDMI can be understood as an ARM7 processor with thumb implementation, JTAG debug, multiplier and ICE macro cell. Similarly ARM926EJ-S is an ARM9 processor with MMU and cache implementation, enhanced instructions, Jazelle state and has a synthesizable core.

ARM Architecture v7 profile:

In order to provide a wide coverage of different application domains, addressing their specific requirements, ARM core is evolved into architecture version7 which has three different profiles: the Application profile(Cortex-A), Real Time profile(Cortex-R) and Microcontroller profile(Cortex-M). Architecture v7 should not be confused with ARM7 which has been explained before under architecture v4.

Application profile (Cortex -A):

Cortex A series of architectures are multicores with power efficiency and high performance. Every Cortex-A implementation is intended for highest performance at ultralow power design. It supports with, in-built memory management unit. Being influenced by multitasking OS system requirements, it has virtualization extensions and provides a trust zone for a safe and extensible system. It has enhanced Java support and provides a secure program execution environment. These architectures are typically designed for high end real time safety critical applications like automotive powertrain system. Some Cortex-A application products are smart phones, tablets, televisions and even high end computing servers.

Real-Time profile (Cortex -R):

Cortex R series of architectures are designed for deeply embedded real time multitasking applications. They have low interrupt latency and predictability features for real time needs. It provides memory protection for supervisory OS tasks being in privileged mode. It also provides tightly coupled memories for fast deterministic access. Typical application examples are: hard disk drive controller and base band controller for mobile applications and engine management unit where high performance and reliability at very low interrupt latency and determinism are critical requirements.

Microcontroller profile (Cortex -M):

Cortex M series of architectures have v6-M as cortex M0, M0+ and M1 and v7-M with Cortex M3, M4 and other successors. This series of architectures developed for deeply embedded microcontroller profile, offer lowest gate count so smallest silicon area. These are flexible and powerful designs with completely predictable and deterministic interrupt handling capabilities by introducing the nested vector interrupt controller (NVIC). The small instruction sets support for high code density and simplified software development. Developers are able to achieve 32-bit performance at 8-bit price. The very low gate count of Cortex M0 facilitates its deployment in analog and mixed mode devices. Due to further

demanding applications requiring even better energy efficiency, Cortex M0+ was designed with two stage pipeline and achieved high performance with very low dynamic power consumption, reduced branch shadow and reduced number of flash memory access. Cortex M1 was designed for implementation in FPGA. It is functionally a subset of Cortex M3 and runs ARM v6 instruction set with OS extension options. It has 32-bit AHB lite bus interface; separate tightly coupled memory interface and JTAG interface to facilitate debug options. It has three stage pipeline implementation and configurable NVIC for reducing interrupt latency.

ARMv7-M architecture:

Key features for ARMv7-M architectures are:

- Enable implementations with industry excelling power, performance and silicon area constraints with simple pipeline design.
- Highly predictable and deterministic operation with Single/low cycle instruction execution and minimum interrupt latency with cache less memory design.
- Exception handlers are standard C/C++ functions align with ARM‟s programming standard.
- Debug and software profiling support.
- Cortex M3 is the first architecture of ARMv7-M profile. Subsequently the architecture was enhanced by DSP extensions and named as Cortex M4. Cortex M3 a general purpose CPU, has optimized debug options for microcontroller applications. It has only Thumb-2 processing core which has blend of ARM 32-bits and Thumb 16 bits instructions which removes the need of ARM-Thumb interworking and offers high code density at high energy efficiency. A hardware divide instruction was introduced in the instruction set and a number of multiply instructions are also available to improve data processing performance. It supports only two modes of operation called thread and handler mode. User programs run in thread mode and exceptions are handled in handler mode which is privileged. All exceptions could be programmed in C/C++. NVIC is part of Cortex-M3 macrocell. It is a 32 bit core with 18 working registers: r0-r7 as low registers, r8-r12 as high registers, three special purpose registers, r13 stack pointer, r14 link register, r15 the program counter, one program status register xPSR and one more stack pointer banked for handler mode. The Cortex-M3 and Cortex-M4 processors also support unaligned data accesses, a feature previously available only in high- end processors. Cortex M4 comes under the nomenclature of ARMv7E-M. It was developed as a high performance digital signal controller with 72 DSP instructions implemented along with Cortex M3 instruction set retained. Single cycle execution of multiply and accumulate instructions provides 45% speed improvement compared to Cortex M3.

Cortex M4 Features:

- Thumb2 instruction set delivers the significant benefits of high code density of Thumb with 32-bit performance of ARM.
- Optional IEEE754-compliant single-precision Floating Point Unit.
- Code-patch ability for memory system updates.
- Power control optimization by integrating sleep and deep sleep modes.
- Hardware division and fast multiply and accumulate for SIMD DSP instructions.
- Saturating arithmetic for noise cancellation in signal processing.
- Deterministic, low latency interrupt handling for real time-critical applications.
- Optional Memory Protection Unit(MPU) for safety-critical applications
- Extensive implementation of debug, trace and code profiling capabilities.

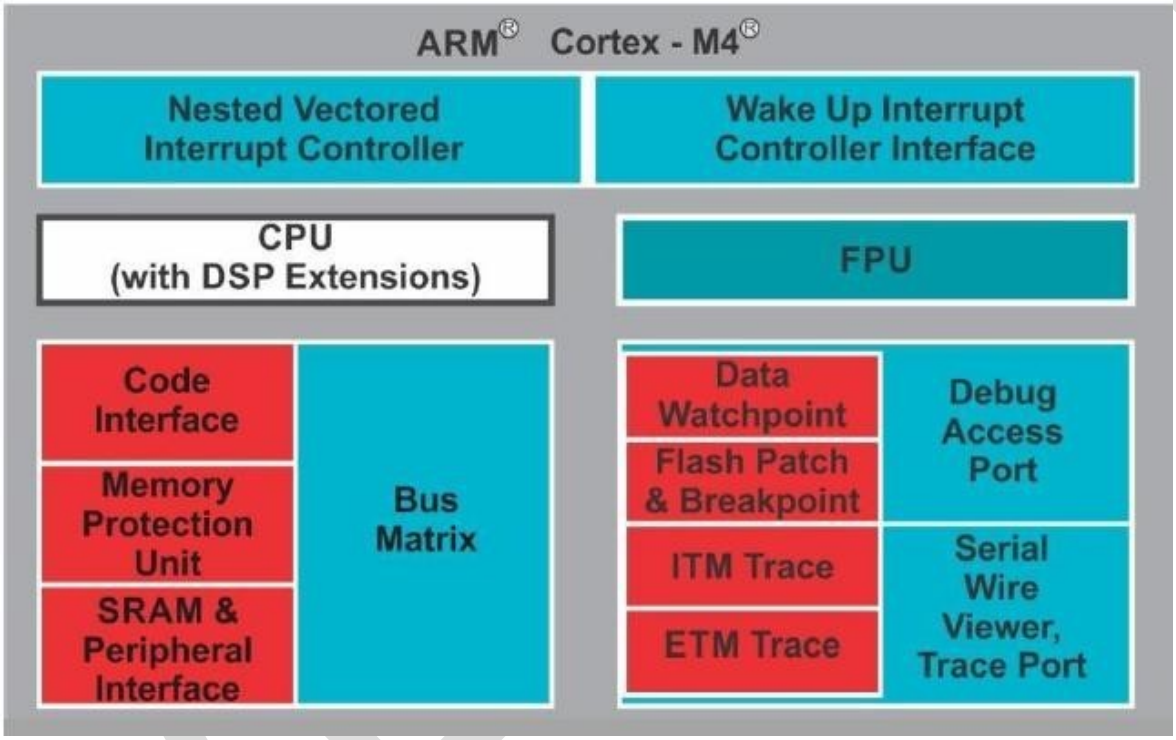


Figure: Cortex M4 Core Architecture

The ARM Cortex-M4 architecture is built on a high-performance processing core, with a 3-stage pipeline. Harvard architecture, 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 features make it typically suitable for high precision digital signal processing applications. The processor delivers excellent energy efficiency at high code density and significantly improving interrupt handling and system debug capabilities. A generic system on chip architecture of Cortex M4 is shown in above figure. The brief description of each functional block is given below.

Nested Vectored Interrupt Controller (NVIC):

Tightly integrated with the processor core, NVIC is a configurable Interrupt Controller used to deliver excellent real time interrupt performance. Very low interrupt latency is achieved through its hardware stacking registers. The processor automatically saves and retrieves its state on exception entry and exit removing the code overhead from ISRs. It also has the ability of interrupting the load and stores multiple atomic instructions that provides faster interrupt response. The NVIC includes a Non Maskable Interrupt (NMI) and can provide up to 256 interrupt priority levels for each of 240 interrupts it supports. A higher priority interrupt can preempt the currently running ISR facilitating interrupt nesting.

Wake Up Interrupt Controller (WIC):

To optimize low-power designs, the NVIC integrates with an optional peripheral called Wake up interrupt controller to implement sleep modes and an optional deep sleep function. When the WIC is enabled, the power management unit powers down the processor and makes it enter deep sleep mode. When the WIC receives an interrupt, it takes few clock cycles to wake-up the processor and restore its state. So it adds to interrupt latency in deep sleep mode. WIC is not programmable and operates completely with hardware signals.

Memory Protection Unit:

In embedded OS, MPU is used for safeguarding memory used for kernel functions from unauthorized access by user program. In OS environment, when any untrusted user program tries to access memory protected by MPU, the processor generates a memory manage fault causing a fault exception. MPU divides the memory map into a number of regions defining memory attributes for each. MPU separates and protects the code, data and stack for each task required for safety critical embedded systems. MPU can be implemented to enforce privilege access rules and separate tasks. It is an optional block in Cortex M4.

Bus Matrix:

The processor contains a bus matrix that arbitrates the processor core and optional Debug Access Port (DAP) memory accesses to both the external memory system, the internal System Control Spaces and to various debug components. It arbitrates requests from different bus masters in the system. Bus matrix is connected to the code interface for accessing the code memory, SRAM and peripheral interface for data memory and other peripherals and the optional MPU for managing different memory regions.

Debug Access Port (DAP):

DAP, the implementation of ARM debug interface enables debug access to various master ports on the ARM SoC. It provides system access for the debugger tool using AHB- AP, APB-AP and JTAG-AP without halting the processor. Embedded Trace Macrocell (ETM) generates instruction trace. Instrumentation Trace Macrocell (ITM) allows software- generated debug messages and also to generate timestamp information. Data Watch point and Trace (DWT) unit can be used to generate data trace, event trace, and profiling trace information. Flash patch and break point (FPB) implements hardware breakpoints, patches code and data from Code space to System space. Serial wire viewer (SWV) is one bit ETM port. SWV provides different types of information like program counter values, data read and write cycles, peripheral values, event counters and exceptions.

Floating Point Unit (FPU):

Cortex M4 architecture suggests an optional FPU which is IEEE 754 single precision compliant. The core instruction set supports various signal processing operations. It executes single instruction multiple data (SIMD) instructions with 16 bit data types. Floating point core supports addition, multiplication and hardware division. It has a 32X32 multiply and accumulate (MAC) unit that produces 64 bit results. Embedded signal processing applications that involve data compression, statistical signal processing, measuring, filtering and compressing real world analog signals can use Cortex M4 with FPU.

Floating point unit supports:

- Conversions between fixed point and floating point data formats and instructions with floating point immediate data.
- Saturation math.
- Decouple 3-stage pipeline.
- Three modes of operations: full compliance mode, flush-to-zero mode and default NaN mode.
- To be disabled when it is not in use to conserve energy.

Introduction to the TM4C family viz. TM4C123x & TM4C129x and its targeted applications:

TIVA TM4C123GH6PM Microcontroller:

Figure: TIVA TM4C123GH6PM Microcontroller Block Diagram

The TM4C microcontroller block diagram shown in above figure has six functional units. The cortex M4F core, on-chip memory, analog block, serial interface, motion control and system integration.

Features:

- TM4C123GH6PM microcontroller has 32 bit ARM Cortex M4 CPU core with 80 MHz clock rate.

- Memory protection unit (MPU) provides protected operating system functionality and floating point unit (FPU) supports IEEE 754 single precision operations.
- JTAG/SWD/ETM for serial wire debugs and traces (SWD/T).
- Nested vector interrupt controller (NVIC) reduces interrupt response latency.
- System control block (SCB) holds the system configuration information.
- The microcontroller has a set of memory integrated in it: 256 KB flash memory, 32 KB SRAM, 2 KB EEPROM and ROM loaded with TIVA software library and boot loader.
- Serial communications peripherals such as: 2 CAN controllers, full speed USB controller, 8 UARTs, 4 I2C modules and 4 Synchronous Serial Interface (SSI) modules.
- On chip voltage regulator, two analog comparators and two 12 channel 12-bit analog to digital converter with sample rate 1 million samples per second (1MSPS) are the analog functions in built to the device.
- Two quadrature encoder interface (QEI) with index module and two PWM modules are the advanced motion control functions integrated into the device that facilitate wheel and motor controls.
- Various system functions integrated into the device are: Direct Memory Access controller, clock and reset circuitry with 16 MHz precision oscillator, six 32-bit timers, six 64-bit timers, twelve 32/64 bit captures compare PWM (CCP), battery backed hibernation module and RTC hibernation module, 2 watchdog timers and 43 GPIOs.

Few Applications:

- Building automation system
- Lighting control system
- Data acquisition system
- Motion control
- IoT and Sensor networks.

TIVA TM4C129CNCZAD Microcontroller:

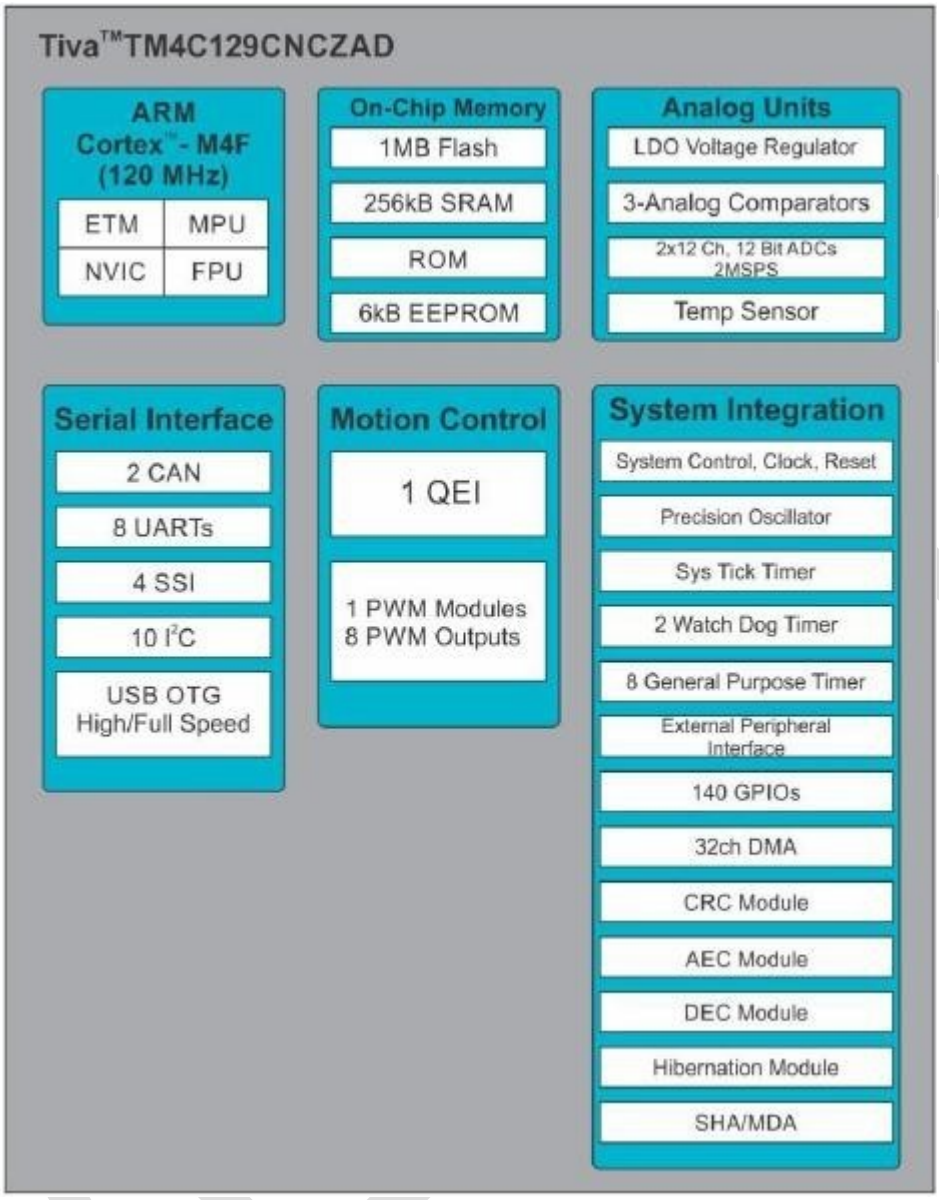


Figure: TIVA TM4C129CNCZAD Microcontroller Block Diagram

Features:

- TM4C129CNCZAD microcontroller has 32 bit ARM Cortex M4F CPU core with 120 MHz clock rate.
- Memory protection unit (MPU) provides a privileged mode for protected operating system functionality and floating point unit (FPU) supports IEEE 754 compliant single precision operations.
- JTAG/SWD/ETM for serial wire debug and trace.
- Nested vector interrupt controller (NVIC) reduces interrupt response latency and high performance interrupt handling for time critical applications.

- The microcontroller has a set of memory integrated in it: 1MB flash memory, 256 KB SRAM, 6 KB EEPROM and ROM loaded with TIVA ware software library and boot loader.

- Serial communications peripherals such as: 2 CAN controllers, full speed and high speed USB controller, 8 UARTs, 10 I2C modules and 4 Synchronous Serial Interface (SSI) modules.
- On chip voltage regulator, three analog comparators and two 12 channel 12-bit analog to digital converter with sample rate 2 million samples per second (2MSPS) and temperature sensor are the analog functions in built to the device.
- One quadrature encoder interface (QEI) and one PWM module with 8 PWM outputs are the advanced motion control functions integrated into the device that facilitate wheel and motor controls.
- Various system functions integrated into the device are: Micro Direct Memory Access controller, clock and reset circuitry with 16 MHz precision oscillator, eight 32-bit timers, and low power battery backed hibernation module and RTC hibernation module, 2 watchdog timers and 140 GPIOs.
- Cyclic Redundancy Check (CRC) computation module is used for message transfer and safety system checks. CRC module can be used in combination with AES and DES modules.
- Advanced Encryption Standard (AES) and Data Encryption Standard (DES) accelerator module provides hardware accelerated data encryption and decryption functions.
- Secure Hash Algorithm/ Message Digest Algorithm (SHA/MDA) provides hardware accelerated hash functions for secured data applications.

Address Space (Memory Map):

A TM4C123GH6PM chip consists of a 256 KB of Flash memory and 32 KB of SRAM. The following figure and table shows the memory map of a TM4C123GH6PM chip with addresses.

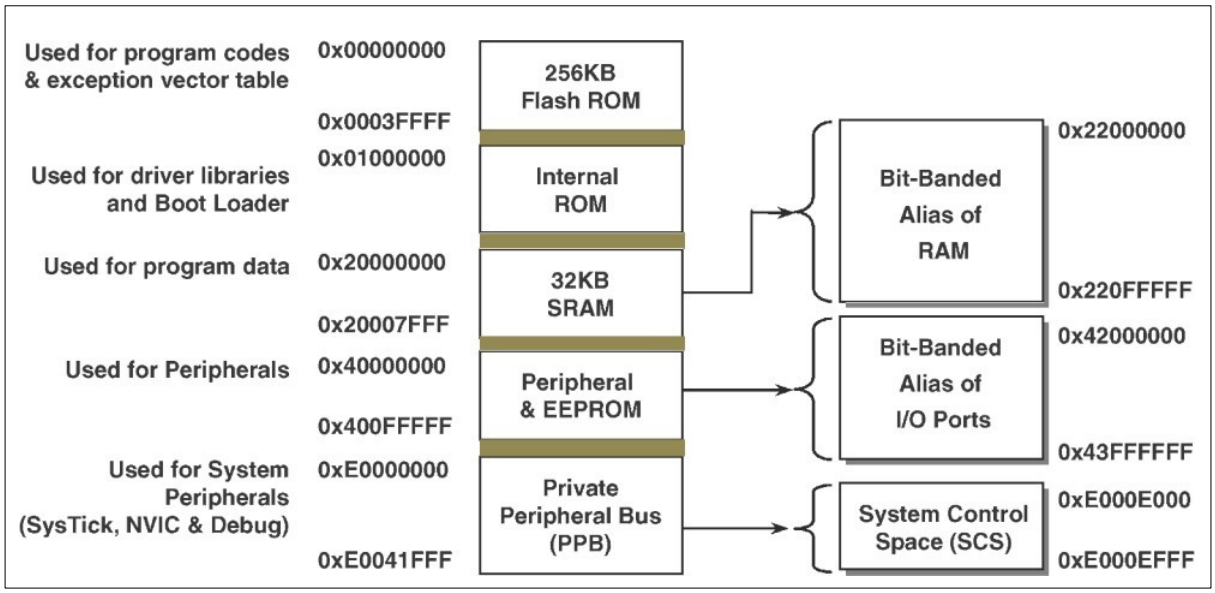


Figure: Memory Mapping in TM4C123GH6PM Chip

Flash Memory:

Flash memory is structured into multiple blocks of single KB size which can be individually written to and erased. Flash memory is used for store program code. Constant data used in a program can also be stored in this memory. Lookup tables are used in many designs for performance improvement. These lookup tables are stored in this memory.

Table: Memory Mapping in TM4C123GH6PM Chip

Memory Type	*Allocated size*	*Allocated address*
Flash	256KB	0x00000000 to 0x0003FFFF
Bit-banded on-chip SRAM	32 KB	0x20000000 to 0x20007FFF
Peripheral	All the peripherals	0x40000000 to 0x400FFFFF

SRAM:

The on-chip SRAM starts at address 0x2000.0000 of the device memory map. ARM provides a technology to reduce occurrences of read-modify-write (RMW) operations called bit-banding. This technology allows address aliasing of SRAM and peripheral to allow access of individual bits of the same memory in single atomic operation. For SRAM, the bit-band base is located at address 0x2200.0000. Bit band alias are computed according to following formula.

Bitband alias= bitband base + byte offset *32 + bit number *4

Note: Bit banding is the technique to access and modifying content of bits in a register. It is helpful to finish the read-modify operation in single machine cycle.

The region of the memory which device consider for modification is known as bit band region and the region of memory to which device maps the selected memory is known as bit band alias.

The SRAM is implemented using two 32-bit wide SRAM banks (separate SRAM arrays). The banks are partitioned in a way that one bank contains all, even words (the even bank) and the other contains all odd words (the odd bank). A write access that is followed immediately by a read access to the same bank. This incurs a stall of a single clock cycle.

Internal ROM:

The internal ROM of the TM4C123GH6PM device is located at address 0x0100.0000 of the device memory map. The ROM contains:

- TivaWare™ Boot Loader and vector table
- TivaWare™ Peripheral Driver Library (DriverLib) release of product-specific peripherals and interfaces
- Advanced Encryption Standard (AES) cryptography tables

- Cyclic Redundancy Check (CRC) error detection functionality

The boot loader is used as an initial program loader (when the Flash memory is empty) as well as an application-initiated firmware upgrade mechanism (by calling back to the boot loader). The Peripheral Driver Library, APIs in ROM can be called by applications, reducing flash memory requirements and freeing the Flash memory to be used for other purposes (such as additional features in the application). Advance Encryption Standard (AES) is a publicly defined encryption standard used by the U.S. Government and Cyclic Redundancy Check (CRC) is a technique to validate if a block of data has the same contents as when previously checked.

Peripheral:

All Peripheral devices, timers, and ADCs are mapped as MMIO in address space 0x40000000 to 0x400FFFFF. Since the number of supported peripherals is different among ICs of ARM families, the upper limit of 0x400FFFFF is variant.

Private Peripheral Bus (PPB):

Private Peripheral Bus is used for System Peripheral like System Timer (SysTick), Nested Vectored Interrupt Controller (NVIC), System Control Block (SCB), Memory Protection Unit (MPU), & Floating Point Unit (FPU).

On-Chip Peripherals (Analog and Digital):

SYSTEM PERIPHERALS:

- Watchdog Timer
- Hibernation Module
- DMA
- GPIOs
- Timers
- EEPROM

SERIAL PERIPHERALS:

- UART
- I2C
- CAN
- USB OTG
- SPI/SSI

ANALOG PERIPHERALS:

- Analog Comparator
- 12-bit ADC

MOTION CONTROL PERIPHERALS:

- PWM
- QEI

Watchdog Timer:

Every CPU has a system clock which drives the program counter. In every cycle, the program counter executes instructions stored in the flash memory of a microcontroller. These instructions are executed sequentially. There exist possibilities where a remotely installed system may freeze or run into an unplanned situation which may trigger an infinite loop. On encountering such situations, system reset or execution of the interrupt subroutine remains the only option. Watchdog timer provides a solution to this.

The primary function of the Watchdog Timer is to perform a controlled system restart after a software problem occurs. If the selected time interval expires, a system reset is generated.

Hibernation Module:

This module manages to remove and restore power to the microcontroller and its associated peripherals. This provides a means for reducing system power consumption. When the processor and peripherals are idle, power can be completely removed if the Hibernation module is only the one powered.

To achieve this, the Hibernation (HiB) Module is added with following features:

(i) A Real-Time Clock (RTC) to be used for wake events

(ii) A battery backed SRAM for storing and restoring processor state. The SRAM consists of 16 32-bit word memory.

DMA:

Direct Memory Access is a way of streamlining transfers of large blocks of data between two different segments of memory or between an I/O device and memory. Reading from disk and store it in memory is a kind of operation we talking about. For this we may prefer either of two mentioned below:

- The processor can read each byte at a time from the memory into a register, then store the contents of the register to the suitable memory location. For each byte,
  - processor must read an instruction,
  - instruction decoding,
  - read the data,
  - execute read for next part of instruction,
  - decode the instruction,
  - Store the data.

  Then the process starts over again for the next byte.

The second option is a special device, called a DMA controller (DMAC), performs high-speed transfers between memory and I/O devices. It is typically used in moving large sized data clusters around the system. Using DMAC, we can bypass the processor by creating a channel between the memory and the I/O device. Thus, data is read from the I/O device and written into memory without executing the code to perform the transfer on a byte-by-byte basis.

Programmable GPIOs:

General-purpose input/output (GPIO) pins offer flexibility for a variety of connections. The TM4C123GH6PM GPIO module is comprised of six physical GPIO blocks, each corresponding to an individual GPIO port. The GPIO module is FiRM-compliant (compliant to the ARM Foundation IP for Real-Time Microcontrollers specification) and supports 0-43 programmable input/output pins. The number of GPIOs available depends on the peripherals being used.

- Up to 43 GPIOs, depending on configuration
- Highly flexible pin muxing allows use as GPIO or one of several peripheral functions
- Fast toggle capable of a change every clock cycle for ports on Advanced High-Performance Bus (AHB), every two clock cycles for ports on Advanced Peripheral Bus (APB)
- Programmable control for GPIO interrupts
  - Interrupt generation masking
  - Edge-triggered on rising, falling, or both
  - Level-sensitive on High or Low values
- Bit masking in both read and write operations through address lines
- Can be used to initiate an ADC sample sequence or a μDMA transfer
- Pin state can be retained during Hibernation mode

Timers:

Timers are basic constituents of most microcontrollers. Today, just about every microcontroller comes with one or more built-in timers. These are extremely useful to the embedded programmer – perhaps second in usefulness only to GPIO. The timer can be described as the counter hardware and can usually be constructed to count either regular or irregular clock pulses. Depending on the above usage, it can be a timer or a counter respectively.

The TM4C123GH6PM General-Purpose Timer Module (GPTM) contains six 16/32- bit GPTM blocks and six 32/64-bit Wide GPTM blocks. These programmable timers can be used to count or time external events that drive the Timer input pins. Timers can also be used to trigger μDMA transfers, to trigger analog-to-digital conversions (ADC) when a time-out occurs in periodic and one-shot modes.

Serial Communications Peripherals:

The TM4C123GH6PM controller supports both asynchronous and synchronous serial communications with:

- Two CAN 2.0 A/B controllers
- USB 2.0 OTG/Host/Device
- Eight UARTs
- Four I2C modules with four transmission speeds including high-speed mode
- Four Synchronous Serial Interface modules (SSI)

Analog Comparators:

An analog comparator is a peripheral that compares two analog voltages and provides a logical output that signals the comparison result. The TM4C123GH6PM microcontroller provides two independent integrated analog comparators that can be configured to drive an output or generate an interrupt or ADC event.

The comparator can provide its output to a device pin, acting as a replacement for an analog comparator on the board, or it can be used to signal the application via interrupts or triggers to the ADC to cause it to start capturing a sample sequence. The interrupt generation and ADC triggering logic is separate. This means, for example, that an interrupt can be generated on a rising edge and the ADC triggered on a falling edge.

Analog to Digital Converter (ADC):

ADCs are peripherals that convert a continuous analog voltage to a discrete digital number. In order to convert to digital, the signal is sampled at higher frequencies to minimize the signal loss. Then the amplitude at those sampled moments is converted with respect to their quantization level. Finally these levels and moments are entitled to a unique code, which are simply the combinations of 0‟s and 1‟s – this is called encoding.

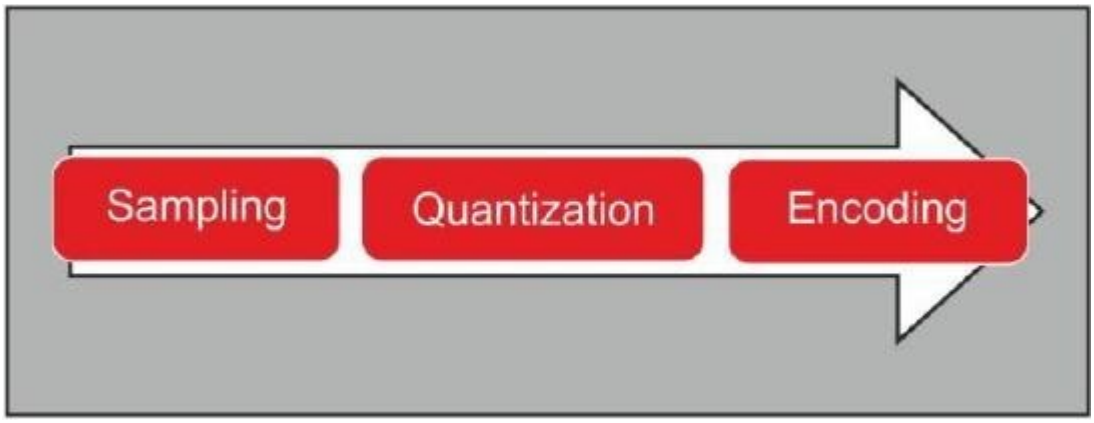


Figure: Block Diagram of working of an ADC

Pulse width modulation (PWM):

Pulse width modulation (PWM) is a simple but powerful technique of using a rectangular digital waveform to control an analog variable or simply controlling analog circuits with a microprocessor's digital outputs. PWM is employed in a wide variety of applications, from measurement & communications to power control and conversion.

Quadrature Encoder Interface (QEI):

A quadrature encoder, also known as a 2-channel incremental encoder, converts linear displacement into a pulse signal. By monitoring both the number of pulses and the relative phase of the two signals, you can track the position, direction of rotation, and speed. In addition, a third channel, or index signal, can be used to reset the position counter.

A classic quadrature encoder has a slotted wheel like structure, to which a shaft of the motor is attached and a detector module that captures the movement of slots in the wheel.

Register Set:

Registers are for temporary data storage within processor architecture. As shown in Figure, ARM processor has sixteen numbers of general purpose registers, R0-R15 and a current program status register (CPSR) defined for user mode of operation. Each of these registers is of 32-bits. Out of these registers, R13, R14 and R15 have special purposes.

R13: Used as the stack pointer that holds the address of the top of the stack in the current processor mode.

R14: Used as the link register that saves the content of program counter on control transfer due to the occurrence of exceptions or using the branch instructions in the program.

R15: Used as the program counter that points to the next instruction to be executed. In ARM state, all instructions are of 32-bits (four bytes) for which, PC is always aligned to a word boundary. This means that the least significant two bits of the PC are always zero. The PC can also be half word (16-bit) aligned for Thumb state (16 bit instructions) or byte aligned for Jazelle state (8-bit instructions) supported by different versions of ARM architecture.

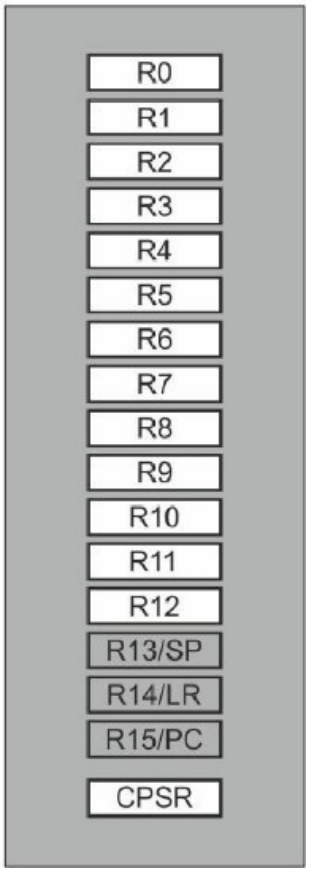


Figure: User Mode Register Set

Current Program Status Register (CPSR):

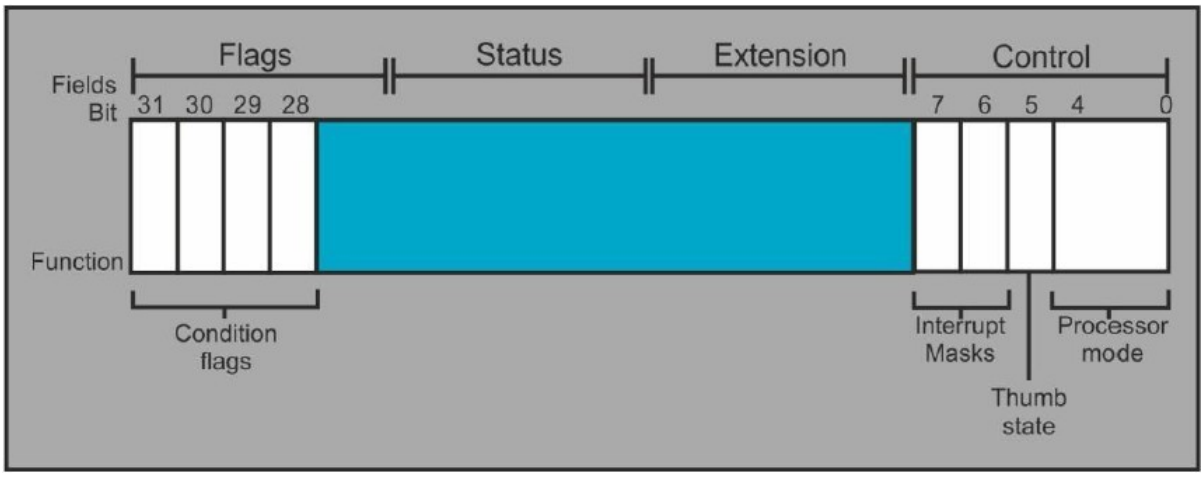


Figure: A Generic CPSR Format

CPSR, a 32-bit status register, holds the current state of the ARM core. As shown in Fig 1.4, the register is divided into four different fields- flags, status, extension and control; each of 8-bits. The flag field has the bit specification for four condition flags; N, Z, C and V and is used for arithmetic and logic instructions.

- N-(Negation flag) = 1 indicates negative result from ALU.
- Z- (Zero flag) = 1 indicates zero result from ALU.
- C- (Carry flag) = 1 indicates ALU operation generated carry.
- V- (Overflow flag) =1 indicates ALU operation overflowed.

Most of the ARM instructions are conditionally executed. Based on the status of these condition flags, condition codes are used along with instruction mnemonics to control whether or not the instruction will be executed. Status and extension fields are reserved for future usage. In the control field, the least significant five bits are used to save the modes of operation of ARM core. Processor mode can be changed by directly modifying these control bits. The most significant three bits I, F and T have significance as below:

- I = 1 indicates IRQ is
  disabled 0 indicates IRQ is
  enabled.
- F = 1 indicates FIQ is disabled
  0 indicates FIQ is enabled.
- T = 1 indicates the Thumb state is active.
  0 indicates ARM state is active.

Operating Modes

ARM core has seven operating modes basically used to isolate users programs from the protected memory or OS services. The operating modes are: user, system, fast interrupt request (FIQ), interrupt request (IRQ), abort, supervisor and undefined mode. Out of these, only user mode is unprivileged, remaining six are privileged modes. The basic difference between privileged and unprivileged mode is the access permission to protected area of the memory and write access permission to CPSR_c given to only privileged modes. All

application programs run in user mode. All operating system kernel functions and services run in system or supervisor mode. After reset, core enters to supervisor mode. FIQ mode is for interrupt requesting faster response and low latency and IRQ mode correspond to the low priority interrupt available on the processor itself. Processor enters abort mode to handle memory access violation. In the execution flow, when processor encounters an instruction that is not supported by the instruction set implementation, it enters to undefined mode. All exceptions are handled in privileged modes. Privileged modes have complete read and write access to both flags and control fields but unprivileged user mode has only read access to the control field while both read and write access to the flags field. Processor mode is changed automatically by the occurrence of exceptions or by modifying the control bits of CPSR by writing its binary pattern as shown in below table, being in a privileged mode.

- User : unprivileged mode under which most tasks run
- FIQ : entered when a high priority (fast) interrupt is raised
- IRQ : entered when a low priority (normal) interrupt is raised
- Supervisor : entered on reset and when a Software Interrupt instruction is executed
- Abort : used to handle memory access violations
- Undef : used to handle undefined instructions
- System : privileged mode using the same registers as user mode

Table: Processor mode with binary Pattern mode Control bits [4:0]

Abort (abt)	10111
Fast interrupt request(fiq)	10001
Interrupt request (irq)	10010
Supervisor(svc)	10011
System (sys)	11111
Undefined(und)	11011
User(usr)	10000

Programming Model:

Programming model of a processor is basically a set of working registers used to perform the operations defined in its instruction set. ARM programming model has total 37 registers in its register bank which are segmented for different modes of operation as shown in below figure. User mode register set is shared by the system mode also.

Each of the remaining privileged modes has a set of banked registers which are active and accessible to the programmer only when the core enters to the corresponding mode. Banked registers for a particular mode are physical replication of few of the user mode registers along with a saved program status register (SPSR) shown by shading in the figure.

If the processor mode is changed, for example from user to FIQ mode due to occurrence of hardware interrupt (fiq), the banked registers R8-R14 from the FIQ mode will replace the corresponding registers in user mode but the remaining user mode registers (R0- R7) can still be used in FIQ mode after saving the previous contents.

It means registers R8-R14 of user mode are unaffected by this mode change. The purpose of these banked registers is to reduce the context saving overhead. There is only one dedicated PC (R15) and one CPSR for all the operation modes.

When a mode is changed, the PC and CPSR contents are saved in the link register (R14) and SPSR of the new mode respectively. While returning back to previous mode, special instructions are used to restore back the saved register contents. There is no SPSR available in user mode and one important feature is that, when a mode change is forced, CPSR content is not saved in SPSR. It happens only when exception occurs.

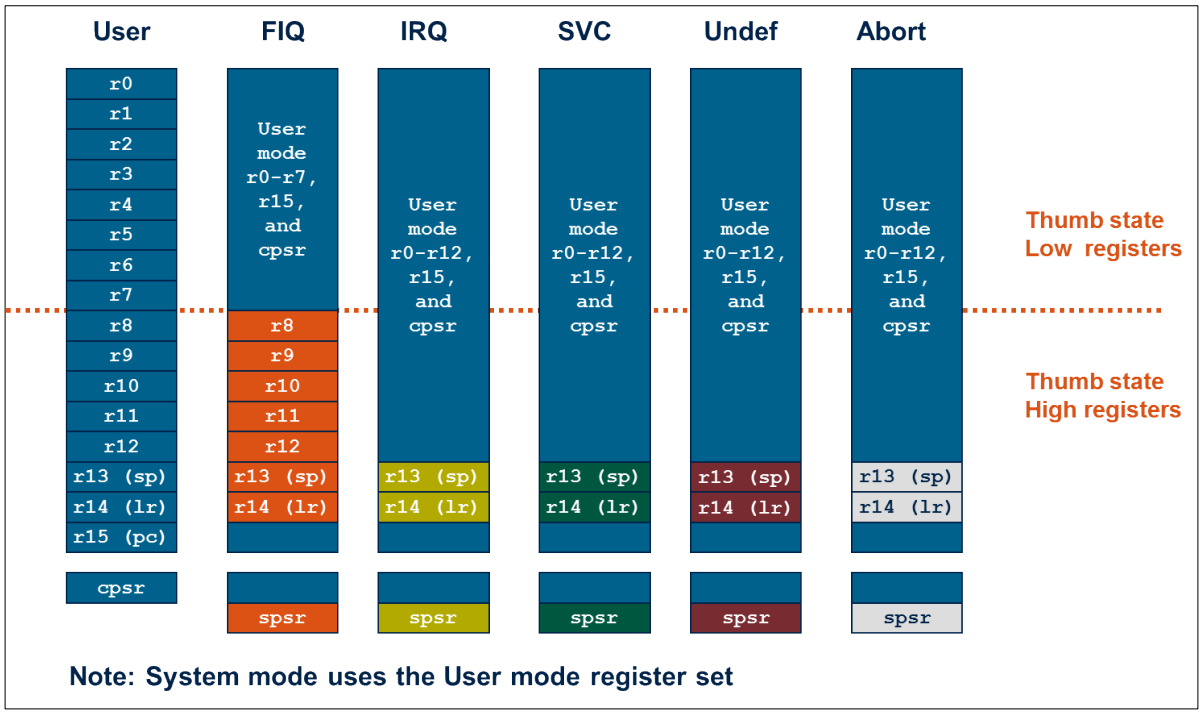


Figure: Complete Register Bank

Addressing Modes:

Addressing mode is the way of addressing data or operand in the instruction. Every processor instruction set offers different addressing modes to determine the address of operands. Some fundamental addressing modes used by most of the processors are: register addressing, immediate addressing, direct addressing and register indirect addressing. In register addressing mode, the operand is held in a register which is specified in the instruction. In immediate addressing mode, the operand is held in the instruction. In direct addressing mode, the operand resides in the memory whose address is specified in the instruction. Similarly in register indirect addressing mode, the operand is held in the memory whose address resides in a register that is specified in the instruction.

The ARM supports the following addressing modes:

1) Register Addressing Mode
2) Relative Addressing Mode
3) Immediate Addressing Mode
4) Register Indirect Addressing
5) Register Offset Addressing Mode
6) Register based with Offset Addressing Mode
   - Pre-Indexed Addressing
   - Pre-Indexed with write back
   - Post-Indexed

1) Register Addressing: The operands are in the registers.

Ex: MOV R1, R2 // move content of R2 to R1
SUB R0, R1, R2 //subtract content of R2 from R1 and move the result to R0

2) Relative Addressing: Address of the memory directly specified in the instruction.

Ex: BEQ LOOP // branch to LOOP if previous instruction sets the zero flag i.e., Z=1

3) Immediate Addressing: Operand2 is an immediate value.

Ex: SUB R0, R0, #1 // Save (R0 –1) to R0
MOV R0, #0xFF00 // Put 0xFF00 to R0

4) Register Indirect Addressing: Address of the memory location that holds the operands there in a register.

Ex: LDR R1, [R2] //Load R1 with the data pointed by register R2.
ADD R0, R1, [R2] //add R1 with the data pointed by R2 and put the result into R0

5) Register Offset Addressing: Operand2 is in a register with some offset calculation.

Ex: MOV R0, R2, LSL #3 // (R2 << 3), then move to R0
AND R0, R1, R2, LSR R3 // (R2 >> R3), logically AND with R1 and move result to R0

6) Register based with Offset Addressing: Effective memory address has to be calculated from a base address and an offset. Offset can be an immediate offset, register offset or scaled register offset.

1) Pre-Indexed Addressing

Ex: LDR R2, [R3, #08] // Take value in R3, add to 08, use it as address and load data from that address to R2
STR R1, [R0, -R2] // Register offset // Use (R0-R2) as address of the memory and store data of R1 to that address.
LDR R3, [R1, R2 LSR #8] // Scaled register offset // Use (R1+ (R2>>8)) as address and load the data from that address to R3.

2) Pre-Indexed with write back also called auto-indexing with pre-indexed addressing. Symbol indicates that the instruction saves the calculated address in the base address register.

Ex: LDR R0, [R1, #4]! // Immediate offset // Use (R1+4) as address and load the data from that address to R0 and update R1 by (R1+4)
STR R1, [R2, R0]! // Register offset // Use (R2+R0) as address and store the data from R1 to that address. Update R2 by (R2+R0)
STR R3, [R1, R2 LSL #4]! // Scaled register offset // Use (R1+ (R2<<4)) as address and store the data from R3 to that address. Update R1 by (R1+ (R2<<4))

3) Post-Indexed also called auto-indexing with post-indexed addressing.
Ex: LDR R0, [R1], #4 // Immediate offset // Load the data pointed to by R1 to R0 and then update R1 by (R1+4).
STR R1, [R3], R4 // Register offset // Store the data in R1 to the memory location pointed to by R3 and then update R3 by (R3+R4)
LDR R2, [R0], -R3, LSR #4 // Scaled register offset // Load the data from the address pointed to by R0 to R2 and then update R0 to (R0-(R3>>4)).

Instruction Set Basics:

In any processor architecture, an instruction includes an opcode that specifies the operation to perform, such as add contents of two registers or move data from a register to memory etc., with specified operands, which may specify registers, memory locations, or immediate data.

ARM architecture has two instruction sets. The ARM instruction set and Thumb instruction set. In ARM instruction set, all instructions are 32 bits wide and are aligned at 4- bytes boundaries in memory. On the other hand, in thumb instruction set, all instructions are of 16 bits wide and are aligned at even or two bytes boundaries in memory.

ARM Instructions can be categorized into following broad classes:

1) Data movement instructions
2) Data Processing Instructions
   - Arithmetic/logic Instructions
   - Barrel shifting instructions
   - Comparison Instructions
   - Multiply Instructions
3) Branch Instructions
4) Load and store Instructions
   - Load and Store register instruction
   - Load and Store multiple register instructions
   - Stack instructions
   - Swap register and memory content
5) Program Status register Instructions
   - Set the values of the conditional code flag
   - Set the values of the interrupt enable bit
   - Set the processor mode

6) Exception generating Instructions

- ¬ Software Interrupt Instruction
- ¬ Software Break Point instruction

Table: Instruction Set Table

Instruction Mnemonic	Description	Example	Working
1) Data Movement instructions			
Syntax: <instruction>{<condition>}{S} Rd, N			
MOV	Move a 32-bit value into a register	MOV r1, r2, LSL #4	Move (r2<<4) to r1.
MVN	Move the NOT of the 32-bit value into a register	MVN r1, r3	Move (~ r3) to r1.
2) Data Processing instructions			
i) Arithmetic Instructions; Syntax:<instruction>{<cond>}{S} Rd, Rn, N			
ADC	Add two 32-bit values and carry	ADC r1, r2, r3	r1= r2+r3+Carry
ADD	add two 32-bit values	ADD r4, r5, r3, LSR # r1	r4= r5+ (r3>> by r1)
RSC	Reverse subtract with carry of two 32- bit values	RSC r3, r2, r1	r3= r1- r2 - ! Carry
RSB	Reverse subtract of two 32-bit values	RSB r3, r2, r1	r3= r1- r2
SBC	Subtract with carry of two 32-bit values	SBC r2,r4, r6	r2=r4-r6- !Carry
SUB	Subtract two 32-bit values	SUB r2,r4, r6	r2=r4-r6
ii) Logical Instructions; Syntax:<instruction>{<cond>}{S} Rd, Rn, N			
AND	logical bitwise AND of two 32-bit values	AND r7, r5, r2	r7= r5 & r2
ORR	logical bitwise OR of two 32-bit values	ORR r6, r4, r1, LSR r2	r6= r4 | (r1>>r2)
EOR	logical exclusive OR of two 32-bit values	EOR r5, r1, r2	r5= r1 ^ r2
BIC	logical bit clear (AND NOT)	BIC r3, r1,r4	r3= r1 & ~ r4
iii) Comparison Instructions; Syntax:<instruction>{<cond>} Rn, N			
CMN	Compare negated	CMN r1, r2	Flags set as results of r1+r2
CMP	Compare	CMP r1, # 0XFF	Flags set as results of r1- 0XFF
TEQ	Test for equality of two 32-bit values	TEQ r3, r5	Flags set as results of r3 ^ r5
TST	Test bits of a 32-bit values	TST r1, r2	Flags set as results of r1& r2
iv) Multiply Instructions; Syntax:MLA{<cond>}{S} Rd, Rm, Rs, Rn; MUL{<cond>}{S} Rd, Rm, Rs			
MLA	Multiply and accumulate	MLA r1,r2,r3,r4	r1=(r2*r3)+r4
MUL	Multiply	MUL r3, r7, r6	R3= r7*r6

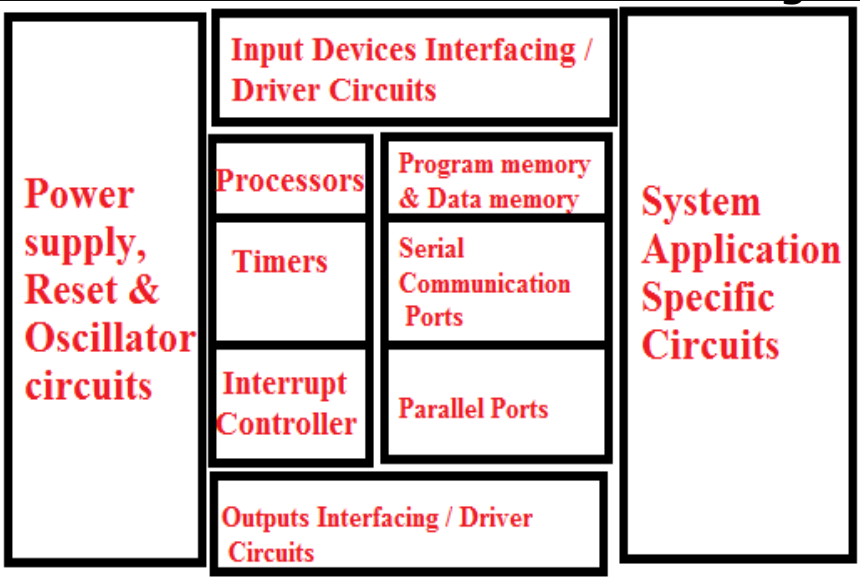
Instruction Mnemonic	Description	Example	Working
3) Branch instructions			
Syntax: B{<cond>} label; BL{<cond>} label; BX{<cond>} Rm; BLX{<cond>} label | Rm			
B	Branch	B label	PC= label
BL	Branch with link	BL label	PC=label and Lr= Address of the next instruction after BL.
BX	Branch exchange	BX r5	PC=r5 & 0Xfffffffe and T= r5 & 1
BLX	Branch exchange with link	BLX r6	PC=r6 & 0Xfffffffe, T=r6 & 1 and lr= address of the next instruction after BLX.
4) Load/Store Instructions			
i) Single register transfer; Syntax:<LDR|STR>{<cond>} Rd, Address			
LDR	Load register from memory	LDR r0, [r2, #0X8]	Load r0 with the content of memory address pointed to by [r2+0X8]
STR	Store register to memory	STR r1, [r4], #0X10	Store r1 into the memory address pointed to by r4 and update r4 by [r4+0X10]
ii) Multiple register transfer; Syntax:<LDM|STM>{<cond>}<addressing mode> Rn{!},{registers}; Addressing modes: IA-Increment after; IB-Increment before; DA-Decrement after; DB-Decrement before:- Increment or decrement the memory pointer after or before the data transfer.			
LDM	Load multiple registers from memory	LDMIA r6!, {r2-r4}	r2=[r6]; r3= [r6+4]; r4=[r6+8] and update r6 by [r6+12]
STM	Store multiple registers to memory	STMDB r1!, {r3-r5}	[r1-4]=r5 [r1-8]=r4 [r1-12]=r3 and update r1 by [r1-12}
iii) Stack Operations ; Syntax:<LDM|STM><addressing mode> SP{!},{registers}; Addressing modes: FA-Full ascending; FD-Full descending; EA-Empty ascending; ED –Empty descending;			
LDM	Load multiple registers from stack memory	LDMED Sp!, {r1, r3}	r1= [Sp+4] r2= [Sp+8] r3= [Sp+12] and Sp is updated by [Sp+12]

STM	Store multiple registers to stack memory	STMFD Sp!, {r4,r6}	[Sp-4]= r6 [Sp-8]= r5 [Sp-12]=r4 and Sp is updated by [Sp-12]
iv) Swap instruction ; Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]			
SWP	swap a word between memory and a register	SWP/SWPB r0, r1, [r2]	Load a 32 bit word or a byte from the memory address in r2 into r0 and store the data in r1 to the memory address in r2.
SWPB	swap a byte between memory and a register		
5) Program status register instructions			
MRS{<cond>} Rd,<cpsr|spsr>;MSR{<cond>} <cpsr|spsr>_<fields>,Rm MSR{<cond>} <cpsr|spsr>_<fields>,#immediate			
MRS	Move the content of cpsr or spsr to a register.	MOV r1, CPSR	Move the content of CPSR register to r1.
MSR	Move an immediate data or register to a specific field of cpsr or spsr.	MSR CPSR_f, r1	Update the flag field of CPSR by the content in r1.
6) Exception generating instructions			
Software interrupt instruction ; Syntax: SWI{<cond>} SWI_number (immediate 24 bit)			
SWI	Software interrupt for an operating system routine. Change to Supervisor mode. CPSR is saved in SPSR. Control branches to interrupt vector.	SWI 0X123456	Execute software interrupt at 0X123456 in ARM state of the core. T =0 in CPSR.

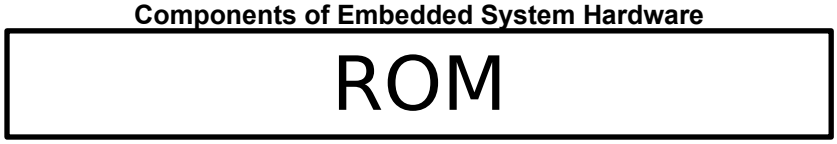
UNIT-III

Overview of Microcontroller and Embedded Systems

3.1 Embedded Hardware and Various Building Blocks:



Components of Embedded System Hardware



- Power Supply
- Oscillator
- Timing Devices
- Inputs, Outputs and I/O ports
- ADC & DAC
- LCD, LED and Touch Screen
- Keypad/Keyboard
- Interrupt Controller

1. Power Source:

- Various units in an embedded system operate in one of the ranges 5.0V±0,25 V. 3.3 V ±0.3V 2.0 V ±0,2 V and 1.5V± 0 2V.

2. Clock Oscillator and Clocking Unit(s)

- The clock controls the time for executing an instruction.
- A processor needs a clock oscillator circuit
- *The clock controls the various clocking requirements of the CPU, of the system timers and the CPU Machine cycles.*

3. System Timer

- A timer circuit is suitably configured and functions as system clock.
- The system clock ticks and generates system-interrupts periodically.
- The system-clock interrupt enables execution of the system supervisory functions in the OS at the periodic intervals.
- System clock ticks can be 60 times in second.

