EMBEDDED SYSTEMS

B.TECH
(IV YEAR – I SEM)

Department of Electronics and Communication Engineering

SVR ENGINEERING COLLEGE
NANDYAL.

AYYALURU METTA, NANDYAL– 518 503 (A.P)
(Affiliated to JNTUA Anantapur, Approved by AICTE, New Delhi)

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR B. Tech
IV-I Sem. (ECE)

LTPC
3103

15A04702

EMBEDDED SYSTEMS

Course Objectives:

- To understand the fundamental concepts of Embedded systems.
- To learn the kernel of RTOS, architecture of ARM processor.

Course Outcomes: After completion the students will be able to

- Design of embedded systems leading to 32-bit application development.
- Understand hardware-interfacing concepts to connect digital as well as analog sensors while ensuring low power considerations.
- Review and implement the protocols used by microcontroller to communicate with external sensors and actuators in real world.
- Understand Embedded Networking and IoT concepts based upon connected MCUs

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.

EMBEDDED SYSTEMS

UNIT-I

Introduction to Embedded system

⇒ Embedded system Introduction:→

*System:- It is an arrangement in which all its units, assembled work together according to set of rules.

Ex: "Watch" is a time displaying system.

*Embedded system:-

Def: ① It is a combination of s/w & H/w, which is designed to perform a particular task and that task has to be completed in a given time.

(or)

- An embedded system is a combination of hardware and software with some attached peripherals to perform a specific task or a narrow range of tasks with restricted resources.
- It is an electronic system that is not directly programmed by the user, unlike a personal computer.

Ex: Mobile phone, Washing machine, Microwave oven, Digital cameras, Air conditions etc..

- Generic block diagram of an embedded system is

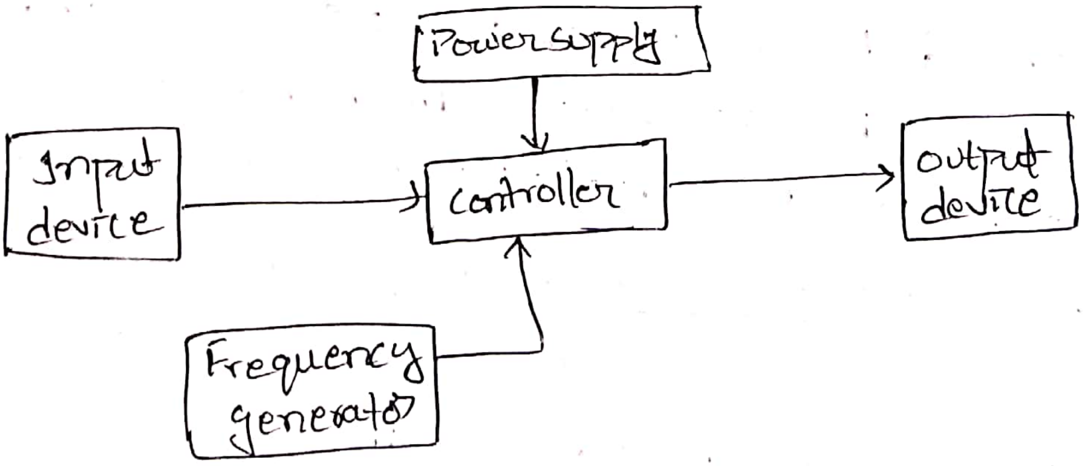


fig:- generic block diagram of an embedded system

- Every embedded system consists of certain input devices such as: key boards, switches, sensors, actuators; output devices such as: displays, buzzers, sensors; Processor along with a control program embedded in the off-chip or on-chip memory, and a real time operating system (RTOS).

* Characteristics of an Embedded System:

- The important characteristics of an embedded system are
  - Speed (bytes/sec) : should be high speed
  - Power (watts) : Low power dissipation
  - Size and weight : As far as possible small in size & low weight.
  - Accuracy (% error) : Must be very accurate
  - Adaptability : High adaptability and accessibility.
  - Reliability : Must be reliable over a long period of time.

So, an embedded system must perform the operations at a high speed so that it can be readily used for real time applications and its power consumption must be very low and the size of the system should be as for as possible small and the readings must be accurate with minimum error. The system must be adaptable for different situations.

* Classifications of Embedded Systems:

- Embedded System can be classified into the following categories based on their functional & performance/size requirements.

Embedded system (E-S)

- Functional
  - standalone E-S
  - Real time E-S
    - Hard RTES
    - soft RTES
  - Networked
  - Mobile
- Performance/size
  - Small scale E-S
  - Medium scale E-S
  - Large scale E-S or Sophistcated E-S

*** standalone E-S:**

- A stand-alone embedded system works by itself.
- It is a self-contained device which does not require any host system like a computer.
- It takes either digital or analog inputs from its input ports, calibrates, converts, and processes the data, and outputs the resulting data to its attached o/p device,
- Which either displays data, or controls and drives the attached devices.

Ex: Temperature measurement systems, video game consoles, MP3 players etc.

*** Real Time Embedded Systems:**

- An embedded system which gives the required output in a specified time or which strictly follows the time deadlines for completion of a task is known as a Real time system.
- A Real Time System, in addition to functional Correctness, also satisfies the time constraints.
- There are two types of Real time systems

(1) Hard Real-Time system

(2) Soft Real-Time system

(1) Hard Real-Time system:-

- A Real time system the complection of an operation after its deadline may lead to a critical failure and loss of life or property damage is known as a Hard Real time system.
- These systems usually interact directly with physical hardware instead of through a human being.
- The hardware and software of hard-Real time system must allow a worst case execution {WCET} analysis that guarantees the execution be completed within a strict deadline.
- The chip selection and RTOS selection become important factors for Hard real-time system design.

Ex: Deadline in a missile control embedded system, Delayed alarm during a Gas leakage, A delayed response in Car airbag control system, pacemakers, Failure in RADAR functioning etc.

(2) Soft Real-Time system:-

- A Real time system the complection of an operation after its deadline will cause only the degraded quality, but the system can continue to operate is known as a Soft real time system.
- In Soft real-time systems, the design focus is to offer a guaranteed bandwidth to each real-time task and to distribute the resources to the tasks.

Ex: A Microwave Oven, Washing machine, TV remote etc..

* Networked Embedded Systems:-

- The networked embedded systems are related to a network with network interfaces to access the resources. The connected network can be a Local Area Network (LAN) or a Wide Area Network (WAN), or the Internet. The connection can be either wired or wireless.
- The networked embedded system is the fastest growing area in embedded systems applications.

Ex: A home security system is an example of a LAN networked embedded system where all sensors are wired and running on the TCP/IP protocol {Transmission control protocol (or) Internet Protocol}

* Mobile Embedded Systems:-

- The Portable embedded devices like mobile and cellular phones, digital cameras, MP3 players, PDA{Personal Digital Assistants} are the example for mobile embedded systems.
- The basic limitation of these devices is the limitation of the memory and other resources.

* Based on the performance of the μC they are also classified into

1. Small scaled Embedded system
2. Medium Scaled Embedded system
3. Large scaled Embedded system

* Small Scaled Embedded System:-

- These systems are designed with a single 8- or 16 bit μC; they have little hardware & software complexities and involve board-level design. They may even be a battery operated systems.
- When developing embedded s/w for these, an editor, assembler and cross assembler, specific to the μC or μP used, are the main programming tools.
- Usually, C is used for developing these systems. C prog compilation is done into the assembly, and executable codes are then appropriately located in the system memory.
- The software has to fit within the memory available

and keep in view the need to limit power dissipation when system is running continuously.

* Medium Scaled Embedded System:-

- These systems are usually designed with a single or few 16 - or 32-bit microcontrollers or DSPs or Reduced Instruction set computers (RISCs).
- These have both hardware and software complexities. They can essentially required of an operating system for complex software design.
- For complex software design, there are some programming tools: RTOS, Source code engineering tool, Simulator, Debugger and Integrated Development Environment (IDE).
- Software tools also provide the solutions to the hardware complexities.
- These systems may also employ the readily available ASSPs and IPs for the various functions.

* Large Scaled (or) Sophistricated Embedded systems:-

* These systems are usually designed with a high End Microcontrollers.

- Sophisticated Embedded systems have enormous hardware and software complexities and may need scalable processors or configurable processors and programmable logic arrays.
- Some of the functions of the hardware resources in the system are also implemented by the software.
- Development tools for these systems may not be readily available at a reasonable cost or may not be availabe at all.

Examples:-

SSEM:- Temparature measurement systems, Robotic arm controller, Automatic chocolate vending machine, stepper motor controllers for a Robotic Arm controller, & Digital multimeter etc..

MSES :- Router, Music systems, ATM machine, Pager, Fax machine etc..,

LSES :- Smart Phones, Multimedia systems.

⇒ Host and Target concept :-

- Embedded system has designed to perform specific functions such as computer hardware, softwar and other parts.
- The Embedded sytem has unique characteristics.

→ The components and functions of hardware & Software can be different for each system.

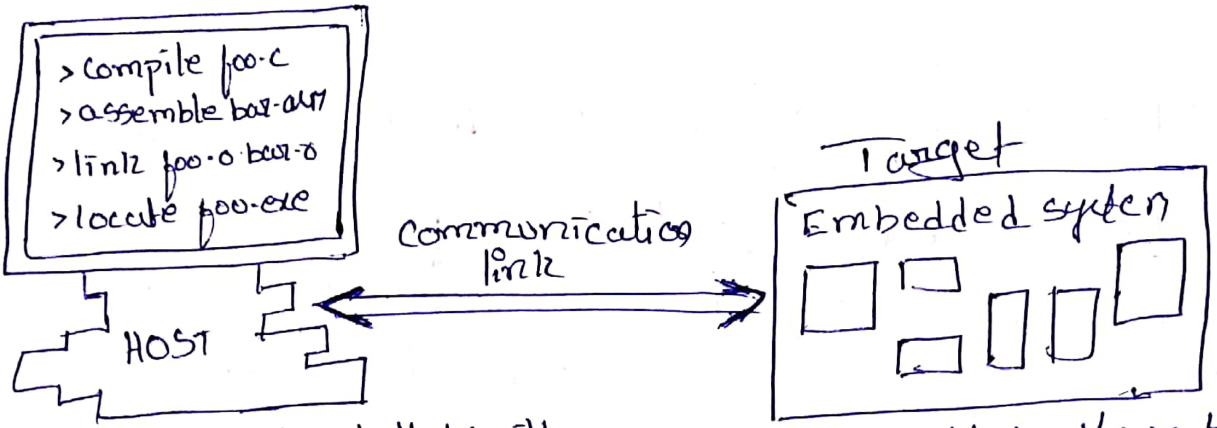
- Now a days, the Embedded software used in all electronic devices such as watches, cellular phones etc..
- This embedded software is similar to gereral programing
- But the embedded hardware is unique.

- The method of communication between interfaces can be vary from processor to processor.

- It leads to more complexity of software

- Engineers need to be aware of the software developing process and tools for Embedded System.

- There are a lot of things that software development tools can do automatically when the target platform is well defined
- This automation is possible because the tools can exploit features of the hardware and operating system on which your program will execute
- Embedded software development tools can rerely make assumptions about the target platform. Hence the user has to provide some explicit instructions of the system to the tools.



The development tools that build the Embedded Software run on a general-purpose computer

The Embedded software that is built by those tools runs on the Embedded system

fig: Embedded system using Host & target machine

* Performance of Host Machin:-

- An application program is developed that runs on host computer.
- It is also called as Development Platform
- It is a General purpose computer
- It has more capable processor and more memory.
- It has different input & output devices.
- Capable operating system.
- It contains many developments tools to create output of 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 of binary image is executed on target hardware platform
- It consists of two entities → Target H/w such as processor
  → Runtime environment such as OS
- It is needed only for final output.
- It is different from development platform & it does not contain any development tools.

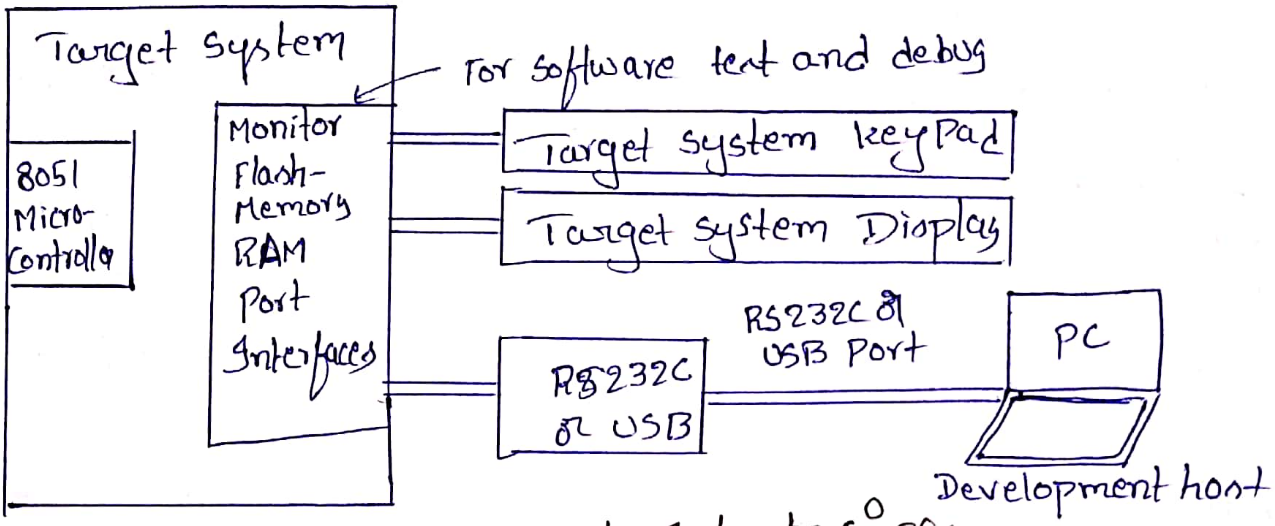
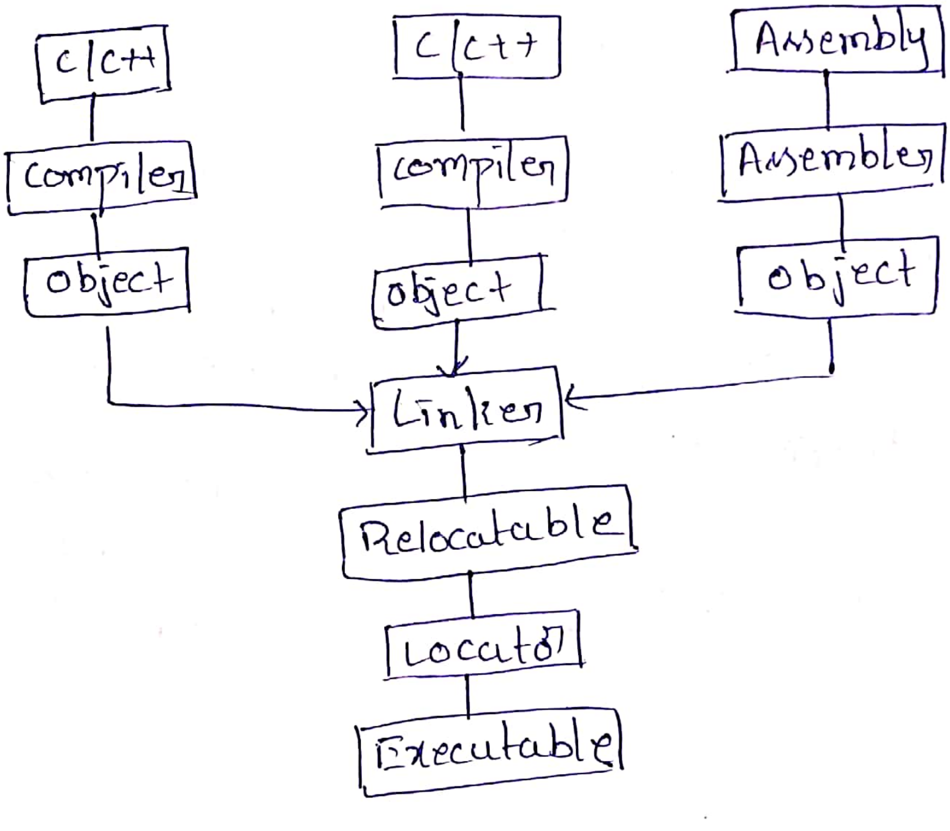