4. Real-Time Clock (RTC):

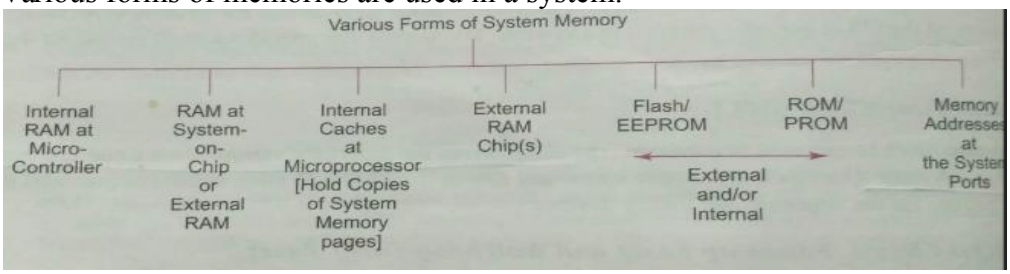
- A real-time clock is required in a system.
- The clock drives the timers for various timing and counting needs in a system.
- The clock also updates time and date in the system.
- A microcontroller provides the internal timer circuits for the counting and timing devices.

5. Reset Circuit, Power-up Reset and Watchdog-Timer Reset

- A circuit for reset enables restart of the system from the beginning using a switch or signal.
- The reset can also be performed by using an instruction to the processor.
- A power-up reset circuit enables restart of the system from beginning whenever power is switched on in the system.
- A **watchdog-timer** reset enables the restart of system when it is stuck up in certain set of instructions for a period more than preset time-interval.
- Reset based on reset-switch, reset-instruction or power-up or power-up reset and watchdog reset can be from same starting instruction or the different starting instructions.

6. Memory

- Various forms of memories are used in a system.



- (a) Internal RAM of 256 or 512 bytes in a MC for registers, temporary data and stack
- (b) Internal ROM/PROM/EPROM -4KB to 64KB of program
- (c) External RAM for the temporary data and stack (in most systems)
- (d) Internal caches (in pipelined and superscalar microprocessors)
- e) Internal EEPROM or flash
- (I) Memory Stick (card): video, images, songs, or speeches and large storage in digital camera, mobile systems
- (g) External ROM or PROM for embedding soft are (in almost all other than microcontrollers-based systems)

- (h) RAM Memory buffers at the ports

ROM or EPROM of Flash	Storing Application Programs
RAM(Internal and External) and RAM for Buffers	Storing Variables during program run and Storing the Stack Storing I/O buffers
Memory Stick	Inserted in Mobile computing system or Digital Camera
EEPROM or Flash	Storing Nonvolatile Results of Processing

Cache	Storing copies of Instructions and Data in advance from External Memory

- A system embeds(locates) the following either in the internal ROM, PROM or in an external ROM or PROM of microcontroller; boot-up programs, initialization data, strings for an initial screen-display or initial state of the system, the programs for various tasks, ISRs and operating system kernel.
- The system has RAMs for saving temporary data, stack and buffers that are needed during a program run.
- The system also has flash for storing non-volatile results.

7. Input, Output and I/O Ports, I/O Buses and I/O interfaces

- The system gets inputs from physical devices through the input ports
- A processor identifies each input port by its memory buffer address, called port address.
- Just as a memory location holding a byte or word is identified by an address, each input port is identified by the address.
- The system gets the inputs by the read operations at the port addresses.
- The system has output ports through which it sends output bytes to the real world.
- Each output port is identified by its memory-buffer address(es) called port address.
- The system sends the output by a write operation to the port address.
- There are also general-purpose ports for both the input and output (I/O) operations.

8. Bus

- Processor of a system might have to be connected to a number of other devices and systems.
- A bus consists of a common set of lines to interconnect the multiple devices, hardware units and systems.
- It enables the communication between two units at any given instance.
- The remaining units remain in an in connected state during communication between these two.
- A bus communication protocol specifies the ways of communication of signals on the bus.
- At any instance, a bus may be a serial bus or a parallel bus transferring one bit or multiple data bits respectively.
- Protocol also specifies ways of arbitration when several devices need to communicate through the bus.
- Alternatively, protocol specifies ways of polling from each device for need of the bus at an instance.
- Protocol also specifies ways of daisy chaining the devices so that, at an instance, the bus is -anted to a device according to the device priority in the chain.

9. Digital to Analog conversion (ADC):

- DAC is a circuit that converts digital 8, 10 r 12 bits to analog output.
- A DAC operation is done with the help of a combination of the Pulse Width Modulation (PWM) unit in a microcontroller and an external integrator chip.

10. Analog to Digital Conversion (ADC):

- ADC is a circuit that converts the analog input to digital.
- The output is of 4, 8 10 or 12 bits from an ADC.
- Analog input is applied between + and – pins.
- ADC circuit converts them into bits.
- The converted bits value depends on the reference voltage.
- When input +ve and -ve pins are at voltage equal to reference +ve and -ve voltage pins, then all output bits = 1.
- When the difference in voltage at inputs +ve and -ve pins = 0v then all output bits = 0.
- An ADC unit in the embedded system microcontroller may have multi-channels.
- It can then take the inputs in succession from the various interconnected to different analog sources.

11. LED, LCD and Touch-Screen Displays:

- A system requires an interfacing circuit and software to display.
- LED is used for indicating ON status of the system.
- A flashing LED may indicate that, a specific task is under completion or is running.
- It may indicate a wait status for a message.
- The display may show the status or message.
- Display may be a line display, a multiline display or a flashing display.
- An LSI (Lower scale integrated circuit) is used as display controller in case of the LCD matrix display.
- Touch screen is an input as well as output device, which is used by the user of a system to enter a command, choose a menu or to give user reply as input.
- The input is on physical touch as a screen position.
- The touch at a position is mostly by the finger or some times by stylus.
- Stylus is a thin pencil shaped long object.
- It is held between the fingers and used just as a pen to mark a dot.
- An LSI (Lower scale integrated circuit) functions as touch-screen controller.
- The display-screen display is similar to a computer display unit screen.

12. Keypad, Keyboard or Virtual Keypad at touch screen:

- The keypad or keyboard, is an important device for getting user inputs.
- A touch screen provides virtual keypad in a mobile computing system.
- Virtual keypad is a keypad displayed on displayed on the LCD display screen on touch plate.
- A user can enter the inputs using touches.
- A tile is displayed for a command on the LCD display screen.
- User can enter the command using touch at the tile.
- A keypad or keyboard may interface to a system.
- The system may provide necessary interfacing circuit and software to receive inputs directly from the keys or touch screen controller

13. Interrupt Handler:

- A system may process a number of devices.
- The system processor controls and handles the requirements of each device by running an appropriate ISR for each interrupting event, An interrupts-handling mechanism must exist in each system.
- It handles interrupts from various events or processors in the system.
- The system handled multiple interrupts, which may be simultaneously pending for service.

3.2 Processor Selection for an Embedded System:

A Hardware Designer must take into account following processor Specific-features

- Should operate at higher clock speed for processing more instructions per second.
- High computing performance of computing when there exist (a) Pipeline(s) and superscalar architectures, (b) Pre-fetch Cache Unit, caches, and register-files and MMU and (c) RISC architecture.
- Register-windows provides fast context switching in a multitasking system.

Suppose, a **low-priority task** is presently being executed by the processor but a **high-priority task** to run.
In this case, CPU will be interrupted through an **interrupt signal.**
The **CPU** will **save** the current tasks information in a stack and **execute** the high priority task.
The mechanism of storing the current CPU registers in a stack to run the other task is known as **context switching.**

- Power-efficient embedded system requires a processor that has auto-shut down feature for its units and programmability for the disabling these use of these when the processing need for a function or instruction set does not have constraint of execution time. Processor uses Stop, Sleep and Wait instructions and special cache design.
- Burst mode accesses external memories fast, reads fast and writes fast.
- Atomic operation unit provides hardware solution to shared data problem when designing embedded software, else special programming skill and efforts are to be made when sharing the variables among the multiple tasks

Processor or Microcontroller Version Selection

- The processor and microcontroller selection process needs the following parameters:

1. Processor instruction cycle in Micro seconds
2. Internal bus width in bits
3. CISC or RISC architecture
4. Pipeline and superscalar architectures
5. On-chip RAM and/or register file sets
6. Instruction cache
7. Program memory EPROM/EEPROM/Flash

8. Program memory capacity in bytes
9. Data/Stack memory capacity in bytes
10. Main memory Harvard or Princeton architecture
11. External interrupts
12. Bit manipulation instructions
13. Floating point processor
14. Interrupt controller
15. DMA controller channels
16. On chip MMU

Capability	Intel 8051 and Intel 8751	Motorola M68HC11 E2	Intel 80196KC	Intel Pentium
Processor instruction cycle in μs (typical)	1	0.5	0.5	0.01[6]
Internal Bus Width in Bits	8	8	16	64
CISC or RISC Architecture	CISC	CISC	CISC	CISC with RISC feature[5]
Program Counter bits with reset value	16 (0x0000)	16 [(0xFFFE)]	16 (0x2080)	32 (0xFFFF FFFF)
Stack Pointer bits with initial reset value in case a processor defines these.	8 (0x07)	16	16	32
Atomic Operations Unit	No	No	No	No
Pipe-line and Super- scalar Architecture	No	No	No	Yes
On-Chip RAM and/or Register file Bytes	128 & 128 RAM	512 RAM	256 & 232	.....
Instruction Cache	No	No	No	8 kB^
Data Cache	No	No	No	8 kB^
Program memory EPROM/EEPROM	4 k	8k	8 k	No
Program memory capacity in Bytes	64 k[1]	64k	64 k	4 GB
Data/ Stack Memory Capacity in Bytes	64 k[1]	64k	64 k	4 GB
Main Memory Harvard or Princeton Architecture	Harvard	Princeton	Princeton	Princeton
External Interrupts	2	2	2	1[#]
Bit Manipulation Instructions	Yes	Yes	Yes	Yes
Floating Point Processor	No	No	No	Yes
Interrupt Controller	No	No	No	Yes
DMA Controller Channels	No	No	1 (PTS)*	4
On - Chip MMU	No	No	No	Yes

Microcontrol

Capability	ARM 7	Intel 80960CA	PowerPC MPC 604
Processor instruction cycle in ms (typical)	0.012	0.03	0.0016
Internal Bus Width in Bits	32	32	64
CISC or RISC Architecture	Both[b]	Both[b]	RISC
Program Counter bits	32	32	32
Stack Pointer bits	32	32	32
Atomic Operations Unit	No	Yes	No
Super- scalar Architecture	Yes	Yes	Yes
On-Chip RAM and/or Register file Bytes	16 kB	1536B RAM	32 kB
Instruction Cache	16 kB[5]	1 kB	16 kB
Data Cache	16 kB[5]	No	16 kB
Program memory EPROM/EEPROM	-	—	...
Program Physical memory capacity in Bytes	4 GB	4 GB	4 GB
Data/ Stack Physical Memory Capacity in Bytes	4 GB	4 GB	4 GB
Harvard or Princeton Main memory Architecture	Princeton	Princeton	Princeton
External Interrupts	32	8	2
Bit Manipulation Instructions	Yes	Yes	Yes
Floating Point Processor	Yes	Yes#	Yes
Interrupt Controller	Yes	Yes	No
DMA Controller Channels	No	4	No
On - Chip MMU	Yes	Yes#	Yes

ler Version Selection:

There are numerous versions of 8051. Additional devices and units arc provided in these versions. A version is selected for embedded system design as per the application as well as its cost.

1. An embedded system in automobile for example requires CAN bus. Then a version with CAN bus controller is selected.
2. An 8051 enhancement, 8052, has an additional timer.
3. Philips P83C528 has I2C serial bus.
4. The 8051-family member 83C152JA (and its sister JB., JC and JD) microcontrollers) have two direct memory access (DMA) channels on-chip.

- The 805196K has a PTS (Peripheral Transactions Server) that supports DMA functions.
- [Only single and bulk transfer modes are supported, not the burst transfer mode] When a system requires direct transfer to memory from external systems, the DMAC is used so that the system performance improves by a separate processing unit for the data transfers nit and to the peripherals.

Example applications:

- Robotics-8086,68HC11

Case Study of a Real Time Robot Control System

1. A robotic system motor needs signaling at the rate above 50 to 100 ms. Hence there is enough time available for signaling and real time control of multiple motors at the robot when we use a processor with instruction cycle time ~ 1 μs.
2. The processor speed need not be very high and performance needed is much below 1 MIPS. So no caches and advanced processing units like pipeline and superscalar processing are required.
3. Four-coil stepper motor needs only 4-bit input and a dc motor needs one bit pulse width modulated output. Therefore, 8-bit processor suffices.
4. Frequent accesses and bit manipulations at I/O ports are needed. CISC architecture therefore suffices.
5. Program can fit in 4 kB or 8 kB of internal ROM on-chip and stack sizes needed in the program are small that can be stacked in on-chip 256 or 512 Byte RAM. Microcontroller is thus needed. No floating-point unit is needed.

   Microcontrollers that are appropriate for the above case are 8051, 68HC11 or 68HC12 or 68HC16 or 80196. Microcontroller 68HC12 or 68HC16 can be best choice due to large number of ports available. [68HC12 instruction cycle and clock cycle time = 0.125 ms. Number of ports = 12 in 68HC12. Therefore, 6 or more degree of freedom robot with 6 or more motors can be driven directly through these ports. STOP and WAIT instructions exist in the processor to save power when robot is at rest!]

- Voice/image processing – ARM or Pentium

Case Study of Voice Data Compression System

1. Voice signals are pulse code modulated. The rate at which bits are generated is 64 kbps. A suitable algorithm can process the data compression of these bits with instruction cycle time ~ 0.01 to 0.04 μs (100 to 25 MHz) when the processor uses advanced processing units and caches.
2. Let us assume that the processor instruction cycle time is 0.02 μs (50 MHz). With a three-stage pipeline and two lines superscalar architecture, the highest performance will be 300 MIPS. [Refer to Example 2.1 for an understanding of the computations of MIPS.] It suffices for not only for voice but also for video compression.
3. Frequent accesses and complex instructions may not be needed. Either a CISC or RISC can be used.
4. Program can't fit in 4 kB or 8 kB of internal ROM on-chip and stack sizes needed in the program are big. Instead large ROM and RAM as well as caches are needed.
5. No floating-point is needed as mostly the bit manipulation instructions are processed during compression.

   Exemplary processors that are appropriate for the above case are 80x86, 80860 and 80960.

- Network – ARM7,ARM9

Case Study of Fast Network Switching System

1. Transfer rates of 100 MHz plus are needed in fast switches on a network. Assuming 10 instructions per switching and transceiver action, instruction cycle time should be 0.001 plus. A multiprocessor system is needed for GHz transfer rates,
2. Let us assume that the processor instruction cycle time is 0.01 μs (100 MHz). With five-stage pipeline and two lines superscalar architecture, the highest performance will be 1000 MIPS. [Refer to Example 2.1 for understanding the computations of MIPS]. The multiprocessor system is thus needed for 100 MHz plus switches.
3. The processor should have RISC architecture for single cycle instruction processing.
4. ROM and RAM as well as caches are needed.
5. No floating-point is needed as mostly the bits are processed for input and output.

   Exemplary processors that are appropriate for the above case are PowerPC and ARM7

- Real time video processing – TMS320, Tiger SHARC

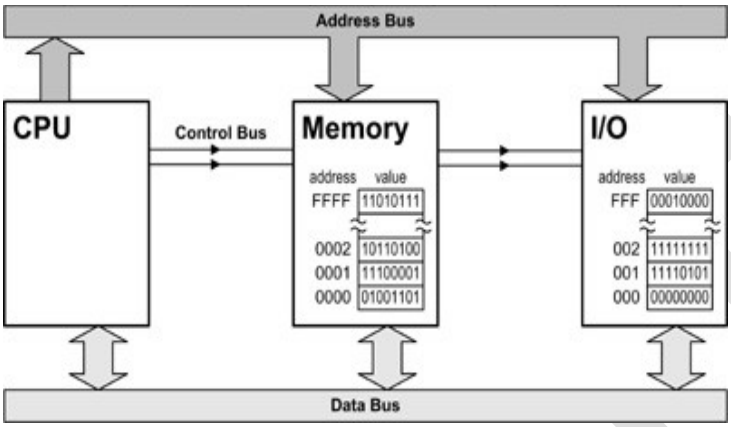
Real-Time Video Processing

1. Real-time video processing requires fast compression of an image. The use of DSP is also essential. A number of real-time functions have to be processed: for instance, scaling and rotation of images, corrections for shadow, colour and hue, image sharpening and filter functions. In such cases, a multiprocessor system with DSP(s) and that has the best processing performance is required.
2. All advanced processing units, which are listed in Section 2.1, are needed.
   Exemplary processors that are appropriate for multiprocessor system are ARM9 integrated with TMS family DSP(s) or ASIC solution using Xilinx FPGA with multiple processors Virtex –II Pro™ (Section 1.6.5).

Case Studies

- Automatic Washing machine
- Data Acquisition Systems for the sixteen parameters channels
- Data Acquisition Systems for the ECG waveforms
- Multi channel Fast Encryption and cum decryption Transceiver System
- Mobile Phone system

3.3 Interfacing of Processor, Memory and I/O Devices

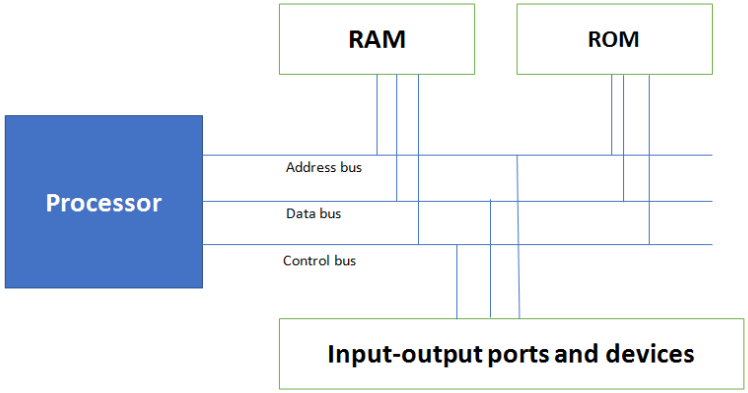


Interfacing of processor, memory and I/O devices using memory system bus

Computer-System Buses:

- Bus is set of parallel lines (wires) which carry signals from one unit to another unit.
- Bus lines interconnect several units, but at a given instance, only two of them communicate.
- Bus enables interconnections among many units in a simple way.
- The signals are in specific sequences according to a method or protocol.
- Computing system buses are as follows:
    - **System bus (Memory bus)**
    - **I/O bus (Peripheral bus)**
- System Bus also called memory bus, which interconnects the subsystems.
- It interconnects the processor to memory system and other hardware units.
- This bus has high speed and bandwidth.
- It is according to the processor, memory system bandwidth, and for read-write cycles of instructions and data. (Bandwidth means number of bits transferred per second.)
- I/O Bus also called peripheral bus.
- Interconnects the memory bus to a variable number of I/O devices functioning at variable speeds.
- Devices or peripherals are designed to interface to the I/O bus.
- The devices can be attached or withdrawn from the I/O bus at any time.
- CPU/Microprocessor System buses Computing system hardware consists of processor, memory and 1/O units (ports, devices and peripherals).
- System bus enables the interconnections among multiple subsystems in the system CPU/microprocessor/processor in a computing system interconnects to memory, I/O units, devices and peripherals through a bus, called system bus or system memory bus.

- Three sets of signals—classified as address bus, data bus and control bus define the system bus.
- A system-bus interfacing-design is according to the timing diagram of processor signals, bus bandwidth and word length.
- A simple structure of system bus is that same bus, which connects the memory also connects the other subs stems (I/O units. ports, devices and peripherals).
- Processor interfaces memory as well as I/O devices using system memory bus.
- Figure (a) shows the interfacing of processor, memory and I/O devices using memory system bus in a simple bus structure.



Interfacing of processor, memory and I/O devices using memory system bus

1. Address Bus

- Address bus signals are from processor to memory or other interfaced units.
- Address bus is unidirectional When it has 'n' signals A0 to An-1, the Processor issues (send) addresses between 0 and 2n-1 using the bus.
- The processor issues the address of the instruction byte or word to the memory system.
- The address bus communicates the address to the memory.
- The address bus of 32 bits fetches the instruction or data when an address specified by a 32-bit number, between 0 and 232-1.

2. Data Bus

- A data bus is bidirectional. If it has 'm' bits then signals are D0-Dm-1 and processor reads a word or instruction or writes a word.
- Data signals are from processor to memory during read cycle and memory to processor during write cycle.
- A data bus of 8, 16, 32 or 64 bits fetches, loads, or stores the instruction or data.

Read Cycle: Read cycle means a sequence of signals during which using the data-bus processor

(i) fetches instruction from program-memory section of memory, or

(ii) loads data word from data-memory section of memory.

- When the processor issues the address of the instruction. It gets back the instruction from memory through the data bus.
- When it issues the address of the data it processor loads the data through the data bus into a register.

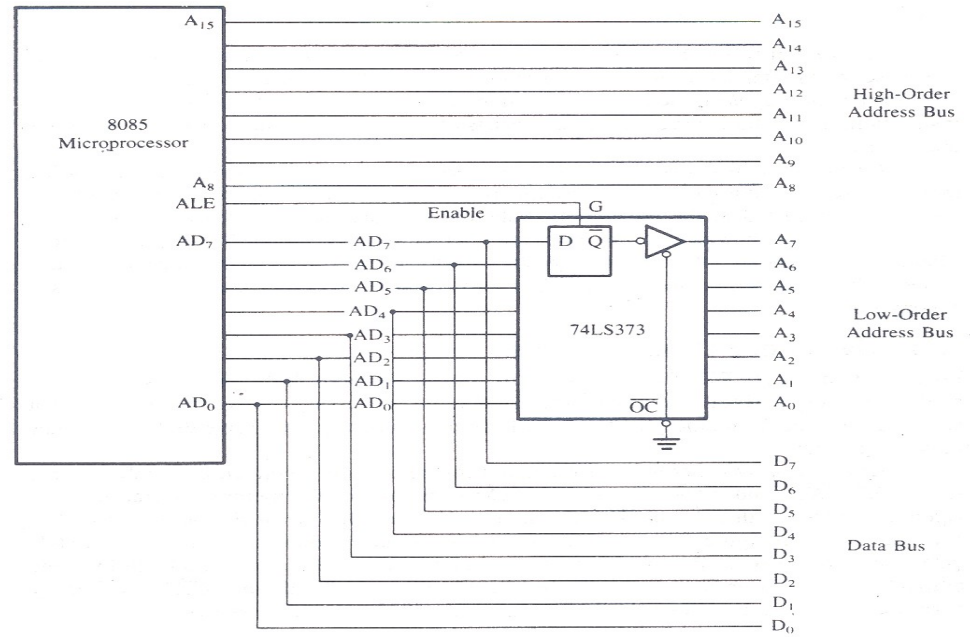
Write Cycle: Write cycle means a sequence of signals during which, using data bus, the processor sends data word to data-memory address in memory.

- When it issues the address of the data it stores the data in the memory through the data bus.

3. Control Bus

- A control bus issues signals to control the timing of various actions during communication of signals.
- These signals synchronize the subsystems.
- (a) When the processor issues the address, after allowing sufficient time for the set-up of all address bits, it also issues a memory-read control signal and waits for the data or instruction after a time interval.
- A memory unit must place the instruction or data during the interval in which memory-read signal is active (not inactivated by the processor).
- (b) When the processor issues the address on the address bus, and (after allowing sufficient time for the set-up of all address bits) it places the data on the data bus, it also then issues memory-write control signal (after allowing sufficient time for the set-up of all data bits) for store signal to memory.
- The memory unit must write (store) the data during the interval in which memory-write is active (not inactivated by the processor).

8051-Address Latch Enable
80196-Assress Strobe(AS)

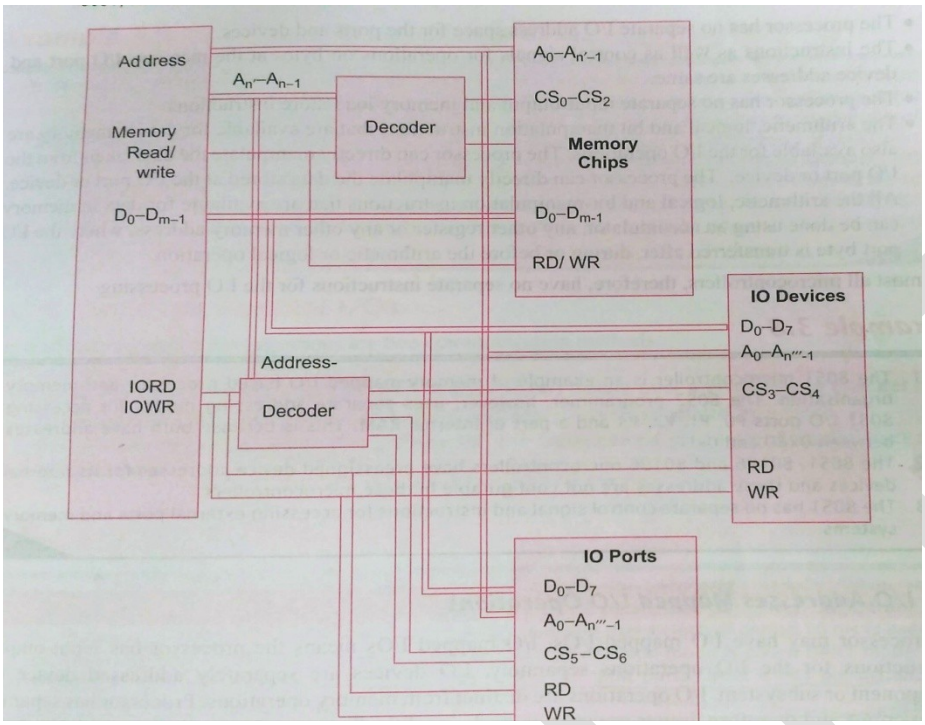


3.4 I/O Interfacing Concepts:

Interfacing methods for I/O device:

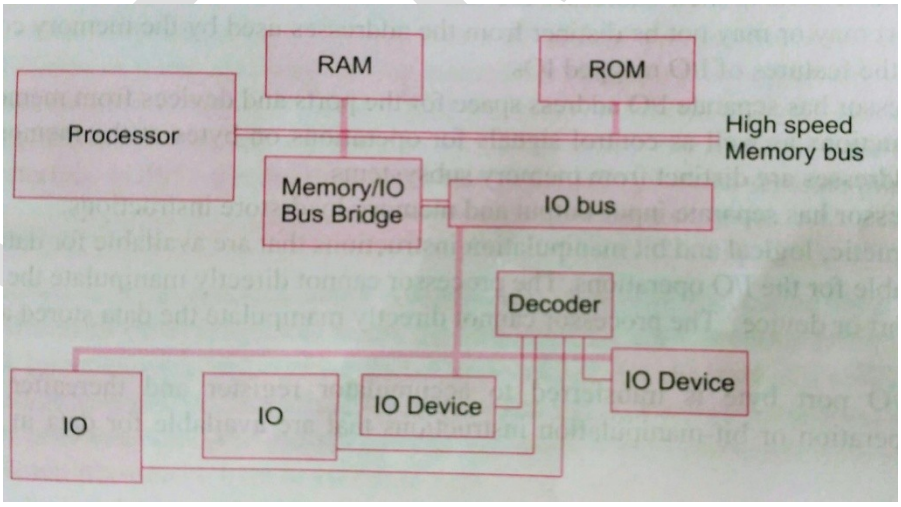
Method 1:

I/O devices or components interface using ports, interfacing circuit consists of decoder. The decoder circuit connects the processor address bus and control signals. A port select output of the decoder is active when the address input corresponds to the port address. Interfacing circuit is designed as per available control signals and timing diagrams of the system bus signals. The below Figure shows interface using ports.



Method 2:

A method is interfacing through an I/O bus. Interfacing circuit consists of I/O controller (called bridge or switch also). The switch circuit connects the processor and memory on system memory bus with the I/O bus. Interfacing-circuit is designed as per available control signals and timing diagrams of the system bus signals and I/O bus. The below Figure shows interface using I/O bus and switch circuit between system memory bus and I/O bus.



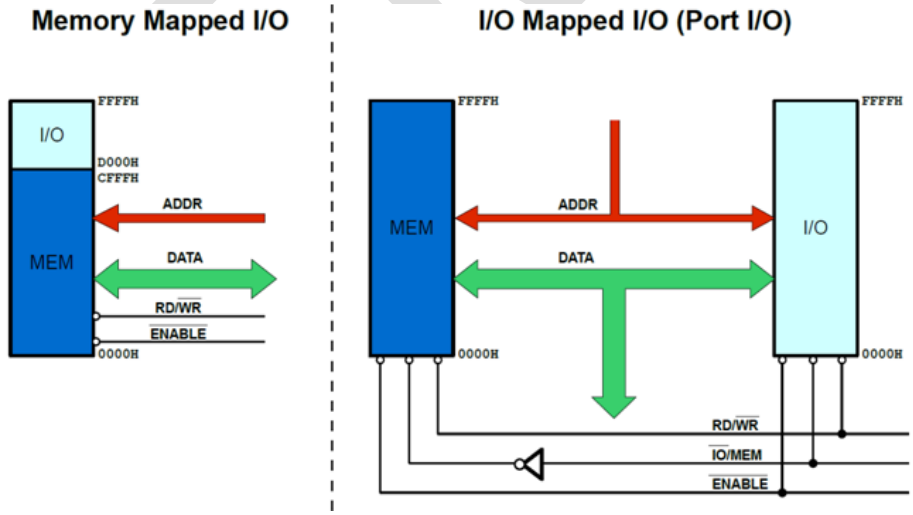
I/O managing data

- Memory mapped I/O and I/O mapped I/O operations are two types of operations based on processor and memory organisation.

- Memory mapped I/O is mapped into the same address space as program memory and/or user memory, and is accessed in the same way.
- **Port mapped I/O uses a separate, dedicated address space and is accessed via a dedicated set of microprocessor instructions.**
- The difference between the two schemes occurs within the microprocessor. Intel has, for the most part, used the port mapped scheme for their microprocessors and Motorola has used the memory mapped scheme.
- As 16-bit processors have become obsolete and replaced with 32-bit and 64-bit in general use, reserving ranges of memory address space for I/O is less of a problem, as the memory address space of the processor is usually much larger than the required space for all memory and I/O devices in a system.
- Therefore, it has become more frequently practical to take advantage of the benefits of memory-mapped I/O. However, even with address space being no longer a major concern, neither I/O mapping method is universally superior to the other, and there will be cases where using port-mapped I/O is still preferable.

Memory-mapped IO (MMIO):

- I/O devices are mapped into the system memory map along with RAM and ROM. To access a hardware device, simply read or write to those 'special' addresses using the normal memory access instructions.
- The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device.
- The disadvantage to this method is that the entire address bus must be fully decoded for every device. For example, a machine with a 32-bit address bus would require logic gates to resolve the state of all 32 address lines to properly decode the specific address of any device. This increases the cost of adding hardware to the machine.



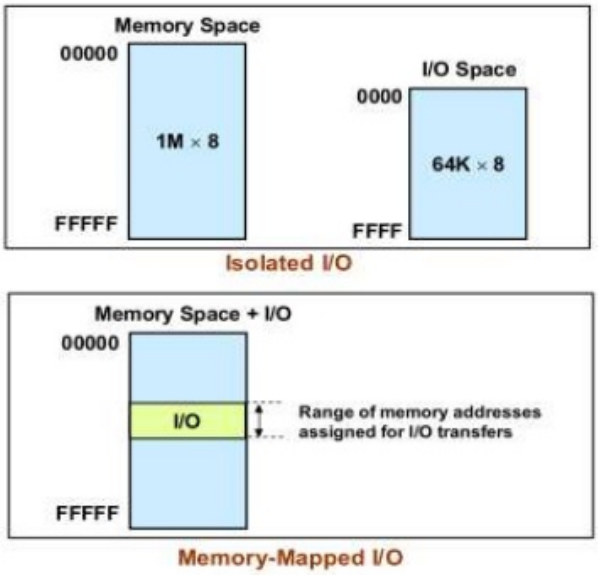
Port-mapped IO (PMIO or Isolated IO):

- I/O devices are mapped into a separate address space. This is usually accomplished by having a different set of signal lines to indicate a memory access versus a port access. The address lines are usually shared between the two address spaces, but less of them are used for accessing ports. An

example of this is the standard PC which uses 16 bits of port address space, but 32 bits of memory address space.

- The advantage to this system is that less logic is needed to decode a discrete address and therefore less cost to add hardware devices to a machine. On the older PC compatible machines, only 10 bits of address space were decoded for I/O ports and so there were only 1024 unique port locations; modern PC's decode all 16 address lines. To read or write from a hardware device, special port I/O instructions are used.
- From a software perspective, this is a slight disadvantage because more instructions are required to accomplish the same task. For instance, if we wanted to test one bit on a memory mapped port, there is a single instruction to test a bit in memory, but for ports we must read the data into a register, then test the bit.

Isolated vs. Memory Mapped I/O



Memory-mapped IO	Port-mapped IO
Same address bus to address memory and I/O devices	Different address spaces for memory and I/O devices
IO is treated as memory	IO is treated as IO
16-bit addressing is used	8-bit addressing is used
More decoder hardware is used	Less decoder hardware is used
Can access 2^{16} = 64k locations theoretically	Can address 2^{16} =256 locations
Access to the I/O devices using regular instructions	Uses a special class of CPU instructions to access I/O devices
Memory instructions are used	Special IN and OUT instructions
Arithmetic and logic operations can be performed directly on data	Arithmetic and logic operations cannot be performed directly on data
Data transfer b/w register and I/O	Data transfer b/w accumulator and I/O

- **Memory Mapped IO**
  - IO is Treated as Memory.
  - More Decoder Hardware.
  - Less memory space is available.
  - Memory Instructions are used.
  - Arithmetic and logic operations can be performed directly on data.
  - Data transfer b/w register and IO.
  - 8051
- **IO Mapped IO**
  - IO is Treated IO.
  - Less Decoder Hardware.
  - Whole memory address space is available.
  - Special Instructions are used like IN, OUT.
  - Arithmetic and logic operations can't be performed directly on data.
  - Data transfer b/w accumulator and IO.
  - 80x96

3.5 I/O Devices:

✓ A port at a device can transmit (send) or receive through wire or wireless. Input port means a circuit to where bit or bits can be input (received) from an external device, peripheral or system. Output port means a circuit from where bit or bits can be output (sent) to an external device, peripheral or system. Input-Output (I/O) port means a circuit where bit(s) can be input or output. There are two types of I/Os, serial and parallel. Serial means in series of successive instants. Parallel means at the same instance.

1. Serial Input- Serial input port means a circuit to where bits can be input (received) in successive time intervals. The time interval is known to the receiver port. The port assembles the bits on receiving at successive instances.

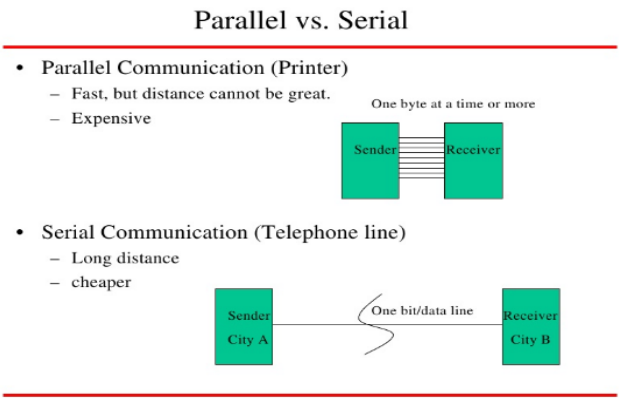
2. Serial Output- Serial output port means a circuit from where bits can be output (sent) in successive time intervals. A time interval is known to the external device or system. The external device assembles the bits on receiving at successive instances from the output port.
3. Serial I/O -Serial Input-Output (I/O) port means a circuit where serially received or sent bits can be input or output.
4. Parallel Input- Parallel input port means a circuit where bits can be input (received) at an instant. The processor at the input device, circuit or system reads the bits from the port at next instant.
5. Parallel Output -Parallel output port means a circuit from where bits can be output (sent) at an instant. The external trial device can read the bits at the output port.
6. Parallel I/O- Parallel Input-Output (I/O) port means a circuit where received or sent bits in parallel can be input or output.
7. I/O Types- (i) Serial Input, (ii) Synchronous Serial Output, (iii) Asynchronous Serial Input, (iv) Asynchronous Sena u..tput, (v) Parallel Port One-bit Input, (vi) Parallel Port One-bit Output, (vii) Parallel Port Input, and (viii) Parallel Port Output.

- ✓ I/O devices can be classified into following **I/O types**:

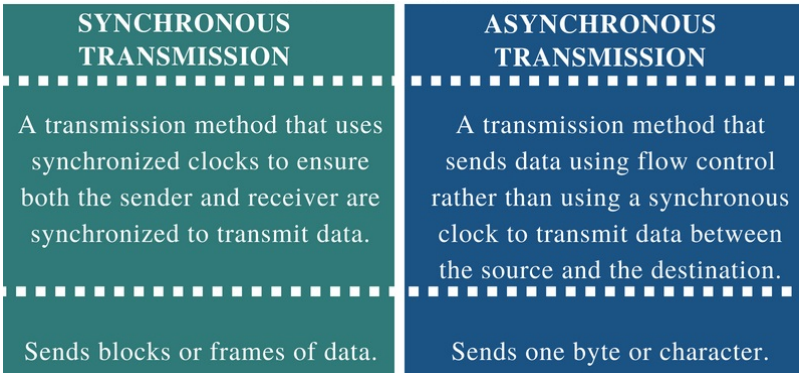
i) Synchronous Serial Input
ii) Synchronous Serial Output
iii) Asynchronous Serial UART Input
iv) Asynchronous Serial UART Output
v) Parallel one bit Input
vi) Parallel one bit Output
vii) Parallel Port Input
viii) Parallel Port Output

- ✓ Some devices function both as input & as output, example –MODEM

- **Serial Vs Parallel**



- **Synchronous Vs Asynchronous**



- **Port**
- A **PORT**is a device
- To receive the bytes from external peripheral(s) for reading them later using instructions executed on the processor **or**
- To send the bytes to external peripheral or device or processor using instructions executed on processor
- A Port connects to the processor using address decoder and system buses
- The processor uses the addresses of the port-registers for programming the port functions or modes, reading port status and for writing or reading bytes.

Synchronous Serial Input Device

Serial Bits and a clock signal used for synchronisation of a port input.

The sender along with the serial bits also sends the clock pulses SCLK (serial clock) to the receiver port pin.

The port synchronizes the serial data input bits with clock bits.

The bytes are received at constant rates.

Each byte at input port separates by 8T and data transfer rate for the serial line bits is (1/T) bps. [1bps = 1 bit per s]

Synchronous Serial Output Device

Device Serial Bits and synchronisation clock signal at a port output.

Each bit in each byte sent in synchronization with a clock.

Bytes sent at constant rates.

If clock period = T, then data transfer rate is (1/T) bps.

- Each bit in each byte is in synchronization at input and each bit in each byte is in synchronization at output with the **master clock output .**
- The bytes are sent or received at constant rates.

Format of bits at UART protocol

Asynchronous Serial Output Device

Asynchronous output serial port line TxD (Transmit Data)

Each bit in each byte transmit at fixed intervals but each output byte is not in synchronization(Separates by a variable interval or phase difference)

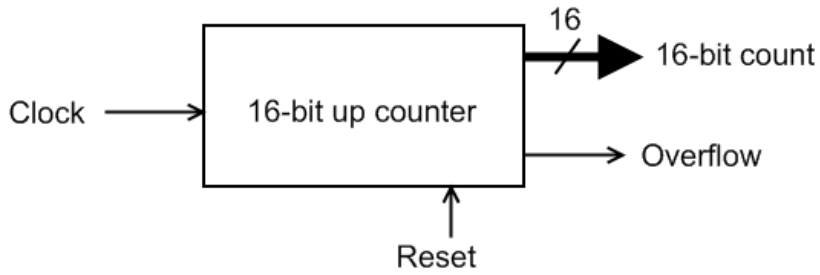
- Examples of Various types of I/O devices

I/O Device Type	Examples
Synchronous Serial Input	Inter-processor data transfer, **reading from CD or hard disk**, audio input, video input, dial tone, network input, transceiver input, scanner input, remote controller input, serial I/O bus input, writing to flash memory using SDIO (**S**ecure **D**ata **A**ssociation **IO** based card)
Synchronous Serial Output	Inter-processor data transfer, multiprocessor communication, **writing to CD or hard disk**, audio Input/output, video Input/output, dialler output, network device output, remote TV Control, transceiver output, and serial I/O bus output or writing to flash memory using SDIO
Serial Asynchronous UART Input	Keypad controller serial data-in, mice, **keyboard controller**, modem input, character send inputs on serial line [also called UART (universal receiverand transmitter) input when according to UART mode]
Serial Asynchronous UART Output	Output from modem, **output for printer**, the output on a serial line [also called UART output when according to UART]

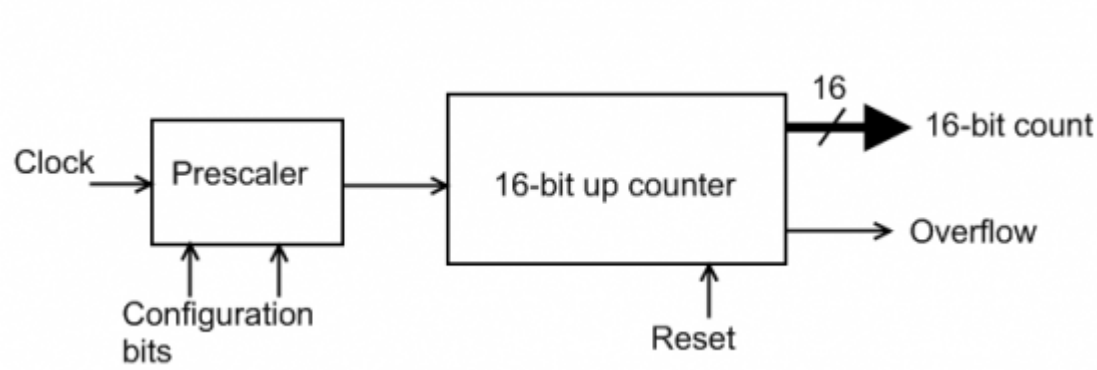
I/O Device Type	Examples
Parallel Port single bit input	i) Completion of a revolution of a wheel, ii) Achieving preset pressure in a boiler, iii) Exceeding the upper limit of permitted weight over the pan of an electronic balance, iv) Presence of a magnetic piece in the vicinity of or within reach of a robot arm to its end point and v) Filling of a liquid up to a fixed level.
Parallel Port Output- single bit	i) PWM output for a DAC, which controls liquid level, or temperature, or pressure, or speed or angular position of a rotating shaft or a linear displacement of an object or a d.c. motor control ii) Pulses to an external circuit / Control signal to an external circuit
Parallel Port Input	i) ADC input from liquid level measuring sensor or temperature sensor or pressure sensor or speed sensor or d.c. motor rpm sensor ii) Encoder inputs for bits for angular position of a rotating shaft or a linear displacement of an object
Parallel Port Output	i) LCD controller for Multilane LCD display matrix unit in a cellular phone to display on the screen the phone number, time, messages, character outputs or pictogram bit-images for display screen or e-mail or web page ii) Print controller output / Stepper-motor coil driving bits

3.6 Timer and Counting Devices:

Timing Devices



- A device, which counts the input at regular interval (δT) using clock pulses at its input.
- The counts are incremented on each pulse and stored in a register, called count register.
- A timer is a specialized type of clock which is used to measure time intervals.
- A timer uses the frequency of the internal clock, and generates delay.
- Evaluation of Time
- The counts multiplied by the interval δT gives the time.
- For example, if a particular clock's frequency is 1 MHz (period 1 μs), and we have counted 3000 pulses on the clock signal, then the elapsed time is 3000 microseconds.
- The (present counts -initial counts) xδT interval gives the time interval between two instances when present count bits are read and initial counts were read or set.
- Suppose, we want to measure the time elapsed between any two successive events.
- Lets assume that when the first event occurs, the timer is reset to zero, and when the second event occurs, the timer output is 25000. If we know that the input clock has period of 1 μs, then the time elapsed between the two events is 25000×1 μs = 25 milliseconds.
- The (present counts -initial counts) xδT interval gives the time interval between two instances when present count bits are read and initial counts were read or set.
- Since this timer's counter can count from 0 to 65535 (2^{16}-1), this particular arrangement can measure time ranging from 0 to 65535 x 1 μs = 65.535 milliseconds, with a resolution of 1 μs.
- Has an input pin (or a control bit in control register) for resetting it for all count bits = 0s.
- Has an output pin (or a status bit in status register) for output when all count bits = 0s after reaching the maximum value, which also means after timeout or overflow.
- Counting Devices



- A device, which counts the input due to the events at irregular or regular intervals.
- A counter is a device that stores (and sometimes displays) the number of times a particular event or process occurred, with respect to a clock signal.
- It is used to count the events happening outside the microcontroller.
- In electronics, counters can be implemented quite easily using register-type circuits such as a flip-flop.
- A counter uses an external signal to count pulses.
- Free Running (Blind Counting) Device with a Pre-scaler, Compare and Capture Registers.
- Pre-scalar can be programmed as p = 1, 2, 4, 8, 16, 32, .. by programming a Pre-scaler register.
- Pre-scalar divides the input pulses as per the programmed value of *p.*
- Count interval = p xδT interval
- δ T = clock pulses period, clock frequency = δT^{-1}
- It has an output pin (or a status bit in status register) for output when all count bits = 0s after reaching the maximum value, which also means after timeout or overflow.
- Free running n-bit counter overflows after

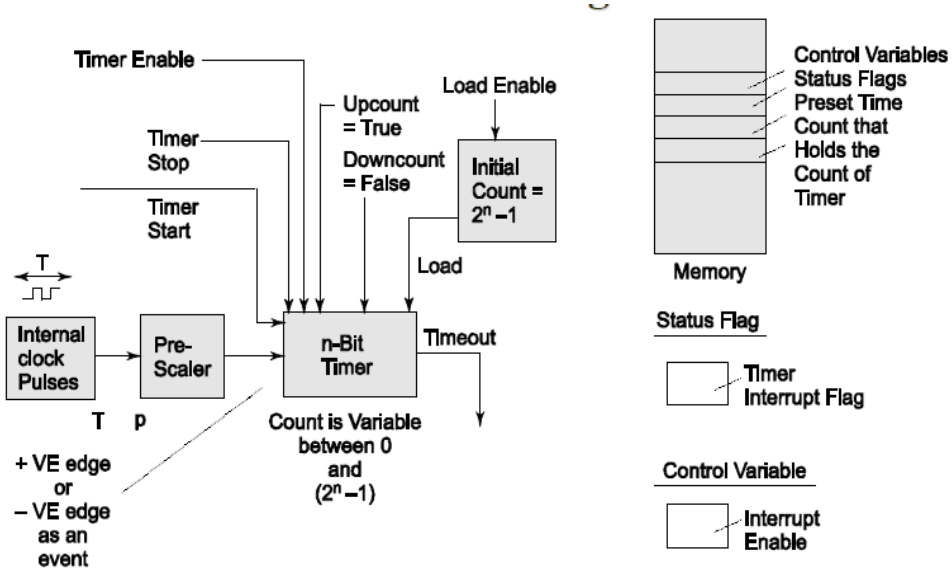
  $p \times (2^n - 1) \times \delta T$ interval
- This device useful for the Alarm or Processor Interrupts after preset intervals with respect to another event from another source.
- For example, if we use a Prescaler to our previous 16-bit timer to divide the clock frequency by 8, then the new range of the timer will be 65535 x 8 μs = 524.280 milliseconds.
- Most microcontrollers are equipped with one or more precision timing systems that can be used to perform a variety of precision timer functions including generating events at specific times, determining the duration between two events, or counting events.
- Example applications that require generating events include generating an accurate 1 Hz signal in a digital watch, keeping a traffic light green for a specific duration, or communicating bits serially between devices at a specific rate, etc.
- **Timer States**

- Reset State (initial count = 0)
- Initial Load State (initial count loaded)
- Present State (counting or idle or before start or after overflow or overrun)
- Overflow State (count received to make count = 0 after reaching the maximum count)
- Overrun State (several counts received after reaching the overflow state)
- Running (Active) or Stop (Blocked) state
- Finished (Done) state (stopped after a preset time interval or timeout)
- Reset enabled/disabled State (enabled resetting of count = 0 by an input)
- Load enabled/disabled State (reset count = initial count after the timeout)
- Auto Re-Load enabled/disabled State (enabled count = initial count after the timeout)
- Service Routine Execution enable/disable State (enabled after timeout or overflow)

- **Uses of Timer Device**

1. Real Time Clock Ticks (**System Heart Beats**).
2. Initiating an event after a preset delay time.
3. Initiating an event (or a pair of events or a chain of events) after a comparison(s) with between the pre-set time(s) with counted value(s).
4. Capturing the *count value* at the timer on an event.
5. Finding the time interval between two events.
6. Wait for a message from a queue or mailbox or semaphore for a preset time when using RTOS.
7. Watchdog timer. It resets the system after a defined time.
8. Baud or Bit Rate Control for serial communication on a line or network. Timer timeout interrupts define the time of each baud.
9. Scheduling, Time Slicing of various tasks.
10. **Time division multiplexing (TDM)**

- A timer cum counting device is a counting device that has two functions.
- (1) It counts the input due to the events at irregular instances.
- (2) It counts the clock input pulses at irregular intervals.



- Control bits are as per the hardware signals and corresponding bits at the control register. Control bits (or signals) can be of nine types.
  1. Timer Enable (to activate a timer).
  2. Timer Start (to start counting at each clock input).
  3. Timer Stop (to stop counting) from next clock input).
  4. Pre-scaling bits (to divide the clock-out frequency signal from the processor).
  5. Up count Enable (to enable up counting by incrementing the count value on each clock input)
  6. Down Count Enable (to decrement on a clock input).
  7. Load Enable (to enable loading of a value at a register into the timer).
  8. Timer Interrupt Enable (to enable interrupt servicing when the timer outs (overflows) and reaches count value = 0)
  9. Timer Interrupt Enable [to enable interrupt servicing when the timer overflows (reaches count =0)].

Ten Forms of a Timer

Hardware internal-Timer

Software timer (SWT)

User software-controlled hardware timer

RTOS controlled hardware timer. An RTOS can define the clock ticks per second of a hardware timer at a system.

Timer with periodic time-out events (auto-reloading after overflow state). A timer may be programmable for auto-reload after each time-out.

One shot timer (No reload after the overflow and finished state). It triggers on event-input for activating it to running state from its idle state. It is also used for enforcing time delays between two states or events.

Up count action Timer. It is a timer that increments on each count-input from a clock.

Down count action Timer. It is a timer which decrements on each count-input.

Timer with its overflow-flag, which auto resets as soon as interrupt service routine starts running.

Timer with overflow-flag, which does not auto reset.

- SWT
- Innovative concept –VIRTUAL Timing device.
- A software, which executes and increases or decreases a count-variable(count value) on an interrupt from system timer output or real time clock interrupt.
- The software timer also generate interrupt on overflow of count-value or on finishing value of the count variable.
- System Clock
- A hardware-timing device programmed to tick at constant intervals δT.
- An interrupt at each tick
- A chain of interrupts thus occur at periodic intervals.
- δ T is as per a preset *count value*
- The interrupts are called system clock interrupts, when used to control the schedules and timings in the system

SWT and System clock

✓ System clock has fixed program to tick at constant intervals δT.

✓ SWTs have fixed but programmable to tick at intervals δT.

✓ An interrupt at each tick in both

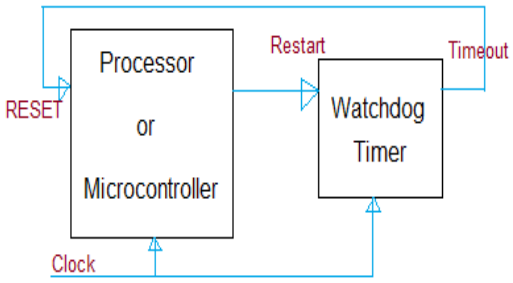
✓ The interrupts called system clock and SWT interrupts, respectively

Watchdog Timer

- ✓ A timing device such that it is set for a preset time interval and an event must occur during that interval else the device will generate the timeout signal on failure to get that event in the watched time interval.
- ✓ On that event, the watchdog timer is disabled for generation of timeout or reset
- ✓ Timeout may result in processor starting a service routine or start from beginning

Example

- ✓ Assume that we anticipate that a set of tasks must finish in 100 ms interval.
- ✓ The watchdog timer is disabled and stopped by the program instruction in case the tasks finish within 100 ms interval.
- ✓ In case task does not finish (not disabled by a program instruction), watchdog timer generates interrupts after 100 ms and executes a routine, which is programmed to run because there is failure of finishing the task in anticipated interval.



Watchdog Timer Application

- ✓ An application in mobile phone is that display is off in case no GUI interaction takes place within a watched time interval.
- ✓ The interval is usually set at 15 s, 20 s, 25 s, 30 s in mobile phone.
- ✓ Mobile thus saves the power
- ✓ An application in temperature controller
- ✓ If controller takes no action to switch off the current within preset watched time interval, the current switched off
- ✓ Warning signal raised as indication of controller failure

✓ Failure to switch off current may burst a boiler in which water is heated.

Provisioning of watchdog timer

✓ A software task can also be programmed as a watchdog timer

✓ Microcontroller may also provide for a watchdog timer.

REAL TIME CLOCK:

- A clock, which is based on the interrupts at preset intervals.
- An interrupt service routine executes on each timeout (overflow) of this clock.
- This timing device once started never resets or never reloaded with another value.
- Used in a system to save the time and date.
- Used in a system to initiate return of control to the system (OS) after the set system clock periods

RTC Application

- Assume that a hardware timer of an RTC for calendar is programmed to interrupt after every 5.15 ms (=1 day period/ 224)
- Assume each tick (interrupt) a service routine runs and updates at a memory location. Within one day (86400 s) there will be 224 ticks, the memory location will reach 0x000000 after reaching the maximum value 0xFFFFFF.

RTC with 5.5 ms tick

- Within 256 days there will be 232 ticks, the memory location will reach 0x00000000 after reaching the maximum value 0xFFFFFFFF.
- A battery is used to protect the memory for long period

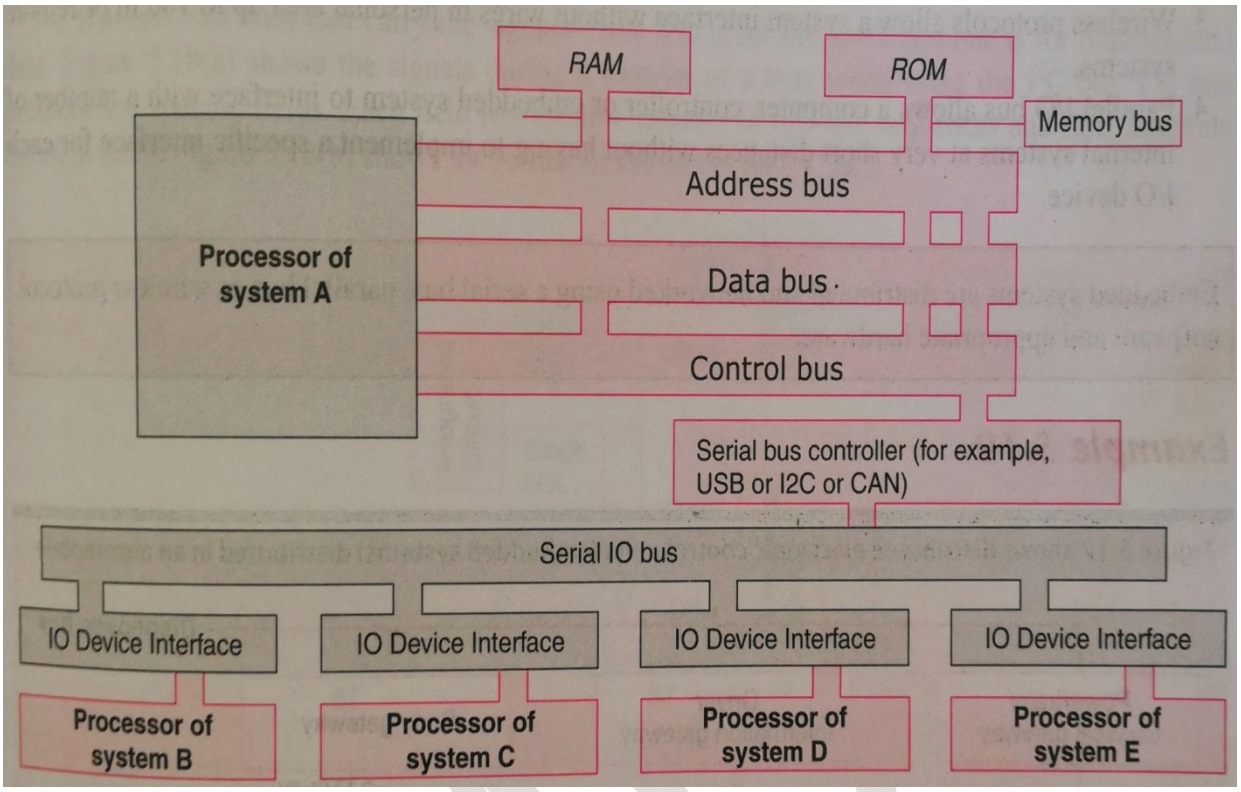
RTC for implementing a software timer

- A hardware 16-bit timer ticks from processor clock after 0.5 μs. It will overflow and execute an overflow interrupt service routine after 2^{15} μs = 32.768 ms.
- The interrupt service routine can generate a port bit output after every time it runs or can call a software routine or send a message for a task. If n = 30, the RTC initiated software will run every 30 × 32.768 ms, which is close to 1 s.

Unmask and reset to mask of real time interrupt

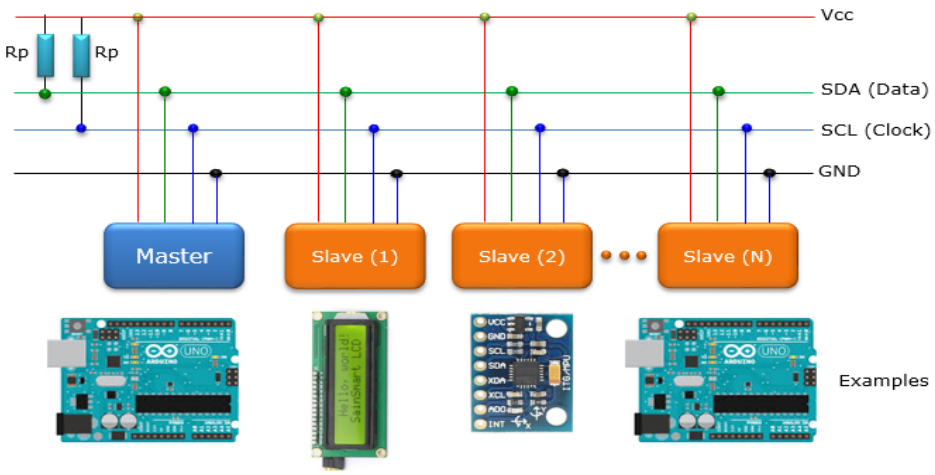
- RTI is set to unmask and reset to mask the real time interrupt locally.
- If RTI and I bits permit the interrupt request for real time, the microcontroller fetches the lower and higher bytes of the interrupt servicing routine address from the addresses 0xFFF0 (higher byte) and 0xFFF1 (lower byte)

3.7 Serial Communication using I2C, CAN and Advanced I/O Buses between the Networked Multiple Devices



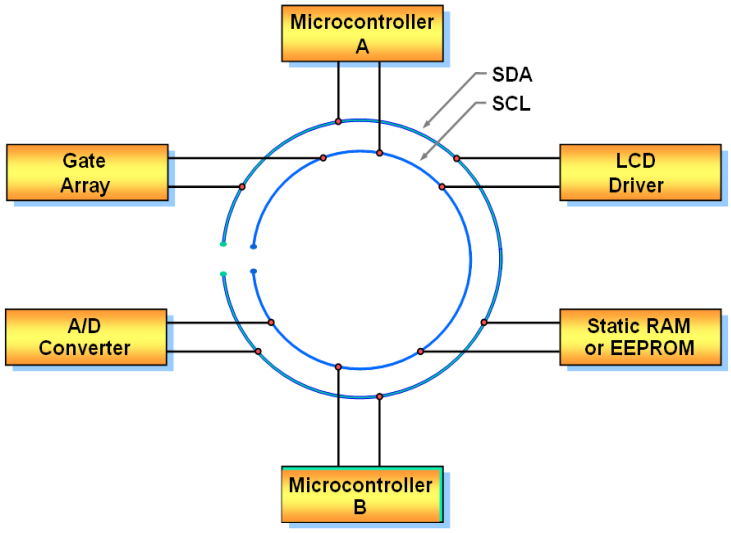
- A distributed networked system means a number of systems on a common bus or a set of buses, where each system interfaces to a bus. Each bus communicates as per a protocol. Bus communication simplifies the number of connections and provides a common way (protocol) of connecting different or same type of I/O devices. A communication system may use protocols such as UART, 12C, CAN, USB, Wi-Fi or Bluetooth for synchronous or asynchronous transmission from interface at device, or from the system to another interface. The above Figure shows a computer-system bus connected to serial-I/O bus using a bus controller, and I/O bus networking the number of embedded systems distributed on a serial bus.
- Embedded systems can be distributed and networked using an IO bus or Networking Protocol
- Serial bus protocols
- I2C bus
- CAN bus
- USB bus

I2C BUS



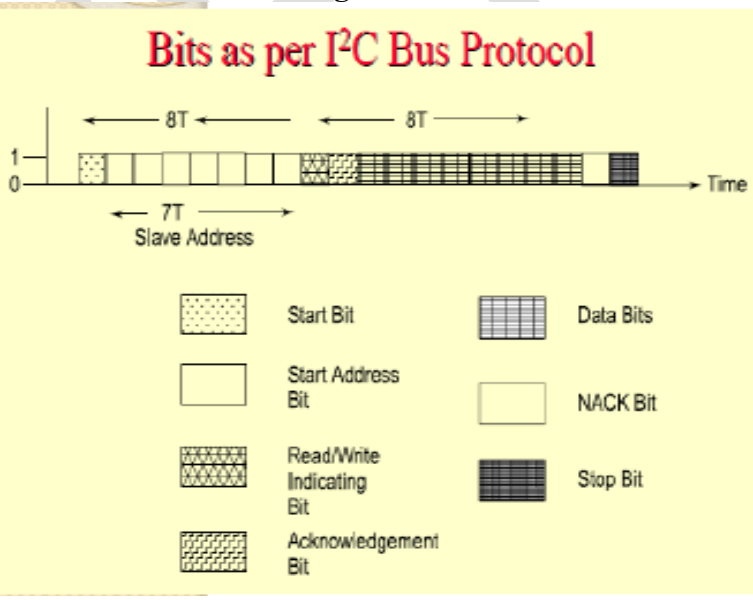
- ✓ ICs mutually network through a common synchronous serial bus I2C ('**Inter Integrated Circuit**')
- ✓ Any device that is compatible with a I2C bus can be added to the system (assuming an appropriate device driver program is available), and a I2C device can be integrated into any system.
- ✓ The Bus has two lines that carry its signals—one line is for the **clock** and one is for bi-directional **data**.
- ✓ There is a standard protocol for the I2C bus.

- ✓ I2C : Inter integrated circuit (I2C) is important serial communication protocol in modern electronic systems. Philips (now NXP semiconductors) invented this protocol in 1986. The objective of reducing the cost of production of television remote control motivated Philips to invent this protocol. I2C is a serial bus interface, can be implemented in software, but most of the microcontrollers support I2C by incorporating it as hard IP (Intellectual Property). IIC can be used to interface microcontroller with RTC, EEPROM and different variety of sensors. IIC is used to interface chips on motherboard, generally between a processor chip and any peripheral which supports I2C. I2C is very reliable wireline communication protocol for an on board or short distances. I2C is a serial protocol for two-wire interface to connect low-speed devices like microcontrollers, EEPROMs, A/D and D/A converters, I/O interfaces and other similar peripherals in embedded systems.



- ✓ I2C combines the best features of SPI and UARTs. With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, o multiple slaves. This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD.
- ✓ I2C protocol uses two wires for data transfer between devices: Serial Data Line (SDA) and Serial Clock Line (SCL). The reduction in number of pins in comparison with parallel data transfer is evident. This reduces the cost of production, package size and power consumption. I2C is also best suited protocol for battery operated devices. I2C is also referred as two wire serial interface (TWI).
- ✓ SDA (Serial Data) – The line for the master and slave to send and receive data.
- ✓ SCL (Serial Clock) – The line that carries the clock signal.
- ✓ I2C is a serial communication protocol, so data is transferred bit by bit along a single wire (the SDA line).

I2C bus fields and its length

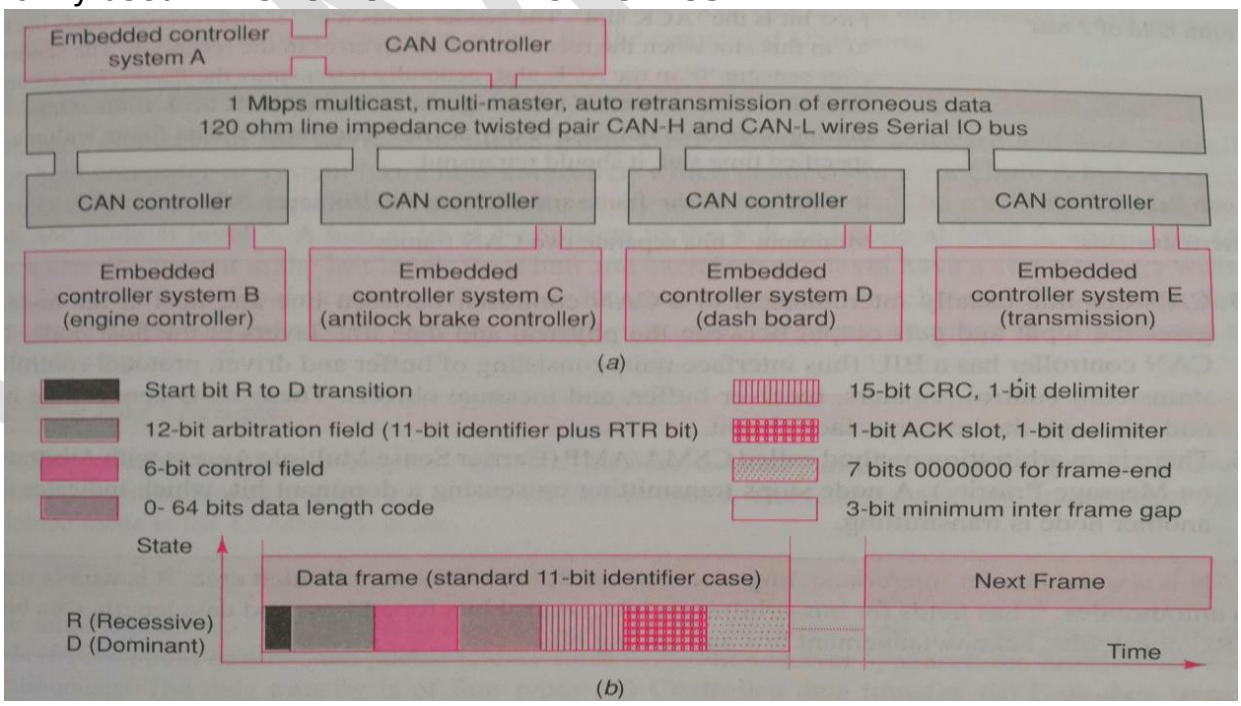


- ✓ **First**field of 1 bit– Start bit similar to one in an UART
- ✓ **Second**field of 7 bits– address field. It defines the slave address, which is being sent the data frame (of many bytes) by the master
- ✓ **Third**field of 1 control bit– defines whether a read or write cycle is in progress
- ✓ **Fourth**field of 1 control bit– defines whether is the present data is an acknowledgment(from slave)
- ✓ **Fifth**field of 8 bits– **I2C device data byte**
- ✓ **Sixth**field of 1-bit– bit NACK(negative acknowledgement) from the receiver. If active then acknowledgment after a transfer is not needed from the slave, else acknowledgement is expected from the slave
- ✓ **Seventh**field of 1 bit – stopbit like in an UART

CAN:

- ✓ A number of devices located and are distributed in a Vehicular Control Network automobile uses a number of distributed embedded controllers. The controllers provide the controls for brakes, engines. electric power. lamps. temperature, air conditioning, car gate. front display panels. and cruising.

- ✓ A number of devices are located and distributed in a car.
- ✓ CAN bus is standard bus in distributed network.
- ✓ Mainly used in **AUTOMOTIVE ELECTRONICS**



The embedded controllers are networked and are controlled through a controller network bit. Figure (a) shows a network of number of CAN controllers and CAN devices on a CAN bus. Figure (b) shows six fields and interframe bits during a transfer of data bits on CAN bus, and timing formats and sequences of frame bits.

First field of 12 bits ─**'arbitration field'**.

◦Contains packet 11-bit destination address and RTR bit (Remote Transmission Request)

- RTR –**1**means packet is for destination address
- RTR –**0**means dominant state, request for data from device
- Second field of 6 bits **control field**.

◦The first bit is for the identifier's extension.

◦The second bit is always '1'.

◦The last 4 bits specify code for data length

- Third field of 0 to 64 bits─Its length depends on the data length code in the control field.
- Fourth field (third if data field has no bit present) of 16 bits─**CRC** (**C**yclic **R**edundancy **C**heck) bits.

◦The receiver node uses it to detect the errors, if any, during the transmission

Fifth field of 2 bits

◦First bit **'ACK slot**'

- Sender sends ACK = '1'and receiver sends back '0'in this slot when the receiver detects an error in the reception.
- Sender after sensing '0' in the ACK slot, generally retransmits the data frame.

◦Second bit **'ACK delimiter'** bit.

- It signals the end of ACK field.
- If the transmitting node does not receive any acknowledgement of data frame within a specified time slot, it should retransmit.
- Sixth field of 7-bits ─**end-of-the frame specification** and has seven '0's

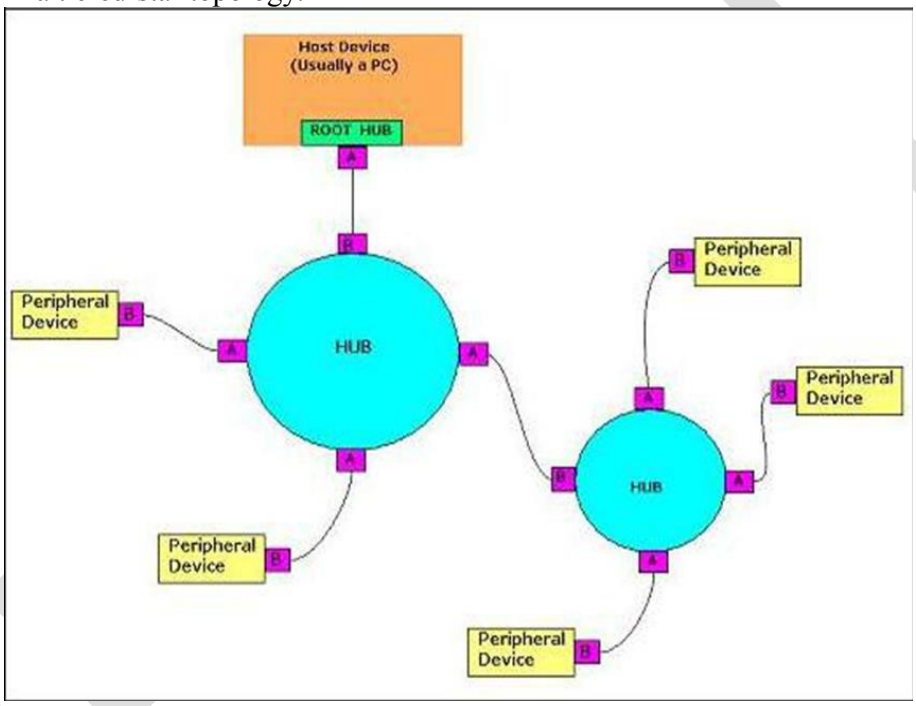
- ✓ The CAN has a serial line, which is bidirectional.
- ✓ A CAN device receives or sends a bit at an instance by operating at the maximum rate of 1Mbps.
- ✓ It employs a twisted pair connection to each node.
- ✓ The pair runs up to a maximum length of 40 m.
- ✓ A CAN version also functions up to 2Mbps. CAN (control area network) bus is a standard bus in distributed network.
- ✓ CAN is mainly used in automotive electronics. It is also used in medical electronics and industrial plant

USB:

- ✓ Universal Serial Bus (USB) is a set of interface specifications for high speed wired communication between electronics systems peripherals and devices with or without PC/computer. The USB was originally developed in 1995 by many of the industry leading companies like Intel, Compaq, Microsoft, Digital, IBM, and Northern Telecom. The major goal of USB was to define an external expansion bus to add peripherals to a PC in easy and simple manner.
- ✓ USB offers users simple connectivity. It eliminates the mix of different connectors for different devices like printers, keyboards, mice, and other peripherals. That means USB-bus allows many peripherals to be connected using a single standardized interface socket. It supports all kinds of data, from slow mouse inputs to digitized audio and compressed video.
- ✓ Various versions USB:
  - ➢ USB1.0: USB 1.0 is the original release of USB having the capability of transferring 12Mbps, supporting up to 127 devices. This USB 1.0 specification model was introduced in January 1996.

- USB1.1: USB 1.1 came out in September 1998. USB 1.1 is also known as full-speed USB. This version is similar to the original release of USB; USB version 1.1 supported two speeds, a full speed mode of 12Mbits/s and a low speed mode of 1.5Mbits/s.
- USB2.0: Hewlett-Packard, Intel, LSI Corporation, Microsoft, NEC, and Philips jointly led the initiative to develop a higher data transfer rate than the 1.1 specifications. USB 2.0, also known as hi-speed USB. This hi-speed USB is capable of supporting a transfer rate of up to 480 Mbps, compared to 12 Mbps of USB 1.1. That's about 40 times as fast!
- USB3.0: It is also called as Super-Speed USB having a data transfer rate of 4.8Gbps(~5Gbps) That means it can deliver over 10x the speed of today's Hi-Speed USB connections
- USB 3.1 : is the latest version of USB also known as Super-Speed USB+, which having data transfer rate of 10Gbps.

✓ The USB system is made up of a host, multiple numbers of USB ports, and multiple peripheral devices connected in a tiered-star topology.



✓ USB can support 4 data transfer types or transfer modes.
- Control
- Isochronous
- Bulk
- Interrupt

Advanced Serial High Speed Buses

✓ An embedded system may need to **connect multi gigabits per second**(Gbps) transceiver (transmit and receive) serial interface(s).

✓ Exemplary products are wireless LAN, Gigabit Ethernet, SONET (OC-48, OC-192, OC-768).

3.8 Embedded System Design and Co-design Issues in System Development Process

There are two approaches for the embedded-system design.
(1) The software development life cycle ends and the life cycle for the process of integrating the software into the hardware begin at the time when a system is designed.
(2) Both cycles concurrently proceed when co-designing a time-critical sophisticated system.
The final design, when implemented, gives the targeted embedded system, and thus the final product. Therefore, an understanding of the (a) software and hardware designs and integrating both into a system, and (b) hardware-software co-designing are important aspects of designing embedded systems. There is a hardware- software trade-off.
The selection of the hardware during hardware design and an understanding of the possibilities and capabilities of hardware during software design are critical especially for a sophisticated embedded-system development.

Choosing the Right Platform

1. Hardware- software trade-off -There is a trade-off between the hardware and software. It is possible that certain subsystems in hardware, I/O memory access, real-time clock, system clock, pulse-width modulation, timer, and serial communication are also implemented by the software.
Hardware implementation provides the following advantages
(i) Reduced memory for the program (ii)Reduced Number of chips but an increased cost (iii) Simple coding for the device drivers (iv) Internally embedded codes, which are more secure than at the external ROM.
Software implementation provides the following advantages:
(i) Easier to change when new hardware versions become available (ii) Programmability for complex operations (iii) faster development time (iv) Modularity and portability (v) Use of a standard software-engineering model (vi) RTOS (vii) Faster speed of operation of complex functions with high-speed microprocessors (viii) Less cost for simple systems.
2. Choosing a Right Platform- System design of an embedded system also involves choosing a right platform. A platform consists of a number of following units.
Processor, ASIP or ASSP, Multiple Processors, System-on-Chip, Memory Other Hardware Units of System, Buses, Software language, RTOS, Code generation tools, tools for finally embedding the software into binary image.
3. Embedded System Processors' Choice

■ Processor-Less System -We have an alternative to a microprocessor microcontroller or DSP. Figure (a) shows the use of a PLC in place of a processor. We can use a PLC for the clothes-in clothes-out type system. A PLC fabricates by the programmable gates, PALs GALs, PLDs and CPLDs.

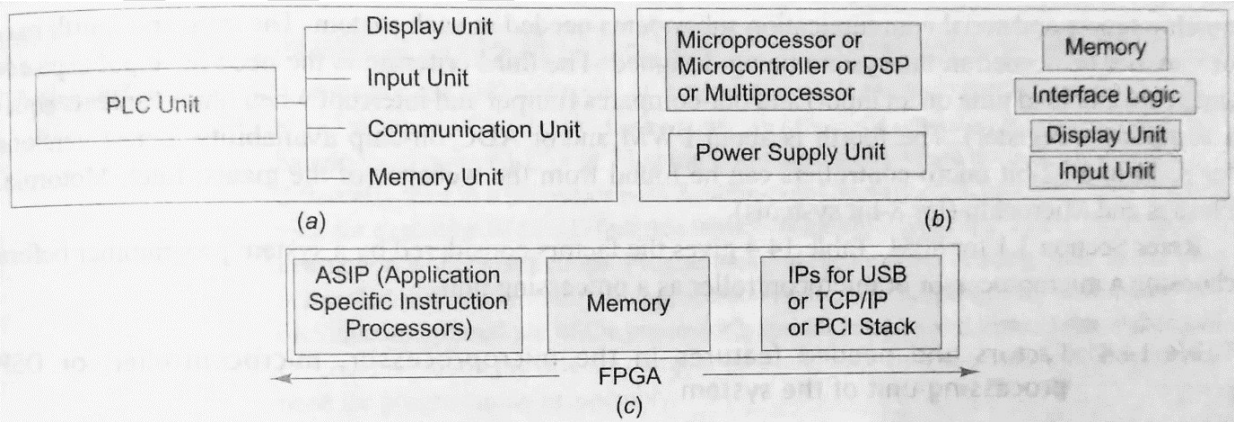


Fig. (a) Use of a PLC in place of a processor (b) Use of a microprocessor, microcontroller or a DSP (c) Processing of functions by using IP embedded into the FPGA instead of processing by the ALU

A PLC has very low operation speed. It also has a very low computational ability. It has very strong interfacing capability with its multiple inputs and outputs. It has system-specific programmability. it is simple in application. Its design implementation is also fast. Automatic chocolate-vending machine another exemplary application of PLC.

■ Fig (b) -System with Microprocessor, Microcontroller or DSP

■ System with Single-purpose Processor or ASSP in VLSI or FPGA Figure (c) shows the processing of functions in using IP embedded into VLSI or FPGA instead of processing b the ALU

A line of action in designing can be use of the IP, synthesising using V HDL like tool and embedding the synthesis into the FPGA. This FPGA implements the functions, which if implemented with the ALU and programmer coding will take a long time to develop.

■ Factors and Needed Features Taken into Consideration- We consider a general. purpose processor choice or choose an ASIP (rnicrocontroller, DSP or network processor). the 32-hit system, 16 KB- on-chip memory. and need of cache, memory management unit, SIMD, M1MD or DSP instructions arise, we use a microprocessor or DSP.

S. No.	Factors for On-Chip Feature	Needed or which one Needed	Available in Chosen Chip
1	8-bit or 16-bit or 32-bit ALU	8/16/32	8/16/32
2	Cache, Memory Management Unit or DSP Calculations	Yes or No	Yes or No
3	Intensive Computations at Fast Rate	Yes or No	Yes or No
4	Total External and Internal Memory up to or more than 64 kB	Yes or No	Yes or No
5	Internal RAM	256/512 B	256/512 B
6	Internal ROM/EPROM/EEPROM	4 kB/8 kB/16 kB	4 kB/8 kB/16 kB
7	Flash	16 kB/64 kB/1 MB/8 MB	16 kB/64 kB/1 MB/8 MB
8	Timer 1, 2 or 3	1/2/3	1/2/3
9	Watchdog Timer	Yes or No	Yes or No
10	Serial Peripheral Interface Full duplex or Serial Synchronous Communication Interface (SI) Half Duplex	Full/Half	Full/Half
11	Serial UART	Yes or No	Yes or No
12	Input Captures and Out-compares	Yes or No	Yes or No
13	PWM	Yes or No	Yes or No
14	Single or Multichannel ADC with or without programmable Voltage reference (single or dual reference)	S/M W/WO V_{ref} S/D	S/M W/WO V_{ref} S/D
15	DMA Controller	Yes or No	Yes or No
16	Power Dissipation	Very low/Low or normal	Very low/Low or normal

Allocation of Addresses to memory, program segments and devices.

1. Functions, Processes, Data and stacks at the various segments of memory

Program routines and processes can have different segments. Each segment has a pointer address and an offset address. Using offset, a code or data word is retrieved from a segment. A stack is a special data structure at the memory. It has a pointer address that always points to the top of a stack.

This pointer address is called a stack pointer. The other data sets, which are also allotted memory are as following: String, Circular queue, A one-dimensional array, A table, A has table, Look-up tables, A list.

2 Device, Internal devices and I/O devices addresses and device drivers

All I/O ports and devices have addresses. These are allocated to the devices according to the system processor and the system hardware configuration. Device addresses are used for processing by the driver. A device has an address, which is usually according to the system hardware or may also be the processor assigned ones. These addresses allocated to the following:

1. Device data Registers or RAM buffers
2. Device Control registers – it saves control bits and may save configuration bits also.
3. Device status registers – it saves flag bits as device ststus. A flag may include the need for servicing and show occurrence of a device-interrupt.

Performance Accelerators …

-An accelerator may include bus interface unit, DMA, read and write units, registers and accelerator cores.

-An accelerator uses a programming model to accelerate, unlike the coprocessor, which has instruction sets for specific tasks.

Performance Accelerators …

-An accelerator may include bus interface unit, DMA, read and write units, registers and accelerator cores.

-An accelerator uses a programming model to accelerate, unlike the coprocessor, which has instruction sets for specific tasks.

- Example - JA108 from Nazonin Communications, a Java accelerator, which accelerates the JAVA code run by 15 to 60 times.
- Another example is a video accelerator, which accelerates video processing tasks.
-Hardware accelerator accelerates code execution. It may be an ASIC, IP core or FPGA.

Porting issues of OS in an embedded platform

Data type may be different on the different platforms, as follows:

(i) unsigned char* (PowerPC, M68HC11/12, M68K, S390),

(ii) unsigned int (ARM)

(iii) unsigned long (Itanium, Alfa, SPARC)

(iv) unsigned short (80x86)

Interrupt vectors are to be defined differently

OS supports these differently on different platforms

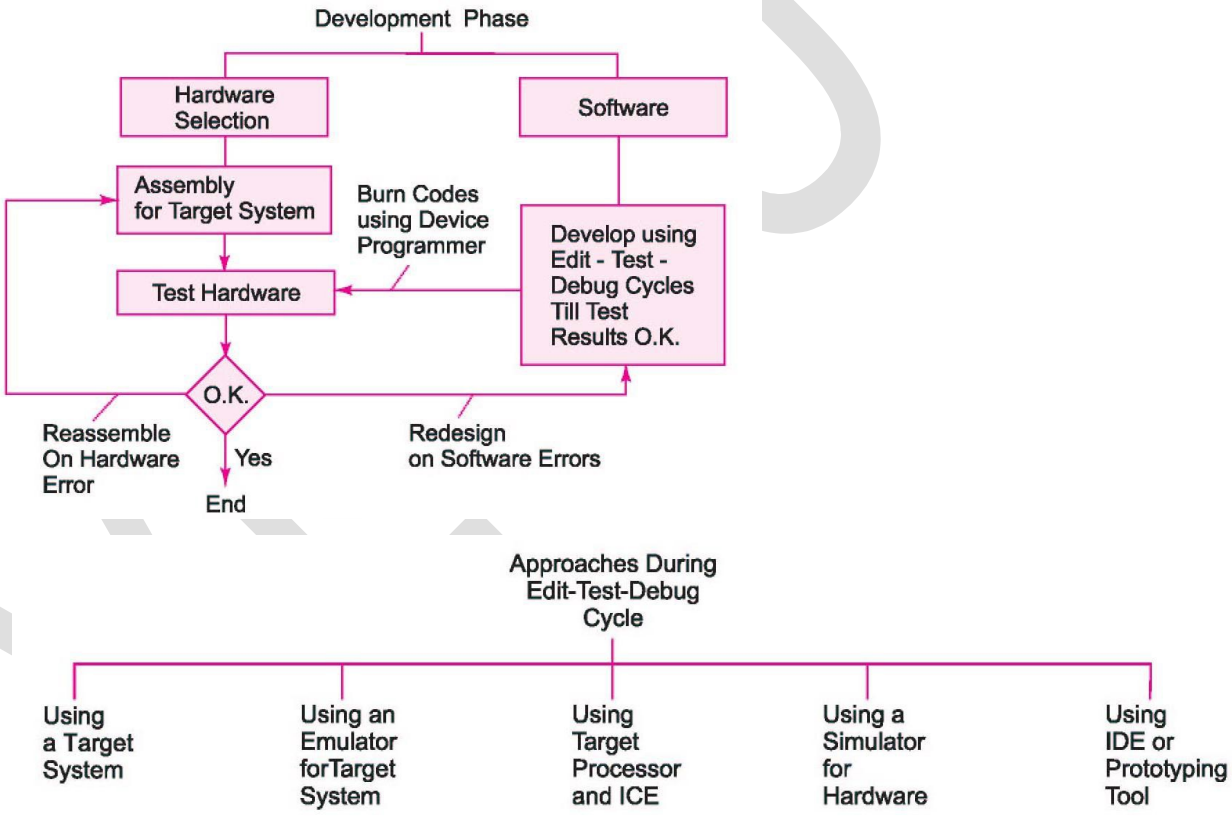
For example, a network interface card supports 32-bit unsigned integers and with a big endian.

(i) Two or three bytes stored at an address from which processor accesses 4-bytes in an access.

(ii) Same data structure at 'C' source file may show differently on different platforms. ['C' takes 16-bit integer on a 16-bit processor and 32-bit integer on a 32-bit processor].

- ✓ Compiler must force the alignment of data by the OS-hardware interface function
- ✓ The following porting issues may arise when the OS is used in an embedded platform: I/O instructions, Interrupt servicing routines, data types, interface specific data types, byte order, data alignment, linked lists, memory page size, time intervals.

3.9 Design Cycle in the Development Phase for an Embedded System



- ✓ Unlike the design of a software application on a standard platform, the design of an embedded system implies that both software and hardware are being designed in parallel. Although this isn't always the case, it is a reality for many designs today. The profound implications of this simultaneous design process heavily influence how systems are designed.

✓ Figure shows the development process or an embedded system and Figure edit-test-debug cycle during implementation phase of the development process. There are cycles of editing-testing-debugging during the development phase, Whereas the processor part once chosen remains fixed, the application software codes have to be perfected by a number of runs and tests. Whereas the cost of the processor is quite small. The cost of developing a final targeted system is quite high and needs a larger time frame Man the hardware circuit design.
The developer uses four main approaches to the edit-test-debug cycles.
I. An IDE or prototype tool
2. A simulator without any hardware
3. Processor only at the target system and uses an in-between ICE (in-circuit-emulator).
4. Target system at the lest stage.

3.10 Uses of Target System or its Emulator and In-Circuit Emulator (ICE):

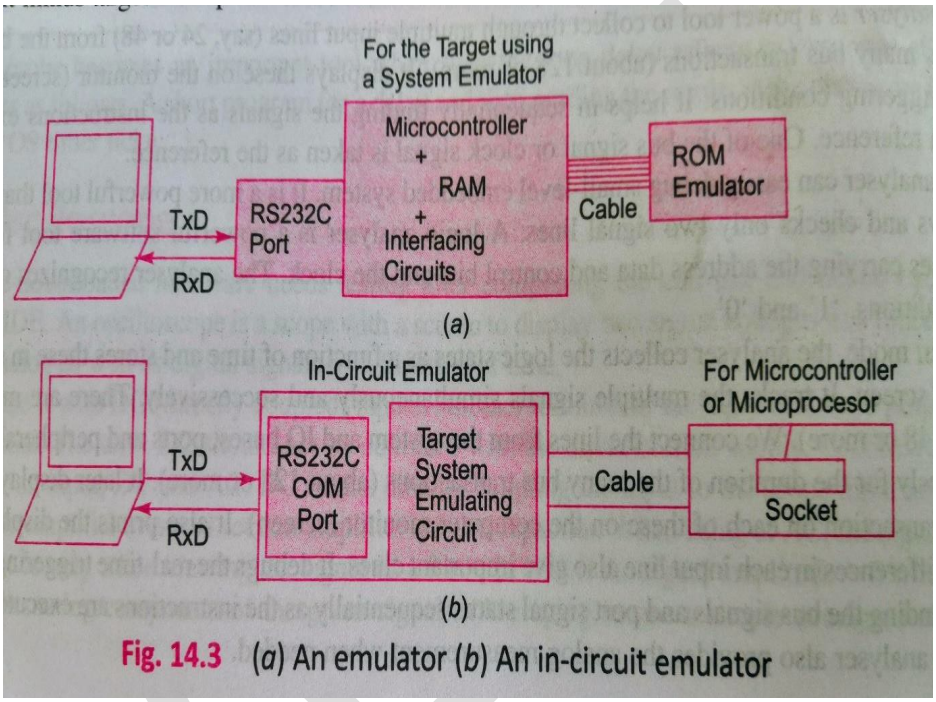


Fig. 14.3 (a) An emulator (b) An in-circuit emulator

- An in-circuit emulator (ICE) is a hardware interface that allows a programmer to change or debug the software in an embedded system. The ICE is temporarily installed between the embedded system and an external terminal or personal computer so that the programmer can observe and alter what takes place in the embedded system, which has no display or keyboard of its own.
- An in-circuit emulator (ICE) provides a window into the embedded system. The programmer uses the emulator to load programs into the embedded system, run them, step through them slowly, and view and change data used by the system's software.
- An emulator gets its name because it emulates (imitates) the central processing unit (CPU) of the embedded system's computer. Traditionally it had a plug that inserts into the socket where

the CPU integrated circuit chip would normally be placed. Most modern systems use the target system's CPU directly, with special JTAG-based debug access. Emulating the processor, or direct JTAG access to it, lets the ICE do anything that the processor can do, but under the control of a software developer.

- ICEs attach a computer terminal or personal computer (PC) to the embedded system. The terminal or PC provides an interactive user interface for the programmer to investigate and control the embedded system. For example, it is routine to have a source code level debugger with a graphical windowing interface that communicates through a JTAG adapter (emulator) to an embedded target system which has no graphical user interface.
- Notably, when their program fails, most embedded systems simply become inert lumps of nonfunctioning electronics. Embedded systems often lack basic functions to detect signs of software failure, such as a memory management unit (MMU) to catch memory access errors. Without an ICE, the development of embedded systems can be extremely difficult, because there is usually no way to tell what went wrong. With an ICE, the programmer can usually test pieces of code, then isolate the fault to a particular section of code, and then inspect the failing code and rewrite it to solve the problem.
- In usage, an ICE provides the programmer with execution breakpoints, memory display and monitoring, and input/output control. Beyond this, the ICE can be programmed to look for any range of matching criteria to pause at, in an attempt to identify the origin of a failure.
- Most modern microcontrollers use resources provided on the manufactured version of the microcontroller for device programming, emulating, and debugging features, instead of needing another special emulation-version (that is, bond-out) of the target microcontroller. Even though it is a cost-effective method, since the ICE unit only manages the emulation instead of actually emulating the target microcontroller, trade-offs must be made to keep prices low at manufacture time, yet provide enough emulation features for the (relatively few) emulation applications.
- **Note: Debugging** is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.

3.11 Uses of software tools for development of an embedded system

- ✓ Software Development Lit (SDK)
- ✓ Source-code Engineering Software
- ✓ RTOS
- ✓ Integrated Development Environment
- ✓ Prototyper
- ✓ Editor
- ✓ Interpreter

- ✓ Compiler
- ✓ Assembler
- ✓ Cross Assembler
- ✓ Testing and debugging tools
- ✓ Locator

3.12 Design metrics of embedded system-low power, high performance, engineering cost, time-to-market.

- ✓ Power dissipation – For many systems, particularly battery operated systems, such as mobile phone or digital camera, the power consumed by the system is an important feature. The battery needs to be recharged less frequently if power dissipation is small.
- ✓ Performance – Execution time taken by instructions in the system is a measure of performance. Smaller execution time for instructions means higher performance. For example, in a mobile phone, if the voice signals process between antenna and speaker in 0.1s, the the phone performance measure is o.1s. Consider another example, a digital camera if shoots and saves a still image of 4M pixels in 0.2s then the camera performance measure is 0.2s(=5 images per second).
- ✓ Process deadlines – Number of processes execute in the system. There are, for example, processes for keypad input processing, refresh of graphic display, and audio and video signals processing. Each process has a deadline within which each of them may be required to complete the computations and give the results.
- ✓ User Interfaces - User Interfaces, for example, user interactions through the keypad and display or GUIs or VUIs.
- ✓ Size - Size of system is measured in terms of (i) physical space required, (ii) RAM in KB and internal flash memory requirements in MB or GB for running software and storing the data, and (iii) number of million logic gates in the hardware.
- ✓ Engineering cost - Initial cost of developing, debugging and testing the hardware and software is called engineering cost. It is a one-time cost. It is therefore, called non-recurring engineering (NRE) cost.
- ✓ Manufacturing cost – Cost of manufacturing each unit tor shipping. Total cost per unit is (NRE cost manufacturing cost for N units)/ N if N units are shipped.
- ✓ Flexibility - Flexibility in a design enables offering the different versions of a product with-out any significant change in engineering cost. Extra functions are necessitated by changing environment or software re-engineering. Software is enhanced for new version by adding the extra functions. The flexible design enables faster marketing. A manufacturer can offer the product in advanced versions later on easily and that too in shorter timeframes.
- ✓ Prototype development time - Time taken in days or months for developing the prototype and in-house testing for the system functionalities. It includes engineering time and prototyping time.
- ✓ Time-to-market - Time taken in days or months after prototype development to put a product for, users and consumers. System safety in terms of accidental tall from hand or table, theft (for phone locking ability and tracing ability), and in terms of user safety when using a product (for example. an automobile brake or engine)

- ✓ Maintenance - Maintenance means changeability and additions in the system, for example, adding or updating soft are. data and hardware. Example of software maintenance is additional set-. ice or functionality of software. Example of data maintenance is additional ring-tones, wallpapers, video-clips in a mobile phone or extending a card expiry-date in case of a smart card. Example of hardware maintenance is providing the additional memory or changing the memory stick in a mobile computer or digital camera.

UNIT-4: Microcontroller fundamentals for basic programming

I/O pin multiplexing, pull up/down registers, GPIO control, Memory Mapped Peripherals, programming System registers, Watchdog Timer, need of low power for embedded systems, System Clocks and control, Hibernation Module on TM4C, Active vs Standby current consumption. Introduction to Interrupts, Interrupt vector table, interrupt programming. Basic Timer, Real Time Clock (RTC), Motion Control Peripherals: PWM Module & Quadrature Encoder Interface (QEI).

I/O Pin Multiplexing:

The System-On-Chip (Microcontroller) has a lot of functionality but a limited number of pins (or pads). Even though a single pin can only perform one function at a time, they can be configured internally to perform different functions. This is called pin multiplexing. i.e one pin can perform multiple functions. The regular function of a pin is to perform parallel I/O. Most of the pins have an alternative function.

I/O pins on Tiva microcontrollers have a wide range of alternative functions:

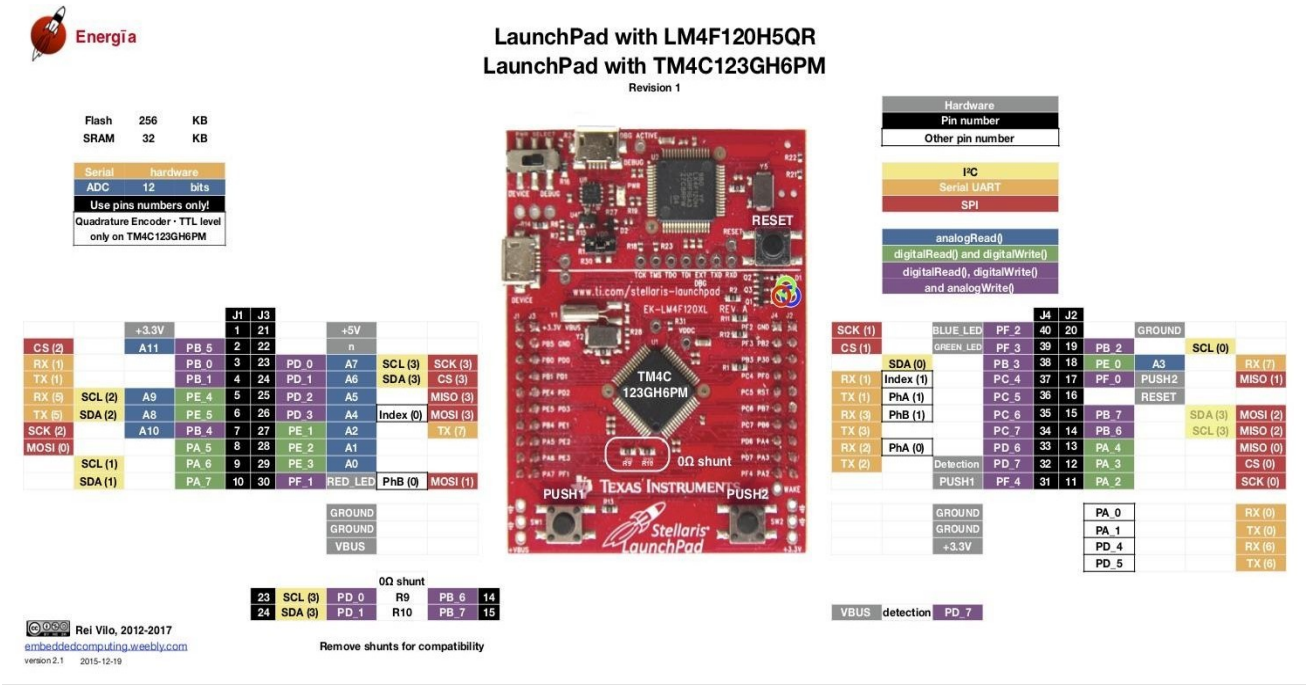
- **UART** **Universal asynchronous receiver/transmitter**
- **SSI** **Synchronous serial interface**
- **I^2C** **Inter-integrated circuit**
- **Timer** **Periodic interrupts, input capture, and output compare**
- **PWM** **Pulse width modulation**
- **ADC** **Analog to digital converter, measure analog signals**
- **Analog Comparator** **Compare two analog signals**
- **QEI** **Quadrature encoder interface**
- **USB** **Universal serial bus**
- **Ethernet** **High-speed network**
- **CAN** **Controller area network**

The **UART** can be used for serial communication between computers. It is asynchronous and allows for simultaneous communication in both directions.
The **SSI** is alternately called serial peripheral interface (SPI). It is used to interface medium-speed I/O devices.
I^2C is a simple I/O bus that we will use to interface low speed peripheral devices. Input capture and output compare will be used to create periodic interrupts and measure period, pulse width, phase, and frequency.
PWM outputs will be used to apply variable power to motor interfaces. In a typical motor controller, input capture measures rotational speed, and PWM controls power. A PWM output can also be used to create a DAC.

The **ADC** will be used to measure the amplitude of analog signals and will be important in data acquisition systems. The analog comparator takes two analog inputs and produces a digital output depending on which analog input is greater.
The **QEI** can be used to interface a brushless DC motor. **USB** is a high-speed serial communication channel.

The **Ethernet** port can be used to bridge the microcontroller to the Internet or a local area network. The **CAN** creates a high-speed communication channel between microcontrollers and is commonly found in automotive and other distributed control applications.

To support above all functionalities, 43pins (available on TIVA) is not enough. So in order to meet the embedded system characteristics small size and weight, all the pins functionality multiplexed together and made it available with in 43GPIO pins. The below figure shows the energia pin map of TIVA microcontroller in which we can observe one pin can perform multiple functionality.



I/O Pull-Up and Pull-Down registers:

Digital logic gates can be used for connection to external circuits or devices but care must be taken to ensure that their inputs or outputs function correctly and provide the expected switching condition. Modern digital logic gates, IC's and micro-controllers contain many inputs, called "pins" as well as one or more outputs, and these inputs and outputs need to be correctly set, either HIGH or LOW for the digital circuit to function correctly.

We know that logic gates are the most basic building block of any digital logic circuit and that by using combinations of the three basic gates, the AND gate, the OR gate and NOT gate, we can construct quite complex combinational circuits. But being digital, these circuits can only have one of two logic states, called the logic "0" state or the logic "1" state.

These logic states are represented by two different voltage levels with any voltage below one level regarded as a logic "0", and any voltage above another level regarded as logic "1". So for example,

if the two voltage levels are 0V and +5V, then the 0V represents a logic “0” and the +5V represents a logic “1”.

If the inputs to a digital logic gate or circuit are not within the range by which it can be sensed as either a logic “0” or a logic “1” input, then the digital circuit may false trigger as the gate or circuit does not recognise the correct input value, as the HIGH may not be high enough or the LOW may not be low enough.i.e. called floating state or undefined state.

The pull up and pull down resistors are used for preventing a node from floating. Consider following images.

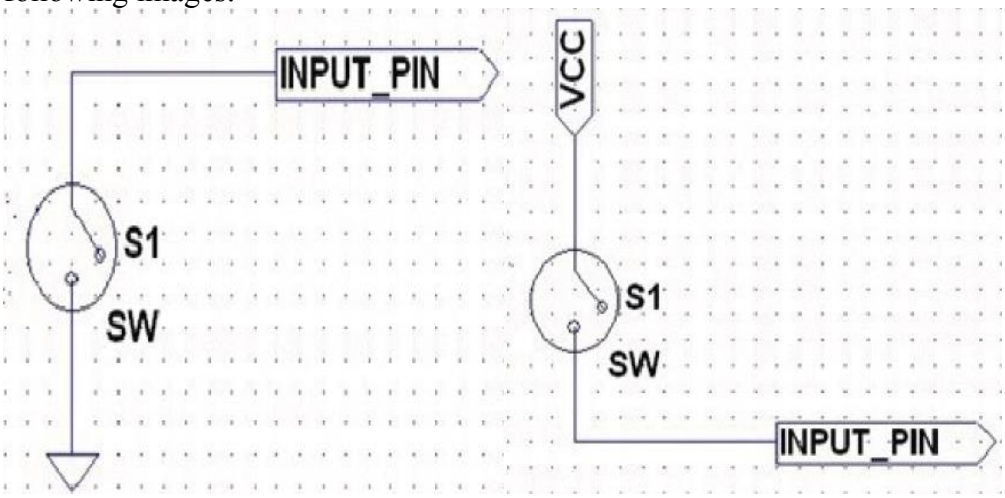


Fig 1: An Example of Floating node

Fig 2: Another Example of Floating Node

So lets consider Fig 1. When the switch is pressed, voltage at input pin will be equal to VCC. But when switch is open (Notpressed), state of input pin is undefined. We can’t be sure about the voltage level of the Input Pin. So we can write the statesof Input Pin based on switch state as:

SwitchState	Open(Not Pressed)	Closed (Pressed)
InputPinState	Undefined	High (Equal to VCC)

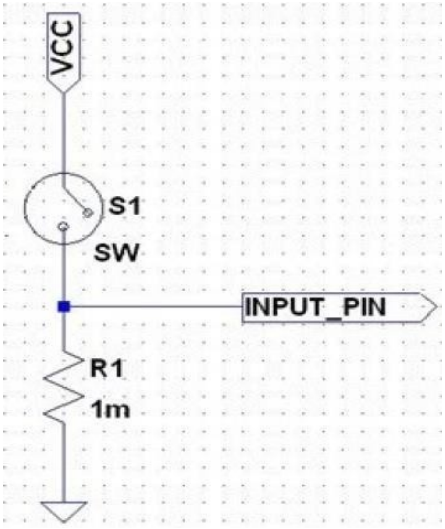
Similarly for Fig 2 we can write the table as

SwitchState	Open(Not Pressed)	Closed (Pressed)
InputPinState	Undefined	Low (Equal to GND)

So we can see that, if we use the Switch as input device to a microcontroller in these configuration, the device is going to facesome undefined states (Floating State). So it may lead to ambiguous readings from the switch. Everyone connecting a switchto a microcontroller must avoid theseconfiguration for their switches.

Pull Down Resistor:

Pull down resistor will pull a floating node to logic level low i.e. 0. So after connecting to a Pull down resistor the Fig 1 will looklike as follow:



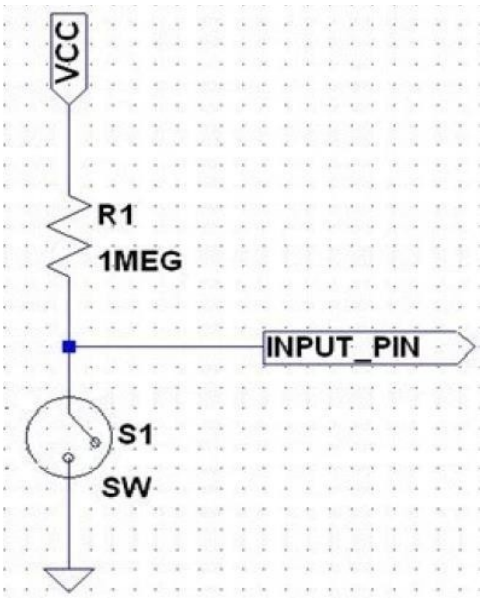
So now we connected a very High value resistor from the floating node to Ground. Lets see how its going to help us.

SwitchState	InputPinState
Open(Not Pressed)	Low
Closed (Pressed)	High (Equal to VCC)

When the switch is in Open state the resistor will try to pull down the node to Ground level. Hence it is named as Pull DownResistor.

Pull Up Resistor:

By now you must have guessed the use of pull up resistor. The pull up resistor pulls up the floating node to a High logic value



Here the resistor R1 will serve the purpose of pulling the Input Pin high when the switch is not pressed. But now the Voltageat Input Pin will be slightly less than VCC based on your R1 value.

But as you are using it as input to a Microcontroller, so it isnot going to cost you in terms of errors as for Digital Devices there is a threshold between High and Low level. Every othervariation is not of much use. So the table for switch with pull up resistor will be

SwitchState	InputPinState
Open(Not Pressed)	High
Closed (Pressed)	Low (Equal to GND)

GPIO Control

General Purpose Input/output (GPIO) refers to pins on a board which are connected to the microcontroller in a special configuration. Users can control the activities of these pins in real-time.

GPIOs are used in devices like SoC, PLDs, and FPGAs, which inherit problems of pin scarcity.
They are used in multifunction chips like audio codecs and video cards for connectivity
They are extensively used in embedded systems designs to interface the microcontroller to external sensors and driver circuits.

GPIO pins can be configured as both input and output. There are generally two states in a GPIO pin, High=1 and Low=0. These pins can be easily enabled and disabled by the user. A GPIO pin can be configured as input and used as an interrupt pin typically for wakeup events.

Voltage levels of GPIOs are critical and it is necessary that users take note of these voltages before interfacing. Tolerant voltages at GPIO pins are not same as the board supply voltage. Some GPIOs have 5 V tolerant inputs: even if the device has a low supply voltage (say 2 V), it can accept 5 V without damage. However, a higher voltage may cause damage to the circuitry or may even fry the board.

GPIO Pins in Tiva Launchpad

In the Tiva Launchpad, the GPIO module is composed of six physical GPIO blocks. Each of these blocks corresponds to an individual GPIO port. There are six ports in Tiva C series microcontrollers namely, Port A through F. This GPIO module supports up to 43 programmable input/output pins. (Although it depends on the peripherals being used.)
The GPIO module has the following features:

The GPIO pins are flexibly multiplexed. This allows it to be also used as peripheral functions. The GPIO pins are 5-V-tolerant in input configuration
Ports A-G are accessed through the Advanced Peripheral Bus (APB)
Fast toggle capable of a change every clock cycle for ports on AHB, every two clock cycles for ports on APB.

Most of the GPIO functions can operate on more than one GPIO pin (within a single module) at a time. Can be configured to be a GPIO or a peripheral pin. On reset, the default is GPIO. Note that

not all pins on all parts have peripheral functions, in which case, e the pin is only useful as a GPIO. The below table shows the pin mux utility in Tiva Launchpad.

Advanced features of GPIO in Tiva Launchpad

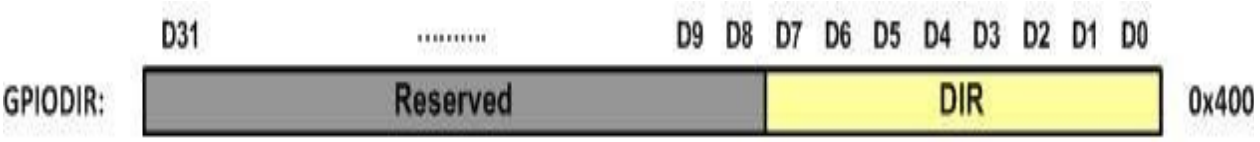
The GPIO module in Tiva Launch Pad can be used in advanced configurations also. They can be used for programmable control through interrupts. These interrupts can be triggered on rising, falling or both edges of the clock. They can also be levelled sensitive for both high and low states. The state of these pins is retained during hibernate mode. The programmable control for GPIO pad configuration includes

Weak pull-up or pull-down resistors
2-mA, 4-mA, and 8-mA pad drive for digital communication; up to four pads can sink 18-mA for high-current applications
Slew rate control for 8-mA pad drive
Open drain enables
Digital input enables

Direction and Data Registers

Generally, every microcontroller has a minimum of two registers associated with each of I/O ports, namely Data Register and Direction Register. As the name suggests, Direction Register decides which way the data will flow; Input or Output. Data register stores the data coming from the microcontroller or from the pin.

The value assigned to Direction register is configure the pin as either input or output. When the direction register is properly configured, the Data register can be used to write to the pin or read data from the pin. When the Direction register is configured as output, the information on the Data register is driven to the microcontroller pin. Similarly, when Direction register is configured as input, the information on the microcontroller pin is written to the Data register.



Note: D0 to D7 are used to set the direction for pins 0 to 7 of the port.
1: Output
0: Input

Direction Register Operation: In Tiva C series Launchpad, the GPIO Direction (GPIODIR) register isused to configure each individual pin as an input or output. When the data direction bit is cleared, the GPIO is configured as an input, and the corresponding data register bit captures and stores the value on the GPIO port. When the data direction bit is set, the GPIO is configured as an output, and the corresponding data register bit is driven out on the GPIO port.