fig:- Host and Target Interfacing

*The Embedded Softwar Developement Process:-



- An each tools can takes one or more files as input and produces a single output file.
- This transformation process is performed by software running on a general-purpose computer.
- The compiler, assembler, linker, & locator run on a host computer.
- The Embedded System target platform runs only itself

- These tools combined to produce an executable binary image.
- This output of binary image runs on target Embedded Stm

★ **Compiling:-**

- A software program that converts source code High level language into low level language.
- Compiler translates program written in human readable language into machine language {source code → object file}
- This object file is binary file that contains set of machine language instructions (opcodes) and data resulting from language translation process.

★ **Native compiler:-** It runs on a computer platform and produces code for that same computer platform

★ **Cross compiler:-** It runs on a computer platform and produces code for another computer platform.

- Compilers can support standard object file formats. Such as COFF [Common Object File Format]
  ELF [Executable and linkable format] etc..

★ **Assembler/Interpreters:-**

- An assembler is a softwar program that converts source code of assembly language into machine language.
- An Interpreter constantly runs & interprets source code as a set of directives.

★ **linking:-** A linker or link editor is a program that takes one or more objects generated by compilers and assemblers them into a single executable program.

- The output of the linker is a new object file that contains all of the code and data from the input object files.

*Locating:- A locator is the tool that performs the conversion from relocatable program to executable binary image.

- The locator assigns physical memory addresses to code and data sections within the relocatable program.

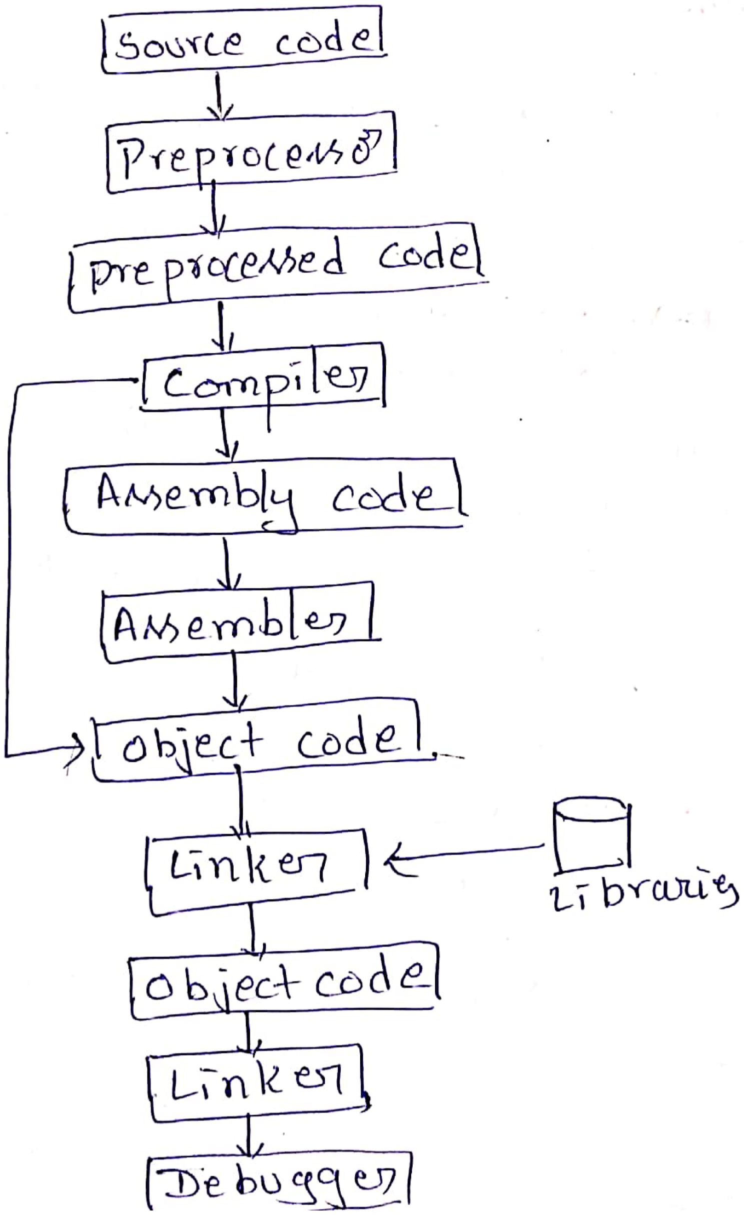
*Types of Embedded tools:-

*Embedded Software tools

- Editor
- Assembler
- Compiler
- Linker
- Simulator
- Profiler

- Embedded Hardware tools
- In-circuit Emulator
- Debugger
- Simulator & Emulator
- Starter kit

*Tools processing:- [Flow of process]:-



⇒ Embedded Applications :-

- Embedded systems used in various applications are listed as ~~as~~

1. Home Appliances: Dishwasher, washing machine, microwave,
Heating ventilation Air conditioning
Top-set box, security system, HVAC system, answering mc
garden sprinkler systems etc.

2. Office Automation: Fax, copy machine, smart phone system, modern scanner, Printers.

3. Security: Face recognition, Finger recognition, eye recognition, building security system, airport security system, and alarm system.

optical character recognition
4. Academia: Smart board, smart room, OCR, calculator, smartcard

5. Instrumentation: Signal generator, signal processor, power supplier, Process instrumentation.

6. Telecommunication: Router, hub, cellular phone, IP phone, web camera.

7. Automobile: Engine management system, Fuel injection controller, anti-locking brake system, air-bag system, GPS, cruise control.

8. Entertainment: MP3, Video game, Mind storm, smart toy.

9. Aerospace: Navigation system, automatic landing system, Flight attitude controller, space explorer, space robotics

10. Industrial automation: Assembly line, data collection system, monitoring systems on pressure, voltage, current, temperature, Hazard detecting system, industrial robot.

11. Personal: PDA, iPhone, palmtop, data organizer.

12. Medical: CT scanner, ECG, EMG, MRI, Glucose monitor, blood pressure monitor, medical diagnostic device.

13. Banking & Finance: ATM, smart vendor machine, cash register, Share market

14. Miscellaneous: Elevators, tread mill, Smard card, security ~~sec~~ room etc.,

ECG: ElectroCardioGraphy
EMG: ElectroMyoGraphy (muscles actin)
MRI: Magnetic Resonance Imaging (body scanner)

⇒ Features of an Embedded System :-

- Embedded systems are called product of Hardware and Software co-design. Embedded system includes different types of processors, power supply unit, clock, reset circuit, memories which are considered to be most essential Hardware components of standalone Embedded systems.

***Different types of processors used:-**

- Processor: A processor is a heart of the Embedded system It is responsible for execution of instruction and controlling flow of data to and from processor.
- Different types of processors available can be categorized into four ~~foo~~ board categorizes.

1. General Purpose Processor [GPP]
2. Application Specific System Processor [ASSP]
3. Multiprocessor System and
4. GPP core or ASIP core [Application specific Instructio Processor]

① **GPP:-** It may be any one of Microprocessor, Microcontrollers Embedded processor, Digital Signal Processor [DSP] & Media processor

② **ASSP:-** It is dedicated for faster processing and useful for applications like real time video processing which ~~in~~ incorporates lots of processing befor transmitting. It may also include some features of RTOS.

- ASSP provides ~~Har~~Hardwired solution for most its time consumming tasks.

③ **Multi processor system:-** As Embedded algorithm has to work with in strict deadline, sometimes it may not be possible to carry out the same with a single processor.

- In such a case an E.S may go for two or more processors
- Multiprocessors are used when a single processor doesn't meet the need of the different tasks that have to be performed concurrently.

④ GPP core or ASGP core:—

·It is integrated into either a application specific Integrated Circuit [ASIC] or a VLSI or an FPGA core ~~integ~~ integrated with processor units.

★ Power Supply unit:—

- Generally Embedded System has its own power supply unit.
- Fourrange of voltage ① 5.0V +0.25V ② 3.3V +0.3V ③ 2.0V +0.2V ④ 1.5V +0.2V are used for operation of different units.
- Additionally 12V + 0.2V supply is needed for a flash or EEPROM and RS232 Serial Interfaces.
- Supply of voltage to the chip depends on number of pins provided in the chip which is generally in pair supply and ground.

★ Clock oscillator:— The function of this oscillator circuit is to provide a accurate and stable periodic circuit signal to a processor

- The processor need a clock oscillator as clock controls and various clocking requirements of CPU. (External to the processor)
- The circuit uses either a crystal or ceramic resonator → Internally associated with the processor or an external IC attached to the processor.
- The machine cycle includes (i) Fetching code and data from memory and Decoding and execution and (ii) Transferring results to memory
- The clock controls the time for executing an instruction.
- The crystal resonator gives the highest stability in frequency with temperature and drift in the circuit
- The internal ceramic generator, if available in a processor, saves the use of the external crysal and gives a reasonable though not very high frequency.
- The external IC based clock oscillator has a significant higher power dissipation compared to the internal processor resonator.

* Real time clock or timer units:-

- A timer is suitably configured as system clock sometime referred as RTC (Real Time clock). RTC is used by scheduler for real time programming.
- A hardware timer is a counter that is incremented at a fixed rate when the system clock pulses. There are several different types of timers available. A timer/counter can perform several different tasks.
- More than one timers using the RTC may be needed for various timing and counting need. There may be a hardware and software implementations of timers.
- At least one hardware timer device is must in a system which is used as system clock.
- A software timer is a software that executes and increases or decreases a count variable or an interrupt on a timer output or on a real time clock interrupt. A software timer can also generate interrupt on overflow of count value or the final value of count variable.

* Interrupt Handler:-

- A system possesses a number of devices and the system processor has to control and handle the requirements of devices by running appropriate Interrupt Service Routine (ISR) for each.
- An interrupt handling mechanism must exist in each system to handle interrupt from various processes in the system.
- An interrupt is an event that suspends regular program operation while the ~~ve~~ event is serviced by another program.
- Different microcontrollers have different interrupt sources which can include external, timer and serial port interrupts
- When an interrupt is received the current operation is suspended, the interrupt is identified and the controller jumps to an interrupt service routine.
- There are two sources of interrupt: hardware and software.

- Hardware interrupts include a signal to a pin, timer overflow and serial port interrupts. Software interrupts are commands given by the programmer.
- There are two different interrupt types: maskable & non-maskable. A maskable interrupt can be disabled & enabled while non-maskable interrupt cannot be disabled and are therefore always enabled.

* Reset circuit and watchdog timer:-

- Reset instruction start execution from starting address otherwise execution start from this address when it is ~~powere~~ powered up. The reset circuit activates for a fixed period and then deactivates to let the program proceed from a default beginning address.
- On deactivation of the reset that succeed the processor activation, a program executes from start-up address.
- Reset can be activated either by external reset circuit that activates on power up or by software instruction or by a programmed timer known as watchdog timer.
- Watchdog timer is a timing device that resets the system after a predefind timeout this time is usually configured and the watchdog timer is activated within the first few clock cycles after power up.
- In many embedded systems reset by a watchdog timer is very essential because it helps in rescuing the system from program hangs. On restart program can function normally

* Memories:-

- Embedded system & makes use of different types of memories based on their features.

① Internal RAM used for registers, temporary data and stack.

② Internal ROM/PROM/EPROM for application program.

③ External RAM for temporary data and stack

④ Internal cache available in case of some microprocessor & microcontroller.

⑤ EEPROM of flash memory for saving the results
⑥ External ROM or PROM for embedding software used in non microcontroller based systems.
⑦ RAM memory buffers at ports. Caches for superscalar ups.

• Different types of memory devices in varying sizes are available for use as per requirement.

① Masked ROM or EPROM of flash which stores the embedded software (ROM image).
② EPROM or EEPROM is used for testing and design stages.
③ EEPROM (5V form) is used to store the results during the system program. run time.
④ RAM is mostly used in SRAM form in a system. Advanced system uses RAM in the form of a DRAM, SDRAM, & RDRAM.
⑤ Parameterised distributed RAM is used when I/o devices and subunits require a memory buffer

⑧ ★ Input/output units and buses :-

• The system gets input from physical devices such as keypads & boards, sensors, transducer circuits etc. It gets the values by read operations at the port address.

• The system has output ports through which it sends output bytes to the real world. It sends the values to output by a write operation at the port address.

• In case of some devices a port may be used as both input as well as output port.

• There are two types of I/o ports ① Parallel port & ② Serial port.

• A serial port facilitates long distance communications and Interconnections.

★ DAC/ADC :-

• For automatic control and signal processing applications, a system must provide necessary interfacing circuit & software for DAC unit & ADC unit.

- ADAC operation is done with the help of a combination of PWM unit in the microcontroller and External Integrator chip.
- ADC operations are needed in systems for voice processing, Instrumentation, Data acquisition system and ~~a~~ automatic control.