Data Register Operation: In Tiva C Series Launchpad, GPIODATA register is the data register in which the values written in this register are transferred onto the GPIO port pins if the respective pins have been configured as outputs through the GPIO Direction (GPIODIR) register. The GPIO ports allow for the modification of individual bits in the GPIO Data (GPIODATA) register by using bits of the address bus as a mask. In this manner, we can modify individual GPIO pins in a single instruction without affecting the state of the other pins.

Programming System Registers

//Blinking GREEN LED with delay of 2 seconds
#include <stdint.h> #include
<stdbool.h> #include
"inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/gpio.h" **int**
main(void)

```
{    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|
SYSCTL_OSC_MA IN);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
while(1){
GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x08);
SysCtlDelay(20000000); GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|
GPIO_PIN_3, 0x00);
SysCtlDelay(20000000);
}
}
```

Memory Mapped Peripherals

Peripheral and Memory Address

A 32-bit processor can have 4 GB (=2^{32}) of address spaces. It depends on the architecture of the CPU how these address spaces are segregated, among the memory and peripherals.

Peripheral Addressing

There are two complementary methods of addressing I/O devices for input and output between CPU and peripheral. These are known as memory mapped I/O (MMIO) and port mapped I/O (PMIO). www.ti.com *Peripheral and Memory Address*

In MMIO, same address bus is used to address both memory and peripheral devices. The address bus of the CPU is shared between the peripheral devices and memory devices attached to the CPU. Thus, any address accessed by the CPU may denote an address in the memory or a register of attached peripheral. In these architectures, same CPU instructions used for memory access can also be used for I/O access.

In PMIO, peripheral devices possess a separate address bus from general memory devices. This is accomplished in most architectures by providing a separate address bus dedicated to the peripheral devices attached to the CPU. In these CPUs, the instruction set includes separate instructions to perform I/O access.

A TM4C123GH6PM chip employs MMIO which implies that the peripherals are mapped into the 32-bit address bus.

A TM4C123GH6PM chip consists of a 256 KB of Flash memory and 32 KB of SRAM. Table 5 shows the memory map of a TM4C123GH6PM chip with addresses.

Flash Memory

Flash memory is structured into multiple blocks of single KB size which can be individually written to and erased. Flash memory is used for store program code. Constant data used in a program can also be stored in this memory. Lookup tables are used in many designs for performance improvement. These lookup tables are stored in this memory.

	Allocated size	*Allocated address*
Flash	256KB	0x00000000 to 0x0003FFFF
Bit-banded on-chip SRAM	32 KB	0x20000000 to 0x20007FFF
Peripheral	All the peripherals	0x40000000 to 0x400FFFFF

Table: Memory Mapping in TM4C123GH6PM Chip

SRAM

The on-chip SRAM starts at address 0x2000.0000 of the device memory map. ARM provides a technology to reduce occurrences of read-modify-write (RMW) operations called bit-banding. This technology allows address aliasing of SRAM and peripheral to allow access of individual bits of the same memory in single atomic operation. For SRAM, the bit-band base is located at address 0x2200.0000.

The SRAM is implemented using two 32-bit wide SRAM banks (separate SRAM arrays). The banks are partitioned in a way that one bank contains all, even words (the even bank) and the other contains all odd words (the odd bank). A write access that is followed immediately by a read access to the same bank. This incurs a stall of a single clock cycle.

Internal ROM

The internal ROM of the TM4C123GH6PM device is located at address 0x0100.0000 of the device memory map. The ROM contains:

TivaWare™ Boot Loader and vector table

TivaWare TM Peripheral Driver Library (DriverLib) release of product-specific peripherals and interfaces

Advanced Encryption Standard (AES) cryptography tables Cyclic

Redundancy Check (CRC) error detection functionality

The boot loader is used as an initial program loader (when the Flash memory is empty) as well as an application-initiated firmware upgrade mechanism (by calling back to the boot loader). The Peripheral Driver Library, APIs in ROM can be called by applications, reducing flash memory requirements and freeing the Flash memory to be used for other purposes (such as additional features in the application). Advance Encryption Standard (AES) is a publicly defined encryption standard used by the U.S. Government and Cyclic Redundancy Check (CRC) is a technique to validate if a block of data has the same contents as when previously checked.

Peripheral

All Peripheral devices, timers, and ADCs are mapped as MMIO in address space 0x40000000 to 0x400FFFFF. Since the number of supported peripherals is different among ICs of ARM families, the upper limit of 0x400FFFFF is variant.

Watchdog Timer

Every CPU has a system clock which drives the program counter. In every cycle, the program counter executes instructions stored in the flash memory of a microcontroller. These instructions are executed sequentially. There exist possibilities where a remotely installed system may freeze or run into an unplanned situation which may trigger an infinite loop. On encountering such situations, system reset or execution of the interrupt subroutine remains the only option. Watchdog timer provides a solution to this.

A watchdog timer counter enters a counter lapse or timeout after it reaches certain count. Under normal operation, the program running the system continuously resets the watchdog timer. When the system enters an infinite loop or stops responding, it fails to reset the watchdog timer. In due time, the watchdog timer enters counter lapse. This timeout will trigger a reset signal to the system or call for an interrupt service routine (ISR).

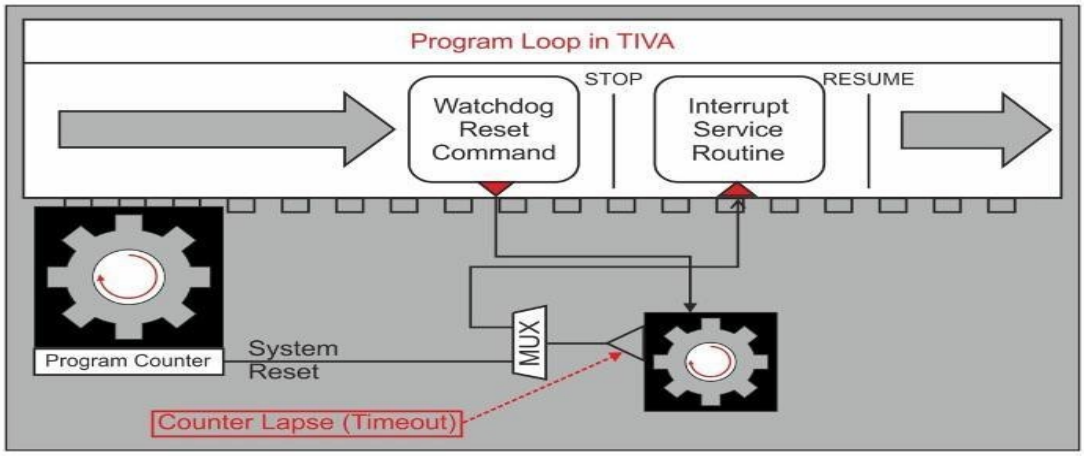


Fig : Operation of Watchdog Timer

TM4C123GH6PM microcontroller has two Watchdog Timer modules, one module is clocked by the system clock (Watchdog Timer 0) and the other (Watchdog Timer 1) is clocked by the PIOSC therefore it requires synchronizers.

Features of Watchdog Timer in TM4C123GH6PM controller:

32-bit down counter with a programmable load register

Separate watchdog clock with an enable

Programmable interrupt generation logic with interrupt masking and optional NMI function

Lock register protection from runaway software

Reset generation logic with an enable/disable

User-enabled stalling when the microcontroller asserts the CPU halt flag during debug

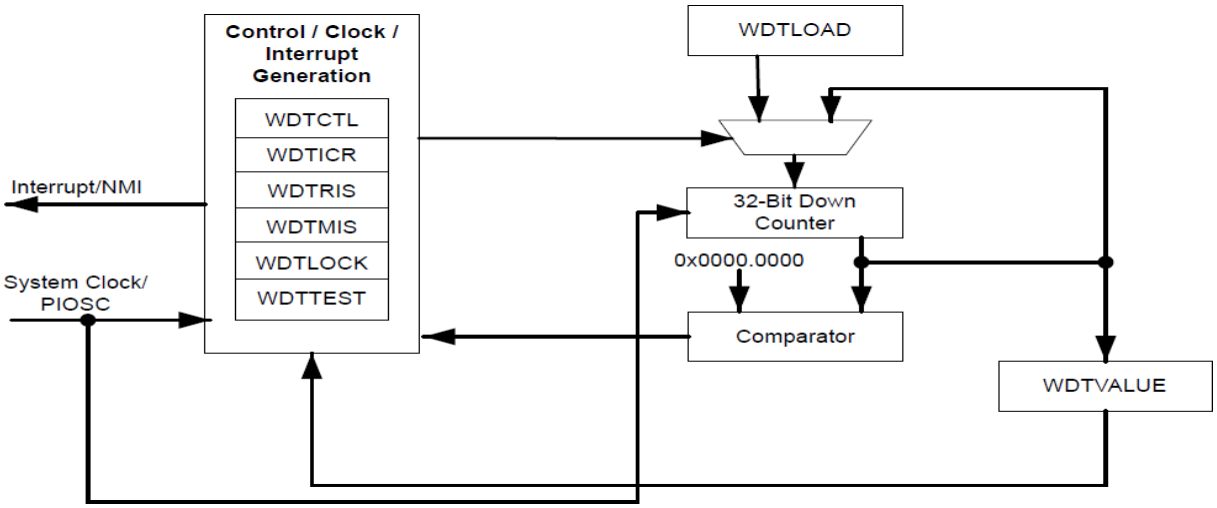


Fig: Block diagram of Watchdog Timer

The watchdog timer can be configured to generate an interrupt to the controller on its first time out, and to generate a reset signal on its second time-out. Once the watchdog timer has been configured, the lock register can be written to prevent the timer configuration from being inadvertently altered.

Need for Low Power Microcontroller

It is imperative for an embedded design to be low on its power consumption. Most embedded systems and devices run on battery. Power demands are increasing rapidly, but battery capacity cannot keep up with its pace. Therefore, a microcontroller which inherently consumes very less power is always encouraging. However, embedded systems engineers usually need to optimize between power and performance. Power and performance are inversely proportional to each other.

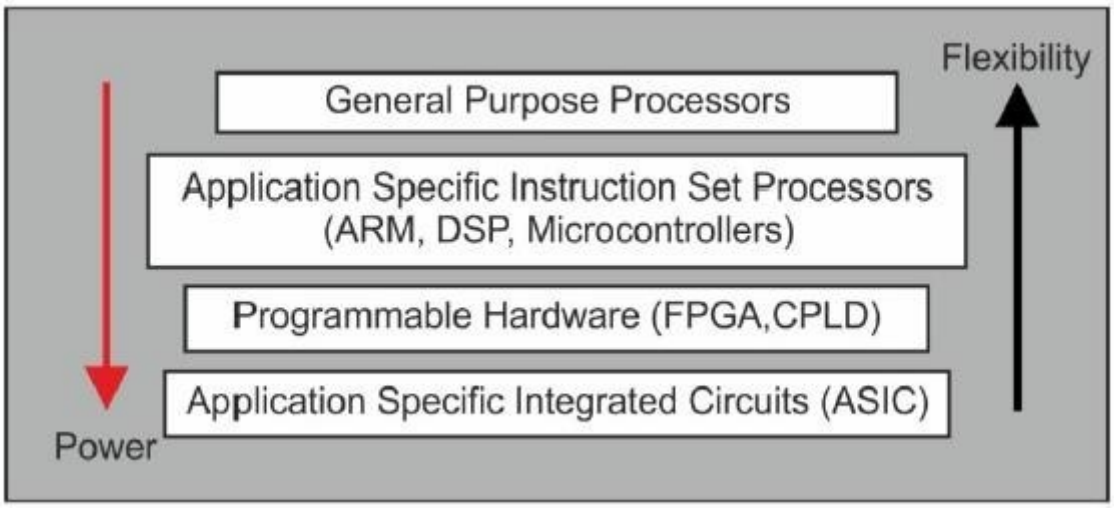


Fig: Power vs Flexibility.

System Clocks and control

System Clock:

Fundamental Clock Sources

Clock is like heart of the controller and used to synchronize all the peripherals in the microcontroller. There are multiple clock sources for use in the TIVA microcontroller:

- Precision Internal Oscillator (PIOSC). The precision internal oscillator is an on-chip clock source that is the clock source the microcontroller uses during and following POR. It does not require the use of any external components and provides a 16-MHz clock with ±1% accuracy with calibration and ±3% accuracy across temperature.

The PIOSC allows for a reduced system cost in applications that require an accurate clock source. If the main oscillator is required, software must enable the main oscillator following reset and allow the main oscillator to stabilize before changing the clock reference. If the Hibernation Module clock source is a 32.768-kHz oscillator, the precision internal oscillator can be trimmed

by software based on a reference clock for increased accuracy. Regardless of whether or not the PIOSC is the source for the system clock, the PIOSC can be configured to be the source for the ADC clock as well as the baud clock for the UART and SSI.

■ Main Oscillator (MOSC). The main oscillator provides a frequency-accurate clock source by one of two means: an external single-ended clock source is connected to the OSC0 input pin, or an external crystal is connected across the OSC0 input and OSC1 output pins. If the PLL is being used, the crystal value must be one of the supported frequencies between 5 MHz to 25 MHz (inclusive). If the PLL is not being used, the crystal may be any one of the supported frequencies between 4 MHz to 25 MHz. The supported crystals are listed in the XTAL bit field in the **RCC** register.

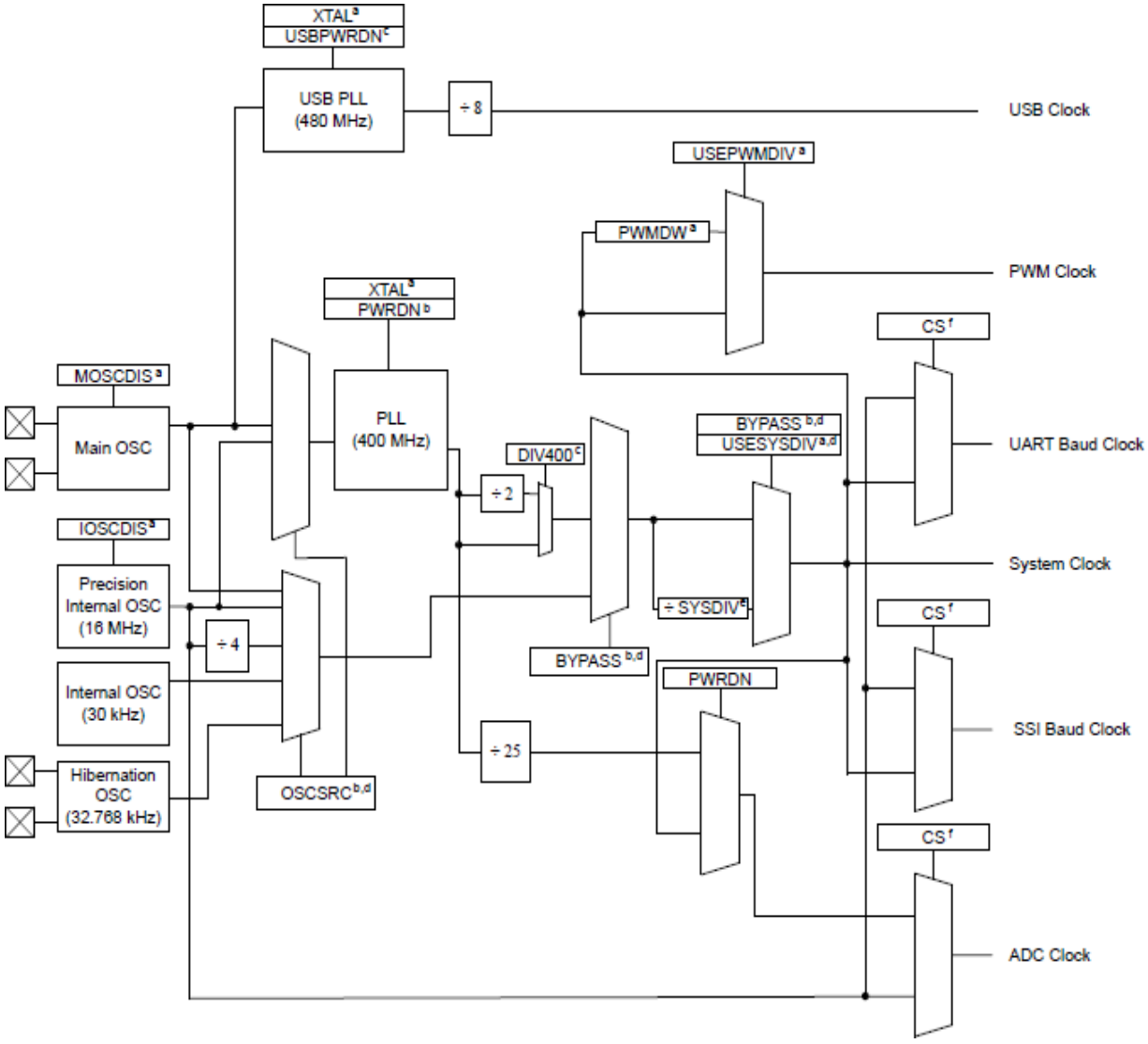


Fig: Clock tree in TIVA

■ Low-Frequency Internal Oscillator (LFIOSC). The low-frequency internal oscillator is intended for use during Deep-Sleep power-saving modes. This power-savings mode benefits from reduced internal switching and also allows the MOSC to be powered down. In addition, the PIOSC can be powered down while in Deep-Sleep mode.

■ Hibernation Module Clock Source**.** The Hibernation module is clocked by a 32.768-kHz oscillator connected to the XOSC0 pin. The 32.768-kHz oscillator can be used for the system clock, thus eliminating the need for an additional crystal or oscillator. The Hibernation module clock source is intended to provide the system with a real-time clock source and may also provide an accurate source of Deep-Sleep or Hibernate mode power savings.

The above Figure shows the clock tree of TIVA microcontroller. From the different clock sources, different clocks are derived. i.e. USB clock, PWM clock, UART baud clock, System clock, SSI baud clock, ADC clock.

System Control:

For power-savings purposes, the peripheral-specific RCGCx, SCGCx, and DCGCx registers (for example, RCGCWD) control the clock gating logic for that peripheral or block in the system while the microcontroller is in Run, Sleep, and Deep-Sleep mode, respectively. These registers are locatedin the System Control register map starting at offsets 0x600, 0x700, and 0x800, respectively. There must be a delay of 3 system clocks after a peripheral module clock is enabled in the RCGC(Run Mode Clock gate control) register before any module registers are accessed.

There are four levels of operation for the microcontroller defined as:

- Run mode
- Sleep mode
- Deep-Sleep mode
- Hibernate mode

Run Mode

In Run mode, the microcontroller actively executes code. Run mode provides normal operation of the processor and all of the peripherals that are currently enabled by the peripheral-specific **RCGC** registers. The system clock can be any of the available clock sources including the PLL.

Sleep Mode

In Sleep mode, the clock frequency of the active peripherals is unchanged, but the processor and the memory subsystem are not clocked and therefore no longer execute code. Sleep mode is entered by the Cortex-M4F core executing a WFI (Wait for Interrupt) instruction. Any properly configured interrupt event in the system brings the processor back into Run mode.

Peripherals are clocked that are enabled in the peripheral-specific SCGC registers when auto-clock gating is enabled (see the RCC register) or the peripheral-specific RCGC registers when the auto-

clock gating is disabled. The system clock has the same source and frequency as that during Run mode.

Deep-Sleep Mode

In Deep-Sleep mode, the clock frequency of the active peripherals may change (depending on the Deep-Sleep mode clock configuration) in addition to the processor clock being stopped. An interrupt returns the microcontroller to Run mode from one of the sleep modes; the sleep modes are entered on request from the code. Deep-Sleep mode is entered by first setting the SLEEPDEEP bit in the System Control (SYSCTRL) register and then executing a WFI instruction. Any properly configured interrupt event in the system brings the processor back into Run mode.

Hibernate Mode

In this mode, the power supplies are turned off to the main part of the microcontroller and only the Hibernation module's circuitry is active. An external wake event or RTC event is required to bring the microcontroller back to Run mode. The Cortex-M4F processor and peripherals outside of the Hibernation module see a normal "power on" sequence and the processor starts running code. Software can determine if the microcontroller has been restarted from Hibernate mode by inspecting the Hibernation module registers.

Hibernation Module on Tiva Microcontroller

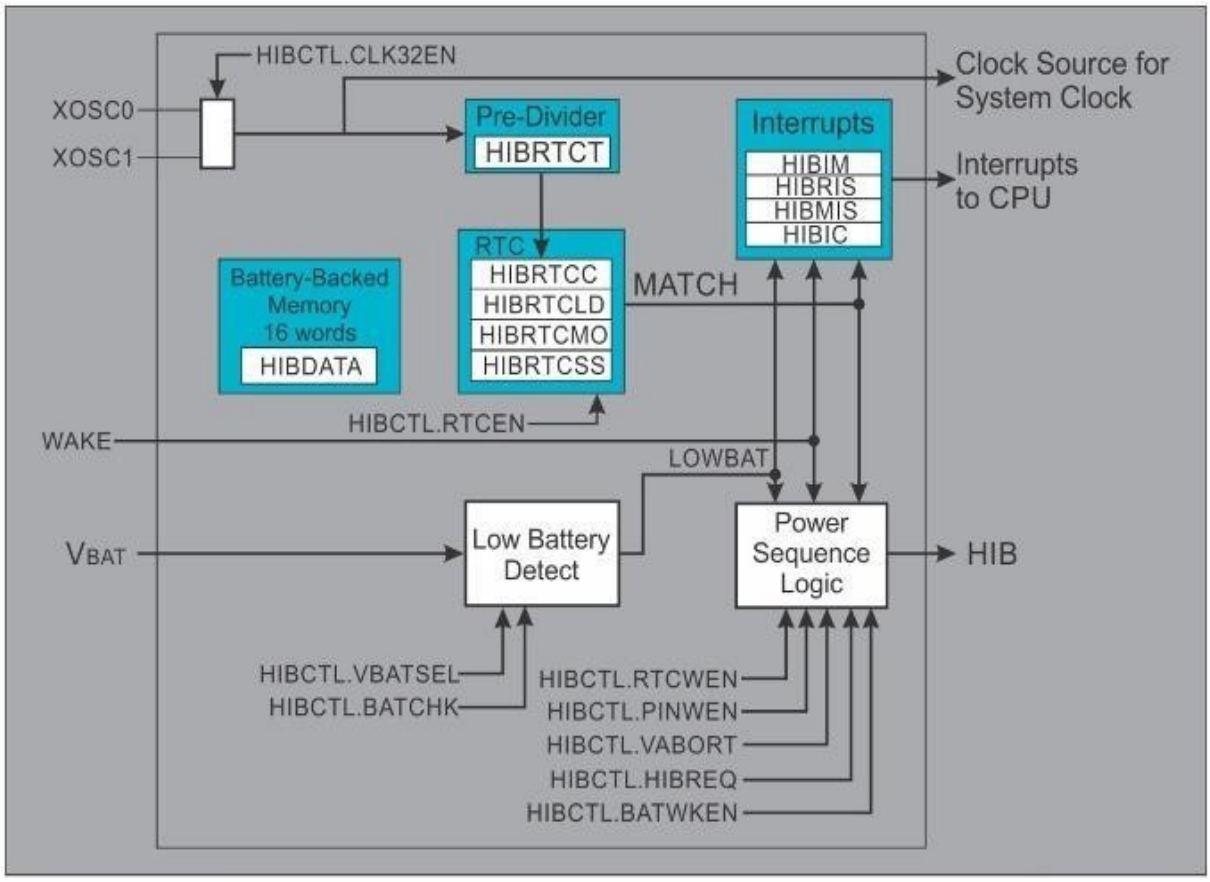


Fig : Block diagram of Hibernation module

This module manages to remove and restore power to the microcontroller and its associated peripherals. This provides a means for reducing system power consumption. When the processor and peripherals are idle, power can be completely removed if the Hibernation module is only the one powered.

To achieve this, the Hibernation (HiB) Module is added with following features:

(i) A Real-Time Clock (RTC) to be used for wake events (ii) A battery backed SRAM for storing and restoring processor state. The SRAM consists of 16 32-bit word memory.

The RTC is a 32- bit seconds counter and 15- bit sub second counter. It also has an add-in trim capability for precision control over time. The Microprocessor has a dedicated pin for waking using external signal. The RTC and the SRAM are operational only if there is a valid battery voltage. There is a VDD30N mode, which provides GPIO pin state during hibernation of the device.

Thus we are actually shutting the power off for the device or part at the lowest power mode. Under such circumstances, it is safe to assume that in the wake up we are actually coming out of reset. But this will allow the device to the keep the GPIO pins in their state without resetting them. A mechanism for power control is used to shut down the part. In TM4C123GH6PM we have an on-chip power controller which controls power for the CPU only. There is also a pin output from the microcontroller which is used for system power control.

It should be duly noted that in TIVA Launchpad, the battery voltage is directly connected to the processor voltage and it is always valid. But in a custom design with TM4C123GH6PM microcontroller running on a battery, if the battery voltage is not valid, it will not go into hibernation mode.

The Hibernation module of TM4C123GH6PM provides two mechanisms for power control:

- The first mechanism uses internal switches to control power to the Cortex-M4F.
- The second mechanism controls the power to the microcontroller with a control signal (HIB) that signals an external voltage regulator to turn on or off.
- The Hibernation module power source is determined dynamically. The supply voltage of the Hibernation module is the larger of the main voltage source (VDD) or the battery voltage source (VBAT).

Hibernate mode can be entered through one of two ways:

The user initiates hibernation by setting the HIBREQ bit in the Hibernation Control (HIBCTL) register.
Power is arbitrarily removed from VDD while a valid VBAT is applied

Programming Hibernation Module

This code can be compiled and executed on a TIVA Launchpad. When this code executes, the GREEN LED glows continuously. We can observe that after 4s, the system automatically goes into sleep and the LED stops glowing. When SW2 (switch on the right hand bottom corner of the

Launchpad) is pressed, it triggers a wake event and the GREEN LED starts glowing again. Now, after 4s, the system goes to sleep again. This shows that, the wakeup process is the same as powering up. When the code starts, we can determine that the processor woke from hibernation and restore the processor state from the memory.

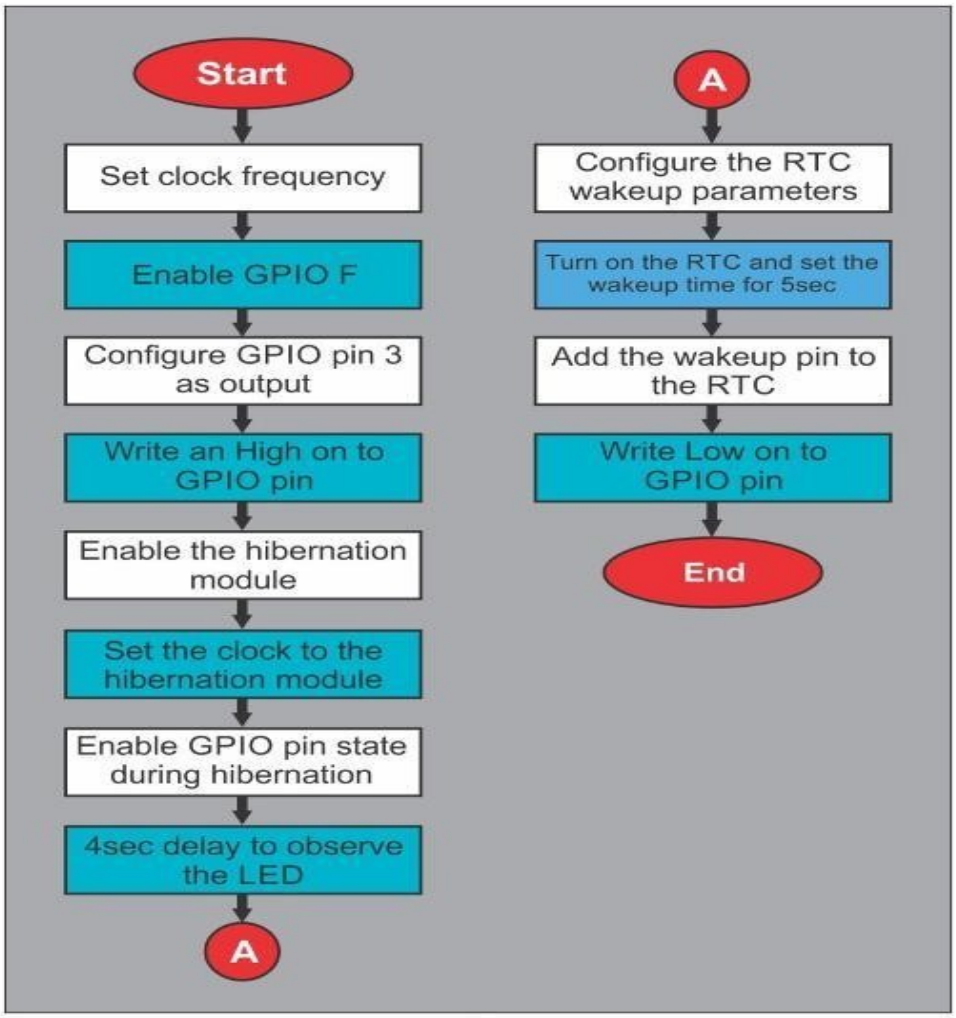


Fig : Flowchart for programming hibernation module

```
#include <stdint.h> #include
<stdbool.h> #include
"utils/ustdlib.h" #include
"inc/hw_types.h"
#include "inc/hw_memmap.h"
#include "driverlib/sysctl.h"

#include "driverlib/pin_map.h"
#include "driverlib/debug.h"
#include "driverlib/hibernate.h"
#include "driverlib/gpio.h"
```

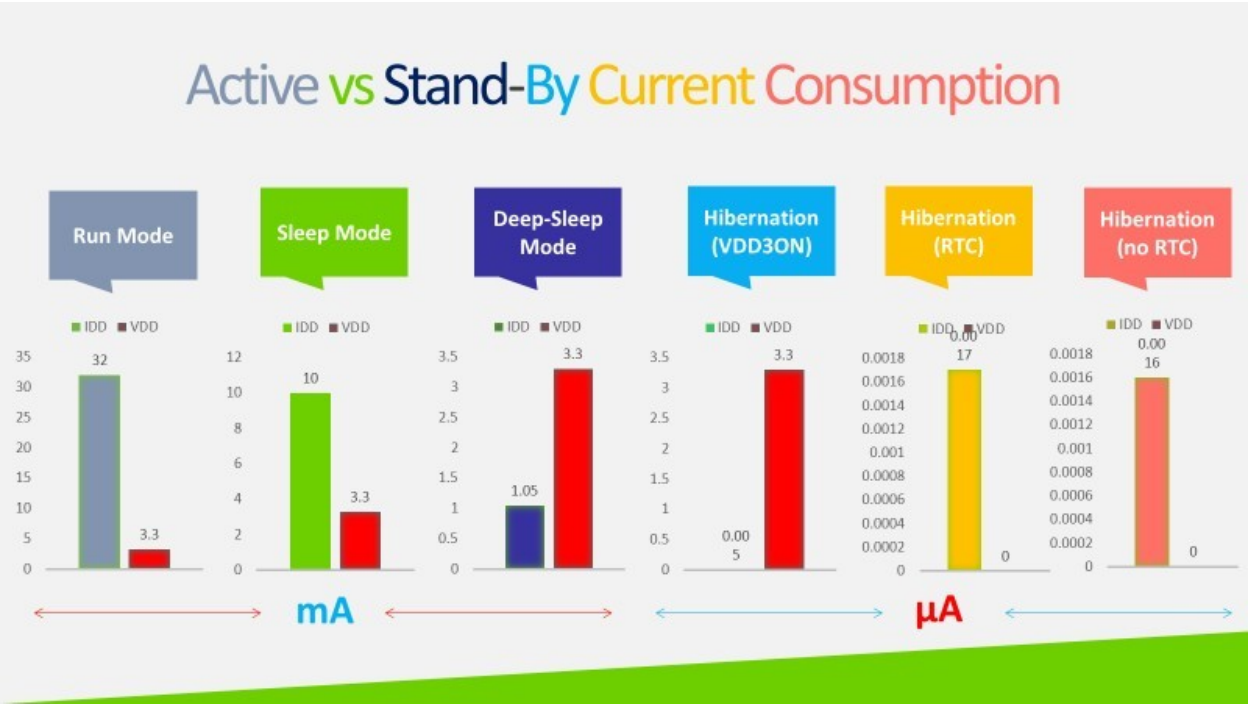
```
int main(void)
{ SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL
_OSC_MAIN); // System Clock set to 40 MHz
/*********************************************************************
******
   * Use the green LED (2=red=pin1, 4=blue=pin2 and 8=green=pin3) as an
   * indicator that the device is in hibernation (off for hibernate and on for wake).
*
*********************************************************************
*****/
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0x08);
SysCtlPeripheralEnable(SYSCTL_PERIPH_HIBERNATE); //enable the hibernation module
HibernateEnableExpClk(SysCtlClockGet()); //defines the clock supplied to the hibernation
module
HibernateGPIORetentionEnable(); //enables the GPIO pin state to be maintained during
hibernation
//and remain active even when waking from hibernation.
SysCtlDelay(64000000); //delay 4 seconds for you to observe the LED
HibernateWakeSet(HIBERNATE_WAKE_PIN); //wake condition to the wake pin is set
GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3, 0x00);//turn the green LED off before the
device
goes to sleep
HibernateRequest(); //HibernateRequest()function requests the Hibernation module
//to disable the external regulator, removing power from the
//processor and all peripherals.
while(1)
{
}
}
```

Power Modes

There are six power modes in which TM4C123GH6PM operates as shown in the below table. They are Run, Sleep, Deep Sleep, Hibernate with VDD3ON, Hibernate with RTC, and Hibernate without RTC. To understand all these modes and compare them, it is necessary to analyze them under a condition. Let us consider that the device is operating at 40 MHz system clock with PLL.

Mode → / Parameter ↓	Run Mode	Sleep Mode	Deep Sleep Mode	Hibernation (VDD3ON)	Hibernation (RTC)	Hibernation (no RTC)
I_{DD}	32 mA	10 mA	1.05 mA	5 µA	1.7 µA	1.6 µA
V_{DD}	3.3 V	3.3 V	3.3 V	3.3 V	0 V	0 V
V_{BAT}	N.A.	N.A.	N.A.	3 V	3 V	3 V
System Clock	40 MHz with PLL	40 MHz with PLL	30 kHz	Off	Off	Off
Core	Powered On	Powered On	Powered On	Off	Off	Off
	Clocked	Not Clocked	Not Clocked	Not Clocked	Not Clocked	Not Clocked
Peripherals	All On	All Off	All Off	All Off	All Off	All Off
Code	while{1}	N.A.	N.A.	N.A.	N.A.	N.A.

Table : Power Modes of Tiva



Introduction to Interrupts, Interrupt vector table, interrupt programming

Introduction to Interrupts and Polling

A microprocessor executes instructions sequentially. Alongside, it is also connected to several devices. Dataflow between these devices and the microprocessor has to be managed effectively. There are two ways it is done in a microprocessor: either by using interrupts or by using polling.

Polling

Polling is a simple method of I/O access. In this method, the microcontroller continuously probes whether the device requires attention, i.e. if there is data to be exchanged. A polling function or subroutine is called repeatedly while a program is being executed. When the status of the device being polled responds to the interrogation, a data exchange is initiated. The polling subroutine consumes processing time from the presently executing task. This is a very inefficient way because I/O devices do not always crave for attention from the microprocessor. But the microprocessor wastes valuable processing time in unnecessarily polling of the devices.

Interrupts

However, in interrupt method, whenever a device requires the attention from the microprocessors, it pings the microprocessor. This ping is called interrupt signal or sometimes interrupt request (IRQ). Every IRQ is associated with a subroutine that needs to be executed within the microprocessor. This subroutine is called interrupt service routine (ISR) or sometimes interrupt handler. The microprocessor halts current program execution and attends to the IRQ by executing the ISR. Once execution of ISR completes, the microprocessor resumes the halted task.

The current state of the microprocessor must be saved before it attends the IRQ in order to be able to continue from where it was before the interrupt. To achieve this, the contents of all of its internal registers, both general purpose and special registers, are required to be saved to a memory section called the stack. On completion of the interrupt call, these register contents will be reinstated from the stack. This allows the microprocessor to resume its originally halted task.

There are two types of interrupts namely software driven interrupts (SWI) and hardware driven interrupts (HWI). SWIs are generated from within a currently executing program. They are triggered by the interrupt opcode. A SWI will call a subroutine that allows a program to access certain lower level service. HWIs are signals from a device to the microprocessor. The device sets an interrupt line in the control bus high.

Microprocessors have two types of hardware interrupts namely, non-maskable interrupt (NMI) and interrupt request (INTR). An NMI has a very high priority and they demand immediate execution. There is no option to ignore an NMI. NMI is exclusively used for events that are regarded as having a higher priority or tragic consequences for the system operation. For example, NMI can be initiated due to an interruption of power supply, a memory fault or pressing of the reset button. An INTR may be generated by a number of different devices all of which are connected to the single INTR control line. An INTR may or may not be attended by the microprocessor. If the microprocessor is attending an interrupt, then no further interrupts, other than an NMI, will be

entertained until the current interrupt has been completed. A control signal is used by the microprocessor to acknowledge an INTR. This control signal is called ACK or sometimes INTA.

Interrupt vector table

It is discussed in the previous section that when an interrupt occurs, the microprocessor runs an associated ISR. IRQ is an input signal to the microprocessor. When a microprocessor receives an IRQ, it pushes the PC register onto the stack and load address of the ISR onto the PC register. This makes the microprocessor execute the ISR. These associated ISRs, corresponding to every interrupt, become a part of the executable program. This executable is loaded in the memory of the device. Under such circumstances, it becomes easier to manage the ISRs if there is a lookup table where address locations of all ISRs are listed. This lookup table is called Interrupt vector table. Table shows an interrupt vector table for ARM cortex-M microcontroller.

In ARM microcontroller, there exist 256 interrupts. Out of these, some are hardware or peripheral generated IRQs and some are software generated IRQs. However, first 15 interrupts, INT0 to INT15 are called the predefined interrupts. In ARM Cortex-M microcontrollers, Interrupt vector table is an on-chip module, called as Nested Vector Interrupt Controller (NVIC).

NVIC is an on-chip interrupt controller for ARM Cortex-M series microcontrollers. No other ARM series has this on-chip NVIC. This means that the interrupt handling is primarily different in ARM Cortex-M microcontrollers compared to other ARM microcontrollers.

VECTOR NO.	PRIORITY	EXCEPTION TYPE	VECTOR ADDRESS
0	-	SP initial Value	0x0000.0000
1	-3	RESET	0x0000.0004
2	-2	NMI	0x0000.0008
3	-1	Hard Fault	0x0000.000C
4	Programmable	Memory Management Fault	0x0000.0010
5	Programmable	BUS Fault	0x0000.0014
6	Programmable	Usage Fault (undefined instructions, divide by zero, unaligned memory access, etc.)	0x0000.0018
7 - 10	-	Reserved	0x0000.001C to 0x0000.0028
11	Programmable	SVCall	0x0000.002C
12	Programmable	Debug Monitor	0x0000.0030
13	-	Reserved	0x0000.0034
14	Programmable	PendSV	0x0000.0038
15	Programmable	SysTick	0x0000.003C
16 -255	Programmable	User Interrupts (interrupts generated from peripherals and software)	0x0000.0040 to 0x0000.03FC

***Predefined Interrupts* (*INT0-INT15*)**

RESET: All ARM devices have a RESET pin which is invoked on device power-up or in case of warm reset. This exception is a special exception and has the highest priority. On the assertion of Reset signal, the execution stops immediately. When the Reset signal stops, execution starts from the address provided by the Reset entry in the vector table i.e. 0x0000.0004. Hereby, to run a program on Reset, it is necessary to place the program in 0x0000.0004 memory address.

NMI: In the ARM microcontroller, some pins are associated with hardware interrupts. They are often called IRQs (interrupt request) and NMI (non-maskable interrupt). IRQ can be controlled by software masking and unmasking. Unlike IRQ, NMI cannot be masked by software. This is why I is named as nonmaskable interrupt. As shown in Table 2.9, "INT 02" in ARM Cortex-M is used only for NMI. On activation of NMI, the microcontroller load memory location 0x0000008 to program counter.

Hard Fault: All the classes of fault corresponding to a fault handler cannot be activated. This may be a result of the fault handler being disabled or masked.

Memory Management Fault: It is caused by a memory protection unit violation. The violation can be caused by attempting to write into a read only memory. An instruction fetch is invalid when it is fetched from non-executable region of memory. In an ARM microcontroller with an on-chip MMU, the page fault can also be mapped into the memory management fault.

Bus Fault: A bus fault is an exception that arises due to a memory-related fault for an instruction or data memory transaction, such as a pre-fetch fault or a memory access fault. This fault can be enabled or disabled.

Usage Fault: Exception that occurs due to a fault associated with instruction execution. This includes undefined instruction, illegal unaligned access, invalid state on instruction execution, or an error on exception return may termed as usage fault. An unaligned address of a word or half- word memory access or division by zero can cause a usage fault.

SVCall: A supervisor call (SVC) is an exception that is activated by the SVC instruction. In an operating system, applications can use SVC instructions to contact OS kernel functions and device drivers. This is a software interrupt since it was raised from software, and not from a Hardware or peripheral exception.

PendSV: PendSV is pendable service call and interrupt-driven request for system-level service. PendSV is used for framework switching when no other exception is active. The Interrupt Control and State (INTCTRL) register is used to trigger PendSV. The PendSV is an interrupt and can wait until NVIC has time to service it when other urgent higher priority interrupts are being taken care.

SysTick: A SysTick exception is generated by the system timer when it reaches zero and is enabled to generate an interrupt. The software can also produce a SysTick exception using the Interrupt Control and State (INTCTRL) register.

User Interrupts: This interrupt is an exception signaled either by a peripheral or by a software request and fed through the NVIC based on their priority. All interrupts are asynchronous to

instruction execution. In the system, peripherals use interrupts to communicate with the processor. An ISR can be also propelled as a result of an event at the peripheral devices. This may include timer timeout or completion of analog-to-digital converter (ADC) conversion. Each peripheral device has a group of special function registers that must be used to access the device for configuration. For a given peripheral interrupt to take effect, the interrupt for that peripheral must be enabled.

Interrupt Programming:

In this experiment, the timer peripheral of the TM4C123GH6PM processor is configured in periodic mode. In periodic mode, the timer load register is loaded with a preset value and the timer count is decremented for each clock cycle. When the timer count reaches zero, an interrupt is generated. On each interrupt, the processor reads the current status of the LED connected to a GPIO port and toggles it. Consequently, the GPIO output is programmed by the timer interrupt.

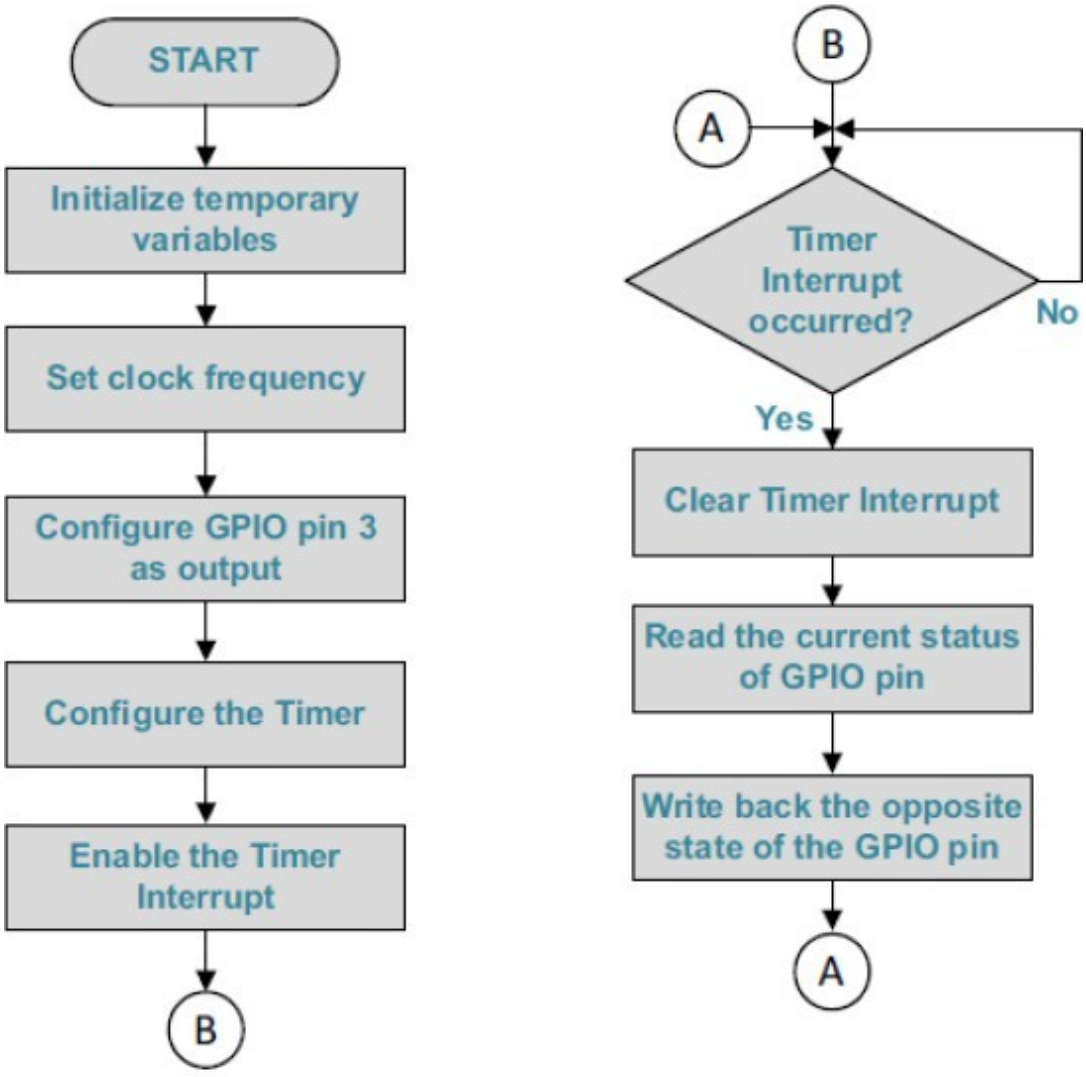


Figure 2-3 Flowchart for Interrupt Programming with GPIO

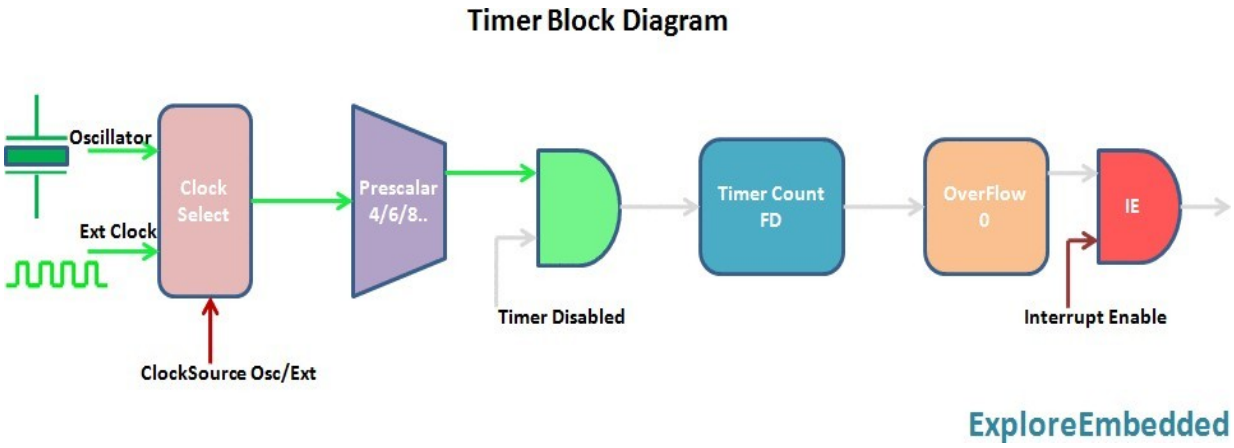
PROGRAM:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/tm4c123gh6pm.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "driverlib/gpio.h"
#include "driverlib/timer.h"
int main(void)
{
uint32_t ui32Period; SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|
SYSCTL_XTAL_16MHZ| SYSCTL_OSC_MAIN);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC); ui32Period
= (SysCtlClockGet() / 10) / 2; TimerLoadSet(TIMER0_BASE, TIMER_A,
ui32Period -1); IntEnable(INT_TIMER0A);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
IntMasterEnable(); TimerEnable(TIMER0_BASE,
TIMER_A); while(1)
{
}
}
void Timer0IntHandler(void)
{
// Clear the timer interrupt
TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
// Read the current state of the GPIO pin and+
// write back the opposite state
if(GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_2))
{
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0);
}
else
{
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 4);
}
}
```

Basic Timer:

Timers are basic constituents of most microcontrollers. Today, just about every microcontroller comes with one or more built-in timers. These are extremely useful to the embedded programmer - perhaps second in usefulness only to GPIO. The timer can be described as the counter hardware and can usually be constructed to count either regular or irregular clock pulses. Depending on the above usage, it can be a timer or a counter respectively.

Sometimes, timers may also be termed as "hardware timers" to distinguish them from software timers. Software timers can be described as a stream of bits of software that achieve some timing function.



A standard timer will comprise a pre-scaler, an N-bit timer/counter register, one or more N-bit capture and compare registers. Usually N is 8, 16 or 32 bits. Along with these, there will also be registers for control and status units responsible to configure and monitor the timer.

To count the incoming pulses, an up-counter is deployed as fundamental hardware. A counter can be converted to a timer by fixing incoming pulses and setting a known frequency. Also note that the size in bits of a timer should not be related directly to the size in bits of the CPU architecture. An 8-bit microcontroller can have 16-bit timers (in fact mostly do), and a 32-bit microcontroller can have 16-bit timers (and some do).

Pre-scaler

The pre-scaler takes the basic timer clock frequency as an input and divides it by some value depending upon the circuit requirements before feeding it to the timer, to configure the pre-scaler register(s). This configuration might be limited to a few fixed values (powers of 2), or integers from 1 to 2^m, where m is the number of pre-scaler bits.

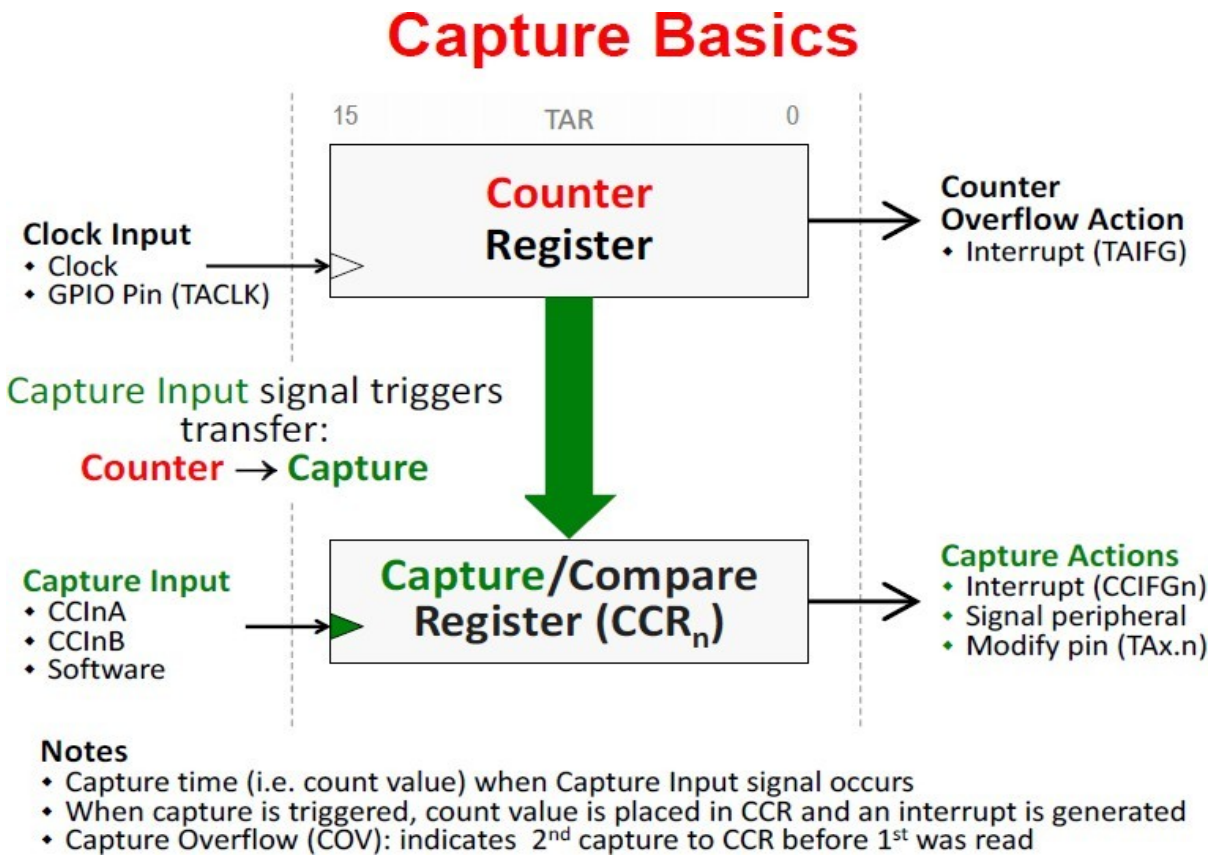
Pre-scaler is used to set the clock rate of the timer as per your desire. This provides a flexibility in resolution (high clock rate implies better resolution) and range (high clock rate causes quicker overflow

Timer Register

The timer register can be defined as hardware with an N-bit up-counter, which has accessibility of read and write command rights for the current count value, and to stop or reset the counter. As discussed, the timer is driven by the pre-scaler output. The regular pulses which drive the timer, irrespective of their source are often called “ticks”. We may understand now that it is not necessary for a timer to time in seconds or milliseconds, they do time in ticks. This enables us the elasticity to control the rate of these ticks, depending upon the hardware and software configuration. We may construct our design to some human-friendly value such as e.g. 1 millisecond or 1 microsecond, or any other design specified

Capture Registers

A capture registers are those hardware which can be routinely loaded with the current counter value upon the occurrence of some event, usually a change on an input pin. Therefore the capture register is used to capture a “snapshot” of the timer at the instant when the event occurs. A capture event can also be constructed to produce an interrupt, and the Interrupt Service Routines (ISR) can save or else use the just-captured timer snapshot.



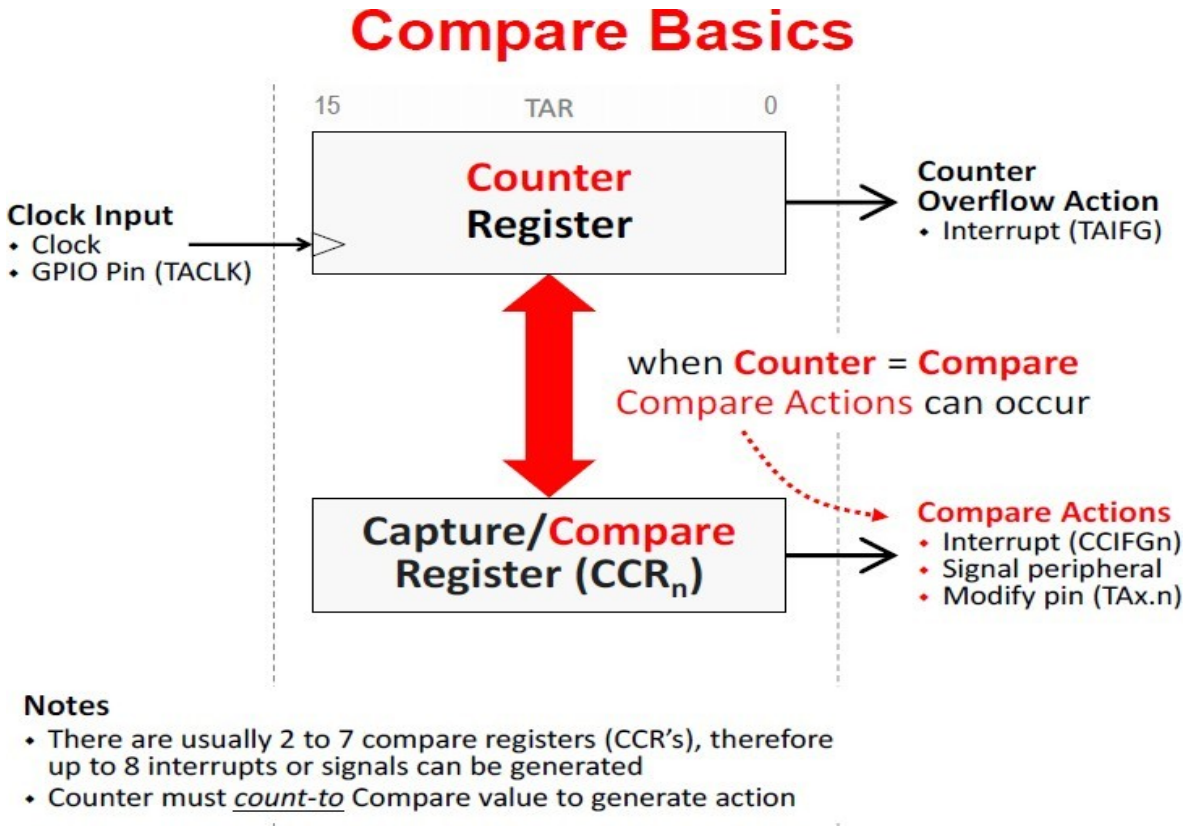
There is no latency problem in snapshot value as the capture occurs in hardware, which would be if the capture was done in software. Capture registers can be used to time intervals between pulses or input signals, to determine the high and low times of input signals.

Compare/Match Registers

Compare or match registers hold a value against which the current timer value is routinely compared and shoots to trigger an event when the value in two registers matches.

If the timer/counter is configured as a timer, we can generate events at known and precise times. Events can be like output pin changes and/or interrupts and/or timer resets.

If the timer/counter is configured as a counter, the compare registers can generate events based on preset counts being achieved.



For instance, the compare registers can be used to generate a timer "tick", a fixed timer interrupt used for system software timing. For example, if a 2ms tick is desired, and the timer is configured with a 0.5us clock, setting a compare register to 4000 will cause a compare event after 2ms. If we set the compare event to generate an interrupt as well as to reset the timer to 0, the result will be an endless stream of 2ms interrupts.

Real Time Clock (RTC)

RTC is a mainframe clock that keeps track of the current time. RTCs are present in approximately every electronic device which needs to maintain accurate time. The term RTC came into picture to avoid confusion with regular hardware clocks which are merely signals that administer digital electronics, and do not count time in human units.

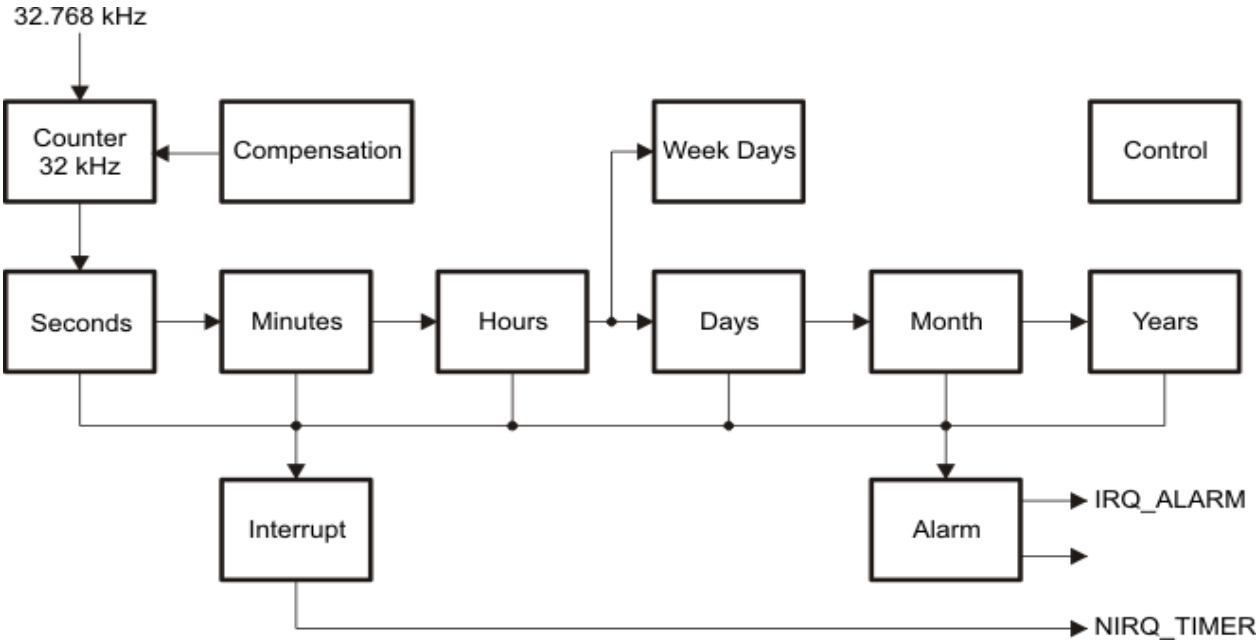
Benefits of using RTC: Low

power consumption

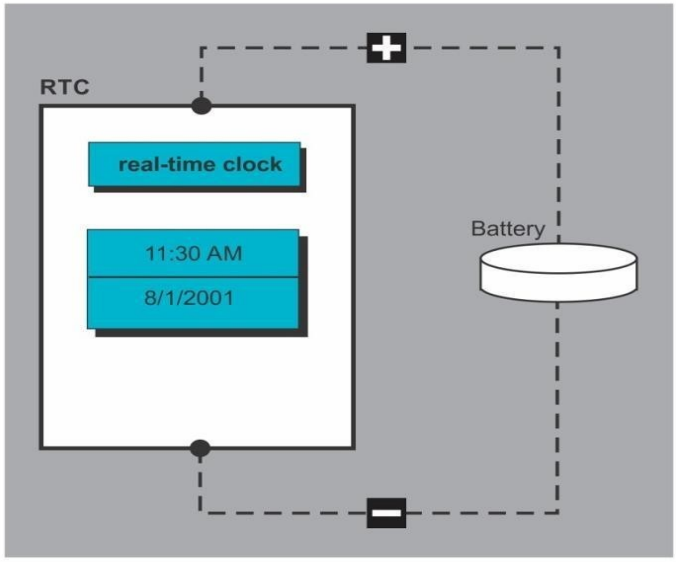
Liberates the main system for time-critical tasks

Increases accuracy if compared to other methods

A GPS receiver can cut down its startup time by comparing the current time as per its RTC, with the moment of last valid signal. If it has been less than a few hours, then the previous ephemeris is still usable.



With the option of alternative power source with RTCs, they can continue to keep time while the primary power source being unavailable. This alternate source may be a lithium battery or a supercapacitor.



Pulse Width Modulation

Pulse width modulation (PWM) is a simple but powerful technique of using a rectangular digital waveform to control an analog variable or simply controlling analog circuits with a microprocessor's digital outputs. PWM is employed in a wide variety of applications, from measurement & communications to power control and conversion.

PWM using TIVA TM4C123HG6PM

TM4C123GH6PM PWM module provides a great deal of flexibility and can generate simple PWM signals, such as those required by a simple charge pump as well as paired PWM signals with deadband delays, such as those required by a half-H bridge driver. Three generator blocks can also generate the full six channels of gate controls required by a 3-phase inverter bridge.

Each PWM generator block has the following features:

One fault-condition handling inputs to quickly provide low-latency shutdown and prevent damage to the motor being controlled, for a total of two inputs
One 16-bit counter

- Runs in Down or Up/Down mode
- Output frequency controlled by a 16-bit load value
- Load value updates can be synchronized
- Produces output signals at zero and load value

Two PWM comparators

- Comparator value updates can be synchronized
- Produces output signals on match

PWM signal generator

- Output PWM signal is constructed based on actions taken as a result of the counter and PWM comparator output signals
- Produces two independent PWM signals

Dead-band generator

- Produces two PWM signals with programmable dead-band delays suitable for driving a half-H bridge.
- Can be bypassed, leaving input PWM signals unmodified.

Can initiate an ADC sample sequence

The control block determines the polarity of the PWM signals and which signals are passed through to the pins. The output of the PWM generation blocks are managed by the output control block before being passed to the device pins.

Block Diagram

TM4C123GH6PM controller contains two PWM modules, each with four generator blocks that generate eight independent PWM signals or four paired PWM signals with dead-band delays inserted. TM4C123GH6PM controller contains two PWM modules, each with four generator blocks that generate eight independent PWM signals or four paired PWM signals with dead-band delays inserted.

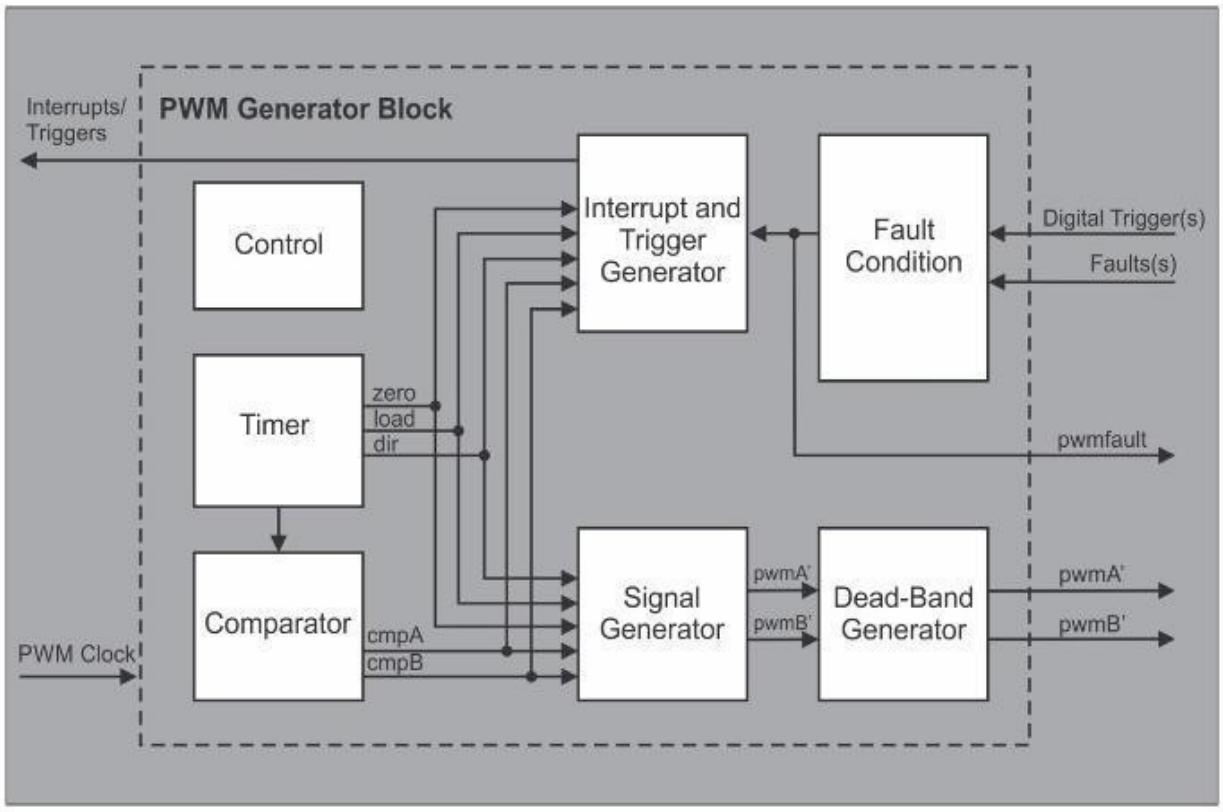


Fig PWM Module Block Diagram

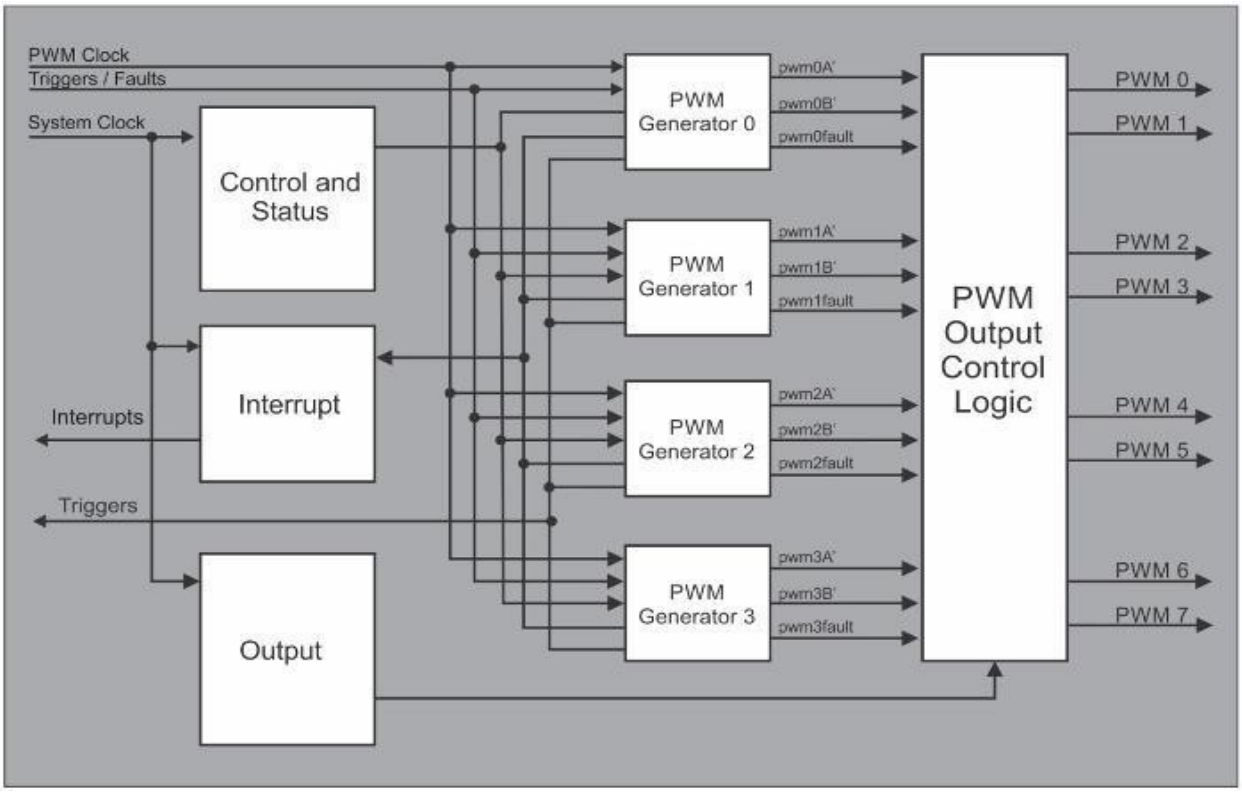


Fig PWM Generator Block Diagram

Functional Description

Clock Configuration

The PWM has two clock source options:

The System Clock

A pre divided System Clock

The clock source is selected by programming the USPWMDIV bit in the Run-Mode Clock Configuration (RCC) register. The PWMDIV bit field specifies the divisor of the system clock that is used to create the PWM Clock.

PWM Timer

The timer in each PWM generator runs in one of two modes: Count-Down mode or Count-Up/Down mode. In Count-Down mode, the timer counts from the load value to zero, goes back to the load value, and continues counting down. In Count-Up/Down mode, the timer counts from zero up to the load value, back down to zero, back up to the load value, and so on. Generally, Count-Down mode is used for generating left- or right-aligned PWM signals, while the Count- Up/Down mode is used for generating center-aligned PWM signals.

The timers output three signals that are used in the PWM generation process: the direction signal (this is always Low in Count-Down mode, but alternates between low and high in Count-Up/Down mode), a single-clock-cycle-width High pulse when the counter is zero, and a single-clock-cycle-width High pulse when the counter is equal to the load value. Note that in Count-Down mode, the zero pulse is immediately followed by the load pulse. In the figures in this chapter, these signals are labelled "dir," "zero," and "load."

PWM Comparators

Each PWM generator has two comparators that monitor the value of the counter, when either comparator matches the counter, they output a single-clock-cycle-width High pulse, labeled "cmpA" and "cmpB" in the figures in this chapter. When in Count-Up/Down mode, these comparators match both when counting up and when counting down, and thus are qualified by the counter direction signal. These qualified pulses are used in the PWM generation process. If either comparator match value is greater than the counter load value, then that comparator never outputs a High pulse.

PWM Signal Generator

Each PWM generator takes the load, zero, cmpA, and cmpB pulses (qualified by the dir signal) and generates two internal PWM signals, pwmA and pwmB. In Count-Down mode, there are four events that can affect these signals: zero, load, match A down, and match B down. In Count-Up/Down mode, there are six events that can affect these signals: zero, load, match A down, match A up, match B down, and match B up. The match A or match B events are ignored when they coincide with the zero or load events. If the match A and match B events coincide, the first signal,

pwmA, is generated based only on the match A event, and the second signal, pwmB, is generated based only on the match B event.

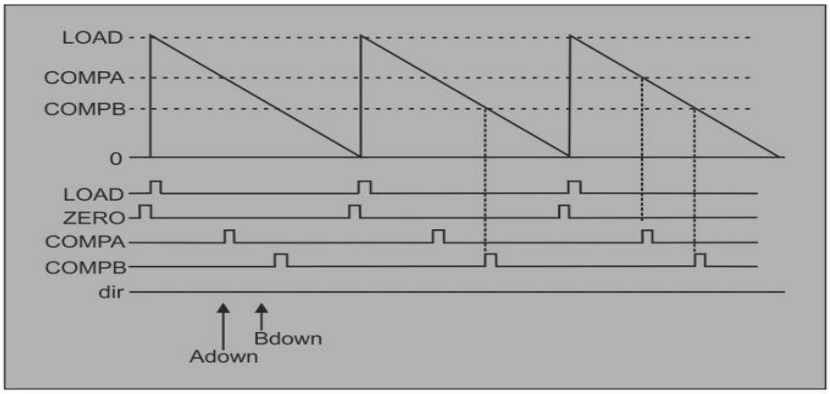
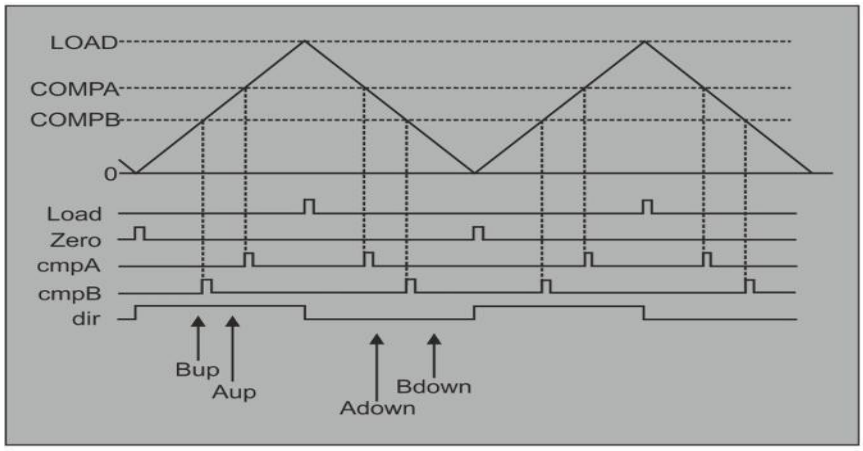


Figure (a): PWM Count- Down Mode



Figure(b): PWM Count- Up/ Down Mode

Dead-Band Generator

The pwmA and pwmB signals produced by each PWM generator are passed to the dead-band generator. If the dead-band generator is disabled, the PWM signals simply pass through to the pwmA' and pwmB' signals unmodified. If the dead-band generator is enabled, the pwmB signal is lost and two PWM signals are generated based on the pwmA signal. The first output PWM signal, pwmA' is the pwmA signal with the rising edge delayed by a programmable amount. The second output PWM signal, pwmB', is the inversion of the pwmA signal with a programmable delay added between the falling edge of the pwmA signal and the rising edge of the pwmB' signal.

The resulting signals are a pair of active high signals where one is always high, except for a programmable amount of time at transitions where both are low. These signals are therefore suitable for driving a half-H bridge, with the dead-band delays preventing shoot-through current from damaging the power electronics.

Quadrature Encoder Interface (QEI)

A quadrature encoder, also known as a 2-channel incremental encoder, converts linear displacement into a pulse signal. By monitoring both the number of pulses and the relative phase of the two signals, you can track the position, direction of rotation, and speed. In addition, a third channel, or index signal, can be used to reset the position counter.

A classic quadrature encoder has a slotted wheel like structure, to which a shaft of the motor is attached and a detector module that captures the movement of slots in the wheel.

Interfacing QEI using Tiva TM4C123GH6PM

The TM4C123GH6PM microcontroller includes two quadrature encoder interface (QEI) modules. Each QEI module interprets the code produced by a quadrature encoder wheel to integrate position over time and determine direction of rotation. In addition, it can capture a running estimate of the velocity of the encoder wheel.

The TM4C123GH6PM microcontroller includes two QEI modules providing control of two motors at the same time with the following features:

Position integrator that tracks the encoder position

Programmable noise filter on the inputs

Velocity capture using built-in timer

The input frequency of the QEI inputs may be as high as 1/4 of the processor frequency (for

example, 12.5 MHz for a 50-MHz system)

Interrupt generation on:

- Index pulse *Quadrature Encoder Interface*www.ti.com
- Velocity-timer expiration
- Direction change
- Quadrature error detection

Functional Description

The QEI module interprets the two-bit gray code produced by a quadrature encoder wheel to integrate position over time and determine direction of rotation. In addition, it can capture a running estimate of the velocity of the encoder wheel. The position integrator and velocity capture can be independently enabled, though the position integrator must be enabled before the velocity capture can be enabled. The two phase signals, **PhAn** and **PhBn**, can be swapped before being interpreted by the QEI module.

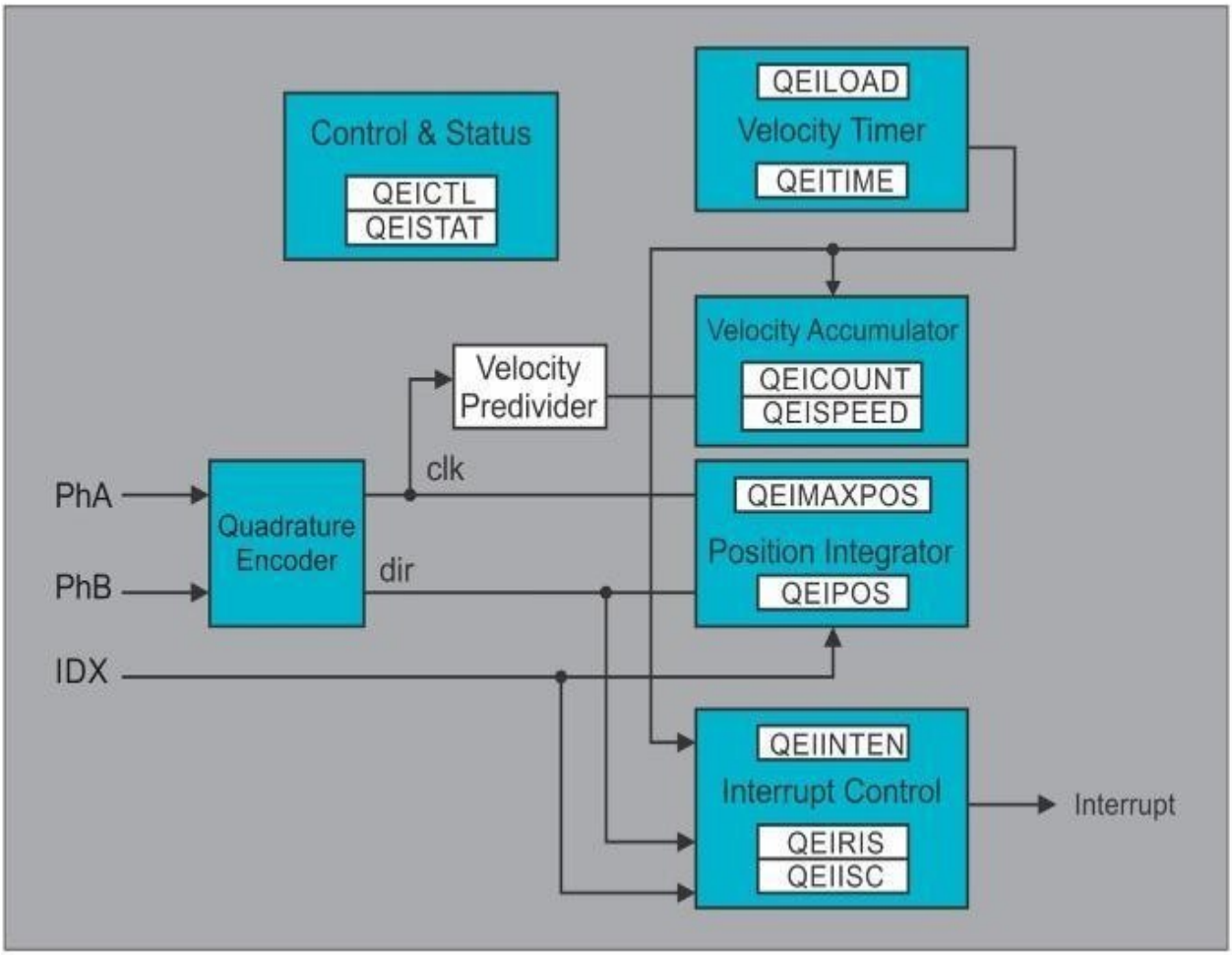


Fig : QEI Block Diagram

The QEI module input signals have a digital noise filter on them that can be enabled to prevent spurious operation. The noise filter requires that the inputs be stable for a specified number of consecutive clock cycles before updating the edge detector. The filter is enabled by the **FILTEN** bit in the **QEI Control (QEICTL)** register. The frequency of the input update is programmable using the **FILTCNT** bit field in the **QEICTL** register.

The QEI module supports two modes of signal operation:

Quadrature phase mode, the encoder produces two clocks that are 90 degrees out of phase, the edge relationship is used to determine the direction of rotation.

Clock/direction mode, the encoder produces a clock signal to indicate steps and a direction signal to indicate the direction of rotation. This mode is determined by the **SIGMODE** bit of the **QEICTL** register.

When the QEI module is set to use the quadrature phase mode (**SIGMODE** bit is clear), the capture mode for the position integrator can be set to update the position counter on every edge of the **PhA** signal or to update on every edge of both PhA and PhB. Updating the position counter on every

PhA and **PhB** edge provides more positional resolution at the cost of less range in the positional counter. When edges on **PhA** lead edges on **PhB**, the position counter is incremented. When edges on **PhB** lead edges on **PhA**, the position counter is decremented. When a rising and falling edge pair is seen on one of the phases without any edges on the other, the direction of rotation has changed.

The positional counter is automatically reset on one of two conditions:

Sensing the index pulse **or**

Reaching the maximum position value.

The reset mode is determined by the **RESMODE** bit of the **QEICTL** register.

Unit 5

Embedded communications protocols and Internet of things

Synchronous/Asynchronous interfaces (like UART, SPI, I2C, USB), serial communication basics, baud rate concepts, Interfacing digital and analog external device, Implementing and programming UART, SPI and I2C, SPI interface using TM4C.

Case Study: Tiva based embedded system application using the interface protocols for communication with external devices "Sensor Hub BoosterPack".

Embedded Networking fundamentals, IoT overview and architecture, Overview of wireless sensor networks and design examples. Adding Wi-Fi capability to the Microcontroller, Embedded Wi-Fi, User APIs for Wireless and Networking applications Building IoT applications using CC3100 user API.

Case Study: Tiva based Embedded Networking Application: "Smart Plug with Remote disconnect and Wi-Fi connectivity.

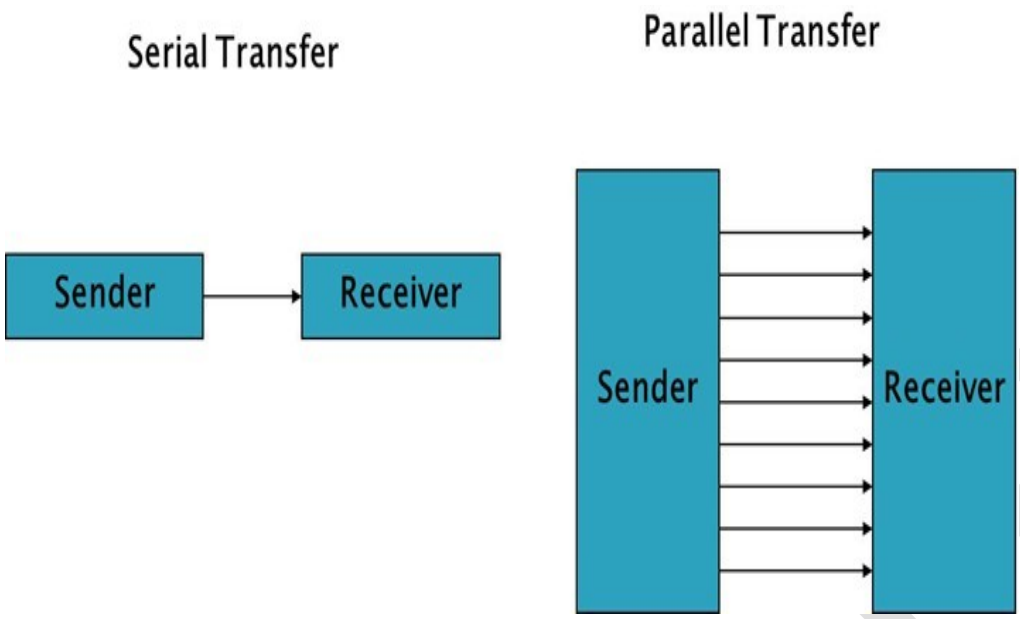
Serial Communication basics:

Communication between electronic devices is like communication between humans. Both sides need to speak the same language. In electronics, these languages are called *communication protocols*. Luckily for us, there are only a few communication protocols (SPI,I2C,UART,USB) we need to know when building most electronics projects.

SPI, I2C, and UART are quite a bit slower than protocols like USB, Ethernet, Bluetooth, and Wi-Fi, but they're a lot simpler and use less hardware and system resources. SPI, I2C, and UART are ideal for communication between microcontrollers and between microcontrollers and sensors where large amounts of high speed data don't need to be transferred.

DATA COMMUNICATION TYPES: (1) PARALLEL

(2) SERIAL: (I) ASYNCHRONOUS (II) SYNCHRONOUS



Parallel Communication:

- In parallel communication, all the bits of data are transmitted simultaneously on separate communication lines.
- Used for shorter distance.
- In order to transmit n bit, n wires or lines are used.
- More costly.
- Faster than serial transmission.
- Data can be transmitted in less time.

Example: printers and hard disk

Serial Communication:

- In serial communication the data bits are transmitted serially one by one i.e. bit by bit on single communication line.
- It requires only one communication line rather than n lines to transmit data from sender to receiver.
- Thus all the bits of data are transmitted on single lines in serial fashion.
- Less costly.
- Long distance transmission.

Example: Telephone.

Serial communication uses two methods:

- Asynchronous.
- Synchronous.

Asynchronous:

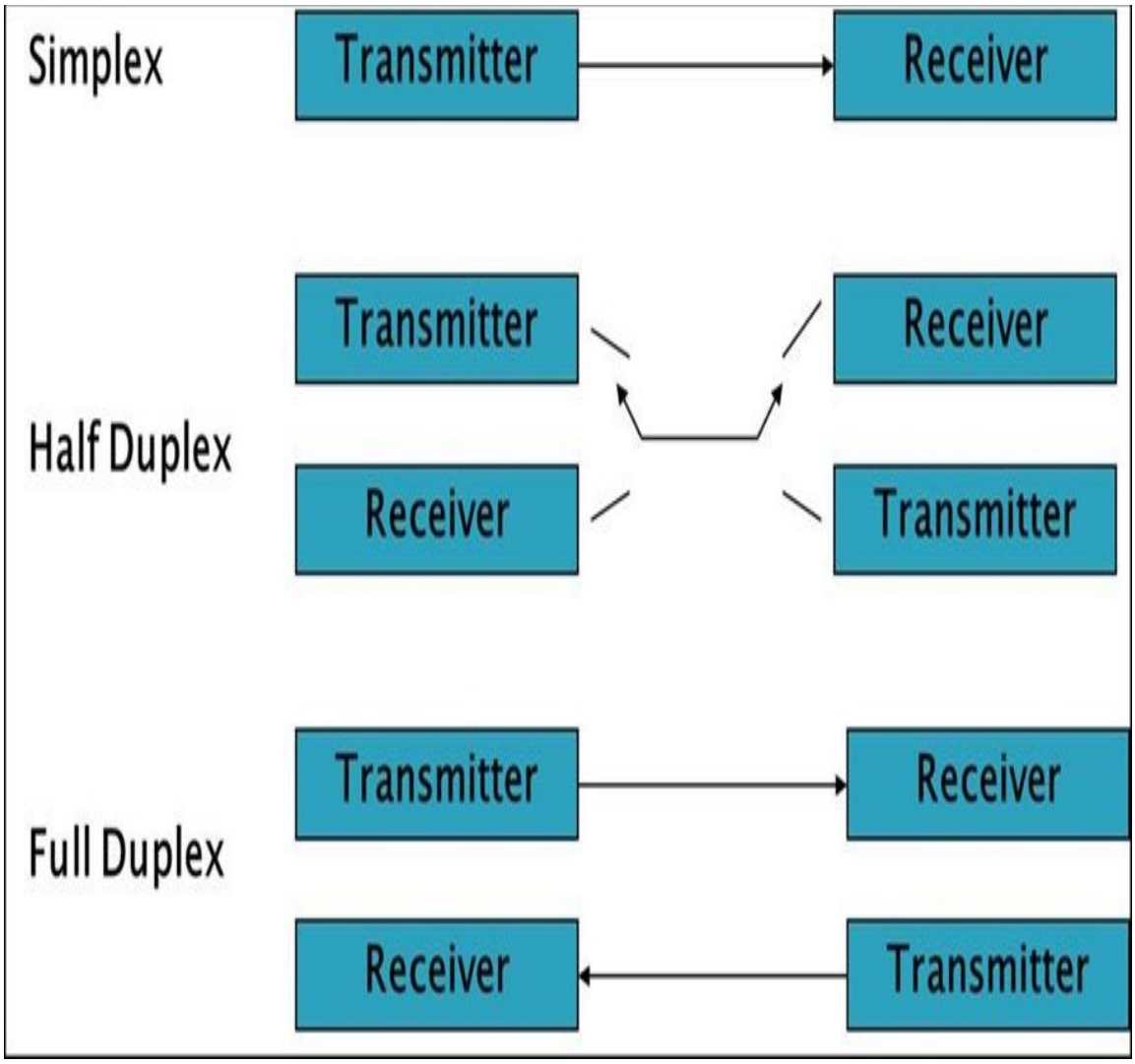
- Ttransfers single byte at a time
- No need of clock signal

Example: UART (universal asynchronous receiver transmitter)

Synchronous:

- Transfers a block of data (characters) at a time.
- Requires clock signal

Example: SPI (serial peripheral interface),
I2C (inter integrated circuit).


Simplex
Transmitter
Receiver
Half Duplex
Transmitter
Receiver
Receiver
Transmitter
Full Duplex
Transmitter
Receiver
Receiver
Transmitter

Data Transmission: In data transmission if the data can be transmitted and received, it is a duplex transmission.

Simplex: Data is transmitted in only one direction i.e. from TX to RX only one TX and one RX only

Half duplex: Data is transmitted in two directions but only one way at a time i.e. two TX's, two RX's and one line

Full duplex: Data is transmitted both ways at the same time i.e. two TX's, two RX's and two lines

A **Protocol** is a set of rules agreed by both the sender and receiver on

- How the data is packed
- How many bits constitute a character
- When the data begins and ends

Serial Protocol	**Synchronous /Asynchronous**	**Type**	**Duplex**	**Data transfer rate (kbps)**
UART	Asynchronous	peer-to-peer	Full-duplex	20
I2C	Synchronous	multi-master	Half-duplex	3400
SPI	Synchronous	multi-master	Full-duplex	>1,000
MICROWIRE	Synchronous	master/slave	Full-duplex	> 625
1-WIRE	Asynchronous	master/slave	Half-duplex	16

Baud Rate Concepts:

Data transfer rate in serial communication is measured in terms of bits per second (bps). This is also called as Baud Rate. Baud Rate and bps can be used inter changeably with respect to UART.

Ex: The total number of bits gets transferred during 10 pages of text, each with 100 × 25 characters with 8 bits per character and 1 start & stop bit is:

For each character a total number of bits are 10. The total number of bits is: 100 × 25 × 10 = 25,000 bits per page. For 10 pages of data it is required to transmit 2, 50,000 bits. Generally

baud rates of SCI are 1200, 2400, 4800, 9600, 19,200 etc. To transfer 2, 50,000 bits at a baud rate of 9600, we need: 250000/9600 = 26.04 seconds (27 seconds).

Synchronous/Asynchronous Interfaces (like UART, SPI, I2C, and USB):

Serial communication protocols can be categorized as Synchronous and Asynchronous protocols. In synchronous communication, data is transmission and receiving is a continuous stream at a constant rate. Synchronous communication requires the clock of transmitting device and receiving device synchronized. In most of the systems, like ADC, audio codes, potentiometers, transmission and reception of data occurs with same frequency. Examples of synchronous communication are: I2C, SPI etc. In the case of asynchronous communication, the transmission of data requires no clock signal and data transfer occurs intermittently rather than steady stream. Handshake signals between the transmitter and receiver are important in asynchronous communications. Examples of asynchronous communication are Universal Asynchronous Receiver Transmitter (UART), USB, CAN etc.

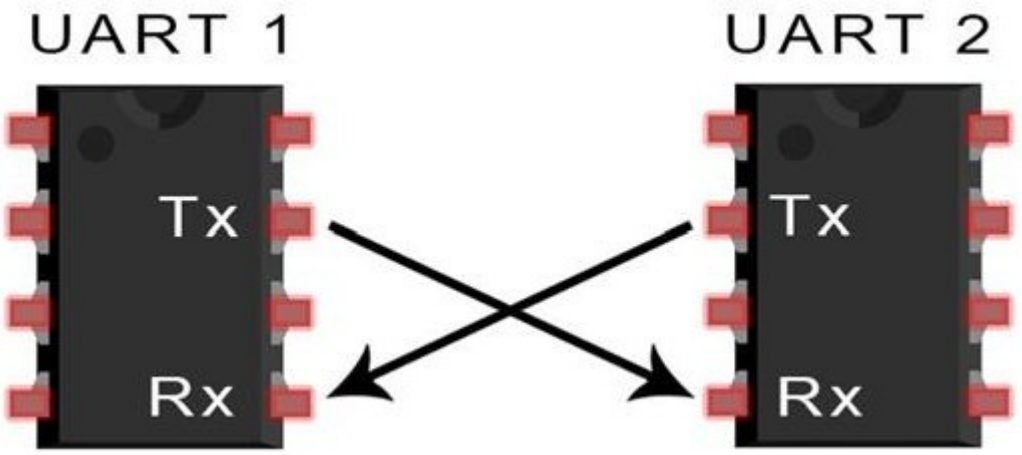
Synchronous and asynchronous communication protocols are well-defined standards and can be implemented in either hardware or software. In the early days of embedded systems, Software implementation of I^2C and SPI was common as well as a tedious work and used to take long programs. Gradually, most the microcontrollers started incorporating the standard communication protocols as hardware cores. This development in early 90‟s made job of the embedded software development easy for communication protocols.

Microcontroller of our interest TM4C123 supports UART, CAN, SPI, I^2C and USB protocols. The five (UART, CAN, SPI, I^2C and USB) above mentioned communication protocols are available in most of the modern day microcontrollers. Before studying the implementation and programming details of these protocols in TM4C123, it is required to understand basic standards, features and applications.

UART COMMUNICATION PROTOCOL

In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device like a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART:

UARTs transmit data asynchronously, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.



When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate*. Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off.

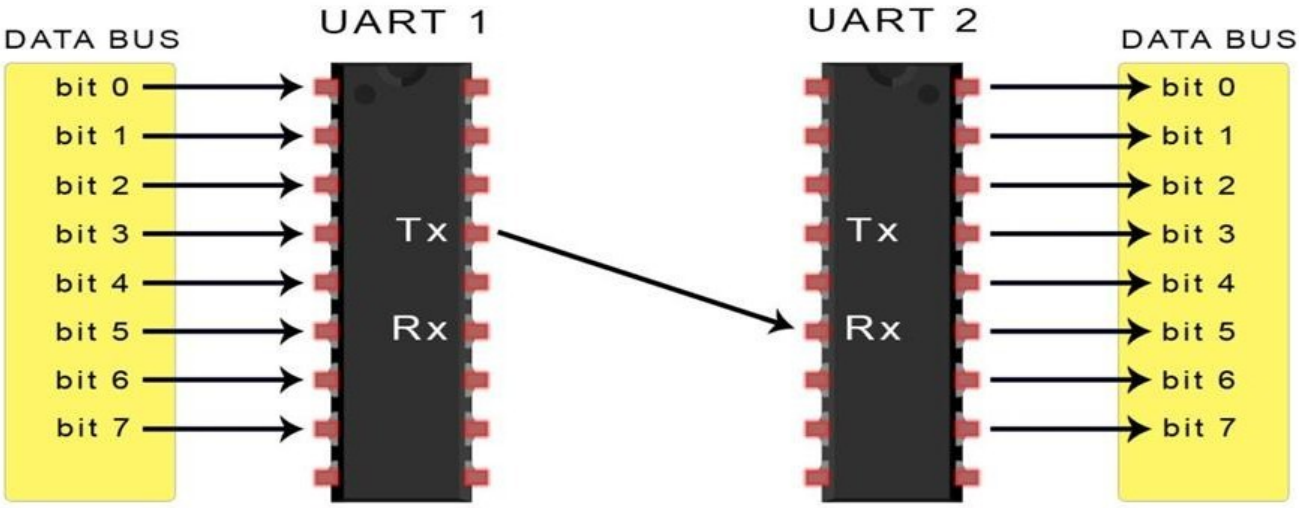
Both UARTs must be configured to transmit and receive the same data packet structure.

Wires Used	2
Maximum Speed	Any speed up to 115200 baud, usually 9600 baud
Synchronous or Asynchronous?	Asynchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	1

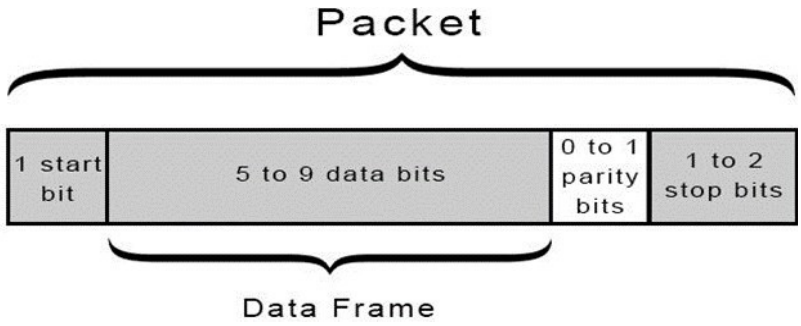
HOW UART WORKS

The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is output serially, bit by bit at the Tx pin.

The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end.



UART transmitted data is organized into *packets*. Each packet contains 1 start bit, 5 to 9 data bits (depending on the UART), an optional *parity* bit, and 1 or 2 stop bits:



START BIT

The UART data transmission line is normally held at a high not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.

DATA FRAME:

The data frame contains the actual data being transferred. It can be 5 bits to 9 bits long if a parity bit is used. If no parity bit is used, the data frame can be 8 bits long. In most cases, the data is sent with the least significant bit first.

PARITY

Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers. After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 bits in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bits in the data frame should total to an odd

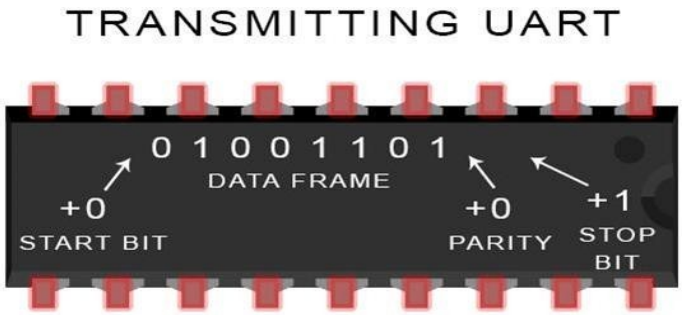
number. When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd; or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.

STOP BITS

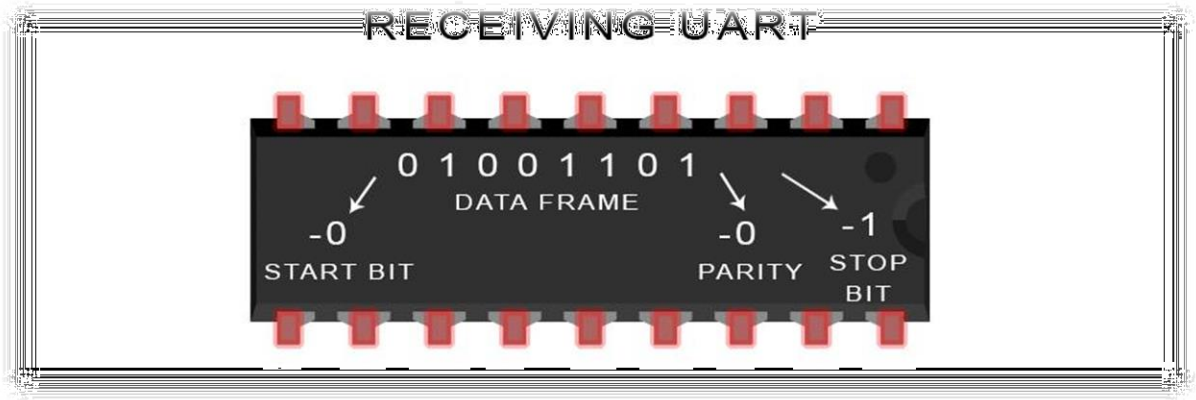
The Stop Bit, as the name suggests, marks the end of the data packet. It is usually two bits long but often only on bit is used. In order to end the transmission, the UART maintains the data line at high voltage (1).

STEPS OF UART TRANSMISSION

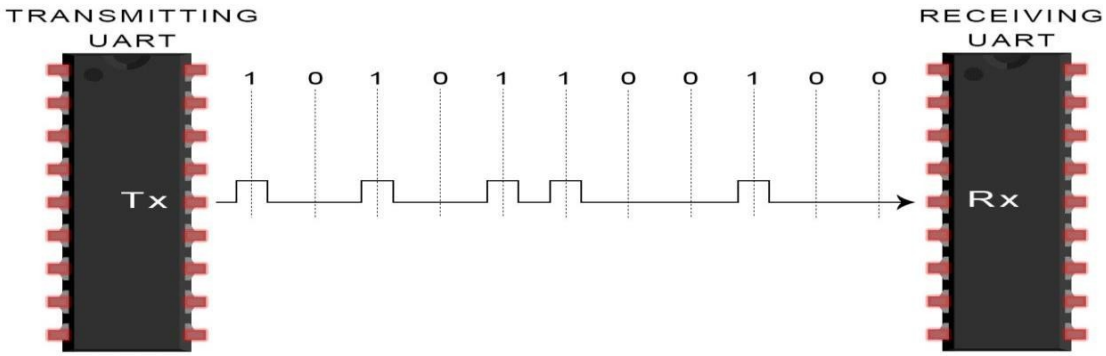
1. The transmitting UART receives data in parallel from the data bus:
2. The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame:



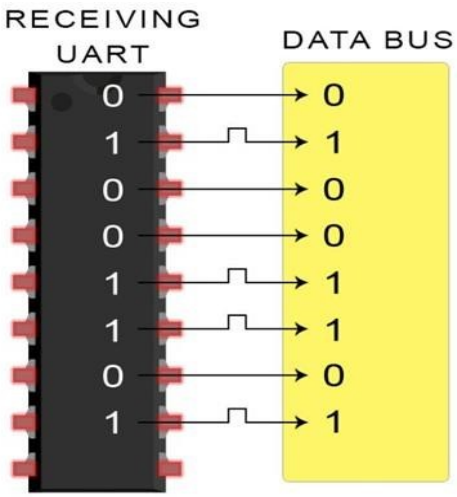
3. The entire packet is sent serially from the transmitting UART to the receiving UART. The receiving UART samples the data line at the pre-configured baud rate:



4. The receiving UART discards the start bit, parity bit, and stop bit from the data frame:



5. The receiving UART converts the serial data back into parallel and transfers it to the data bus on the receiving end:



ADVANTAGES AND DISADVANTAGES OF UARTS

No communication protocol is perfect, but UARTs are pretty good at what they do. Here are some pros and cons to help you decide whether or not they fit the needs of your project:

ADVANTAGES

Only uses two wires

No clock signal is necessary

Has a parity bit to allow for error checking

The structure of the data packet can be changed as long as both sides are set up for it

Well documented and widely used method

DISADVANTAGES

The size of the data frame is limited to a maximum of 9 bits

Doesn't support multiple slave or multiple master systems

The baud rates of each UART must be within 10% of each other

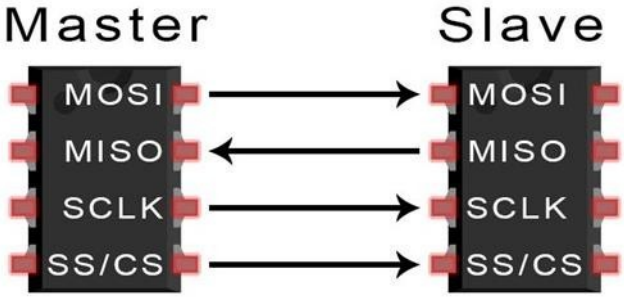
UART or Universal Asynchronous Receiver Transmitter is a dedicated hardware associated with serial communication. The hardware for UART can be a circuit integrated on the microcontroller or a dedicated IC. This is contrast to SPI or I2C, which are just communication protocols. UART is one of the most simple and most commonly used Serial Communication techniques. Today, UART is being used in many applications like GPS Receivers, Bluetooth Modules, GSM and GPRS Modems, Wireless Communication Systems, RFID based applications etc.

SPI COMMUNICATION PROTOCOL

SPI is a common communication protocol used by many different devices. For example, SD card modules, RFID card reader modules, and 2.4 GHz wireless transmitter/receivers all use SPI to communicate with microcontrollers.

One unique benefit of SPI is the fact that data can be transferred without interruption. Any number of bits can be sent or received in a continuous stream. With I2C and UART, data is sent in packets, limited to a specific number of bits. Start and stop conditions define the beginning and end of each packet, so the data is interrupted during transmission.

Devices communicating via SPI are in a master-slave relationship. The master is the controlling device (usually a microcontroller), while the slave (usually a sensor, display, or memory chip) takes instruction from the master. The simplest configuration of SPI is a single master, single slave system, but one master can control more than one slave (more on this below).



MOSI (Master Output/Slave Input) – Line for the master to send data to the slave.

MISO (Master Input/Slave Output) – Line for the slave to send data to the master

SCLK (Clock) – Line for the clock signal.

SS/CS (Slave Select/Chip Select) – Line for the master to select which slave to send data to.

Wires Used	4
Maximum Speed	Up to 10 Mbps
Synchronous or Asynchronous?	Synchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	Theoretically unlimited*

*In practice, the number of slaves is limited by the load capacitance of the system, which reduces the ability of the master to accurately switch between voltage levels.

HOW SPI WORKS

THE CLOCK

The clock signal synchronizes the output of data bits from the master to the sampling of bits by the slave. One bit of data is transferred in each clock cycle, so the speed of data transfer is determined by the frequency of the clock signal. SPI communication is always initiated by the master since the master configures and generates the clock signal.

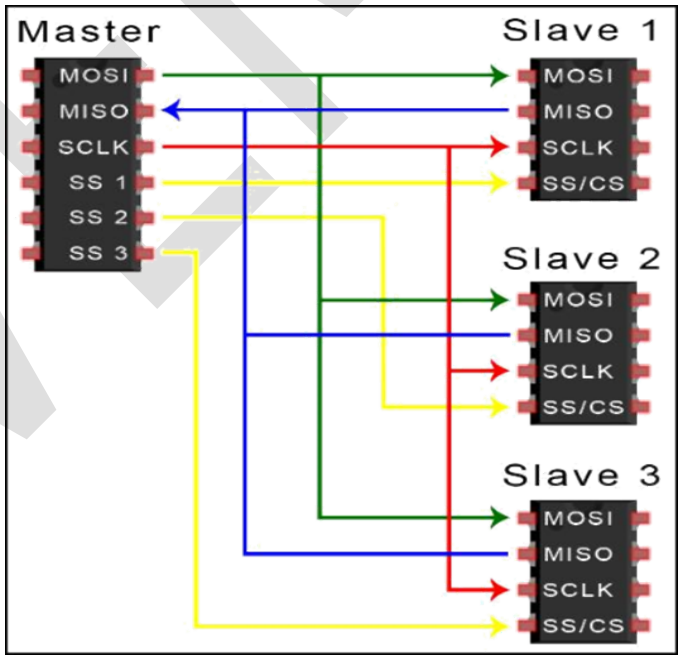
Any communication protocol where devices share a clock signal is known as *synchronous*. SPI is a synchronous communication protocol. There are also *asynchronous* methods that don't use a clock signal. For example, on, in both sides are set to a pre-configured baud rate that dictates the speed and timing of data transmission.

The clock signal in SPI can be modified using the properties of *clock polarity* and *clock phase*. These two properties work together to define when the bits are output and when they are sampled. Clock polarity can be set by the master to allow for bits to be output and sampled on either the rising or falling edge of the clock cycle. Clock phase can be set for output and sampling to occur on either the first edge or second edge of the clock cycle, regardless of whether it is rising or falling.

SLAVE SELECT

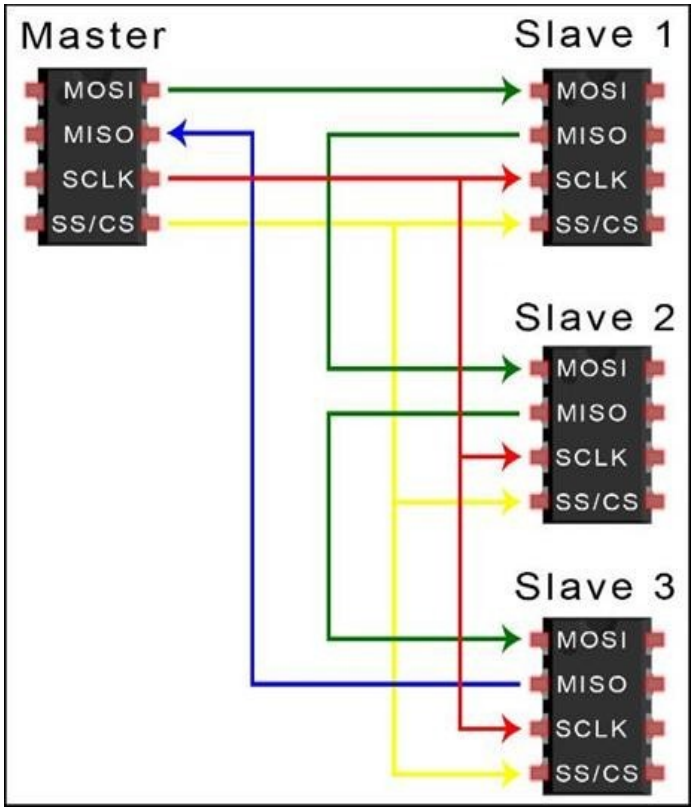
The master can choose which slave it wants to talk to by setti a low voltage level. In the idle, non-transmitting state, the slave select line is kept at a high voltage level. Multiple CS/SS pins may be available on the master, which allows for multiple slaves to be wired in parallel. If only one CS/SS pin is present, multiple slaves can be wired to the master by daisy-chaining.

MULTIPLE SLAVES



SPI can be set up to operate with a single master and a single slave, and it can be set up with multiple slaves controlled by a single master. There are two ways to connect multiple slaves to the master. If the master has multiple slave select pins, the slaves can be wired in parallel like this:

If only one slave select pin is available, the slaves can be daisy-chained like this:

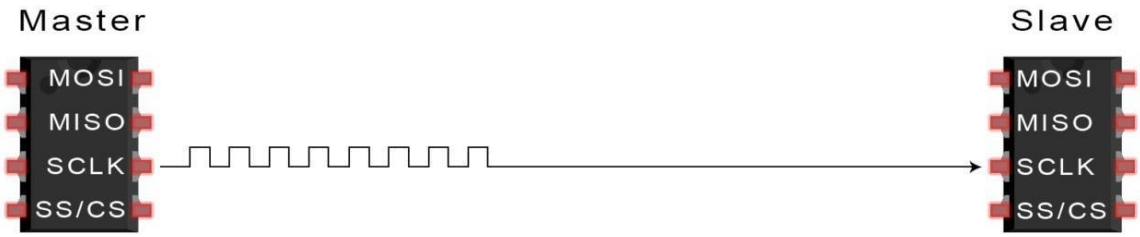


MOSI AND MISO

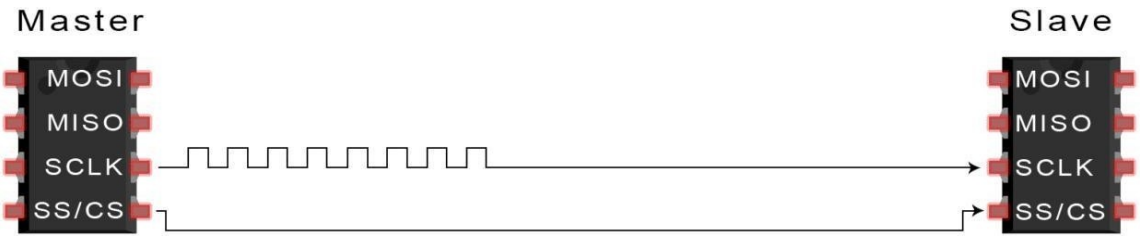
The master sends data to the slave bit by bit, in serial through the MOSI line. The slave receives the data sent from the master at the MOSI pin. Data sent from the master to the slave is usually sent with the most significant bit first.The slave can also send data back to the master through the MISO line in serial. The data sent from the slave back to the master is usually sent with the least significant bit first.

STEPS OF SPI DATA TRANSMISSION

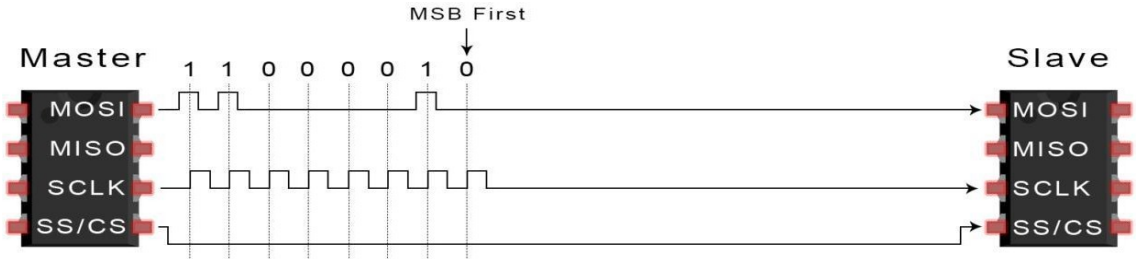
1. The master outputs the clock signal:



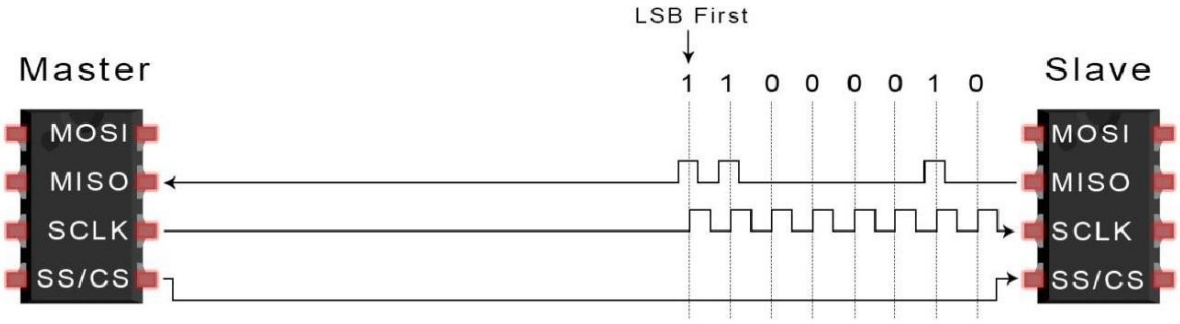
2. The master switches the SS/CS pin to a low voltage state, which activates the slave:



3. The master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received:



4. If a response is needed, the slave returns data one bit at a time to the master along the MISO line. The master reads the bits as they are received:



ADVANTAGES

No start and stop bits, so the data can be streamed continuously without interruption No

complicated slave addressing system like I2C

Higher data transfer rate than I2C (almost twice as fast)

Separate MISO and MOSI lines, so data can be sent and received at the same time

DISADVANTAGES

Uses four wires (I2C and UARTs use two)

No acknowledgement that the data has been successfully received (I2C has this) No

form of error checking like the parity bit in UART

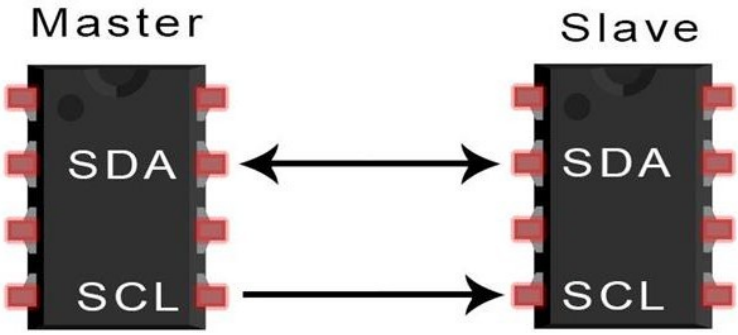
Only allows for a single master.

I2C COMMUNICATION PROTOCOL

Inter IC (i2c) (IIC) is important serial communication protocol in modern electronic systems. Philips invented this protocol in 1986. The objective of reducing the cost of production of television remote control motivated Philips to invent this protocol. IIC is a serial bus interface, can be implemented in software, but most of the microcontrollers support IIC by incorporating it as hard IP (Intellectual Property). IIC can be used to interface microcontroller with RTC, EEPROM and different variety of sensors. IIC is used to interface chips on motherboard, generally between a processor chip and any peripheral which supports IIC. IIC is very reliable wireline communication protocol for an on board or short distances. I2C is a serial protocol for two-wire interface to connect low-speed devices like microcontrollers, EEPROMs, A/D and D/A converters, I/O interfaces and other similar peripherals in embedded systems

I2C combines the best features of SPI and UARTs. With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, or multiple slaves. This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD.

IIC protocol uses two wires for data transfer between devices: Serial Data Line (SDA) and Serial Clock Line (SCL). The reduction in number of pins in comparison with parallel data transfer is evident. This reduces the cost of production, package size and power consumption. IIC is also best suited protocol for battery operated devices. IIC is also referred as two wire serial interface (TWI).

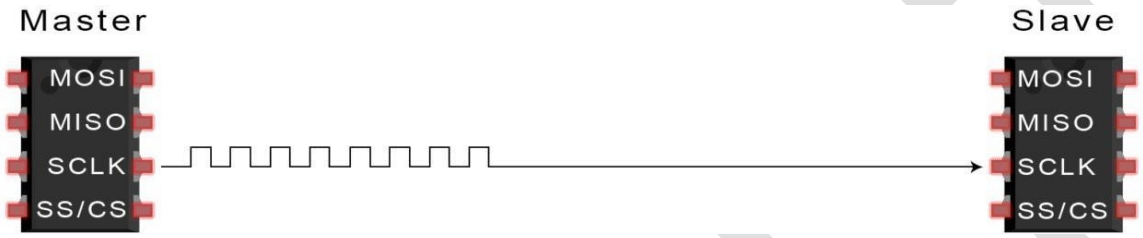


MOSI AND MISO

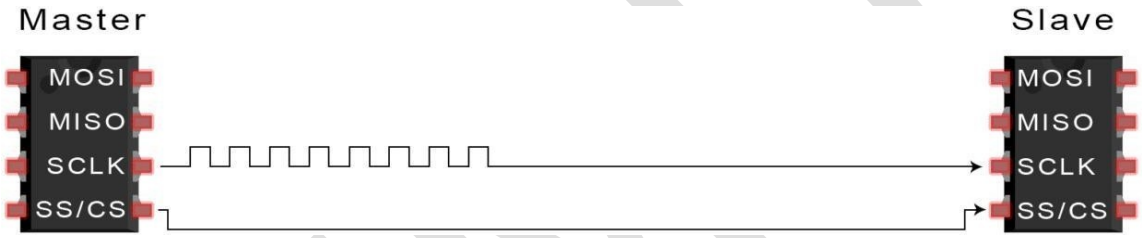
The master sends data to the slave bit by bit, in serial through the MOSI line. The slave receives the data sent from the master at the MOSI pin. Data sent from the master to the slave is usually sent with the most significant bit first.The slave can also send data back to the master through the MISO line in serial. The data sent from the slave back to the master is usually sent with the least significant bit first.

STEPS OF SPI DATA TRANSMISSION

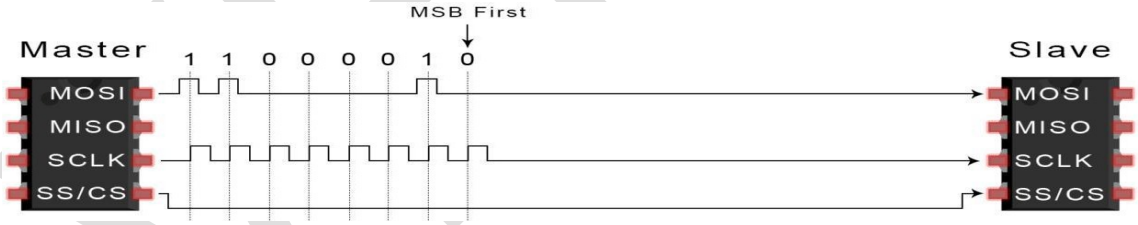
1. The master outputs the clock signal:



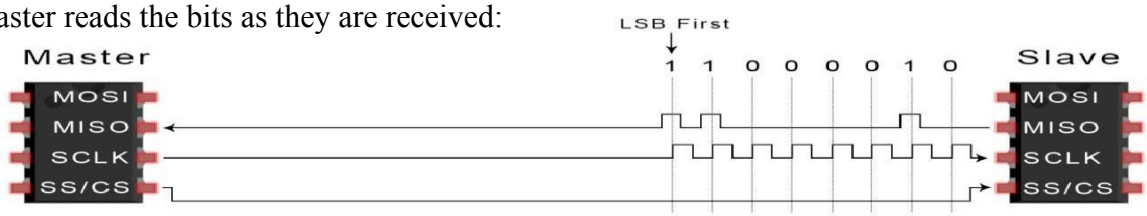
2. The master switches the SS/CS pin to a low voltage state, which activates the slave:



4. The master sends the data one bit at a time to the slave along the MOSI line. The slave reads the bits as they are received:



5. If a response is needed, the slave returns data one bit at a time to the master along the MISO line. The master reads the bits as they are received:



ADVANTAGES

No start and stop bits, so the data can be streamed continuously without interruption No complicated slave addressing system like I2C

Higher data transfer rate than I2C (almost twice as fast)

Separate MISO and MOSI lines, so data can be sent and received at the same time

DISADVANTAGES

Uses four wires (I2C and UARTs use two)

No acknowledgement that the data has been successfully received (I2C has this) No form of error checking like the parity bit in UART

- Only allows for a single master.

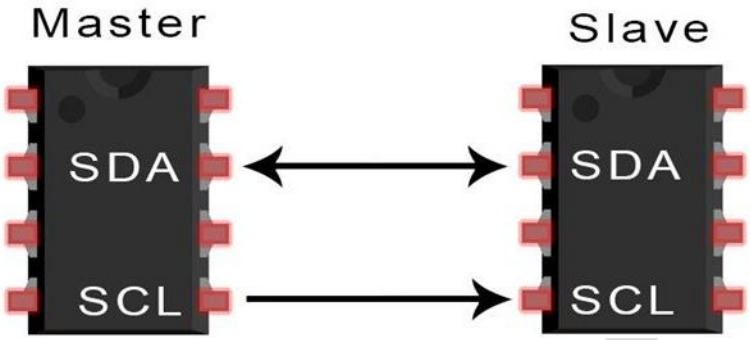
I2C COMMUNICATION PROTOCOL

Inter IC (i2c) (IIC) is important serial communication protocol in modern electronic systems. Philips invented this protocol in 1986. The objective of reducing the cost of production of television remote control motivated Philips to invent this protocol. IIC is a serial bus interface, can be implemented in software, but most of the microcontrollers support IIC by incorporating it as hard IP (Intellectual Property). IIC can be used to interface microcontroller with RTC, EEPROM and different variety of sensors. IIC is used to interface chips on motherboard, generally between a processor chip and any peripheral which supports IIC. IIC is very reliable wireline communication protocol for an on board or short distances. I2C is a serial protocol for two-wire interface to connect low-speed devices like microcontrollers, EEPROMs, A/D and D/A converters, I/O interfaces and other similar peripherals in embedded systems

I2C combines the best features of SPI and UARTs. With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, or multiple slaves. This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD.

IIC protocol uses two wires for data transfer between devices: Serial Data Line (SDA) and Serial Clock Line (SCL). The reduction in number of pins in comparison with parallel data transfer is evident.

This reduces the cost of production, package size and power consumption. IIC is also best suited protocol for battery operated devices. IIC is also referred as two wire serial interface (TWI).



SDA (Serial Data) – The line for the master and slave to send and receive data.

SCL (Serial Clock) – The line that carries the clock signal.

I2C is a serial communication protocol, so data is transferred bit by bit along a single wire (the SDA line).Like SPI, I2C is synchronous, so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The clock signal is always controlled by the master.

Wires Used	2
Maximum Speed	Standard mode= 100 kbps
	Fast mode= 400 kbps
	High speed mode= 3.4 Mbps
	Ultra fast mode= 5 Mbps
Synchronous or Asynchronous?	Synchronous
Serial or Parallel?	Serial
Max # of Masters	Unlimited
Max # of Slaves	1008

HOW I2C WORKS

With I2C, data is transferred in *messages*. Messages are broken up into *frames* of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame:

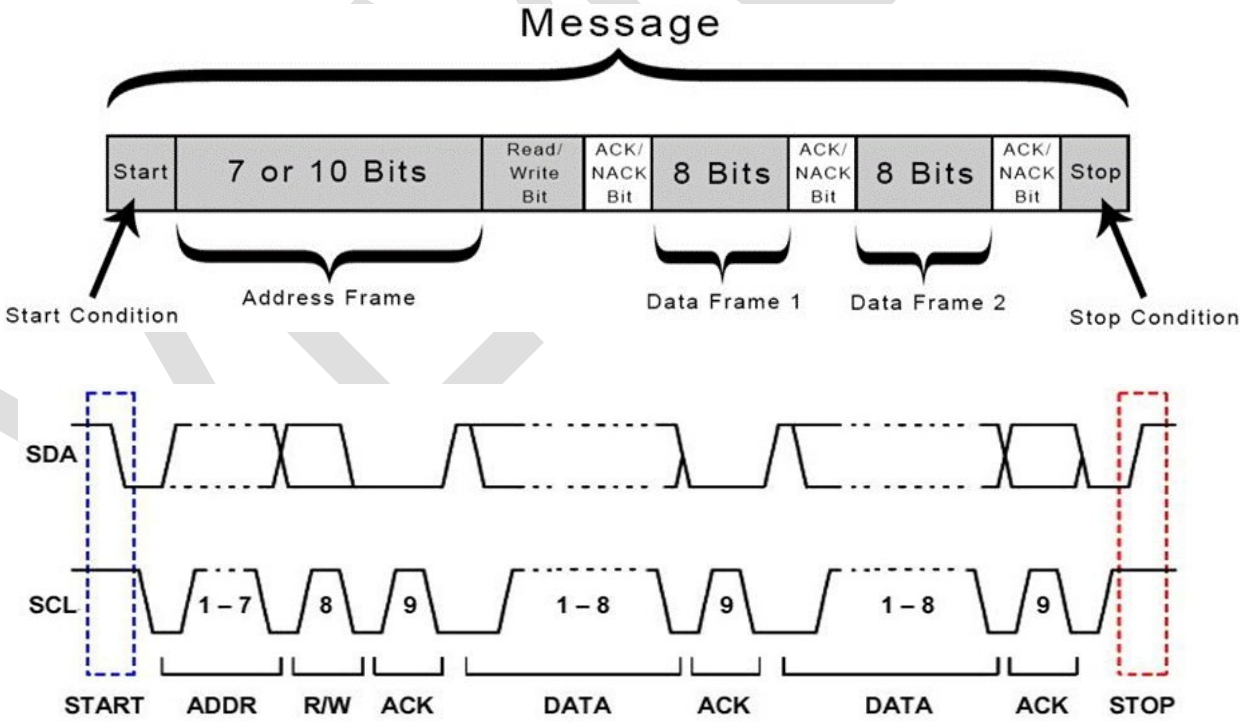
Start Condition: The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.

Stop Condition: The SDA line switches from a low voltage level to a high voltage level *after* the SCL line switches from low to high.

Address Frame: A 7 or 10 bit sequence unique to each slave that identifies the slave when the master wants to talk to it.

Read/Write Bit: A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

ACK/NACK Bit: Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.



ADDRESSING

I2C doesn't have slave select it needs lines another like way to SPI let the slave so know that data is being sent to it, and not another slave. It does this by *addressing*. The address frame is always the first frame after the start bit in a new message.

The master sends the address of the slave it wants to communicate with to every slave connected to it. Each slave then compares the address sent from the master to its own address. If the address matches, it sends a low voltage ACK bit back to the match, the slave does nothing and the SDA line remains high.

READ/WRITE BIT

The address frame includes a single bit at the end that informs the slave whether the master wants to write data to it or receive data from it. If the master wants to send data to the slave, the read/write bit is a low voltage level. If the master is requesting data from the slave, the bit is a high voltage level.

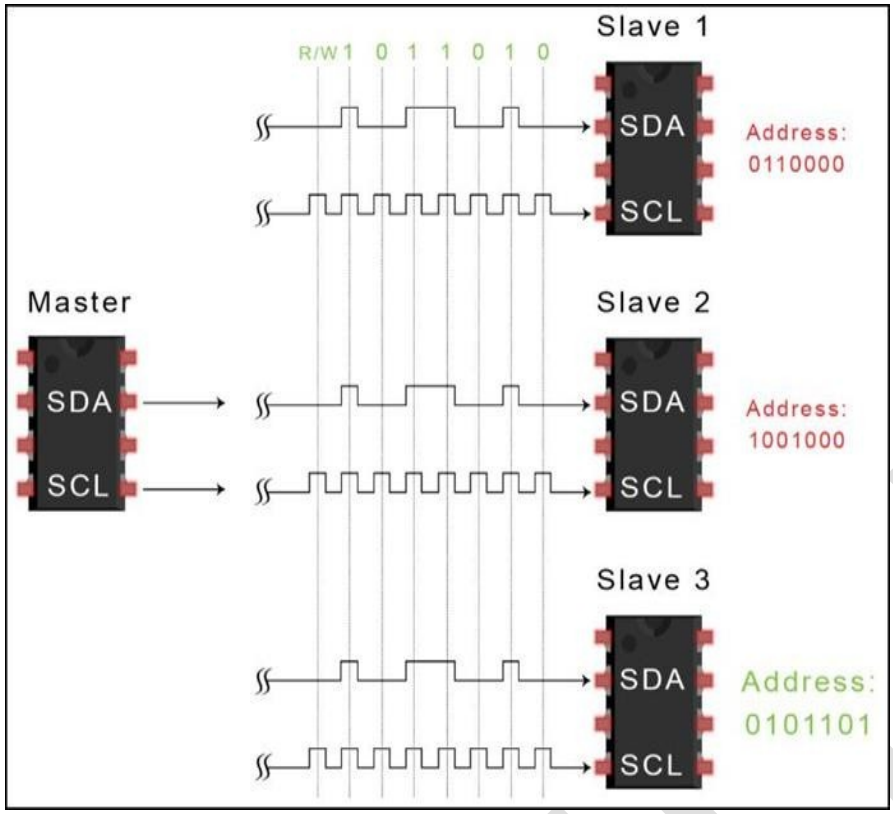
THE DATA FRAME

After the master detects the ACK bit from the slave, the first data frame is ready to be sent.

The data frame is always 8 bits long, and sent with the most significant bit first. Each data frame is immediately followed by an ACK/NACK bit to verify that the frame has been received successfully. The ACK bit must be received by either the master or the slave (depending on who is sending the data) before the next data frame can be sent. After all of the data frames have been sent, the master can send a stop condition to the slave to halt the transmission. The stop condition is a voltage transition from low to high on the SDA line after a low to high transition on the SCL line, with the SCL line remaining high.

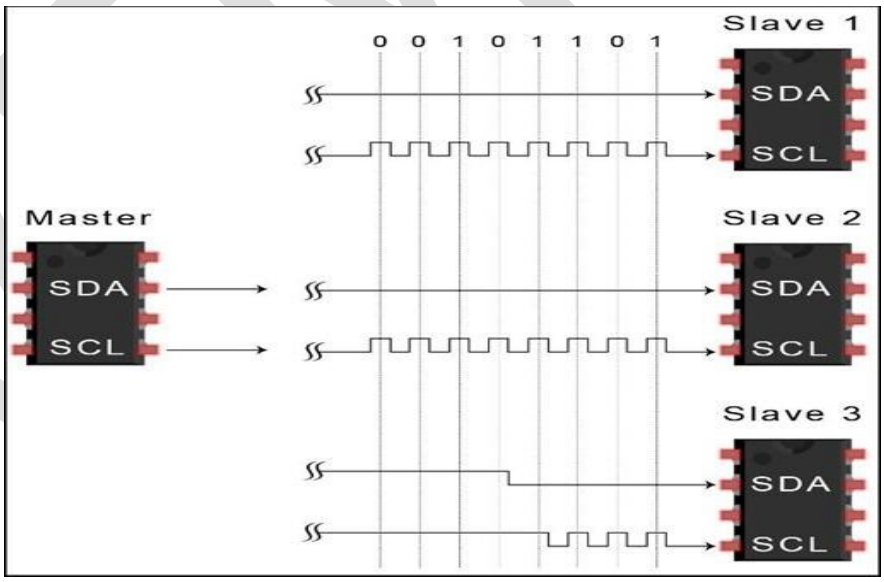
STEPS OF I2C DATA TRANSMISSION

1. The master sends the start condition to every connected slave by switching the SDA line from a high voltage level to a low voltage level *before* switching the SCL line from high to low:
2. The master sends each slave the 7 or 10 bit address of the slave it wants to communicate with, along with the read/write bit:
3. Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave's own address, SDA line high.
4. The master sends or receives the data frame:



5. After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame:

6. To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high:



SINGLE MASTER WITH MULTIPLE SLAVES

Because I2C uses addressing, multiple slaves can be controlled from a single master. With a 7 bit address, 128 (2^7) unique address are available. Using 10 bit addresses is uncommon, but provides 1,024 (2^{10}) unique addresses. To connect multiple slaves to a single master, wire them like this, with 4.7K/10K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:

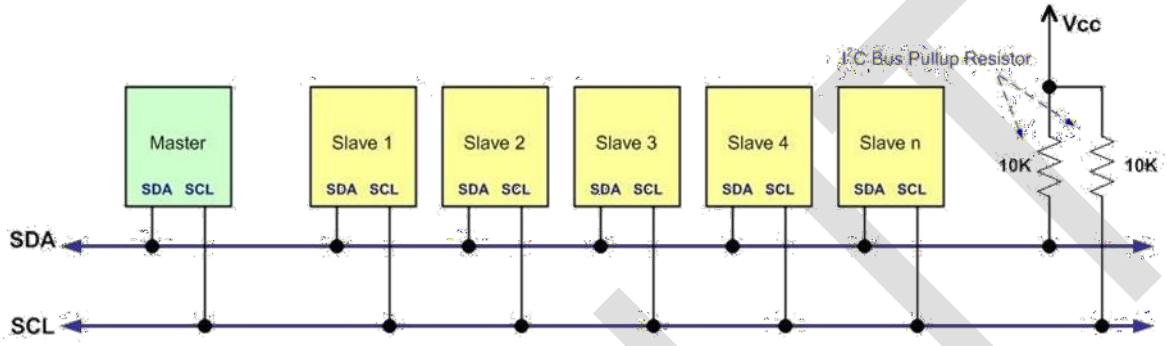
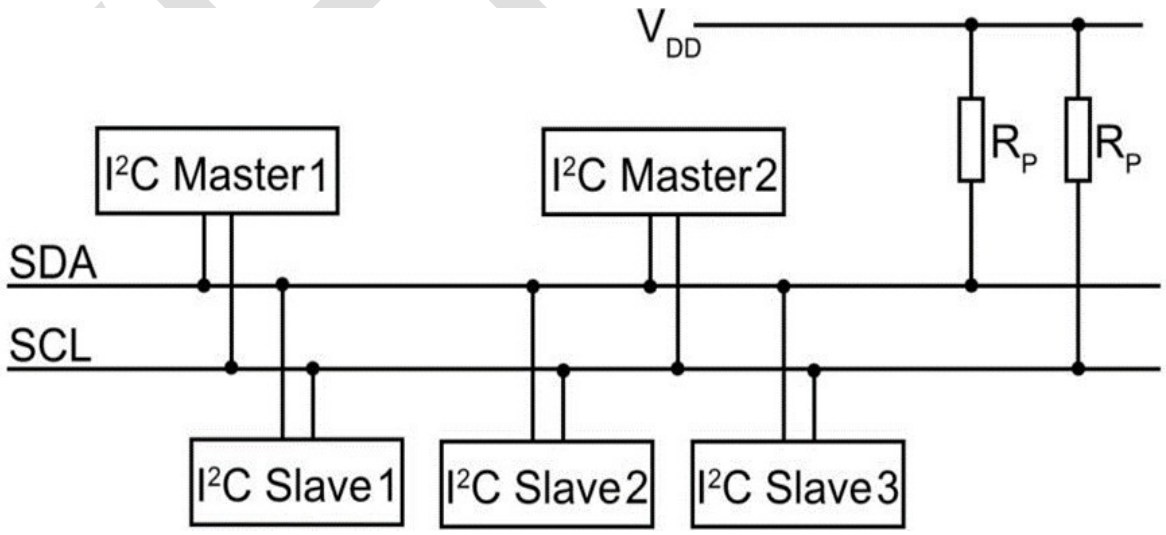


Figure: I2C Bus Connection

MULTIPLE MASTERS WITH MULTIPLE SLAVES

Multiple masters can be connected to a single slave or multiple slaves. The problem with multiple masters in the same system comes when two masters try to send or receive data at the same time over the SDA line. To solve this problem, each master needs to detect if the SDA line is low or high before transmitting a message. If the SDA line is low, this means that another master has control of the bus, and the master should wait to send the message. If the SDA line is high, then it's safe to transmit To connect the multiple masters to .multiple slaves, use the following diagram, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:



ADVANTAGES

Only uses two wires

Supports multiple masters and multiple slaves

ACK/NACK bit gives confirmation that each frame is transferred successfully

Hardware is less complicated than with UARTs

Well known and widely used protocol

DISADVANTAGES

Slower data transfer rate than SPI

The size of the data frame is limited to 8 bits

More complicated hardware needed to implement than SPI

UNIVERSAL SERIAL BUS (USB)

Universal Serial Bus (USB) is a set of interface specifications for high speed wired communication between electronics systems peripherals and devices with or without PC/computer. The USB was originally developed in 1995 by many of the industry leading companies like Intel, Compaq, Microsoft, Digital, IBM, and Northern Telecom.

The major goal of USB was to define an external expansion bus to add peripherals to a PC in easy and simple manner. USB offers users simple connectivity. It eliminates the mix of different connectors for different devices like printers, keyboards, mice, and other peripherals. That means USB-bus allows many peripherals to be connected using a single standardized interface socket. It supports all kinds of data, from slow mouse inputs to digitized audio and compressed video.

USB sends data in serial mode i.e. the parallel data is serialized before sends and de-serialized after receiving. The benefits of USB are low cost, expandability, auto-configuration, hot-plugging and outstanding performance. It also provides power to the bus, enabling many peripherals to operate without the added need for an AC power adapter.

Various versions USB:

USB1.0: USB 1.0 is the original release of USB having the capability of transferring 12Mbps, supporting up to 127 devices. This USB 1.0 specification model was introduced in January 1996.

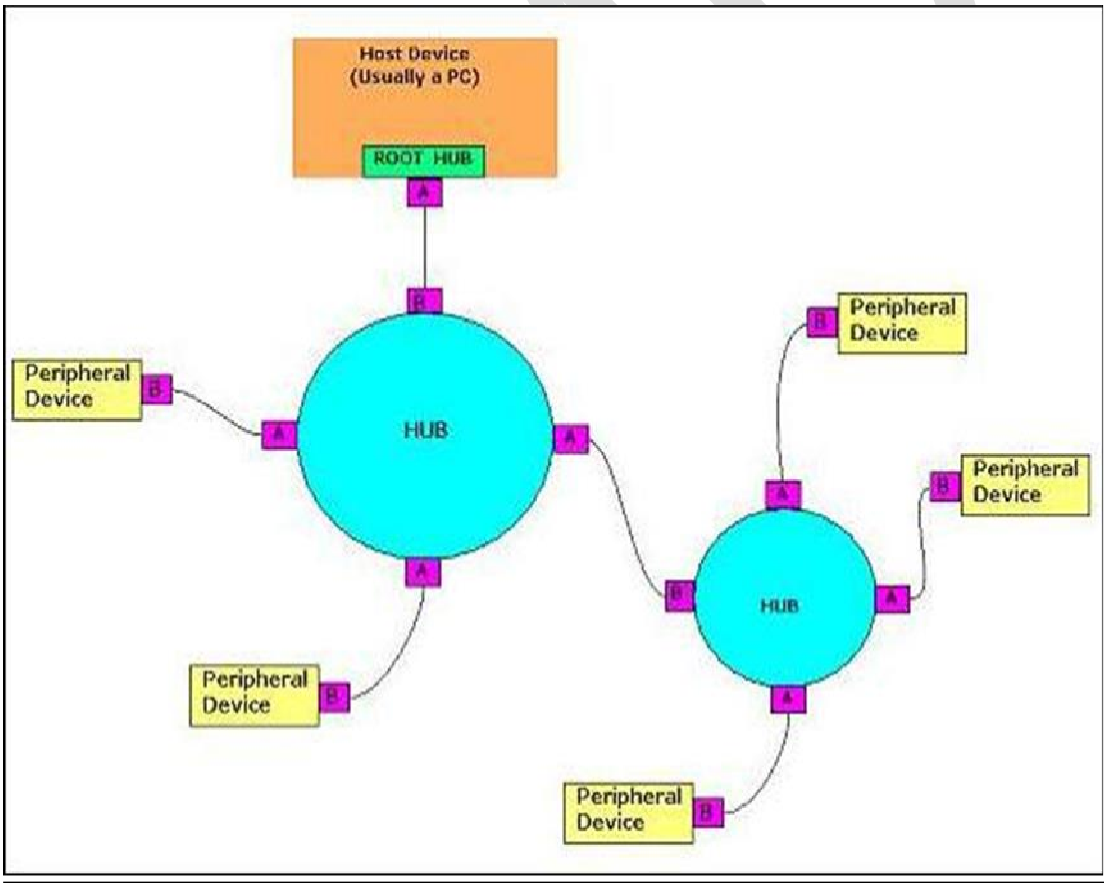
USB1.1: USB 1.1 came out in September 1998. USB 1.1 is also known as full-speed USB. This version is similar to the original release of USB; however, there are minor modifications for the hardware and the specifications. USB version 1.1 supported two speeds, a full speed mode of 12Mbits/s and a low speed mode of 1.5Mbits/s.

USB2.0: USB 2.0, also known as hi-speed USB. This hi-speed USB is capable of supporting a transfer rate of up to 480 Mbps, compared to 12 Mbps of USB 1.1. That's about 40 times as fast! Wow!

USB3.0: It is also called as Super-Speed USB having a data transfer rate of 5Gbps. That means it can deliver over 10x the speed of today's Hi-Speed USB connections.

USB3.1: It is also called as Super-Speed USB+ having a data transfer rate of 10Gbps.

The USB "tiered star" topology:



The USB system is made up of a host, multiple numbers of USB ports, and multiple peripheral devices connected in a tiered-star topology.

The host is the USB system's master, and as such, controls and schedules all communications activities. Peripherals, the devices controlled by USB, are slaves responding to commands from the host. USB devices are linked in series through hubs. There always exists one hub known as the root hub, which is built in to the host controller.

By using different connectors on the upstream and downstream end, it is impossible to install a cable incorrectly, because the two types are physically different.

Pin No	Signal	Color of the cable
1	+5V power	Red
2	- Data	White / Yellow
3	+Data	Green / Blue
4	Ground	Black/Brown

Table: USB pin connections

USB can support 4 data transfer types or transfer modes.

1. Control
2. Isochronous
3. Bulk
4. Interrupt

Control transfers exchange configuration, setup and command information between the device and host. The host can also send commands or query parameters with control packets.

Isochronous transfer is used by time critical, streaming device such as speakers and video cameras. It is time sensitive information so, within limitations, it has guaranteed access to the USB bus.

Bulk transfer is used by devices like printers & scanners, which receives data in one big packet.

Interrupt transfer is used by peripherals exchanging small amounts of data that need immediate attention.

All USB data is sent serially. USB data transfer is essentially in the form of packets of data, sent back and forth between the host and peripheral devices. Initially all packets are sent from the host, via the root hub and possibly more hubs, to devices.

Each USB data transfer consists of a…

1. Token packet (Header defining what it expects to follow)
2. Optional Data Packet (Containing the payload)
3. Status Packet (Used to acknowledge transactions and to provide a means of error correction).

Implementing and Programming UART:

TM4C123GH6PM microcontroller has got eight UART ports. They are named as UART0-UART7. In the TI Launchpad, the UART0 port is connected to the ICDI (In-Circuit Debug Interface). ICDI is further connected to USB port. Users can use UART0 for flash programming, debugging using JTAG. The UART features of TI Tiva TM4C123GH6PM microcontroller is: -

- UART‟s have programmable baud-rate generator allowing speeds up to 5 Mbps for regular speed and 10 Mbps for high speed.
- Separate 16x8 transmit (TX) and receive (RX) FIFOs to reduce CPU interrupt service loading with programmable FIFO length
- Standard asynchronous communication bits for start, stop, and parity, Line-break generation and detection
- Fully programmable serial interface characteristics o 5, 6, 7, or 8 data bits
- Even, odd, stick, or no-parity bit generation/detection o 1 or 2 stop bit generation
- IrDA serial-IR (SIR) encoder/decoder providing
- Programmable use of IrDA Serial Infrared (SIR) or UART input/output
- Support of IrDA SIR encoder/decoder functions for data rates up to 115.2 Kbps half duplex
- Support of normal 3/16 and low-power (1.41-2.23 μs) bit durations
- Programmable internal clock generator enabling division of reference clock by 1 to 256 for low-power mode bit duration
- Support for communication with ISO 7816 smart cards
- Modem flow control (on UART1)
- EIA-485 9-bit support
- Standard FIFO-level and End-of-Transmission interrupts

- Efficient transfers using Micro Direct Memory Access Controller (µDMA) o Separate channels for transmit and receive
- Receive single request asserted when data is in the FIFO; burst request asserted at programmed FIFO level Transmit single request asserted when there is space in the FIFO; burst request asserted at programmed FIFO level.

UART Register Map

TI Tiva TM4C123GH6PM UART has got several Special Function Registers (SFR‟s) which needs to program with appropriate values to achieve required UART functionality. In this section, UART0 is taken as example in which virtual connection is possible on TI Tiva launch pad.

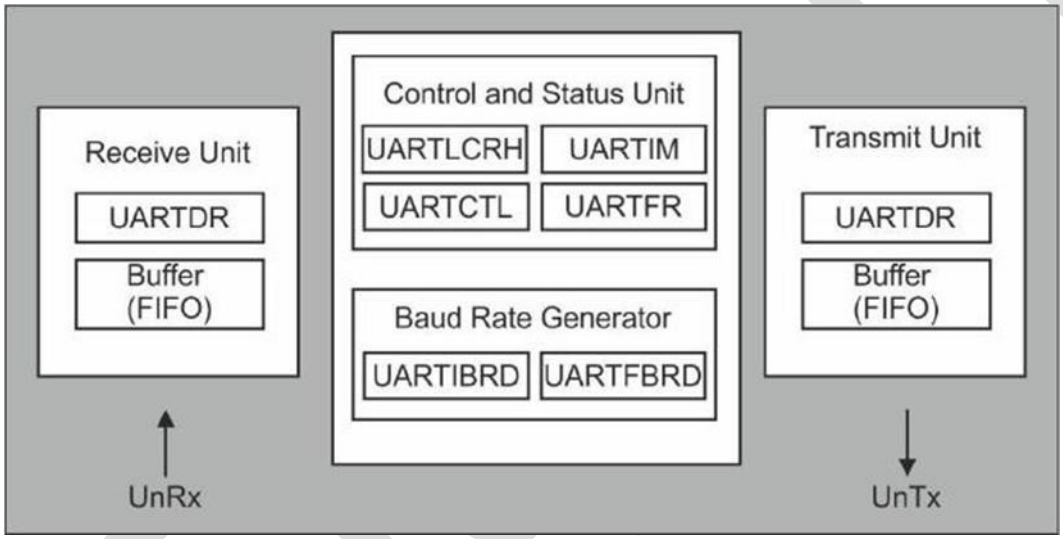


Figure: Simplified block diagram of UART

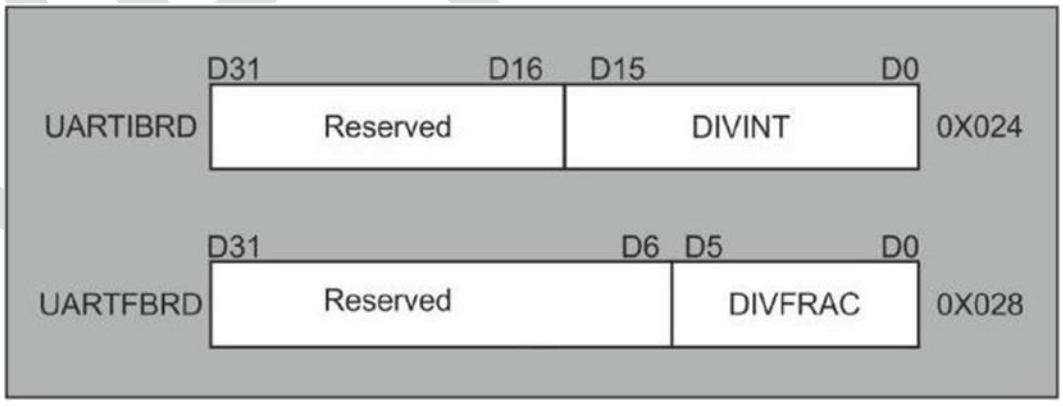


Figure: Baud Rate Registers

Baud Rate Generators: The SFR‟s used in setting the baud rate are UART Integer Baud-Rate Divisor (UARTIBRD) and UART Fractional Baud-Rate Divisor (UARTFBRD). The block diagram of the registers is given above.

The physical addresses for these UART baud rate registers are: 0x4000:C000+0x024 (UARTIBRD) and 0x4000:C000+0x028 (UARTFBRD). Only lower 16 bit are used in UARTIBRD and lower 6-bits are used in UARTFBRD. So it comes to total of 22 bits (16-bit integer + 6 bit of fraction). To reduce the error rate and use the standard baud rate supported by the terminal programs it is required to use both the registers when we program for the baud rate. The standard baud rates are: 2400, 4800, 9600, 19200, 57600 and 115200.

Baud rate can be calculated using the below formula:

$$\text{Desired Baud Rate} = \text{SysClk} / (16 \times \text{ClkDiv})$$

Where the SysClk is the working system clock connected to the UART and ClkDiv is the value programmed into baud rate registers.

The baud-rate divisor (BRD) has the following relationship to the system clock, where BRDI is the integer part of the BRD and BRDF is the fractional part, separated by a decimal place.

$$\text{BRD} = \text{BRDI} + \text{BRDF} = \text{UARTSysClk} / (\text{ClkDiv} * \text{Baud Rate})$$

UARTSysClk is the system clock connected to the UART, and ClkDiv is 16 (if HSE in UARTCTL is clear) or 8 (if HSE is set).

Alternatively, the UART may be clocked from the internal precision oscillator (PIOSC), independent of the system clock selection. This will allow the UART clock to be programmed independently of the system clock PLL settings.

TI Tiva Launchpad system clock is 16 MHz so desired Baud Rate can be calculated as:

$$\text{Baud Rate} = 16\text{MHz} / (16 \times \text{ClkDiv}) = 1\text{MHz} / \text{ClkDiv}$$

The ClkDiv value includes both integer and fractional values loaded into UARTIBRD and UARTFBRD registers. The integer part is easy to calculate and fraction part requires manipulations based on trial and error.

Example: System clock of TI Tiva Launchpad is16 MHz 16MHz is divided by 16 and it is fed into UART. So UART operates at 1MHz frequency. So ClkDiv = 1MHz.

To generate a baud rate of 4800: 1MHz/4800 = 208.33

(a) 1MHz/4800 = 208.33, UARTIBRD=208 & UARTFBRD = (0.33×64) + 0.5 = 21.83 =21

(b) 1MHz/9600 = 104.166666, UARTIBRD = 104 & UARTFBRD = (0.16666×64) +0.5=11

(c) 1MHz/57600 = 17.361, UARTIBRD = 17 and UARTFBRD = (0.361 × 64) + 0.5 =23

(d) 1MHz/115200 = 8.680, UARTIBRD = 8 and UARTFBRD = (0.680 × 64) +0.5=44

Serial IR (SIR):

UART includes an IrDA (Infrared) serial IR encoder-decoder block. SIR block converts the data between UART and half-duplex serial SIR interface. The SIR block provides a digitally encoded output and decoded input to UART. SIR block uses UnTx and UnRx pins for SIR interface. These pins are connected to IrDA SIR physical layer link. SIR block supports half-duplex communication. The IrDA SIR physical layer specifies a minimum 10-ms delay between transmission and reception. The SIR block has two modes of operation normal mode and low power mode.

ISO 7816 Support: UART support ISO 7816 smartcard communication. The UnTx signal is used as a bit clock and the UnRx signal is used as the half-duplex communication line connected to the smartcard. Any GPIO signal can be used to generate the reset signal to the smartcard.

UART Control Register (UARTCTL):

This is a 32-bit register. The most important bits are RXE, TXE, HSE, and UARTEN.

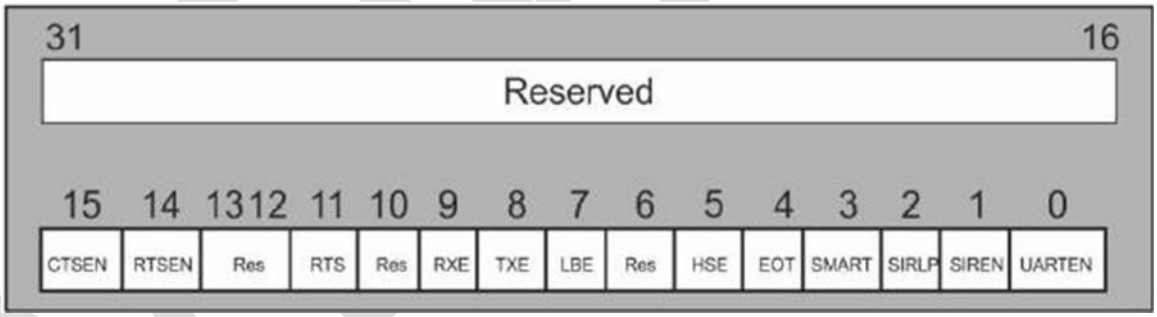


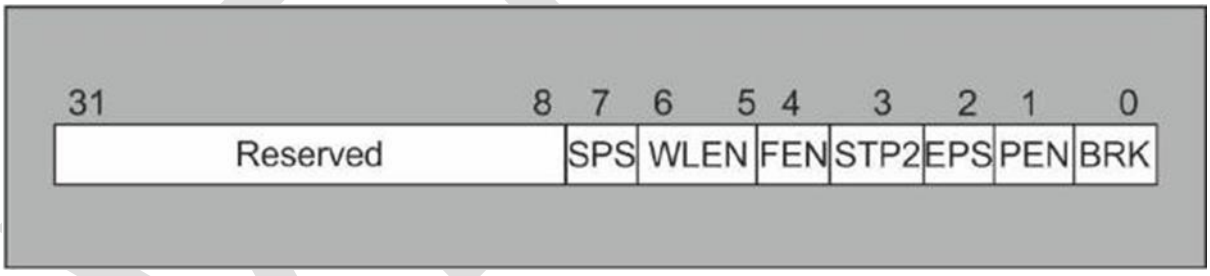
Figure: UART Control Register (UARTCTL)

- RXE (Receive enable): This bit should be enabled to receive data.
- TXE (Transmit Enable): This bit should be enabled to transmit data.
- HSE (High Speed enable): This bit is used to set the baud rate. By default the system clock is divided by 16 before it is fed to the UART. The user can program HSE =1, to make system clock divide by 8.

- UARTEN (UART enable): This bit allows user to enable or disable the UART. During the initialization of the UART registers, this is disabled. To disable UART under any circumstances, this bit is used.
- SIREN (SIR Enable): IrDA SIR Block is enabled. UART will transmit and receive data using SIR protocol.
- SIRLP (SIR Low Power Mode): This bit selects the IrDA encoding mode: Normal mode or low power mode.
- SMART (ISO 7816 Smart Card support): The UART operates in Smart Card mode when SMART = 1. UART does not support automatic retransmission on parity errors. If a parity error is detected on transmission, all further transmit operations are aborted and software must handle retransmission of the affected byte or message.
- LBE (Loop Back Enable): The UnTx path is fed through the UnRx path when LBE =1.
- RTSEN (Enable Request to send): RTS hardware flow control is enabled. Data is only requested when receive FIFO has available entries.
- RTS (Request to send): When RTSEN is clear, the status of this bit is reflected on the U1RTS signal. If RTSEN is set, this bit is ignored on a write and should be ignored on read.

UART Line Control Register (UARTLCTH)

This register is used to set the length of data. The bits per character in a frame and number of stop bits are also decided.



- STP2 (Stop bit2): The stop bits can be 1 or 2. The default is 1 stop bit at the end of each frame. If the receiving device is slow, we can use 2 stop bits by making the STP2=1.
- FEN (FIFO Enable): UART has an internal 16-byte FIFO (first in first out) buffer to store data for transmission to keep the CPU getting interrupted for the reception and transmission of every byte. Enabling FEN bit, we can write up to16 bytes of data block into its transmission FIFO buffer and let transfer happen one byte at a time. There is also a separate 16 byte FIFO for the receiver to buffer the incoming data. Upon Reset, the default for FIFO buffer size is 1 byte.

- WLEN (Word Length): The number of bits per character data in each frame can be 5, 6, 7, or 8. we use 8 bits for each character data frame. Default world length mode is 5.
- BRK (Send Break): A Low level is continually output on the UnTx signal, after completing transmission of the current character. For the proper execution of the break command, software must set this bit for at least two frames (character periods).
- PEN (Parity Enable): Parity is enabled and parity bit is added to the data frame by making PEN = 1. Parity checking is also enabled.
- EPS (Even Parity Select): Odd parity is performed, which checks for an odd number of 1s when EPS = 0. Even parity generation and checking is performed during transmission and reception, which checks for an even number of 1s in data and parity bits when EPS = 1.

UART Data Register (UARTDR):

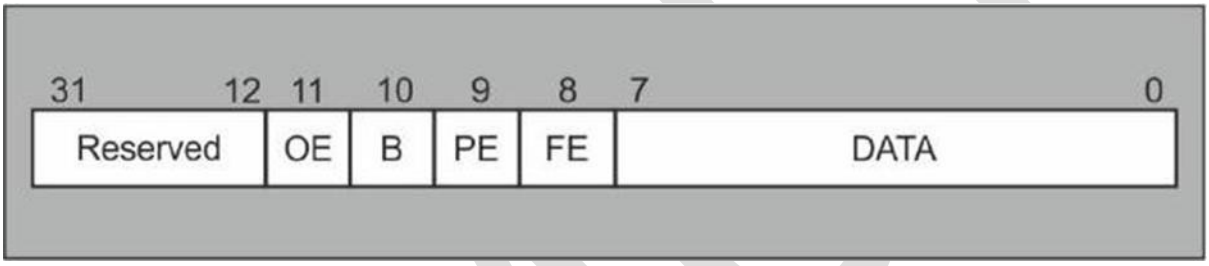


Figure: UART Date Register (UARTDR)

Data should be placed in data register before transmission. Only lower 8 bits are used. In a similar way, the received byte should be read and saved in memory before it gets overwrite by next byte. During reception, we use other four bits (8, 9, 10 and 11) to detect error, parity etc. Another set of registers are used to check the source of error. (UARTRSR/UARTRCR)

- OE: Overrun error (OE = 0: No data is lost).
- BE: Break error
- PE: Parity error
- FE: Framing error.

UART Flag Register (UARTFR):

The UART Flag Register holds one byte of data when FIFO buffer is disabled.

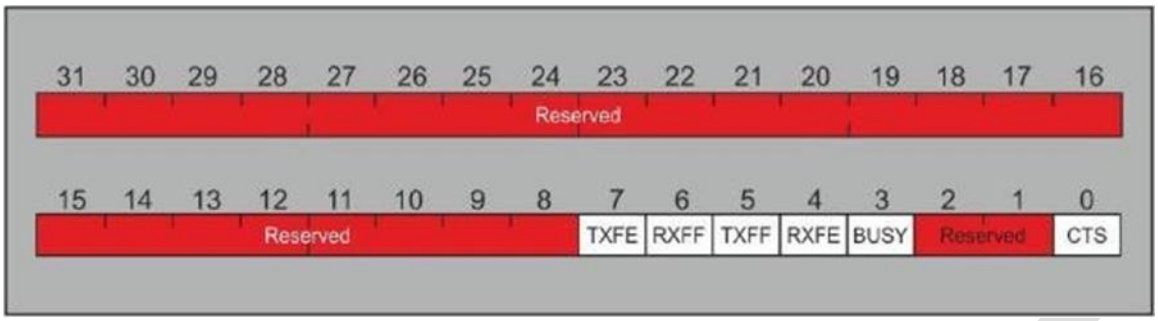


Figure: UART Flag Register (UARTFR)

- TXFE (TX FIFO Empty): Transmitter loads one byte for transmission from the FIFO buffer.
- When FIFO becomes empty, the TXFE is raised. The transmitter then frames the byte and sends it out via TxD pin bit by bit serially.
- RXFF (RX FIFO Full): When a byte of data is received, byte is placed in Data register and RXFF (RX FIFO full) flag bit is raised after receiving the complete byte.
- TXFF (TX FIFOI Full): When the transmitter is not busy, it loads one byte from the FIFO buffer and the FIFO is not full anymore and the TXFF is lowered. We can monitor TXFF flag and upon going LOW we can write another byte to the Data register.

UART Transmission

Step to perform UART Transmission:

- Program the RCGCUART register to get clock on UART0.
- Program the RCGCGPIO register to get the clock for PORTA.
- Program UARTCTL to disable UART0.
- Program the integer part and fractional part into baud rate registers: UARTIBRD and UARTFBRD for UART0.
- Program UARTCC to select the system clock as UART clock.
- Set the bits in UARTLCRH register for 1 stop bit, no interrupt, no FIFO use, and for 8-bit date size (for UART 0).
- Program TxE and RxE in UARTCTL to enable transmitter and receiver.
- Make PA0 and PA1 pins to use as digital pins.
- Configure PA0 and PA1 pins for UART.

- Loop the program for wait on TxD output. Monitor the TXFF flag bit and when it goes low, write a data into data register.

UART Reception

Step by Step Execution of UART Reception:

- Program the RCGCUART register to get clock on UART0.
- Program the RCGCGPIO register to get the clock for PORTA.
- Program UARTCTL to disable UART0.
- Program the integer part and fractional part into baud rate registers: UARTIBRD and UARTFBRD for UART0.
- Program UARTCC to select the system clock as UART clock.
- Set the bits in UARTLCRH register for 1 stop bit, no interrupt, no FIFO use, and for 8 -bit data size (for UART 0).
- Program TxE and RxE in UARTCTL to enable transmitter and receiver.
- Make PA0 and PA1 pins to use as digital pins.
- Configure PA0 and PA1 pins for UART.
- Loop the program for wait on TxD output. Monitor the TXFF flag bit and when it goes low, write a data into data register.
- Monitor the RXFE flag bit in UART Flag register and when it goes LOW read the received byte from Data register and save before it gets overwrite.

Basic UART programing

Example 1:

Program to send the characters "HELLO" to HyperTerminal of PC