⇒ Data and Address Bus Concept:-

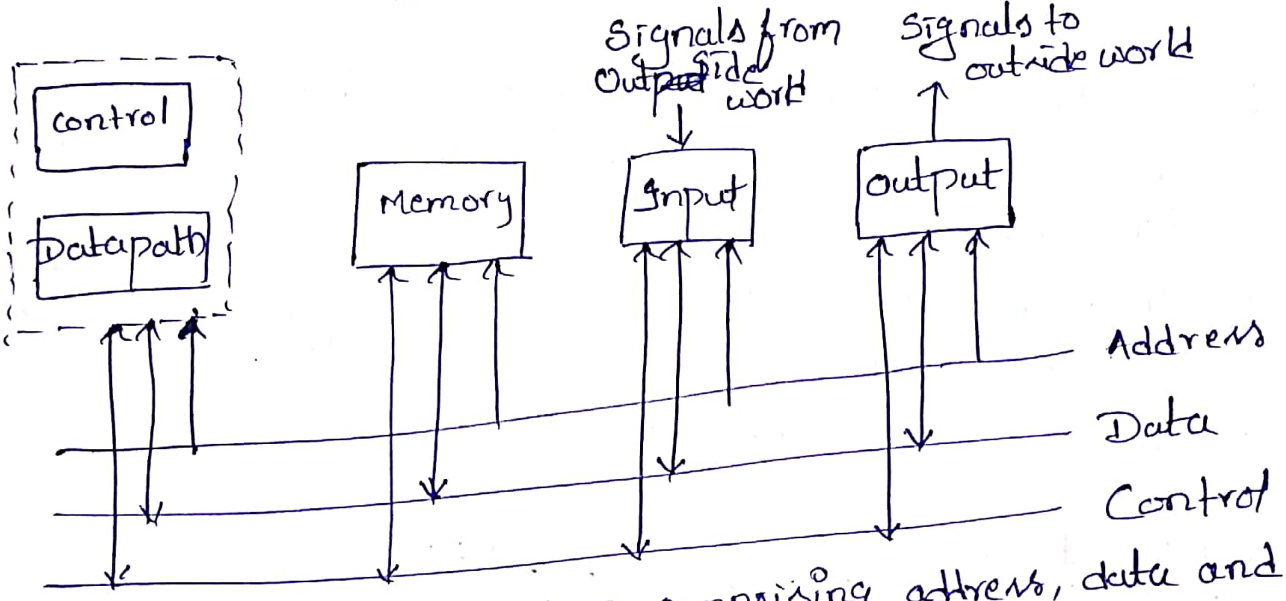


fig:- A typical Bus structure compoising address, data and control signals.

- According to computer architecture, a bus is defined as a system that transfers data between hardware components of a computer or between two separate computers.
- Initially, buses were made up using electrical wires.
- computer buses can be parallel or serial and can be connected as multi drop, daisy chain, or by switched hubs
- system bus is a single bus that helps all major components of a computer to communicate with each other.
- It is made up of an address bus, data bus and a control bus.

★ Address bus:-

- Address bus is a part of the computer system bus that is dedicated for specifying a physical address.
- When the computer processor needs to read or write from or to the memory, it uses the address bus to specify the physical address of the individual memory block it needs to access.
- When the processor wants to write some data to the memory, it will assert the write signal, set the write address on the address bus and put the data on to the data bus.
- Similarly, when the processor wants to read some data residing in the memory, it will assert the read signal and set the read address on the address bus.
- The size of the memory that can be addressed by the system determines the width of the address bus.

Ex: if the width of the address bus is 32 bits, the system can address 2^{32} memory blocks.

★ Data bus:-

- A data bus simply carries data. Internal buses carry information within the processor, while external buses carry data between the processor and the memory.
- Typically, the same data bus is used for both read/write operations. When it is write operation, the processor will put the data on to the data bus.
- When it is read operation, the memory controller will get the data from the specific memory block & put it in to the data bus.

★ Control bus:-

- control bus simply carries control information between the processor and other devices with in the computer.

- The control bus carries signals that report the status of various devices

*** Difference between Address bus and data bus:-**

- Data bus is bidirectional, while address bus is unidirectional. That means data travels in both directions but the addresses will travel in only one direction. ~~to~~ Because address is always specified by the processor.
- The width of the data bus is determined by the size of the individual memory block, while the width of the address bus is determined by the size of the memory that should be addressed by the system.

⇒ Architecture Considerations for Embedded System:

[ROM, RAM, Timers]:-

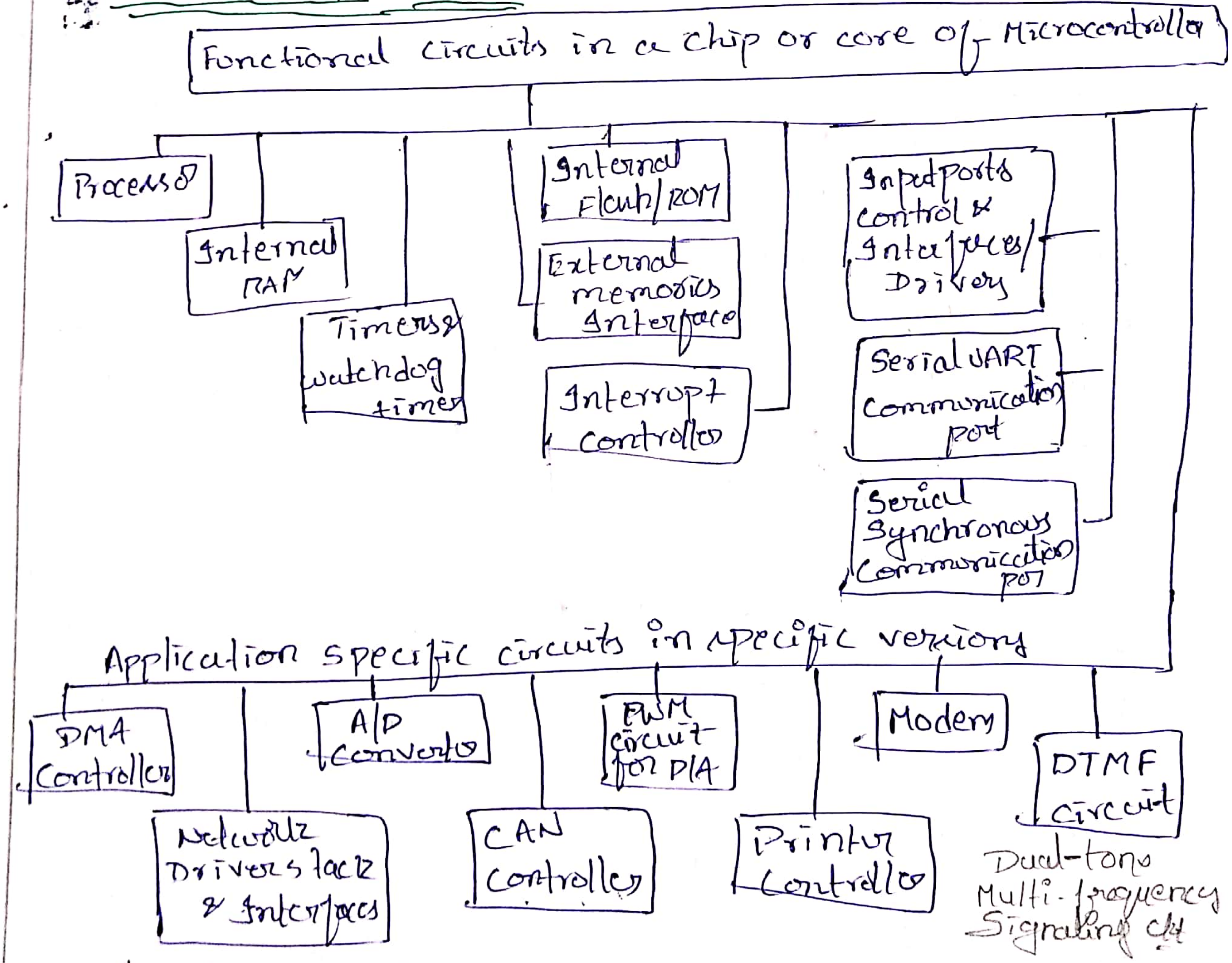

Functional circuits in a chip or core of Microcontroller
Processor
Internal RAM
Timers & watchdog timer
Internal Flash/ROM
External memories Interface
Interrupt controller
Input ports control & Interfaces/ Drivers
Serial UART communication port
Serial Synchronous Communication port
Application specific circuits in specific versions
DMA Controller
Network Drivers tack & Interfaces
A/D Converter
CAN controller
PWM circuit for D/A
Printer Controller
Modem
DTMF circuit
Dual-tone Multi-frequency Signaling ckt

fig:- Functional circuit diagram for Embedded system.