```
#include <stdint.h>
#include "tm4c123gh6pm.h"
void UART0Tx(char c); void
delayMs(int n);
int main(void)
SYSCTL->RCGCUART |= 1;            /* enable clock supply to UART*/
```

```
SYSCTL->RCGCGPIO |= 1; /* enable clock supply to PORTA */ /* UART0 initialization */
UART0->CTL = 0;       /* disable UART0 */
UART0->IBRD = 104;            /* 9600 baud rate


UART0->FBRD = 11;             /*    fractional    portion*/
UART0->CC = 0; /* configured to system clock */
UART0->LCRH = 0x60; /* 8-bit, no parity, 1-stop bit, no FIFO */ UART0->CTL = 0x301; /* configure UART0 and TXE, RXE*/ /* UART0 TX0 and RX0 use PA0 and PA1. Set them up. */ GPIOA->DEN = 0x03; /* Make PA0 and PA1 as digital */
GPIOA->AFSEL = 0x03; /* Use PA0, PA1 alternate function */ GPIOA->PCTL = 0x11; /* configure PA0 and PA1 for UART */ delayMs(1); /* wait for output line to stabilize */
for(;;)
{ UART0Tx('H')
;
UART0Tx('E');
UART0Tx('L');
UART0Tx('L ');
UART0Tx('O');
}
}
/* UART0 Transmit */                    /* wait until Tx buffer not full */
void UART0Tx(char c)                    /* before giving it another byte */
{
while((UART0->FR & 0x20) != 0);
UART0->DR = c;
}
```

Example 2:

Program to receive data serially via UART0

```
#include <stdint.h>
#include "tm4c123gh6pm.h"
char UART0Rx(void);
void delayMs(int n);
```

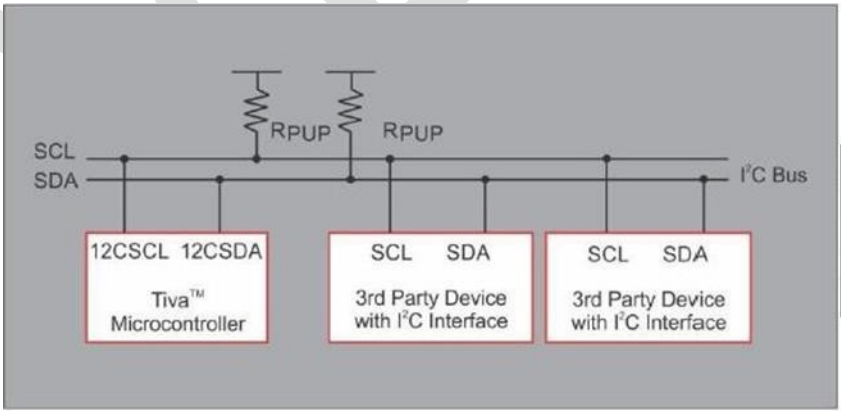
```
int main(void)
{
char c;

SYSCTL->RCGCUART |= 1;            /* enable clock supply to UART*/
SYSCTL->RCGCGPIO |= 1; /* enable clock supply to PORTA */
/* UART0 initialization */
UART0->CTL = 0;      /* disable UART0 */
UART0->IBRD = 104; /* 9600 baud rate */ UART0-
>FBRD = 11;              /* fractional portion*/
UART0->CC = 0;       /* configured to system clock */
UART0->LCRH = 0x60;        /* 8-bit, no parity, 1-stop bit, no FIFO */
UART0->CTL = 0x301;        /* configure UART0 and TXE, RXE */
/* UART0 TX0 and RX0 use PA0 and PA1. Set them up. */ GPIOA-
>DEN = 0x03;               /* Make PA0 and PA1 as digital */
GPIOA->AFSEL = 0x03;       /* Use PA0, PA1 alternate function */
GPIOA->PCTL = 0x11;        /* configure PA0 and PA1 for UART */
for(;;)
{
c = UART0Rx();         /* get a character from UART */
}
}
/* UART0 Receive */
char UART0Rx(void)
{ char c;
while((UART0->FR & 0x10) != 0);          /* wait until the buffer is not empty */
c = UART0->DR;                           /* read the received data */ /* and return it */
return c;
}
```

Implementing and Programming I2C:

- The TM4C123GH6PM controller includes four I2C modules with the following features:
- Devices on the I2C bus can be designated as either a master or a slave
- Supports both transmitting and receiving data as either a master or a slave
- Supports simultaneous master and slave operation
- Four I2C modes
  - o Master transmit o Master receive o Slave transmit o Slave receive
- Four transmission speeds: o Standard (100 Kbps) o Fast-mode (400 Kbps)
  - o Fast-mode plus (1 Mbps) o High-speed mode (3.33 Mbps)
- Clock low timeout interrupt
- Dual slave address capability
- Glitch suppression
- Master and slave interrupt generation
- Master generates interrupts when a transmit or receive operation completes (or aborts due to an error)
- Slave generates interrupts when data has been transferred or requested by a master or when a START or STOP condition is detected
- Master with arbitration and clock synchronization, multi-master support, and 7-bit addressing mode.

I2C Network:

There are four on chip IIC modules in this Tiva microcontroller. The base address of each IIC module is shown in below table:



Module	Base Address
I2C 0	0x4002.0000
I2C 1	0x4002.1000
I2C 2	0x4002.2000
I2C 3	0x4002.3000

Figure: I2C Networking using Tiva microcontroller

Clock should be enabled to IIC module and system control register (SYSCTL) RCGCI2C needs to be programmed. To enable the clock SYSCTL ->RCGCI2C | = 0x0F will enable clock to all four modules

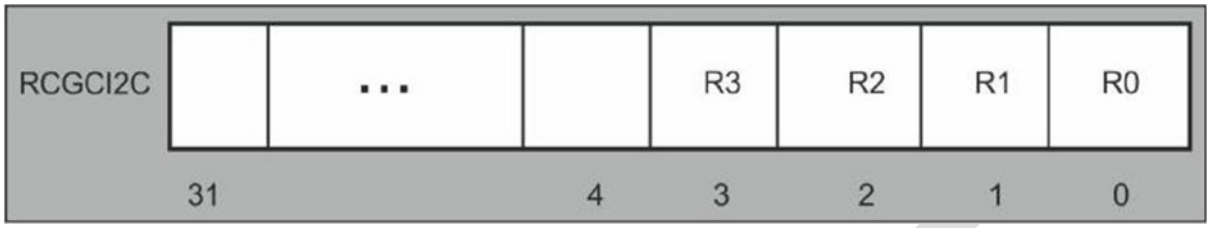


Figure: RunMode Clock Gating Control Register (RCGCI2C)

Clock should be enabled to IIC module and system control register (SYSCTL) RCGCI2C needs to be programmed.

To enable the clock SYSCTL ->RCGCI2C | = 0x0F will enable clock to all four modules. Clock Speed: I2CMTPR (I2C Master Timer Period) register is programmed to set the clock frequency for SCL.

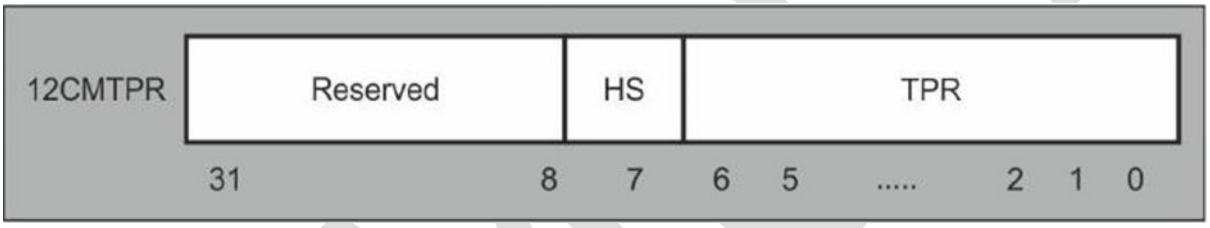


Figure: I2C Master Time Period Register

Bit	Function	Description
R0	I2C0 clock gating control	1: Enable, 0: disable
R1	I2C1 clock gating control	1: Enable, 0: disable
R2	I2C2 clock gating control	1: Enable, 0: disable
R3	I2C3 clock gating control	1: Enable, 0: disable

Table: RCG12C Register Description The formula used to set the clock

speed is given below:

SCL_PERIOD = 2 x (1+TPR) x (SCL_LP+ SCL_HP) x CLK_PRD

Where

CLK_PRD: System Clock period

SCL_LP: SCL low period and it is fixed at 6.

SCL_HP: SCL High period and it is fixed at 4.

Finally, the above equation can be written as:

SCL_PERIOD = (20 x (1+TPR)/ System clock frequency

The TPR can be calculated as:

TPR = ((System clock frequency x SCL_PERIOD)/20) - 1)

TPR = (System Clock frequency)/ (20 x I2C clock) - 1

With System clock frequency of 20MHz and with I2C clock is 333 KHz, we get TPR (Timer period) = 2.

TPR value to generate Standard, Fast and Fast mode plus SCL frequencies is given in below table: Table:

TPR Values for I2C modes

Table: TPR Values for I^2C modes

System Clock	Timer Period	Standard Mode	Timer Period	Fast Mode	Timer Period	Fast Mode Plus
4 MHz	0x01	100 Kbps	-	-	-	-
6 MHz	0x02	100 Kbps	-	-	-	-
12.5 MHz	0x06	89 Kbps	0x01	312 Kbps	-	-
16.7 MHz	0x08	93 Kbps	0x02	278 Kbps	-	-
20 MHz	0x09	100 Kbps	0x02	333 Kbps	-	-
25 MHz	0x0C	96.2 Kbps	0x03	312 Kbps	-	-
33 MHz	0x10	97.1 Kbps	0x04	330 Kbps	-	-
40 MHz	0x13	100 Kbps	0x04	400 Kbps	0x01	1000 Kbps
50 MHz	0x18	100 Kbps	0x06	357 Kbps	0x02	833 Kbps
80 MHz	0x27	100 Kbps	0x09	400 Kbps	0x03	1000 Kbps

The HS bit in the I2CMTPR register needs to be set for the TPR value to be used in High-Speed mode.

Table: TPR Values for High-Speed Mode

System Clock	Timer Period	Transmission Mode
40 MHz	0x01	3.33 Mbps
50 MHz	0x02	2.77 Mbps
80 MHz	0x03	3.33 Mbps

I2CMCR (I2C Master Configuration register) is used to configure microcontroller as master or slave. The description of I2CMCR is below:

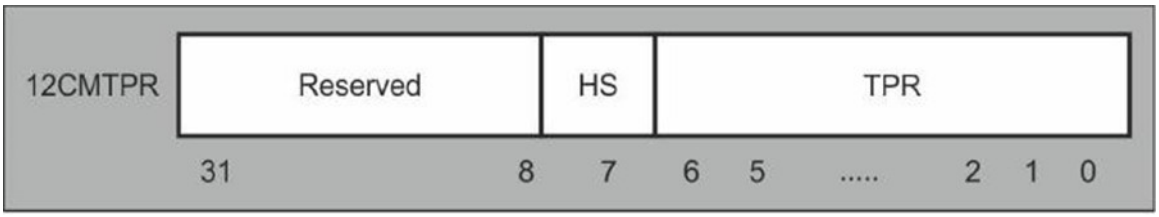


Figure: I2C Master Configuration Register

Name	Function	Description
LPBK	I2C Loopback	0: Normal Operation; 1: Loopback
MFE	I2C Master function Enable	1: Enable Master function; 0: Disable master
SFE	I2C Slave function Enable	1: Enable Slave function; 0: Disable
GFE	I2C Glitch Filter Enable	1: Enable Glitch filter; 0: Disable

Table: I2CMCR Register Description

Slave Address:

In a master device, the slave address is stored in I2CMSA. Addresses in I2C communication is 7-bits. I2CMSA stores D7 to D1 bits and LSB of D0 indicate master is receiver of transmitter.

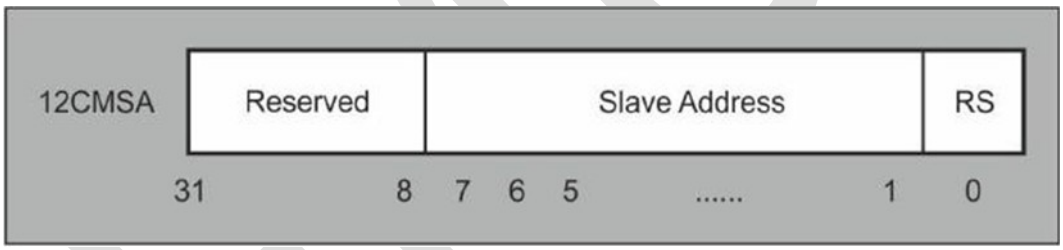


Figure: I2C Master Slave Address Register

Data Register:

In transmit mode, a byte of data will be placed in I2CMDR (I2C Master Data Register) for transmission.

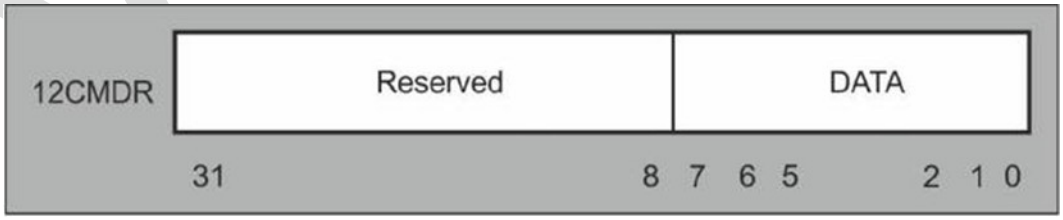


Figure: I2C Master Data Register

Control and Status Flag Register:

The I2CMCS (I2C Master Control/Status) register is programmed for both control and status. I2CMCS register configures the I2C controller operation. The status whether a byte has been transmitted. That is, transmission buffer is empty and ready to transmit the next byte. After writing a data into I2C Data register and the slave address into I2C Master Slave address register, we can configure I2CMCS register for the I2C to start a data transmission from Master to slave device. Writing 0x07 to I2CMCS register has all the three of STOP = 1, RUN = 1, and START = 1 in it. To check the status of transmission, we poll the BUSBSY bit of I2CMCS register. BUSBSY bit goes low after transmission complete. Program should also check the ERROR bit to confirm that no error has occurred during transmission. For any error in transmission, detected by transmitter or raised by slave, the ADRACK and DATACK will be set. The bit ARBLST should be polled, to confirm transmitter has got access to bus and not lost arbitration.

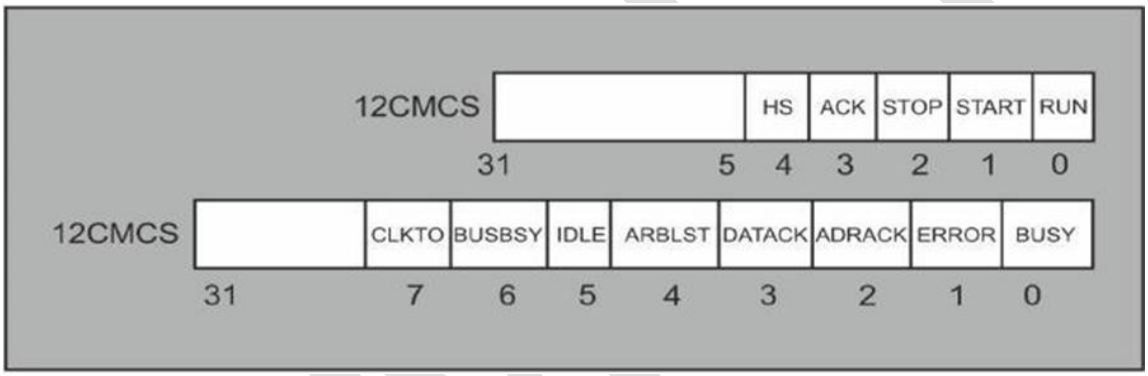


Figure: I2C Master Control/Status Register

Bit	Function	Description
RUN	I2C Master Enable	1: Enables the master, so that transmission can be started.
START	Generate START	1: Enabled to generate START condition
STOP	Generate STOP	1: Enabled to generate STOP condition
ACK	Data Acknowledge Enable	1: To generate auto ACK condition
HS	High Speed Enable	1: High Speed operation enabled
BUSY	I2C Busy	0: I2C controller is idle
ERROR	Error in network	0: No Error detected in network
ADRACK	Acknowledge address	0: Transmitted address acknowledged
DATRACK	Acknowledge Data	0: Transmitted data acknowledged
ARBLST	Arbitration lost	0: IIC controller won arbitration
IDLE	I2C Idle	0: Bus is not idle
BUSBSY	Bus Busy	0: Bus is idle
CLKTO	Clock Timeout Error	0: No clock timeout error

Table: I2C MCS Register Description

Configuring GPIO for I2C Network:

- GPIO pins are configured for I2C as follows:
- Enable the clock to GPIO pins by using system control register RCGCGPIO.
- Set the GPIO AFSEL (GPIO alternate function) for I2C pins.
- Enable digital pins in the GPIODEN register.
- I2C signals are assigned to specific pins using GPIOCTL register.

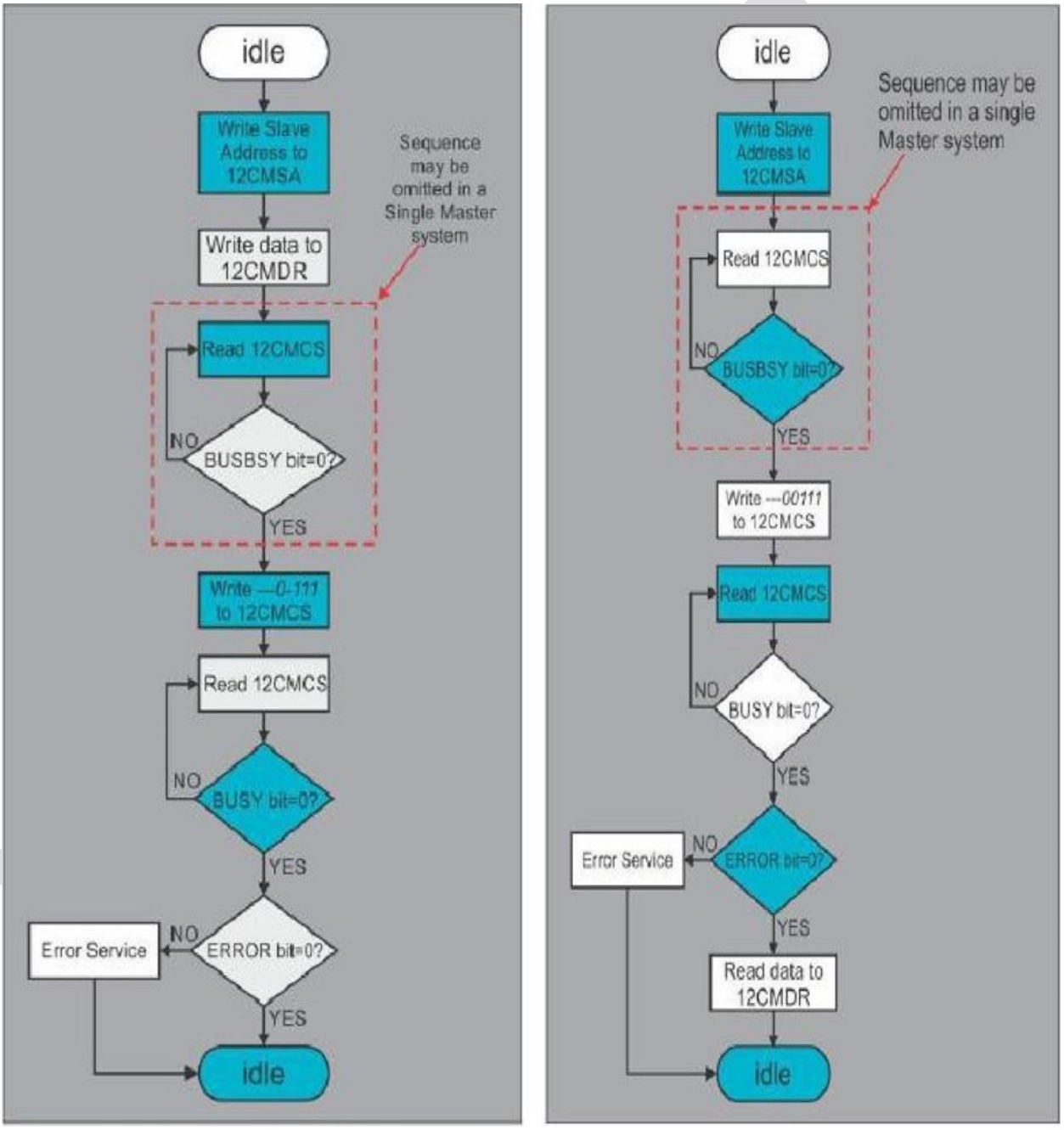


Figure: Data transmission using (a) Master Single Transmit, (b) Single Master Receive

Implementing and Programming SPI:

Serial peripheral interface (SPI) is a serial communication interface originally designed by Motorola in late eighties. SPI and I2C came into existence almost at the same time. Most of the modern day microcontrollers will support SPI protocol. Both SPI and I2C offer good support for communication with low-speed devices, but SPI is better suited to applications in which devices transfer data streams. Some devices use the full-duplex mode to implement an efficient, swift data stream for applications such as digital audio, digital signal processing, or telecommunications channels, but most off-the-shelf chips stick to half-duplex request/response protocols.

SPI is used to talk to a variety of peripherals, such a

- Sensors: temperature, pressure, ADC, touchscreens, video game controllers
- Control devices: audio codecs, digital potentiometers, DAC
- Camera lenses: Canon EF lens mount
- Memory: flash and EEPROM
- Real-time clocks
- LCD, sometimes even for managing image data
- Any MMC or SD card

Description: SPI is a synchronous serial communication protocol like I2C, where master generates clock and data transfer between master and slave happens with respect to clock. Both master and slave devices will have shift registers connected to input (MISO for master and MOSI for slave) and output (MOSI for master and MISO for slave) as shown in figure.

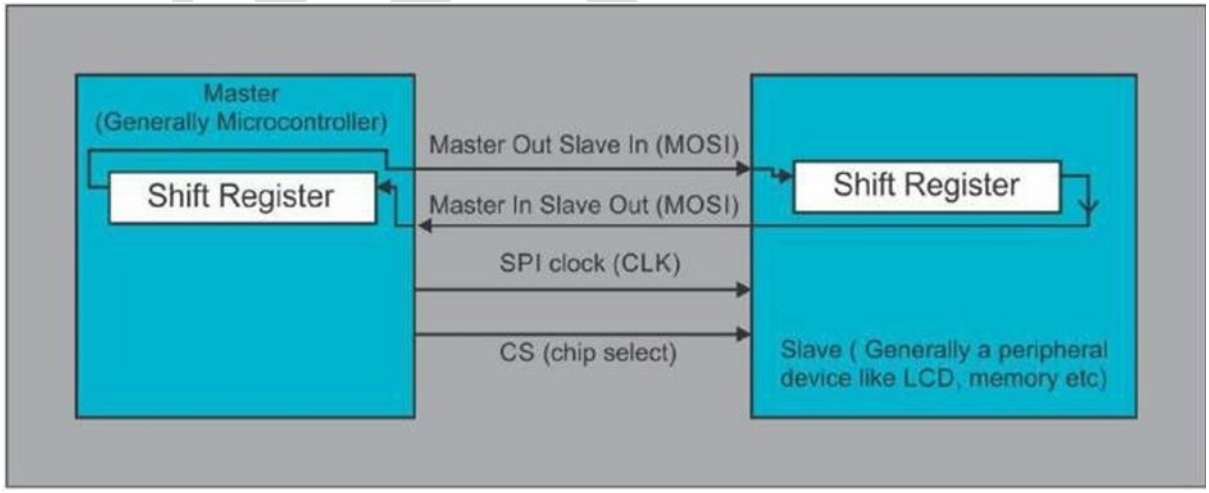


Figure: Serial Peripheral Interface

Communication between the devices will start after CS (chip select) pin will go low. (CS is an active low pin). In SPI, the 8-bit shift registers are used. After passing of 8 clock pulses, the contents of two shift registers are interchanged. SPI is full duplex communication.

In SPI protocol both master and slaves use the same clock for communication When CPOL= 0 the idle value of the clock is zero while at CPOL=1 the idle value of the clock is one.CPHA=0 means sample data on the leading (first) clock edge, while CPHA=1 means sample data on the trailing (second) clock edge. The idle value of the clock is zero the leading clock edge is a positive edge but if the idle value of the clock is one, the leading clock edge is a negative edge.

In SPI protocol both master and slaves use the same clock for communication When CPOL= 0 the idle value of the clock is zero while at CPOL=1 the idle value of the clock is one.CPHA=0 means sample data on the leading (first) clock edge, while CPHA=1 means sample data on the trailing (second) clock edge. The idle value of the clock is zero the leading clock edge is a positive edge but if the idle value of the clock is one, the leading clock edge is a negative edge.

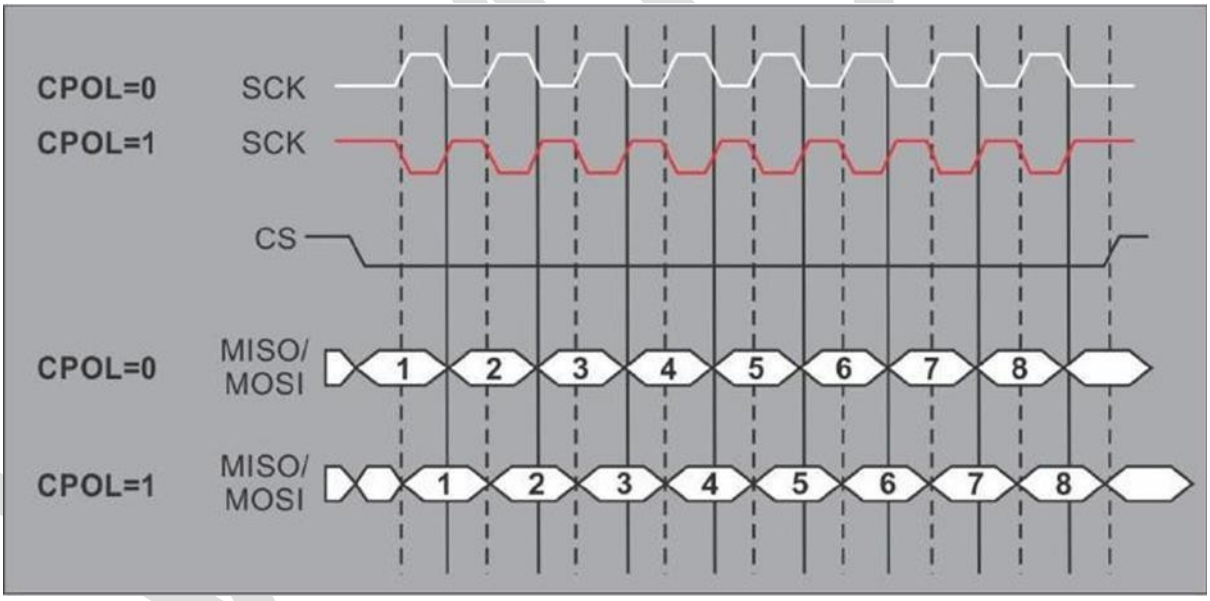


Figure: SPI Timing Diagram

SPI Mode	CPOL	CPHA	Data Read and Change time
0	0	0	Read on positive (rising) edge, changed on falling edge
1	0	1	Read on negative (falling) edge, changed on rising edge
2	1	0	Read on negative (falling) edge, changed on rising edge
3	1	1	Read on positive (rising) edge, changed on falling edge

Table: SPI Modes

SPI in Tiva Microcontroller:

The TM4C123GH6PM microcontroller includes four Synchronous Serial Interface (SSI) modules. Each SSI module is a master or slave interface for synchronous serial communication with peripheral devices that have Freescale SPI, MICROWIRE, or Texas Instruments synchronous serial interfaces.

The TM4C123GH6PM SSI modules have the following features:

- Programmable interface operation for Freescale SPI, MICROWIRE, or Texas Instruments synchronous serial interfaces
- Master or slave operation
- Programmable clock bit rate and prescaler
- Separate transmit and receive FIFOs, each 16 bits wide and 8 locations deep
- Programmable data frame size from 4 to 16 bits
- Internal loopback test mode for diagnostic/debug testing
- Standard FIFO-based interrupts and End-of-Transmission interrupt
- Efficient transfers using Micro Direct Memory Access Controller (μDMA)
- Separate channels for transmit and receive
- Receive single request asserted when data is in the FIFO; burst request asserted when FIFO contains 4 entries
- Transmit single request asserted when there is space in the FIFO; burst request asserted
- When four or more entries are available to be written in the FIFO.

Most SSI signals are alternate functions for some GPIO signals and default to be GPIO signals at reset. The exceptions to this rule are the SSI0Clk, SSI0Fss, SSI0Rx, and SSI0Tx pins, which default to the SSI function. The AFSEL bit in the GPIO Alternate Function Select (GPIOAFSEL) register should be set to choose the SSI function.

Each data frame is between 4 and 16 bits long depending on the size of data programmed and is transmitted starting with the MSB. There are three basic frame types that can be selected by programming the FRF bit in the SSICR0 register:

- Texas Instruments synchronous serial
- Freescale SPI
- Microwire

For all three formats, the serial clock (SSInClk) is held inactive while the SSI is idle, and SSInClk transitions at the programmed frequency only during active transmission or reception of data. The idle

state of SSInClk is utilized to provide a receive timeout indication that occurs when the receive FIFO still contains data after a timeout period.

For Freescale SPI and MICROWIRE frame formats, the serial frame (SSInFss) pin is active Low, and is asserted (pulled down) during the entire transmission of the frame.

We focus on the SPI features of SSI module. This microcontroller supports four SSI modules. The SSI modules are located at the following base addresses:

Table: SPI Modules base address

Module	SSI0	SSI1	SSI2	SSI3
Base Address	0x40008000	0x40009000	0x4000A000	0x4000B000

Clock to SSI: RCGCSSI register is used to enable the clock to SSI modules. We need to write RCGSSI = 0x0F to enable the clock to all SSI modules.

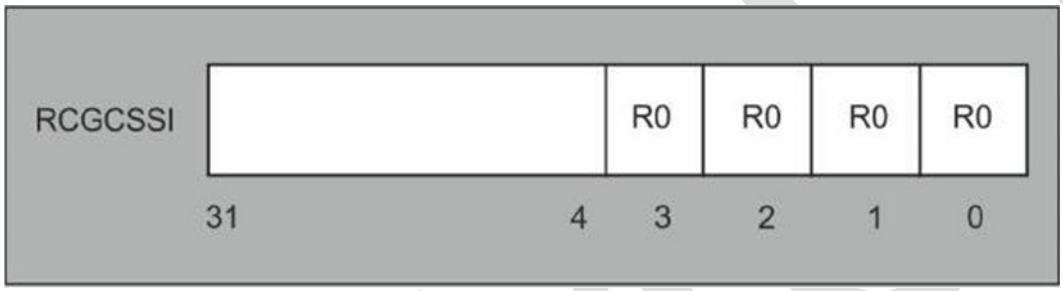


Figure: Synchronous Serial Interface Run Mode Clock Gating Control CRCG (SSI) Register

Configuring the SSI:

SSICR0 (SSI control register 0) is used to configure the SSI. The generic SPI is used to transfer the byte size of data, the SSI in Tiva microcontroller allows transfer of data between 4 bits to 16bits.

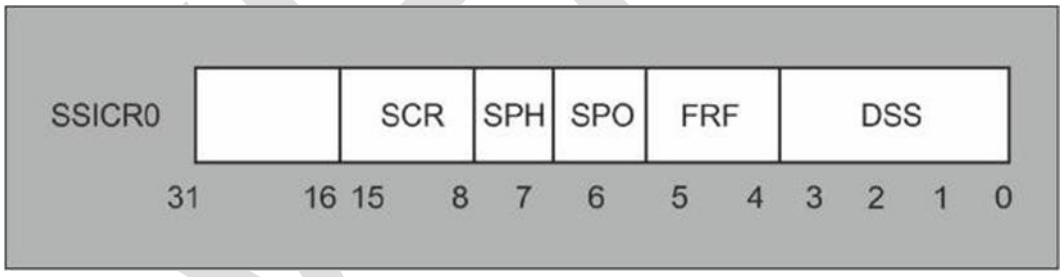


Figure: SSI Control O Register

Bits	Name	Function	Description
0-3	DSS	SSI Data Size select	0x3: for 4-bit; 0x7: for 8-bit; 0xF: 16-bit data
4-5	FRF	SSI frame format select	0 for SPI, 1 for TI, and 2 for MICROWIRE frame format
6	SPO	SSI serial clock polarity	Clock Polarity
7	SPH	SSI serial clock phase	Clock Phase
8-15	SCR	SSI serial clock rate	BR = SysClk/(CPSDVSR * (1+SCR))

Table: SSICRO Register Description

Bit Rate:

SSI module clock source can be either from System Clock or PIOSC (Precision Internal Oscillator). The selected frequency is fed to pre-scaler before it is used by the Bit Rate circuitry. The CPSDVSR (CPS Divisor) value comes from the pre-scaler divisor register. The lower 8 bits of SSICPSR (SSI Clock Prescale) register are used to divide the CPU clock before it is fed to the Bit Rate circuitry. Only even values can be used for the pre-scaler since the D0 must be 0. For the pre-scaler register, the lowest value is 2 and the highest is 254.

The SSICR0 (SSI Control register 0) allows the Bit Rate selection among other things. The output of clock pre-scaler circuitry is divided by 1 + SCR and then used as the SSI baud rate clock. The value of SCR can be from 0 to 255. The below formula is used to calculate the bit rate.

Bit Rate (BR): BR=SysClk/(CPSDVSR × (1 + SCR))

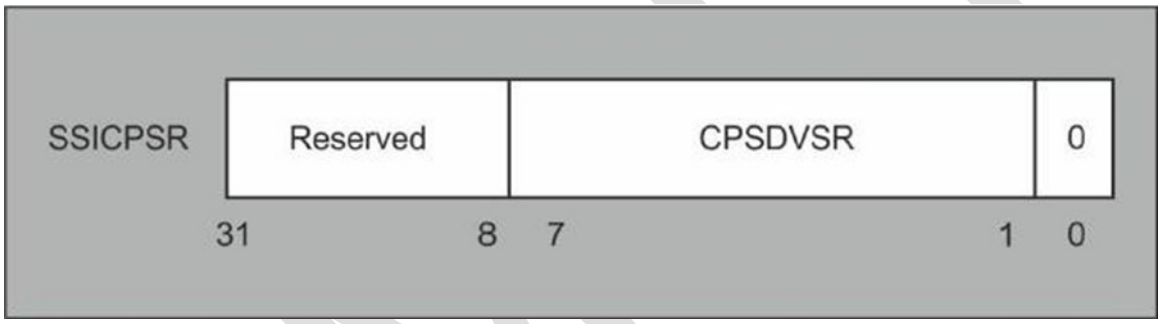


Figure: SSI Clock Prescaler Register

Example:

For a Bit Rate=50 KHz and SCR=03 in SSICR0 register.

The pre-scaler register value for a given system clock frequency of 16MHz, the BR can be calculated using above formula as:

BR = SysClk / (CPSDVSR × (1 + SCR)) 50 KHz = 16

MHz / (X × (1 + 3).

The pre-scaler value is 0x50 in Hex.

SPI module can act like slave or a master. The value in a MS bit in SSI control register 1 (SSICR1) decide the microcontroller as master or slave. SSE bit in the SSICR1 register is used to enable/ disable the SPI.

Figure: SSI Control 1 Register

Data Register: The SSIDR is used for both as transmitter and receiver buffer. In SPI handling 8-bit data, will be placed into the lower 8-bits of the register and the rest of the register are unused. In the receive mode, the lower 8-bit holds the received data.

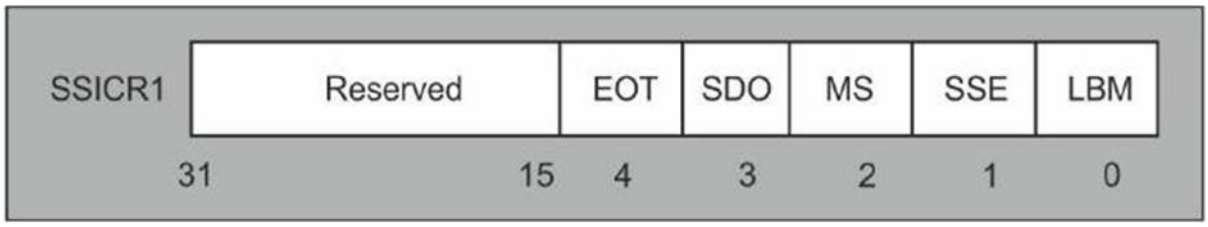


Figure: SSI Data Register

Status Flag Register: SSISR is used to monitor transmitter/receiver buffer is empty.

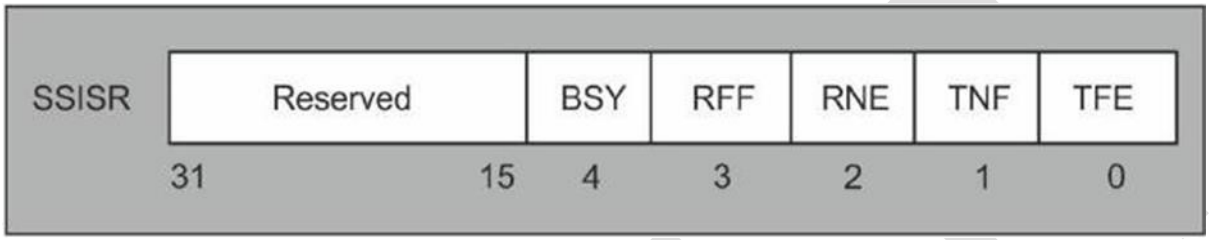


Figure: SSI Status Register Table: SSI Status Register Description

Name	Function	Description
TFE	Transmit FIFO empty	1: Transmit FIFO is empty
TNF	Transmit FIFO full	1: Transmit FIFO is not empty
RNE	Receive FIFO not empty	1: Receive FIFO is not empty
RFF	Receive FIFO full	1: Receive FIFO is full
BSY	SSI Busy Bit	1: transmission or reception is under progress

SPI data Transmission:

- To perform SPI data transmission, follow the steps given below:
- Enable the clock to SPI module in system control register RCGCSSI.
- Before initialization, disable the SSI via bit 1 of SSICR1 register.
- Set the Bit Rate with the SSICPSR prescaler and SSICR0 control registers.
- Select the SPI mode, phase, polarity, and data width in SSICR0 control register.
- Set the master mode in SSISCR1 register.
- Enable SSI using SSICR1 register.
- Assert slave select signal.
- Wait until the TNF flag in SSISR goes high, then load a byte of data into SSIDR.
- Wait until transmit is complete that is, transmit FIFO empty and SSI not busy.
- De-assert the slave signal

NVIC interrupt for SSI:

Interrupt handler can be used for transmission and reception of data. By enabling the interrupt in SSIIM (SSI Interrupt mask) register, NVIC interrupt controller will enable interrupts from SSI and execute the corresponding interrupt service routine. All SSI interrupts are masked upon reset.

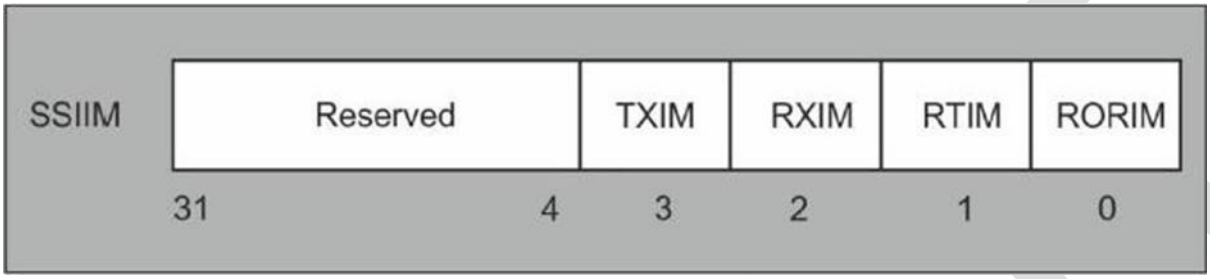


Figure: SSI Interrupt Mask Register Table: SSI Interrupt Mask Register Description

Bit	Function	Description
RORIM	Receive overrun interrupt mask	0: Receive FIFO overrun interrupt is masked; 1: not masked
RTIM	Receive Time out interrupt mask	0: Receive FIFO time out interrupt is masked; 1: not masked
RXIM	Receive FIFO interrupt mask	0:Receive FIFO interrupt is masked ; 1: not masked
TXIM	Transmit FIFO interrupt mask	0: Transmit FIFO interrupt is masked; 1: not masked

/* Program for Tiva Microcontroller to use SSI1 (SPI) to transmit A to Z characters*/

```
#include "TM4C123GH6PM.h"
void init_SSI1(void);
void SSI1Write(unsigned char data);
int main(void)
{
unsigned char i;
init_SSI1(); for(;;)
{
for (i = 'A'; i <= 'Z'; i++)
{
SSI1Write(i); /* write a character */
}
void SSI1Write(unsigned char data)
{
GPIOF->DATA &= ~0x04; /* assert SS low */
while((SSI1->SR & 2) == 0); /* wait until FIFO not full */ while(SSI1->SR & 0x10); /* wait until
transmit complete */
GPIOF->DATA |= 0x04;          /* keep SS idle high */
void init_SSI1(void)
```

```
{
SYSCTL->RCGCSSI |= 2;      /* enable clock to SSI1 */
/* configure PORTD 3, 1 for SSI1 clock and Tx */
GPIOD->DEN |= 0x09;        /* and make them digital */
GPIOD->AFSEL |= 0x09;      /* enable alternate function */
GPIOD->PCTL &= ~0x0000F00F;    /* assign pins to SSI1 */
GPIOD->PCTL |= 0x00002002;     /* assign pins to SSI1 */
/* configure PORTF 2 for slave select */
GPIOF->DEN |= 0x04; /* make the pin digital */
GPIOF->DIR |= 0x04;        /* make the pin output */
GPIOF->DATA |= 0x04; /* keep SS idle high */
/* SPI Master, POL = 0, PHA = 0, clock = 4 MHz, 16 bit data */
SSI1->CR1 = 0;         /* disable SSI and make it master */
SSI1->CC = 0;          /* use system clock */
SSI1->CPSR = 2;        /* prescaler divided by 2 */
SSI1->CR1 |= 2;        /* enable SSI1 */
}