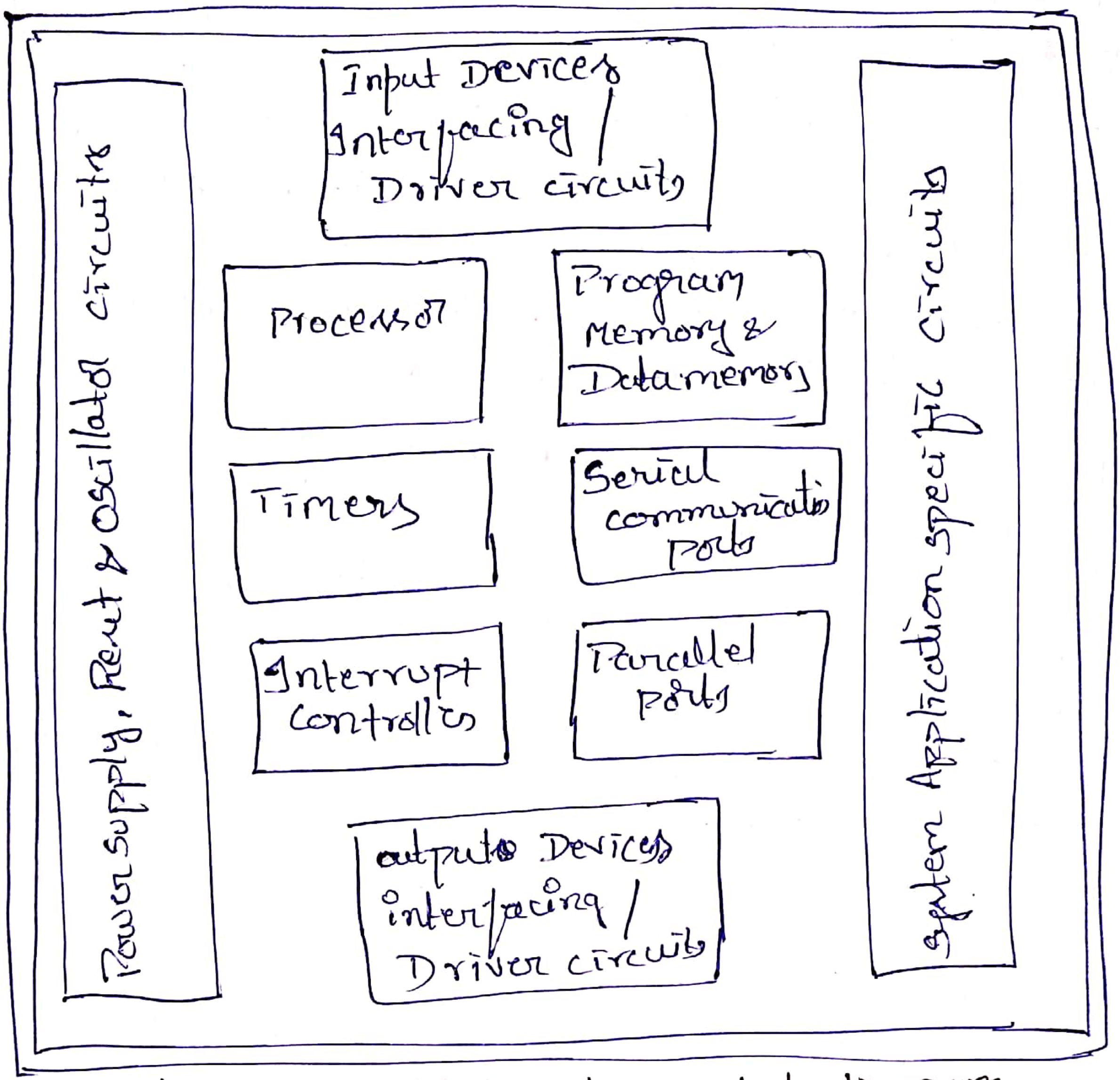

Input Devices Interfacing / Driver circuits
Processor
Program Memory & Data memory
Timers
Serial communication ports
Interrupt Controllers
Parallel ports
outputs Devices interfacing / Driver circuits
Power supply, Reset & oscillator circuits
system Application specific circuits

fig: Embedded system Block diagram

★ Processor:- Processor is similar to CPU it perfors four functions such as Fetch, Decode, Executes & stores the results

- It Fetchs instructions from memories after fetchs instructions from the memory it decodes the instructions which is fetched from the memory, After decode it Executes the instructions, After Executes it stores the instructions.

- Any processor has two essential units.
  ① Bus Interface unit {BIU}; ② Execution unit {EU}:

- BIU:- It basically deals with interfacing operations and take care of every operations related to a BUS.

- It includes a fetch unit for fetching instructions from the memory.

- EU:- It takecare of every operations of executing an instructions. It consist of Registers, st
- ~~Exe~~It has circuits that implement the instructions pertaining to data transfer operation and data conversion from one form to another form.
- EU Includes the ALU and also the circuits that execute instructions for program.

★ Program memory and data memory :-

memoryes can clarified as two types.
such as (1) Program memory (2) Data memory

★ Program memory:-

- Also called as ~~RAM~~ ROM
- ~~It is~~ ROM is Nonvolatile memory [Retains its contents when power is removed]
- It can't be written or modified at run time

★ Data memory:-

- Also called as RAM
- RAM is a volatile memory [Loses its contents when power is removed]
- Data can be read or written with equal care
- It has ability to access any memory cell directly
- RAM is much faster than ROM.

- There is another memory is there i.e cache memory, it is very faster than RAM.

★ Power Supply circuit:-

- Power supply is nothing but the supply which is given to the circuit.

★ clock oscillator:- The function of this oscillator circuit is to provide a accurate and stable periodic circuit signal to the processor.

★ Reset circuit:- In many embedded systems reset by a watchdog timer, is very essential because it helps ir rescuing the system from program hangs.

★ **Timers:-**

- Timers are generaly used to generate Time delays.
- It also be made to work as counter.
- Timers have many use cases. Such as Creating accurate delays, executing a periodic task, implementing a PWM o/p or capturing the elapsed time between two events, to vary the speed of data transfer rate i.e Baud rate in case of serial commu etc.

★ **Interrupt controller:-**

- set of special instructions which has to be executed at that perticular point of time.
- Manages all the interrupt requests on a priority basic and provide individual interrupt service by CPU.

★ **Input and output Ports:-**

- Port is nothing but an external pin which is used to interface a device to this processor.

★ **Interfacing:-**

★ Serial communication:- Transfer data bit by bit.

★ Parallel Communication: Transfer bulk of data.

★ Driver Circuit:- Which is use to controll other Circuits

⇒ Embedded Processors and their types:→

- Processor is the heart of an embedded system. It is the basic unit that takes inputs and produces an output after processing the data.
- For an embedded system designer, it is necessary to have the knowledge of both microprocessors & microcontrollers
- A processors has two essential units.
  - Program Flow control unit (CU) / BIU
  - Execution Unit [EU]

- Control unit:- This unit in processors performed the program flow control operation inside an Embedded system. The control unit also acts as a fetching unit for fetching the set of instructions stored inside a memory.
- Execution unit:- This unit is used for execution the various tasks inside a processors. It mainly comprises of arithmetic and logical unit (ALU) and it also include a circuit that executes the instruction sets used to perform program control operation inside the processors.
- Processors inside an embedded system are of the following categories.

*(1) General purpose processor:- [GPP]:-

- GPP is used for processing signal from input to output by controlling the operation of the system bus, address bus and data bus inside an Embedded system.
- GPP with instruction set designed not specific to the applications.
  - Microprocessor
  - Embedded processor

*(2) Application specific Instruction set processor [ASIP]:-

- An ASIP is a processor with an instruction set designed for specific applications on a VLSI chip.
- ASIP is application dependent instruction processors.
- It is used for processing the various instruction set inside a combinational circuit of a an Embedded system.
  - Microcontroller
  - Embedded ~~controll~~ microcontroller
  - Digital signal Processor (DSP)
  - media processor.

*(3) Application specific System Processor [ASSP]:-

- ASSP is application dependent system processor used for processing signal of Embedded system. Therefore for different application performing task a unique set of system processors is required.

*④ single purpose processors [SPP]:-

- This type of processor is designed to execute exactly one program. An embedded designer creates a single purpose processor by designing a custom digital circuit.
- SPP are used for specific applications or computations or as controllers for peripherals, direct memory accesses and buses.
- SPP used in Embedded systems include:

1. Coprocessor {Ex: Floating point processing].
2. Graphics processor: An image consists of a number of pixels.
3. Pixel coprocessor: High-resolution pictures formats.
4. Encryption engine: A suitable algorithm runs in this processor to encrypt data for secure transmission.
5. Decrption engine: A suitable algorithm runs in this processor to ~~encrypt data for secure~~ decrypt the encrypted data at receiver's end.
6. DCT & DCIT = {A discrete cosin transformation & invers transformation] processor: is required in speech & video processing.
7. Protocol stack processor: which has a number of header words, is prepared before an application data is send to network.
8. Network processor: Functions are to establish a connection, finish, send & receive acknowledgements, send and receive retransmission requests and check and correct received data frame errors.
9. CODEC {Coder & Decoder]:- is a processor circuit that encodes input an decodes the encoded information or bits or signals into a complete set of bits or original signal.

★⑤: GPP core or ASIP core:-

- GPP core or ASIP core is integrated into either an ASIC [Application specific Integrated circuit] or a VLSI or an FPGA core integrated with processor units.
- For a number of applications GPP core may not be a suitable solution.
- For various security application, smart card, video game, mobile Internet, Gbps transceiver, Gbps LAN, missile system needs a special processing unit on a VLSI design circuit to function as a processor.
- These units are called Application Specific Instruction Processor. somtime for an application both configurable processor (FPGA or ASIP) and non-configurable processor (DSP or μP or μC) might be needed on a chip. Generally this type of applications are vary important in some killer applications such as HDTV, cell-phone etc. [applications which is useful to millions of people].

★⑥ Multicore processors or multiprocessor:-

- As embedded algorithm has to work within strict deadline, sometimes it may not be possible to carry out the same with a single processor.
- In such a case an embedded system may go for two or more processors. similar requirement may be needed in modern cell phones which has to perform number of tasks.
- Multiprocessors are used when a single processor doesn't meet the need of the different tasks that have to be performed concurrently.
- The operations of all processors are synchronized to obtain an optimum performance.

★ Microprocessor:- μP 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 MC 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 no of applications

★ Embedded Processor:-

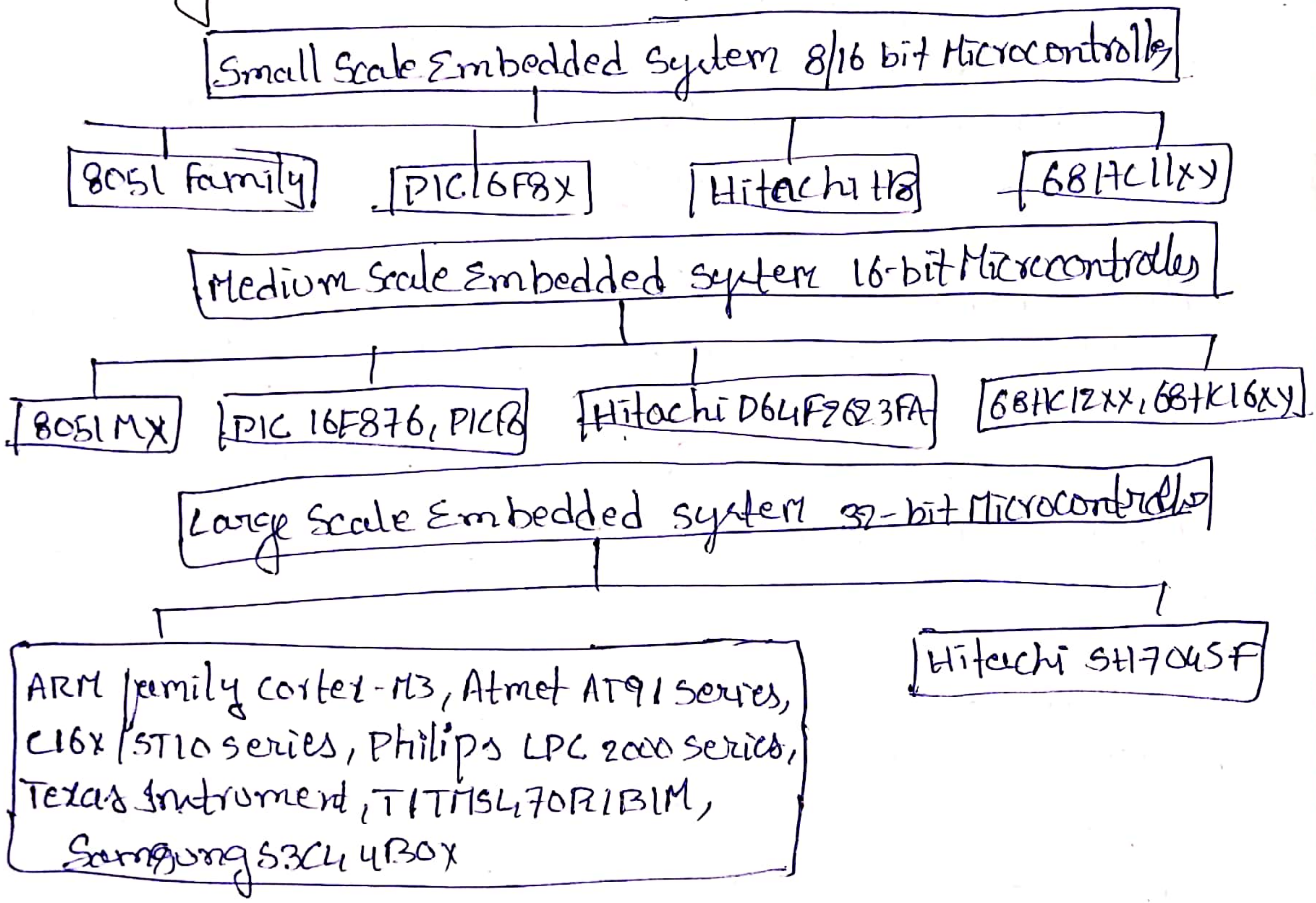
- Embedded processors are specifically designed for embedded systems to meet design constraints.
- They have the potential to handle multitasking applications.
- The performance and power efficiency requirements of embedded systems are satisfied by the use of embedded processors.

★ DSP: DSP are used for signal processing applications, such as voice or video compression, data acquisition, image processing or noise and echo cancellation.

★ ASIC: ASIC is basically a proprietary device designed and used by a company for a specific line of products. It is specifically an algorithm called intellectual property core implemented on a chip.

★ FPGA: 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.

- Commonly used microcontrollers in small-, medium- and large-scale embedded systems.



Small Scale Embedded System 8/16 bit Microcontrollers

8051 family

PIC16F8X

Hitachi H8

68HC11xy

Medium Scale Embedded system 16-bit Microcontrollers

8051MX

PIC 16F876, PICF8

Hitachi D64F2623FA

68HC12XX, 68HC16xy

Large Scale Embedded system 32-bit Microcontrollers

ARM family cortex-M3, Atmel AT91 series, C16x / ST10 series, Philips LPC 2000 series, Texas Instrument, TITMS470R1B1M, Samsung S3C44B0X

Hitachi SH7045F

Memory Types :-

• Memory is the most essential element of a system because without it system can't perform simple tasks.