void SystemInit(void)
{
SCB->CPACR |= 0x00f00000;
}
```

Case Study: Tiva based embedded system application using the interface protocols for communication with external devices "Sensor Hub BoosterPack"

Weather monitoring balloon using sensor hub:

Microcontrollers or System on Chips (SoC) are important components of modern day electronic systems. But microcontrollers alone can't make systems. Microcontrollers need to communicate with other devices on the same PCB or with devices of other PCB from the same system, to achieve the system functionality in applications such as digital audio, digital signal processing, or telecommunications channels. Microcontroller is an intelligent component on the system, which runs the software program to take decisions and control the other components of the system.

Consider a system shown in Fig below, which highlight the various components of weather monitoring systems. The weather monitoring systems usually monitor atmospheric properties such as humidity, temperature, pressure etc. The various parameters are gathered using sensors that are interfaced with microcontroller using popular serial interfaces such as SPI, I2C, and UART etc. The same system can be used for upper atmospheric data gathering missions by incorporating Wi-Fi and satellite connectivity with the ground station. Therefore, communication between microcontroller and other devices on the system is very important to achieve intended functionality.

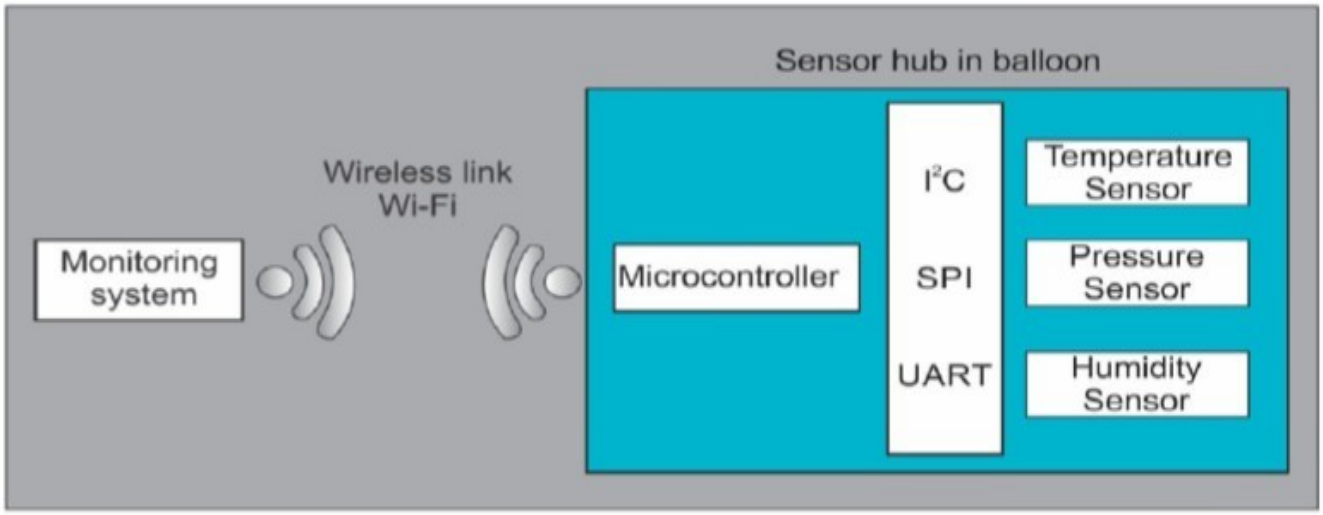
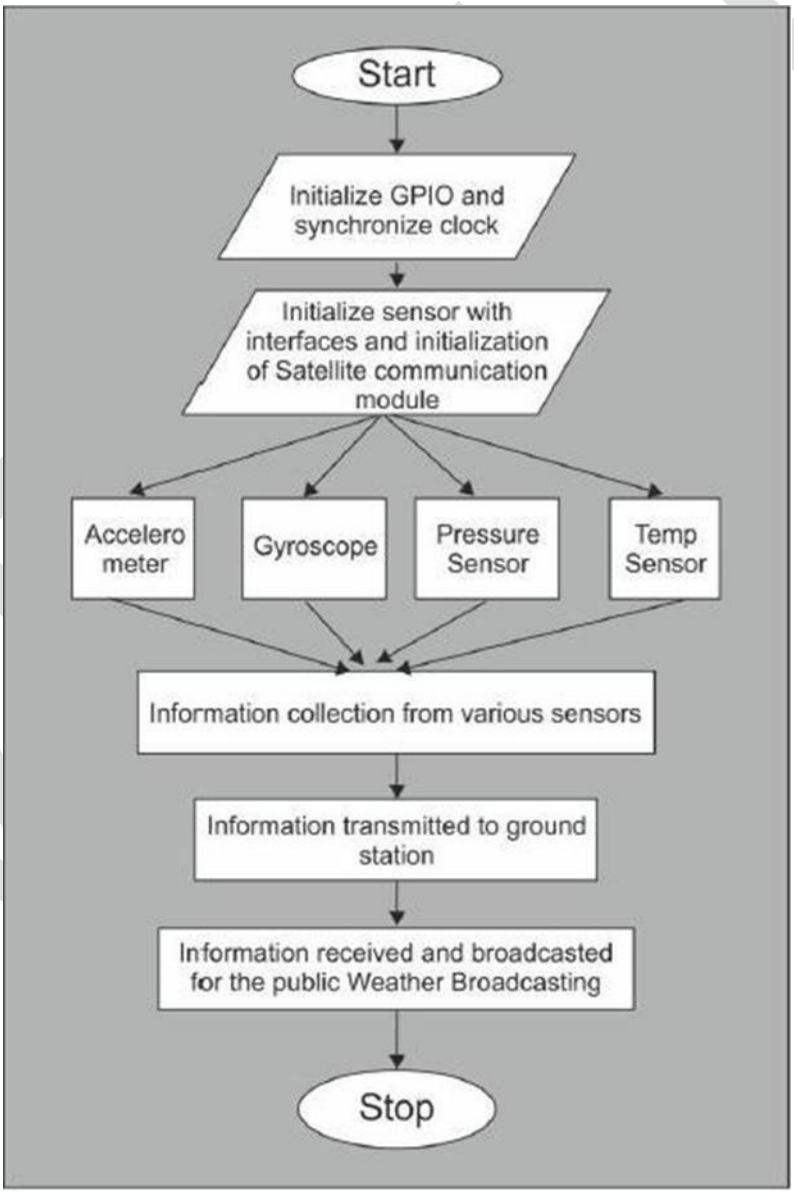


Fig: Weather monitoring using sensor hub balloon



Flowchart: Interfacing TIVA with Sensor Hub Booster Pack

Weather broadcasting system require some smart technique to monitor the weather conditions of different places. It is useful for the meteorological department for the detection of the environmental condition with the help of a balloon. In this case study we are using four sensors Accelerometer, gyroscope, temperature sensor and pressure sensor. The Tiva booster pack with various sensors is mounted on the balloon and accelerometer used for the detection of acceleration of the balloon and gyro scope is used for the position detection of the balloon and pressure and temperature sensor senses pressure and temperature of the environment respectively. These all gathered information sent to the ground station with the help of satellite communication system installed at the balloon and the meteorological department's ground station. The collected information is used for the public weather broadcasting.

Embedded Networking Fundamentals:

Microcontrollers are used to design intelligent embedded systems such as smartphones, netbooks, digital TVs, mp3 players, smart-watches, smart-sensors, etc. These smart *things* can be connected together to form an *embedded network* that imparts intelligence to bigger *things* like homes, buildings, fields, forests and cities. An embedded network of smart things like automatic home appliances, lights, door sensors, CCTV cameras, refrigerators, etc. can provide smart-home users with more convenient and high-quality living experience.

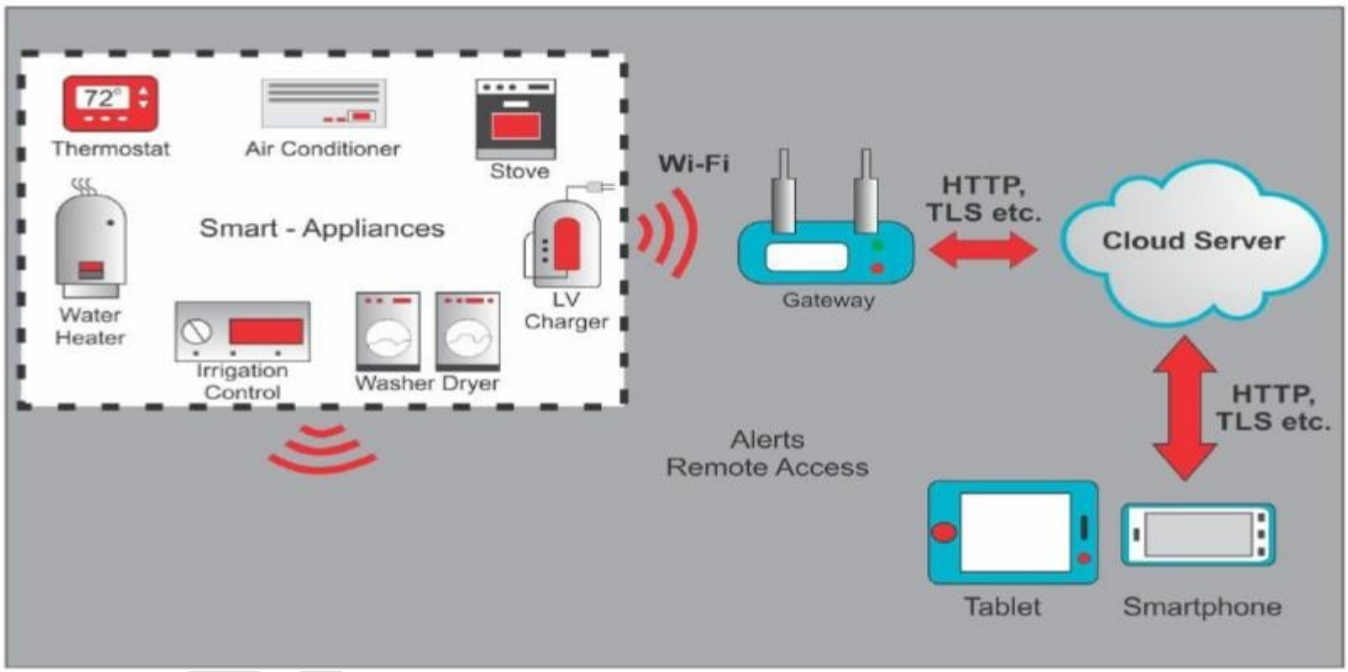
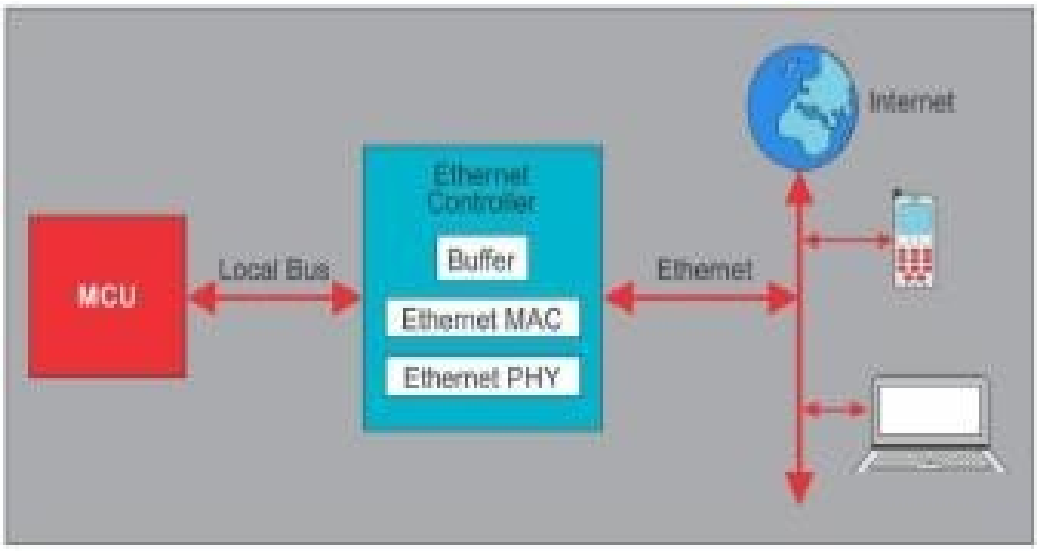


Fig: Embedded Network

Embedded Network and Ethernet:

Ethernet is a local area network (LAN) technology that is widely used to connect computers using wires or cables. Ethernet is similar to Wi-Fi technology, but with a different medium. Ethernet is wired, and Wi-Fi is wireless. Ethernet is based on standards (IEEE 802.3) that ensure reliability of network connections and data transmission and interoperability. Ethernet networks are scalable from the simplest to most complex networks or up to 2^48 network nodes.

Once equipment is connected to an Ethernet network, it can be monitored or controlled through the Internet removing any distance barrier that may have inhibited remote communication previously. Based on its ease of use, low cost, high bandwidth, stability, security and compatibility across devices, Ethernet has become the de facto standard of network access for 32, 16 and even 8-bit microcontrollers.
From the below block diagram, MCU and Ethernet controller can make any device connected to the World Wide Web. Thus, it helps in monitor, control or access devices over internet.



User Datagram Protocol (UDP): UDP is a Internet protocol suite and it uses a simple connectionless transmission model with a minimum of protocol mechanism. It has no handshaking dialogues, and thus exposes the user's program to any unreliability of the underlying network protocol. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram.

UDP is smaller, faster, and more suitable for fast embedded network communication. UDP allows embedded devices to use the Internet Protocol (IP) to send data to, and receive data from remote network nodes. The remote nodes can be on the same local network, or on remote networks that are accessible over the Internet.

UDP data is sent in connectionless packets. That means the UDP and IP protocols do not guarantee delivery, the node sending data cannot assume the intended recipient received the data just because the data was sent onto the network. So UDP packets requires acknowledgement of receipt must be manually acknowledged by the receiving application.

Network Addressing A network address is an identifier for a node or network interface of a telecommunications network. Different nodes on the same network are identified by their IP address. Each single network node can run multiple applications that use the same network interface, and therefore use the same IP address. Different applications running on the same network node, and therefore at the same IP address, are identified by their port number. The source and destination address of each UDP packet is therefore a combination of an IP address and a port number.

Sockets and Binding Sockets are conceptual end points of a network communication. Sockets can send and receive data. Each socket needs a unique address. As already stated, an address is the combination of an IP address and a port number. When a socket is created it assumes the IP address of the network node that created it. If a socket has an IP address but not a port number, it is said to be 'unbound'. An unbound

socket cannot receive data because it does not have a complete address. When a socket has both an IP address and a port number it is said to be 'bound to a port', or 'bound to an address'. A bound socket can receive data because it has a complete address. The process of allocating a port number to a socket is called 'binding'.

Client and Server Servers are applications that wait for and then reply to incoming requests. Clients are applications that send requests to servers. In this context, the requests and replies go over the network and clients need to locate servers. Servers do not need to know the client's address in advance, they just send their replies to the address from which the client's request originated and therefore clients can bind to nearly any port number.

The sequence diagram below shows a socket being created and bound on both an echo client and an echo server, and then a single echo transaction between the client and the server. Echo servers simply echo back the data sent to them by clients.

Static IP Address: An Internet Protocol address is an address used in order to uniquely identify a device on an IP network. The address is made up of 32 binary bits, which can be divisible into a network portion and host portion with the help of a subnet mask. If the IP address is 'static' then it is pre-assigned and never changed. Static IP addresses are useful during application development.

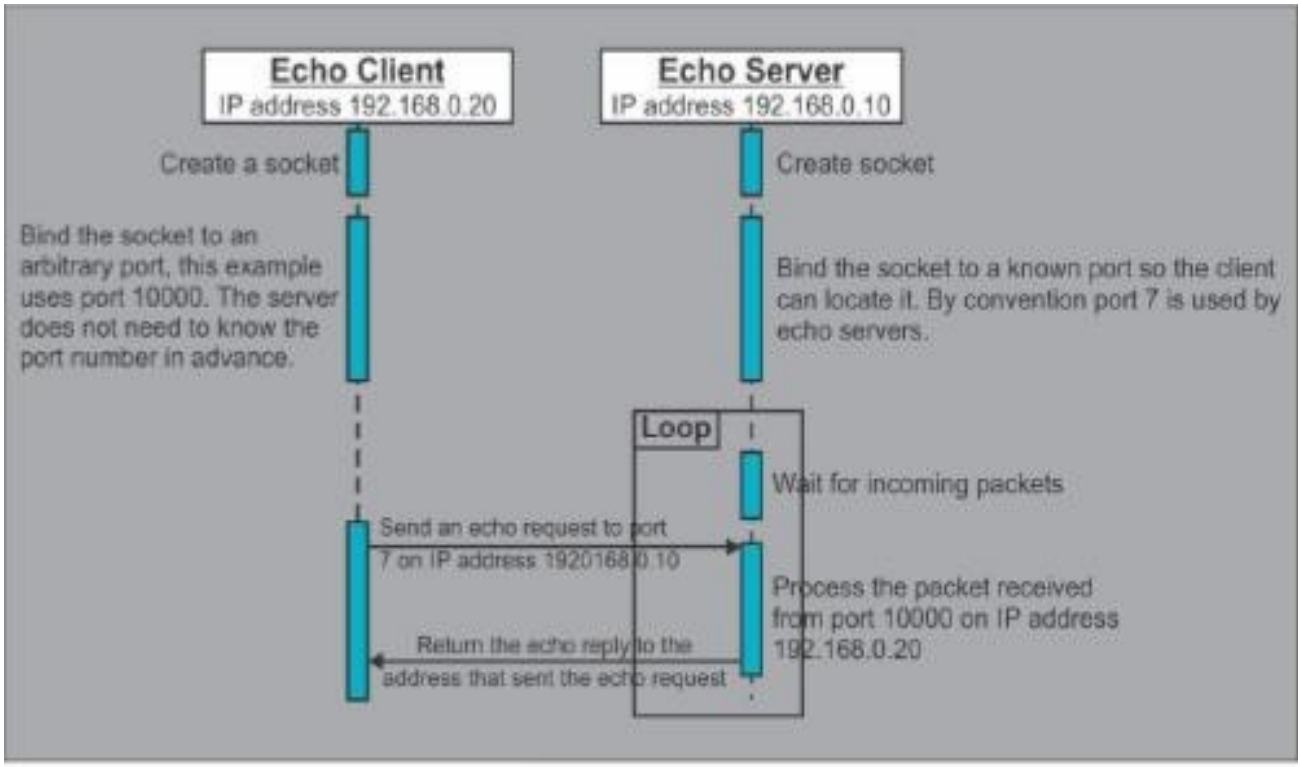


Fig . Transaction between an echo client and an echo server

Dynamic Host Configuration Protocol (DHCP): The IP address can be pre-assigned but it is impractical for product deployment and cannot be pre-assigned to products without prior knowledge of the network environment in which the products will be deployed. And also there is no prior knowledge of how many nodes will exist on the network, or indeed how many of the possible total number of nodes will be active at any one time.

DHCP provides an alternative to static IP address assignment. DHCP servers exist on local networks to dynamically allocate IP addresses to nodes on the same network. When a network enabled product boots up it contacts the DHCP server to request its IP address, removing the need for each node to be statically configured.

Sub-netting / Netmask: A subnetwork, or subnet, is a logical, visible subdivision of an IP network. The practice of dividing a network into two or more networks is called sub-netting. Sub-netting is a way of determining whether a destination IP address exists on the local network or a remote network. Like the IP address, the subnet mask can be configured either statically or dynamically from a DHCP server. If a destination IP address bitwise ANDed with the subnet mask matches the local IP address bitwise ANDed with the subnet mask then the two IP addresses exist on the same network.

Gateways and Routers: A gateway acts as a conversion from one protocol to another. A router works by looking at the IP address in the data packet and decides if it is for internal use or if the packet should move outside the network. If a destination IP address bitwise ANDed with the subnet mask does not match the local IP address bitwise ANDed with the subnet mask then the two IP addresses do not exist on the same network. In this case the packet being sent to the destination address cannot be sent directly, and must instead be sent to a gateway for intelligent inter-network routing.

Domain Name System (DNS): The Domain Name System (DNS) is a hierarchical distributed naming system for computers, services, or any resource connected to the Internet or a private network. It associates various information with domain names assigned to each of the participating entities. Most prominently, it translates domain names, which can be easily memorized by humans, to the numerical IP addresses needed for the purpose of computer services and devices worldwide. The Domain Name System is an essential component of the functionality of most Internet services because it is the Internet's primary directory service.

For example, entering "ping www.freertos.org" in the command console of a desktop computer will show a ping request being sent to the IP address 195.8.66.1 (today anyway) a DNS server resolved the string "www.freertos.org" to the IP address 195.8.66.1.

Address Resolution Protocol (ARP): The Address Resolution Protocol (ARP) is a telecommunication protocol used for resolution of network layer addresses into link layer addresses, a critical function in multiple-access networks. Assuming a conventional wired network is used, UDP messages are sent in Ethernet frames. UDP messages are sent between IP addresses, but Ethernet frames are sent between MAC (hardware) addresses. Therefore, the MAC address of the destination IP address must be known before an Ethernet frame can be created. The Address Resolution Protocol (ARP) is used to obtain MAC address information.

TCP/IP Introduction IoT overview and architecture:

Communication between computers or embedded devices in a network involves exchanging useful messages over a medium like air, telephone line, Ethernet, etc. Each device must have an address or ID using which, it can be uniquely identified in the network. The devices must follow some rules while communicating with each other, so that messages are exchanged in a proper manner. IP (Internet Protocol) provides a set of unique addresses to the devices, whereas TCP (Transport Control Protocol) describes a set of rules to be followed to exchange messages in a proper way.

In the smart home application shown in fig. below, TCP/IP protocol can be used over Ethernet to provide Internet connectivity to the outside world. This will enable the user to monitor or control the smart home functions from anywhere in the world using a PC, laptop or a smartphone.

Internet Protocol version 6 (IPv6):

IPv6 is the most recent version of the Internet Protocol (IP), the communications protocol that provides an identification and location system for computers on networks and routes traffic across the Internet.

IPv6 advantages for IoT:

Adoption: The Internet Protocol is a must and a requirement for any Internet connectivity. It is the addressing scheme for any data transfer on the web.

Scalability: IPv6 offers a highly scalable address scheme. The present scheme of Internet Governance provides at most 2 x 1019 unique, globally routable, addresses.

Solving the NAT barrier: The Network Address Translation (NAT). It enables several users and devices to share the same public IP address. The NAT users are borrowing and sharing IP addresses with others.

Multi-Stakeholder Support: IPv6 provides for end devices to have multiple addresses. Multiple stakeholders can deploy their own applications, sharing a common sensor/actuation infrastructure, without impacting the technical operation or governance of the Internet.

Internet of Things (IOT):

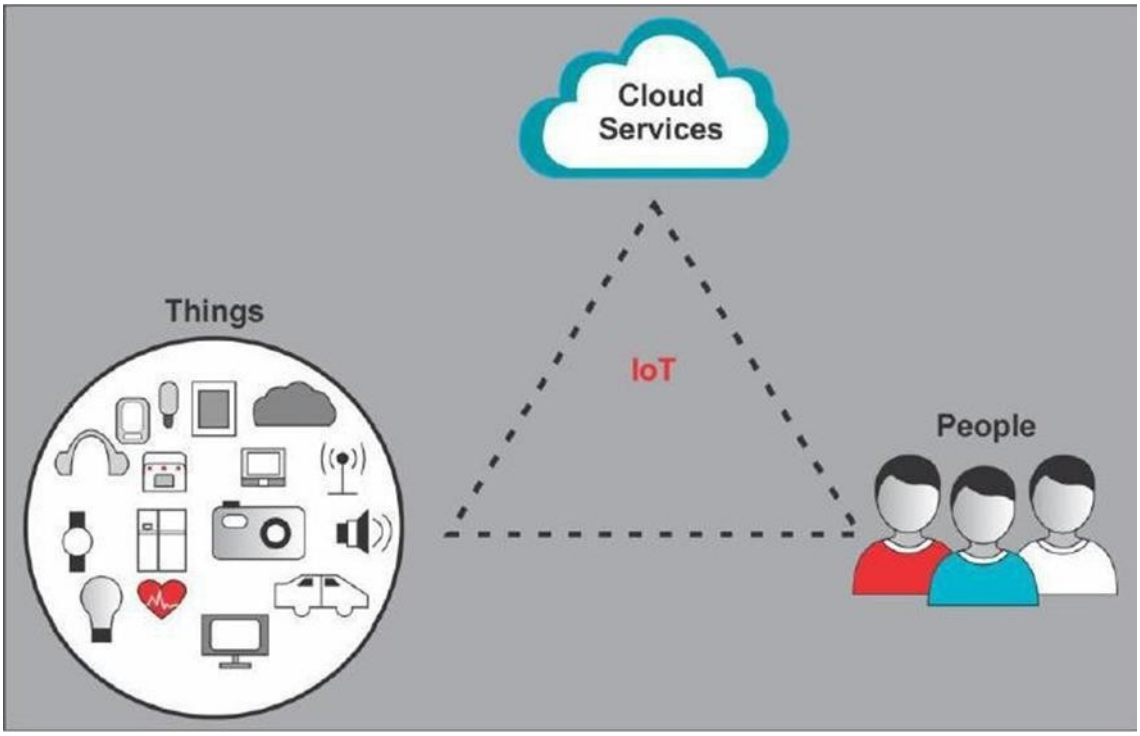
Klevin Ashton introduced the term “Internet of Things” (IOT), to the world of technology in 1999. All the real-life things (including objects, people and animals) are connected to internet, and can transfer data over it preferably to a cloud. This data can then be used by businesses and the people, to create a world of new possibilities and to benefit from it. Fig. below shows the three main components of IoT i.e. things, data (cloud) and the people. For e.g. a smart refrigerator can sense the quantity of items inside it, and then automatically generate a shopping list to be ordered on-line. This list is put by the smart refrigerator on the cloud, where the best deals are offered for online purchase.

IOT is considered as a scenario of accessing any information from anywhere and accessible to everyone. This is described as follows:

Anything: Eventually, any device, appliance or entity will be seamlessly connected to the Internet. Connectivity will not be the main feature of the device, but will extend the device capabilities.

Anywhere: Any conceived wireless connectivity framework should be abstract enough to run from any location.

Anyone: IoT accessible to anyone. Anyone will be able to connect their product to the Internet, and also customize it to their personal preferences.



Main components of IoT

Applications of IOT:

Wired and wireless connectivity technologies
Microcontrollers
Processors
Sensors
Analog signal chain and power solutions
High-performance home
Industrial applications
Automotive applications
Battery-powered wearable
Portable electronics
Energy-harvested wireless sensor nodes.

In **automotive appliances**, IoT is mainly used for infotainment purposes such as connecting between the phones and the speakers of the car, activating the engine through voice control etc.

In **the personal area network** we encounter wearable devices for entertainment and location tracking. For example, it can be a Bluetooth headset or a GPS tracker. These devices facilitate the user to help enhance their health and wellness, and to gather information around the user.

At **home** we are surrounded with an ever-growing number of appliances, multimedia devices and other consumer gadgets.

In home automation systems, IoT applications include monitoring and controlling the devices inside a home in an intelligent way. They include lighting and temperature control among the connected appliances for effective use of energy.

While **on-the-go**, we use private or public transportation vehicles and infrastructure to improve our mobility time utilization.

In **industries**, sensors might be introduced for production efficiency, maintenance and failure management.

And at a **metropolitan** level smart building management system include smart cities equipped with smart city lights, residential e-meters, surveillance cameras for traffic control, pipeline leak detection etc.

Healthcare IoT applications include remote monitoring of patients for example heart rate, blood pressure level etc.

Architecture of IOT:

The IoT players: We need to get a wider view of the IoT playground. To do that, the key players must first be identified. We classify the players into three clusters: users, things and services

Users are human participants that use services and their own end equipment. They mostly consume information and may inspire actions through profile settings and other decision making processes.
Things are physical or virtual endpoints representing either a data source, data sink or both. They feed or consume information to and from the Internet.
Services are information aggregators and may provide tools for data analysis of different kinds.

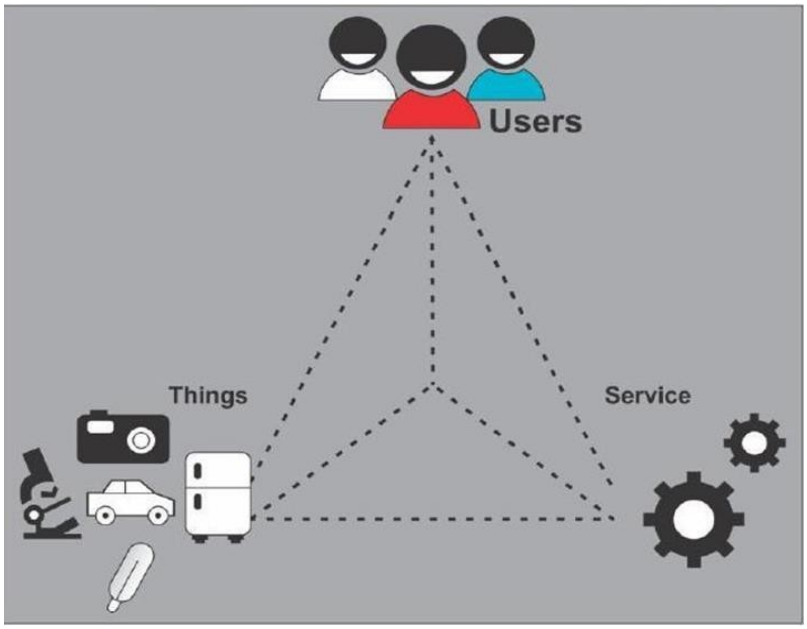


Figure: The IOT Players

The different devices and environments needed in IoT can be layered as shown in the figure. The sensors and devices needed in the IoT environment are the bottom layer. The different types of sensors can be temperature, pressure, moisture etc. The data captured by the sensors needs to be processed using processors and enabling technologies.

The technologies include RFID detection, motion sensing etc. Some of the technologies that enable these devices are discussed further in the Wireless Sensor networks section. Examples include Bluetooth, Wi-Fi etc. The processed data can be stored using cloud infrastructures and thus in turn provide different IoT services. The different types of IoT services include Home automation, healthcare services, energy management, emergency services among others.

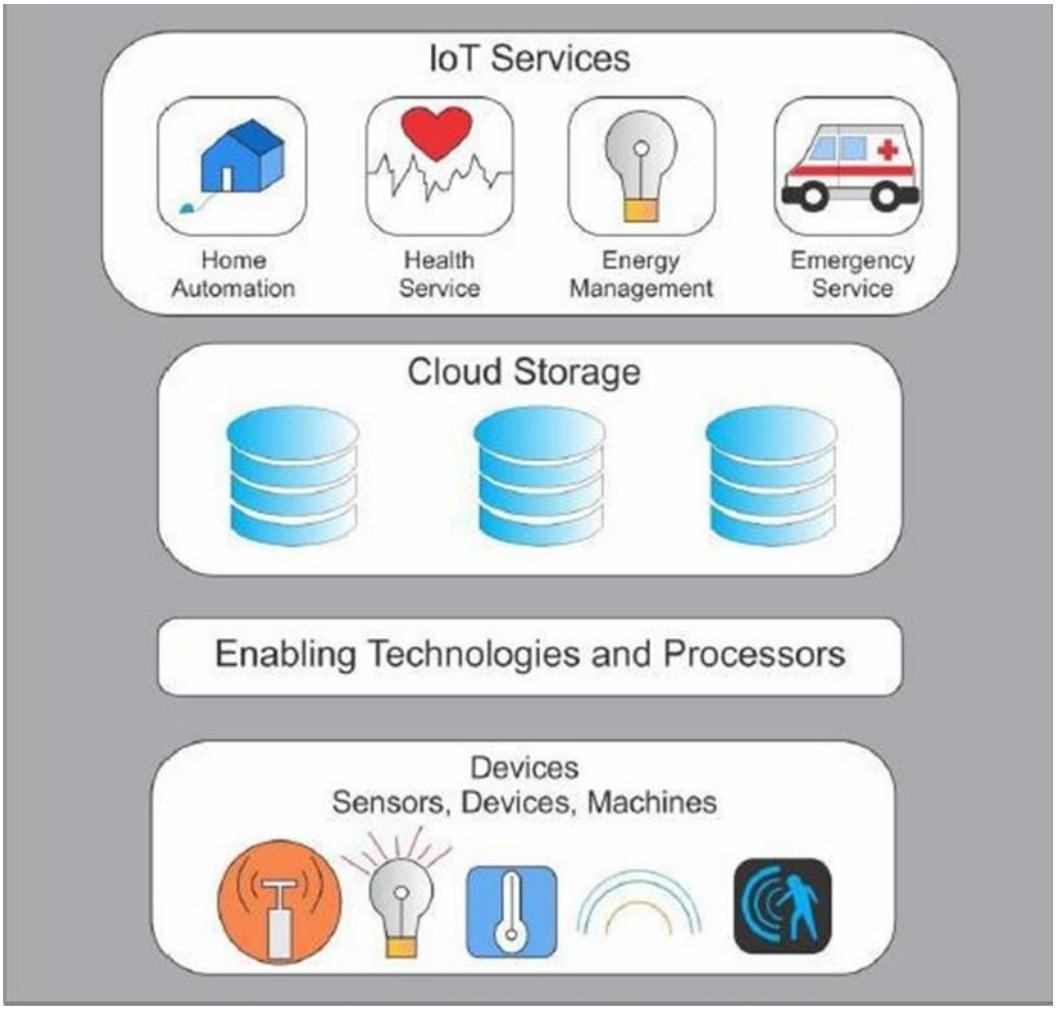


Figure: Architecture of IoT

Challenges of IOT

Connectivity: There is not one connectivity standard that "wins" over the others. There are a wide variety of wired and wireless standards as well as proprietary implementations used to connect the things in the IoT. The challenge is getting the connectivity standards to talk to one another with one common worldwide data currency.

Power management: More things within the IoT need to be battery powered or use energy harvesting to be more portable and self-sustaining. Line-powered equipment need to be more energy efficient. The challenge is making it easy to add power management to these devices and equipment. Wireless charging incorporates connectivity with charge management.

Complexity: Manufacturers are looking to add connectivity to devices and equipment that has never been connected before to become part of the IoT. Ease of design and development is essential to get more

things connected especially when typical RF programming is complex. Additionally, the average consumer needs to be able to set-up and use their devices without a technical background

Rapid evolution: The IoT is constantly changing and evolving. More devices are being added every day. The challenge facing the industry is the unknown; unknown devices, unknown applications, unknown use cases. There needs to be flexibility in development.

Energy Efficiency: When it comes to power, the challenge is to ensure that adding Internet connectivity does not impose a change to the power supply. In other words, ideally it should fit within the existing power budget headroom.

Security: Security is always a challenge in data networks. This challenge intensifies in the case of the IoT simply because there are more entry points thereby creating more penetration points. This increased system vulnerability makes the battle for security inevitable. In an IoT solution, threats also take a new level of magnitude since it is not just data that is put at risk. With IoT the damage potential is much higher (e.g., opening a door remotely, taking a burglar alarm system offline). There will surely be a never-ending fight towards better security. This provides inbuilt security features to address major security requirements.

Data handling: Massive deployment of endpoints results in higher node density. This requires demand for higher capacity. Furthermore, large quantities of data that are generated create a need for accessible storage. In addition, real network latency introduces a challenge to limited resource systems.

Wireless Sensor Networks:

Wireless Sensor Networks (WSNs) are networks of tiny, battery powered sensor nodes with limited onboard processing, storage and radio capabilities. Recent advances in micro-electro-mechanical systems (MEMS) technology, embedded electronics and wireless communication have made it possible to develop low-power and low-cost sensor nodes that are small in size and communicate using wireless medium over short distances.

The sensor units in the nodes can sense any desired parameter (like temperature, pressure humidity, movement etc.) in an area that is covered by the network. The sensed data is then relayed through the network to the base station, where information can be generated and acted upon to serve the purpose for which the network has been deployed.

WSNs are on the verge of being utilized for many challenging real-life applications like early earthquake warning systems, battlefield surveillance, environment and habitat monitoring, healthcare, smart homes and buildings etc... This involves deploying a large number of nodes in the area to be sensed by the network. This large-scale deployment often requires the nodes to possess self-organizing capability to form a network without any human intervention.

A typical cluster-based sensor network topology as shown in Fig. 5.10 consists of a base station, cluster-head nodes and sensor nodes. The base station is normally connected to the outside world through internet link or a user terminal.

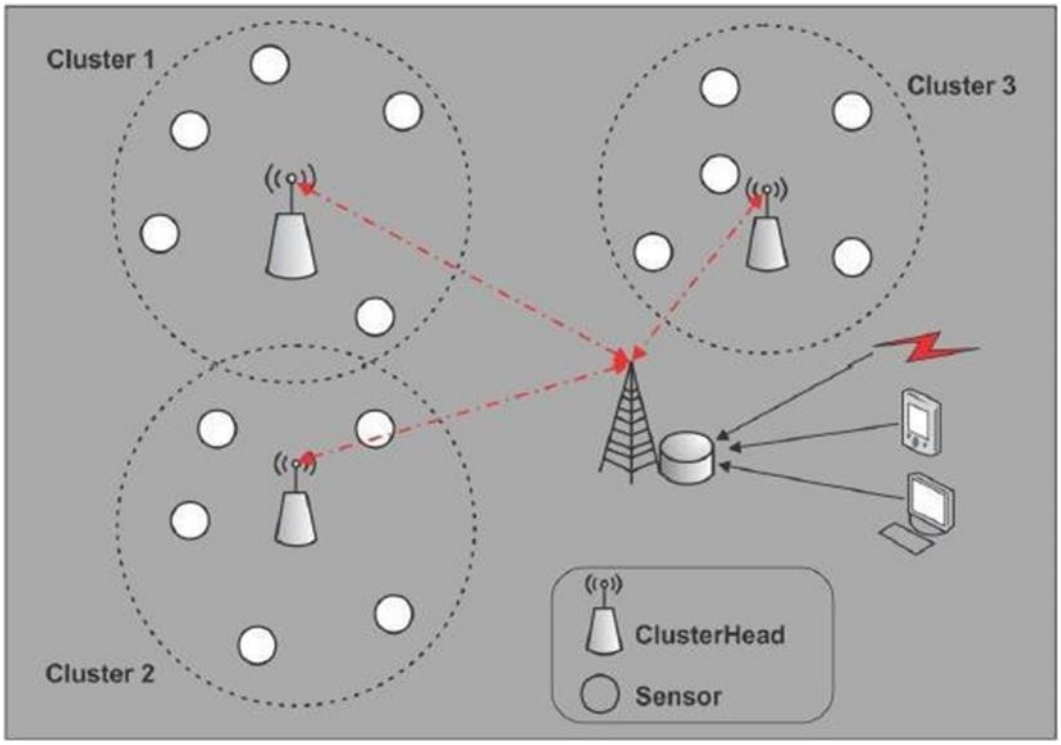


Fig . A Typical Sensor Network Architecture

Wireless Connectivity in Embedded Networks:

Wireless communication has become a preferred choice for connecting the devices in embedded networks. Communication technologies like NFC, ZigBee, Bluetooth, WiFi, and cellular have already become popular with developers working on Smart Homes, Sensor Networks and IoT based applications. The choice of a connectivity option depends upon various factors like communication range, bandwidth requirements, security issues, and power consumption.

	Low Energy Bluetooth	ZigBee	NFC	Low Power WiFi
Frequency (MHz)	2402 – 2482	868 - 868.8, 902 - 928, 2402 – 2482	13.56	2400 - 2500
Channels	3	16	1	3
Modulation	GFSK	BPSK & QPSK	ASK	64QAM
Max potential data rate	1 Mbps	250 Kbps	424 Kbps	54 Mbps
Range	10m	100+m	10cm	30m
Power Profile	Days	Months/Years	Months/Years	Hours
Complexity	Complex	Simple	Simple	Complex
Nodes/Master	7	65,000	1+1	
Extendibility	No	Yes	No	Yes

37

Fig. Comparison of different wireless technologies

***Near Field Communication* (*NFC*):**

NFC is a wireless communication technology that enables two devices to interact when they are in very close proximity to each other. For example, smartphones and other such devices can use NFC to interact with NFC-enabled machines (for e.g. battery-charging portals, ticket-vending machines, ATM, etc.) to exchange information at a distance of less than 10cm. NFC-enabled devices use globally available unlicensed radio frequency ISM band of 13.56 MHz on standard ISO/IEC 18000-3 air interface at rates ranging from 106 kbit/s to 424 kbit/s. Each full NFC device can work in 3 modes:

1. **NFC Card Emulation mode** – devices act like smart cards, to allow users to perform transactions like payment or ticketing.
2. **NFC Reader/ Writer mode** – allows NFC-enabled devices to read information stored on NFC tags embedded in labels or smart posters.
3. **NFC Peer-to-peer mode** – NFC-enabled devices can form an adhoc network to communicate and exchange information with each other.

ZigBee:

ZigBee is an industry-standard wireless networking technology that is suitable for applications that require infrequent low-power data transfer at low data rates within a 100m range, such as inside a home or a building. It is an IEEE 802.15.4 based specification for a suite of high-level communication protocols used to create personal area networks (PAN's) with small, low-power digital radios.

ZigBee based PAN's are suitable for applications like - home entertainment and control, building automation, industrial control and implementing wireless sensor networks. It operates in the ISM radio bands with data rates that can vary from 20 kbit/s to 250 kbit/s. The protocol stack for ZigBee builds on the physical layer and MAC layer defined in IEEE standard 802.15.4 for low-rate wireless PAN's (WPAN's). To this, ZigBee adds on specifications for network layer and application layer.

The ZigBee network layer supports star,tree and mesh network topologies. The application layer provides an interface between ZigBee system and the end user applications. A ZigBee network may consist of the following three types of devices:

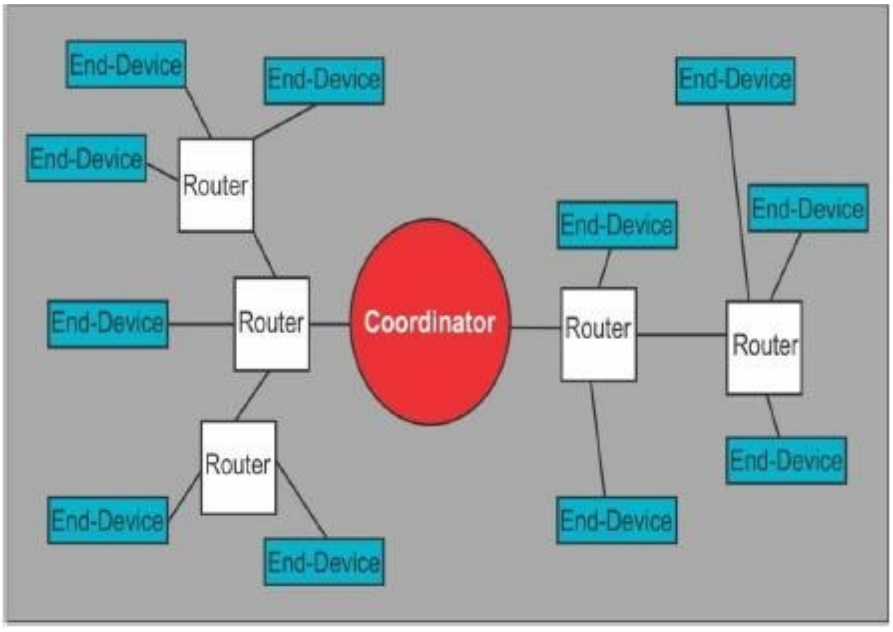


Fig 5.42. **Zigbee co-ordinator with end device**

ZigBee Coordinator (**ZC**): It is the most capable device in the network. Each ZigBee network must have exactly one coordinator device, and it is responsible to build and maintain the network. The ZC forms the root of the network and also connects to the other networks.
ZigBee Router (**ZR**): In addition to communicating with ZC, a ZR device can perform the function of forwarding/routing data received from other devices.
ZigBee End Device (**ZED**): A ZED device can only talk to its parent node, a ZC or a ZR device.

Bluetooth

It is an important short-range communication technology that is widely used in smartphones and many other fixed as well as mobile devices, for data transfer and building personal area networks. It operates in the 2.4 GHz ISM frequency band and uses frequency hopping spread spectrum technique. Bluetooth is a packet-based protocol with a master-slave structure. One master may communicate with upto 7 slaves in a piconet. Two or more piconets can be connected to form a bigger network, called a scatternet.

Bluetooth is widely used in applications like handsfree headset, smartphone-to-smartphone data transfer, wireless communication between smartphone and car-stereo system, cable-free connection between PC and I/O modules like mouse, keyboard, printer etc. Bluetooth in its new avatar as *Bluetooth Low Energy* (BLE) or *Bluetooth Smart,* is expected to be a key technology in near future for wearable devices that will connect to the IoT, probably through the smartphones and other such options.

Bluetooth Smart is meant to provide low-cost and low-power consumption while maintaining the same range as Bluetooth. Application development with Bluetooth Smart: Texas Instrument's CC2460 is a wireless MCU that can be used to design Bluetooth Smart enabled applications. The CC2640 contains a 32-bit ARM Cortex-M3 processor running at 48- MHz as the main processor and a rich peripheral feature set, including a unique ultra-low power sensor controller, ideal for interfacing external sensors and/or collecting analog and digital data autonomously while the rest of the system is in sleep mode. The Bluetooth Low Energy controller is embedded into ROM and run partly on an ARM Cortex®-M0 processor.

Wi-Fi

Wi-Fi is a wireless local area network (WLAN) technology that allows electronic devices to network using the 2.4 GHz or 5 GHz ISM radio bands. It is based on the IEEE 802.11 MAC and physical layer standards for WLAN and is the most pervasive choice for connectivity with the Internet, especially in the home LAN environment. Wi-Fi supports very fast data transfer rates, but consumes a lot of power which makes it unviable for low-power applications. Nevertheless, the embedded networks, wireless sensor network applications and Internet-of-Things implementations explicitly make use of Wi-Fi as a preferred choice for connectivity to the Internet.

Adding Wi-fi to a Microcontroller-Based System using CC3100 Simple link Wi-fi Module:

To illustrate the use of wireless connectivity in embedded networks, this section discusses the usage of Wi-Fi technology with a microcontroller. Wi-Fi is very widely used to provide connectivity between user and embedded systems. For example, a user can interact with utility systems (like AC, Garage door, Coffee machine, etc.) in a smart-home using a smartphone, provided both (smart-home and smartphone) are connected to the internet. TI provides low-power and easy-to-use Wi-Fi solutions that include battery-operated Wi-Fi designs with more than a year of battery life on two AA batteries. TI's Simple Link Wi-Fi CC3100 module is a wireless network processor with on-chip Wi-Fi, internet, and robust security protocols. It can be used to connect any low-cost microcontroller (MCU).

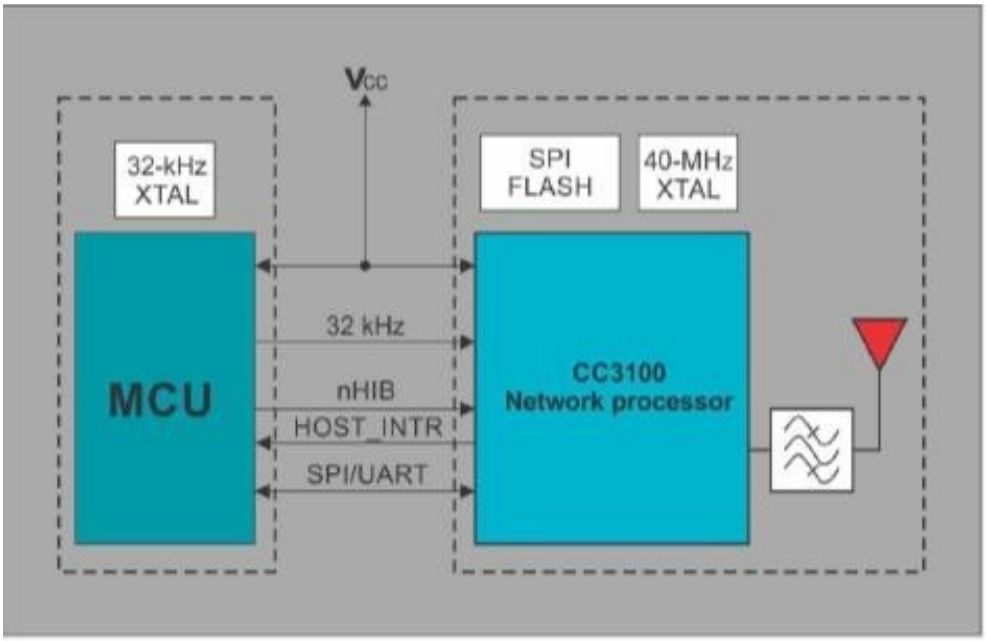


Fig Functional diagram of SimpleLink Wi-Fi CC3100 Module

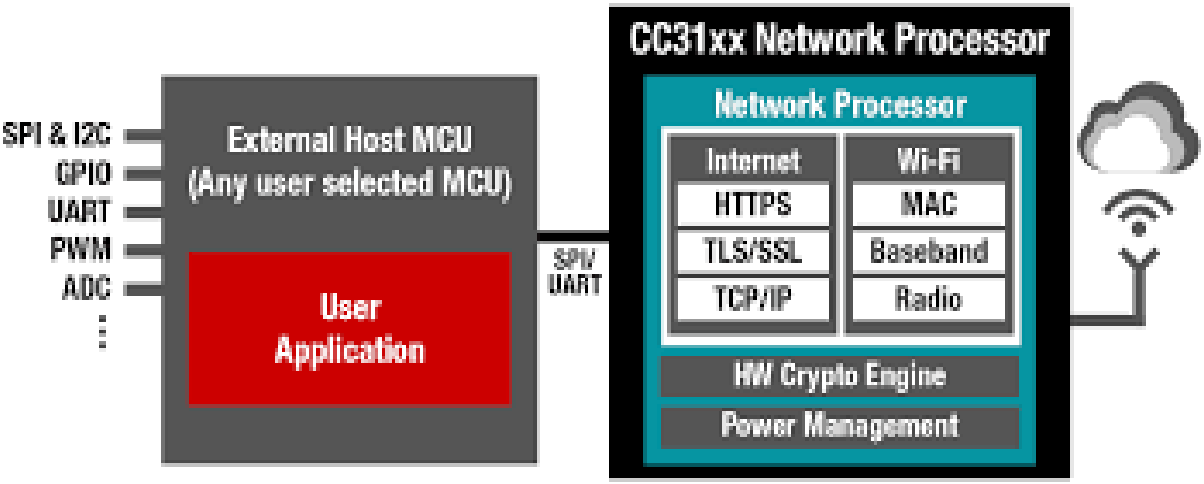


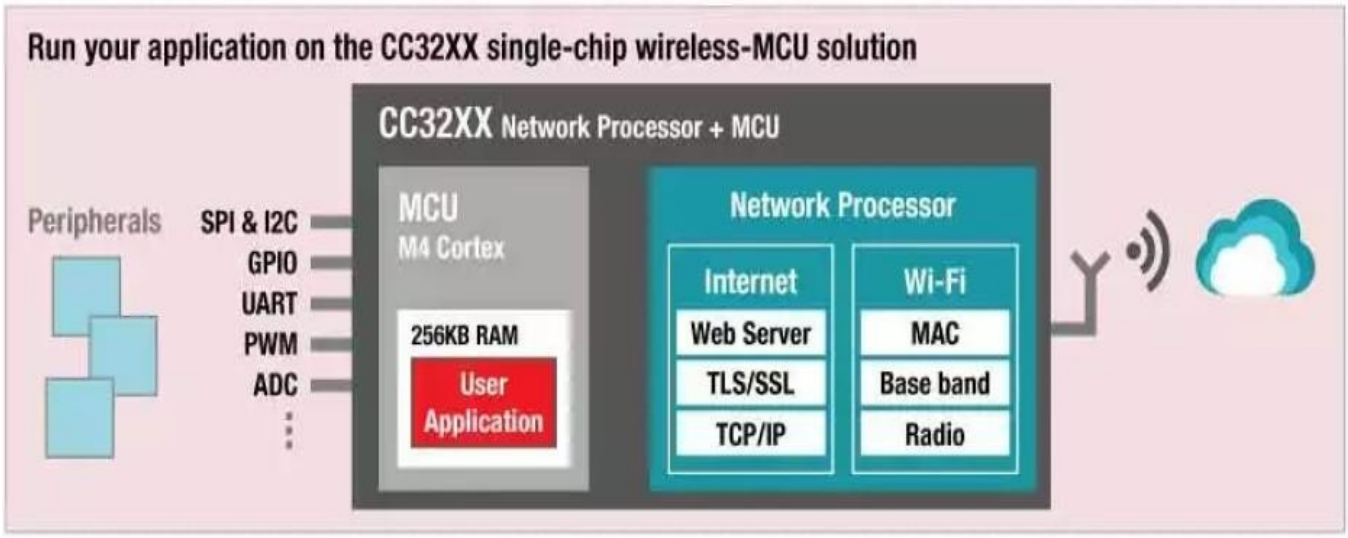
Fig SimpleLink Wi-Fi CC3100 Module

Embedded Wi-Fi:

Architecture of Simple Link Wi-Fi CC3100 Module

It is important to understand the hardware and software architecture of any device before using it in a design. Fig. shows the hardware architecture for Simple Link Wi-Fi CC3100 module, that can be used to provide Wi-Fi connectivity to any micro-controller based system. It consists mainly of two parts:

I. Wi-Fi Network Processor Subsystem
II. Power-management Subsystem



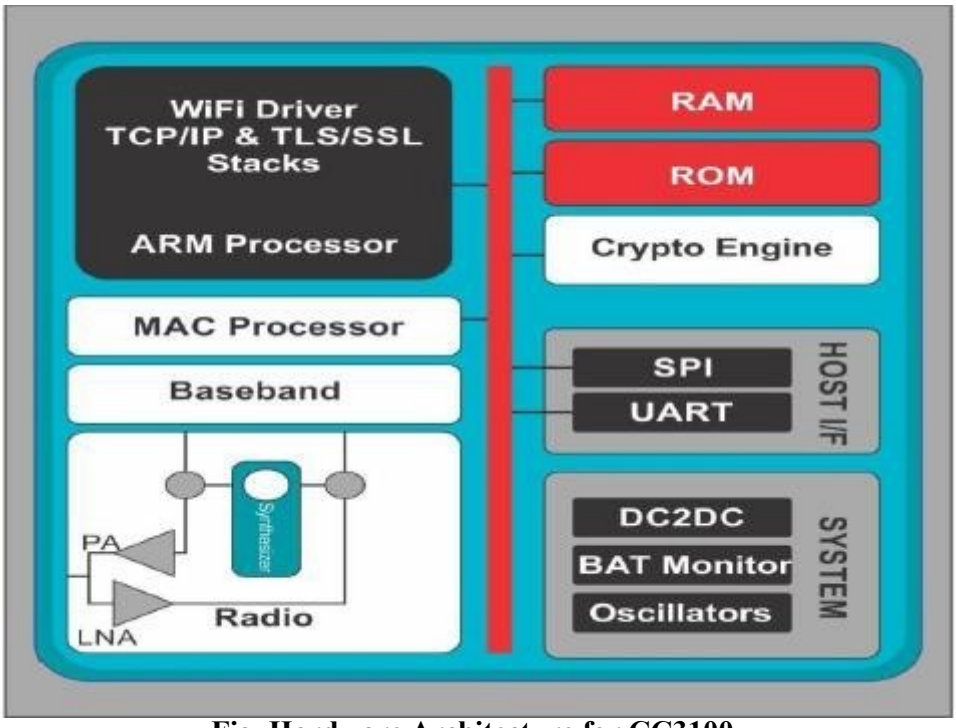


Fig. Hardware Architecture for CC3100

Wi-Fi Network Processor Subsystem:

The Wi-Fi Network Processor subsystem mainly consists of the following:

1. Dedicated ARM MCU – It executes the Wi-Fi and Internet protocols required to communicate over the Internet using Wi-Fi connectivity.
2. ROM – stores preprogrammed Wi-Fi driver and multiple Internet protocols
3. TCP/IP Stack – supports communication with computer systems on the Internet
4. Crypto Engine – provides fast, and secure Wi-Fi as well as Internet connectivity
5. 802.11 b/g/n Radio, Baseband and Medium Access Control - for wireless transmission and reception of data
6. SPI/ UART Interface – connects the CC3100 module to the host MCU.

Power Management Subsystem

The power management subsystem of CC3100 module provides the CC3100 module with an integrated DC-to-DC converter with a wide range of power supply from 2.3 to 3.6 V. This subsystem enables low-power consumption modes such as hibernate with RTC mode, which requires approximately 7 μA of current.

Features of Wi-Fi supported by CC3100 chip

The Wi-Fi network processor sub-system in SimpleLink Wi-Fi CC3100 device integrates all protocols for Wi-Fi and Internet, greatly minimizing MCU software requirements. With built-in security protocols, SimpleLink Wi-Fi provides a simple yet robust security experience. This section discusses the features of Wi-Fi supported by the CC3100 device. A list of features and the functionality provided by them is given in Table-1.

Table 5.1: Wi-Fi features

Sr. No.	Wi-Fi Feature	Function/ Utility
1	Supports 1-13 Wi-Fi channels	Provides 13 channels in 2.4 GHz frequency band
2	Support for WEP, WAP, WAP2	Secure Wi-Fi access
3	Enterprise Security	Provides additional security for enterprise networks
4	Wi-Fi Protected Set-up with WPS2	Provisioning methods to connect to Wi-Fi
5	Access Point mode with internal HTTP server	
6	SmartConfig technology	
7	802.11 Transceiver	Transmits and receives Wi-Fi packets
8	Supports IPv4	Internet Protocol
9	802.11 Power save and device deep sleep power with three user configurable policies	Low Power Operation
10	Up to 8 open sockets Up to 2 secured application sockets	User Application Sockets

USER APIs:

CC3100 SimpleLink Driver and its Application Programming Interface (API)

In order to simplify the development using the SimpleLink Wi-Fi devices, TI provides a simple and user friendly host driver software. This driver software allows any MCU (like TIVA platform) to interact with a SimpleLink device and performs the following functions:

1. Provides a simple API for user application development.
2. Handles the communication of MCU with the SimpleLInk device.
3. Provides flexibility in working with a MCU, with or without an OS.
4. Works with existing UART or SPI physical interface drivers
5. Compatible with 8-bit, 16-bit or 32-bit MCUs

The SimpleLink Host Driver includes a set of six logical and simple API modules:
Device API – Manages hardware-related functionality such as start, stop, set, and get device configurations.
WLAN API – Manages WLAN, 802.11 protocol-related functionality such as device mode (station, AP, or P2P), setting provisioning method, adding connection profiles, and setting connection policy.
Socket API – The most common API set for user applications, and adheres to BSD socket APIs.
NetApp API – Enables different networking services including the Hypertext Transfer Protocol (HTTP) server service, DHCP server service, and MDNS client\server service.
NetCfg API – Configures different networking parameters, such as setting the MAC address, acquiring the IP address by DHCP, and setting the static IP address.
File System API – Provides access to the serial flash component for read and write operations of networking or user proprietary data.

Programmer's model for CC3100 SimpleLink driver and its API

A programmer using a SimpleLink device needs to know about the different software blocks needed to build a networking application. This topic describes the recommended flow for most applications. However, program developers have complete flexibility on how to use the various software blocks. Programs using the SimpleLink device consist of the following software blocks:

- **Wi-Fi subsystem initialization** – Wakes the Wi-Fi subsystem from the hibernate state.
- **Configuration** – WiFi sub-system. This phase refers to time configuration that does not happen very often. For instance, changing the WiFi sub-system from a WLAN STA to WLAN soft AP, changing the MAC address and so forth.
- **WLAN connection** – The physical interface needs to be established. There are numerous ways to do so, all of which will be explained in this document. The simplest way is to manually connect to an AP as a wireless station.

DHCP – Although not an integral part of the WLAN connection, you need to wait for the receiving IP address before continuing to the next step of working with TCP\UDP sockets.

- **Socket connection** – At this point, it is up to the application to set up their TCP\IP layer. Separate this phase into the following parts:

Creating the socket – Choosing to use TCP, UDP or RAW sockets, whether to use a client or a server socket, defining socket characteristics such as blocking\nonblocking, socket timeouts, and so forth.
Querying for the server IP address – In most occasions, when implementing a client side communication, you will not know the remote server side IP address, which is required for establishing the socket connection. This can be done by using DNS protocol to query the server IP address by using the server name.

- **Creating socket connection** – When using the TCP socket, it is required to establish a proper socket connection before continuing to perform data transaction.
  - **Data transactions** – Once the socket connection was established, it is possible to transmit data both ways between the client and the server. Basically implementing the application logic.
  - **Socket disconnection** – Upon finishing the required data transactions, it is recommended to perform a graceful closure of the socket communication channel.
  - **Wi-Fi subsystem hibernate** – When not working with the Wi-Fi subsystem for a long period of time, it is recommended to put it into hibernate mode.

Case Studies with SimpleLink Wi-Fi CC3100 and TIVA Launchpad:-

Case Study: **Smart Plug with Remote Disconnect and Wi-Fi Connectivity**

In this application, the WiFi enabled Smart plug helps you to control any connected device from home or remotely from anywhere in the world with internet access such as home appliances like control portable heaters or window ac, turn on a light, Smart Grid and in building automation. A smart plug is an electronic device, generally connected to other devices or networks via different wireless protocols such as Bluetooth, NFC, WiFi, 3G, etc., that can operate to some extent interactively and autonomously. Now an day all application like home automation and building automation requires two main aspects of Smart Plug technology.

Android and cloud based remote access.
Remote disconnect and Wi-Fi connectivity based upon power consumption.

In this case study the WiFi enabled Smart Plug utilizes a TIVA Launchpad to monitor the energy consumption for a single load and control the high-voltage side of the design. This data is then passed to a CC3100 module to communicate the data over Wi-Fi to a Cloud server. A solid state relay enables the application to control the load, based on its energy consumption. And this system is powered from a highly compact and efficient UCC28910D High-Voltage Flyback Switcher with Primary-Side Regulation and Output Current Control.

Block Diagram

Figure shows a high-level overview of the connections between the the various devices in the Smart Plug reference design. The only physical connections from the system are the AC voltage input and the output for an AC load. On this high-voltage line, the flyback power supply, load control relay, and metrology sensors are connected.

Circuit Design and Component Selection

Embedded Metrology – Analog Inputs

The analog front-end of the MSP430i204x, which consists of the ΣΔ ADC, is differential and requires thatthe input voltages at the pins do not exceed ±928 mV (gain = 1). To meet this specification, the current and voltage inputs must be scaled down. Additionally, the ΣΔ24 allows a maximum negative voltage of - 1V. Therefore, the AC current signal from mains can be directly interfaced without the need for level shifters. This section describes the analog front end used for the voltage and current channels.

Voltage Inputs

The voltage from the mains is usually 230 V or 120 V, and must be scaled down to within 928 mV. The analog front end for voltage consists of spike protection varistors followed by a voltage divider network, and an RC low-pass filter that acts as an anti-alias filter.

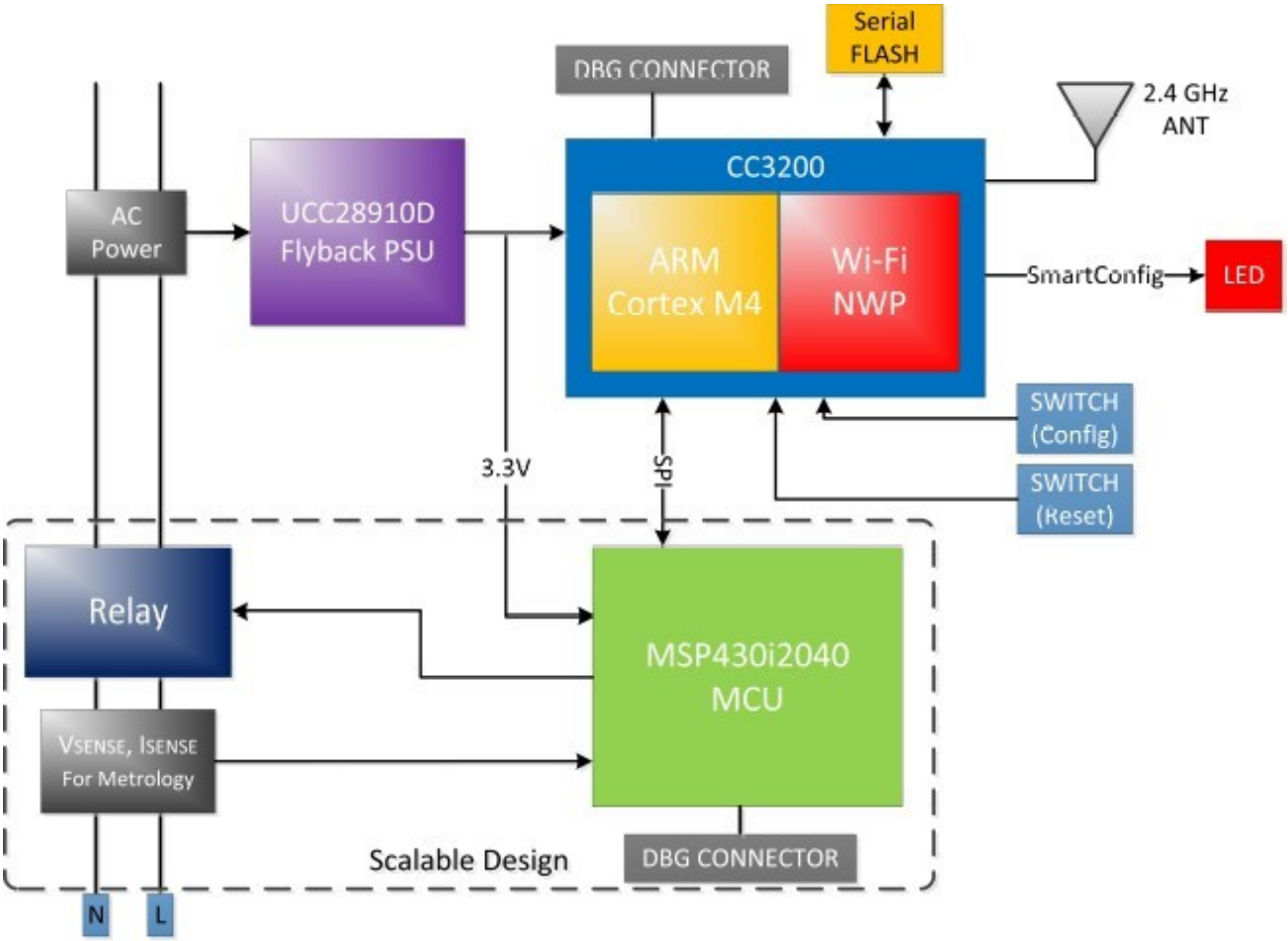


Figure 1. Smart Plug Block Diagram

Current Input
The analog front-end for current inputs is slightly different from the analog front-end for the voltage inputs.

Embedded Wi-Fi
The Smart Plug uses two primary design considerations for the CC3200 Wi-Fi SoC, the antenna design and the power management sections

Antenna Landing
As with many RF designs, the placement and control of the antenna and matching circuit is critical to ensure peak performance. By utilizing the work already done on the Simple Link Wi-Fi Launchpad, this process is highly simplified.

Power Management
The CC3200 SoC provides a wide range of potential power options to the designer, including wide input voltage ranges and battery power support. For the Smart Plug, however, the design has only a single 3.3V power rail, so the power management system needs to be tweaked from the standard design

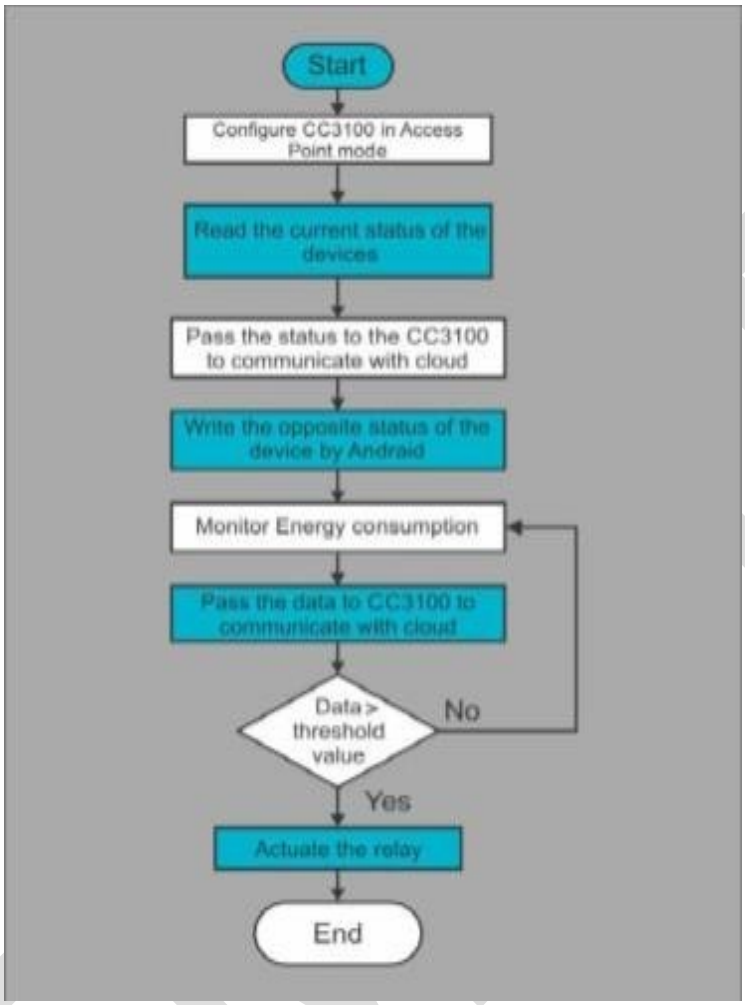


Fig Flow chart of Smart Plug with Wi-Fi connectivity