• The semiconductor memory can be classified into two types.

① Volatile Memory
② Non-Volatile Memory

*** Volatile memory:-**

→ RAM
- Static RAM
- Dynamic RAM

*** Non-Volatile memory:-**

→ ROM
- Masked ROM
- OTPROM | PROM
- EPROM
- EEPROM
- FLASH

→ NVRAM (Battery Backup RAM)

*** RAM (Random Access Memory):-**

• It is also called as "reade write memory" or the main memory or the primary memory.

• The programs and data that the CPU requires during execution of a program are stored in this memory.

• It is a volatile memory as the data loses when the power is turned off.

• It has ability to access any memory cell directly.

• Volatile memory is used for data, and small microcontrollers often have very little RAM. RAM is much faster than ROM

• RAM is further classified into two types.

- SRAM {Static Random Access Memory}
- DRAM {Dynamic Random Access Memory}.

*** SRAM:-**

• The SRAM are memories that consist of circuits capable of retaining the stored information as long as the power is applied.

- That means this type of memory requires constant power.
- SRAM memories are used to build cache memory.
- A single cell of SRAM needs six transistors. RAM therefore takes up a large of silicon, which makes it expensive

★ Dynamic RAM:-

- DRAM stores the binary information in the form of electric charges that applied to capacitors.
- The stored information on the capacitors tend to lose over a period of time and thus the capacitors must be periodically recharged to retain their usage. The main memory is generally made up of DRAM chip
- This needs only one transistor per cell but must be refreshed regularly to maintain its contents, so it is not used in small microcontrollers.
-

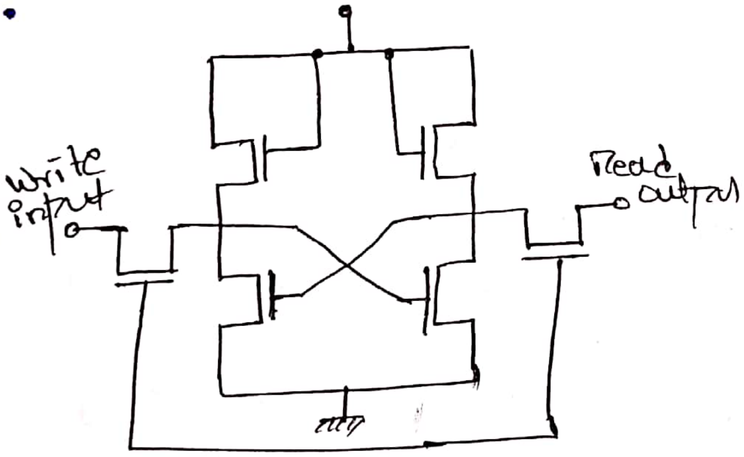


fig:- SRAM ~~porcell~~ single cell

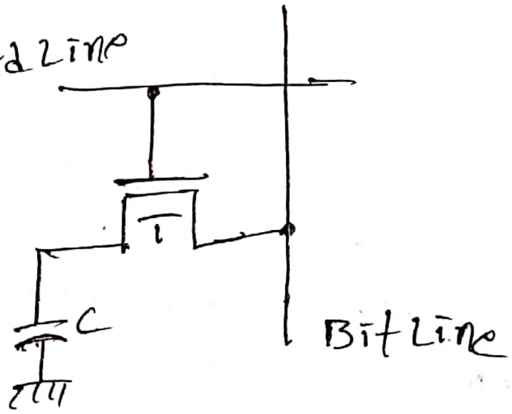


fig:- Single cell DRAM.

- There are mainly 5 types of DRAM:

★1. ADRAM [Asynchronous DRAM]:

- The DRAM described above is the asynchronous type DRAM.
- The timing of memory device is controlled asynchronously.
- A specialized memory controller circuit generated the necessary control signals to control the timing.

★2. SDRAM [Synchronous DRAM]:-

- These RAM chips access speed is directly synchronized with the CPU's clock. For this, the memory chips remain ready for operation when the CPU expects them to be ready.

- These memories operate at the CPU-memory bus without imposing wait states.
- SDRAM is commercially available as modules incorporating multiple SDRAM chips and forming the required capacity for the modules.

***3 DDRSDRAM {Double-Data-Rate SDRAM]:**

- This faster version of SDRAM performs its operations on both edges of the clock signal; whereas a standard SDRAM performs its operations on the rising edge of the clock signal
- ∴ they transfer data on both edges of the clock, the data transfeer rate is doubled.
- To access the data at high rate, the memory cells are organized into two groups. Each group is accessed separately.

***(4) RDRAM {Rambus DRAM]:-**

- The RDRAM provides a very high data transfer rate over a narrow CPU memory bus.
- It uses various speedup mechanisms, like synchronous memory interface, caching inside the DRAM chips and very fast signal timing.
- The Rambus data bus width is 8 or 9 bits. (5) CDRAM cache DRAM

SRAM	DRAM
(1) SRAM has lower access time, so it is faster compared to DRAM	(1) DRAM has higher access time, so it is slower than SRAM.
(2) SRAM is costlier than DRAM	(2) DRAM costs less than SRAM
(3) SRAM requires constant power supply, which means this type of memory consumes more power	(3.) DRAM offers reduced power consumption, due to the fact that the information is stored in the capacit
(4) Due to complex internal circuitry less storage capacity is available compared to the same physical size of DRAM memory chip.	(4) Due to the small internal circuitry in the one-bit memory cell of DRAM, the large storage capacity is available
(5) SRAM has low packaging density.	(5) DRAM has high packaging density.

* ROM [Read-Only Memory]:-

- It is not volatile memory. Always retains its data.
- Used in embedded systems or where the programming needs no change.
- 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

* Masked ROM:-

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

* OTPROM (One-Time Programmable ROM):-

- This is just PROM 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 OTPROM 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. Device must be exposed to ultraviolet (UV) light for about ten to twenty minutes to erase them.
- We can erase the contents of the chip and rewrite it with new contents, typically several thousand times.

[UV light exposed, can erase all the previous data]

* EEPROM {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 tim (similar to RAM), but remains permanently saved even if the power supply is off (similar to ROM.)

17/10

- EEPROMs are also limited to the number of erase-writes that can be performed (e.g., 100,000) but support updates (erase-writes) to individual bytes. whereas EPROM ~~up~~

★ FLASH Memory :-

- This can be both Programmed and erased electrically & is now by far the most common type of memory.
- Flash Memory is designed for high speed & high density, at the expense of large erase blocks.
- The practical difference is that individual bytes of EEPROM can be erased but FLASH can be erased only in blocks.
- 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

★ characteristics of the various memory types:-

Type	Volatile?	Writeable?	Erase size	Max Erase cycles	Cost (Per Byte)	speed
SRAM	Yes	~~NO~~ yes	Byte	unlimited	Expensive	Fast
DRAM	Yes	yes	Byte	Unlimited	Moderate	Moderate
Masked ROM	NO	NO	n/a	n/a	Inexpensive	Fast
PROM	NO	once, with a device Programmer	n/a	n/a	Moderate	Fast
EPROM	NO	Yes, with a device Programmer	Entire chip	Limited [consult datasheet]	Moderate	Fast
EEPROM	NO	Yes	Byte	Limited [consult datasheet]	Expensive	Fast to read, slow to erase/write.
Flash	NO	yes	Sector	Limited [consult datasheet]	Moderate	Fast to read Slow to erase/write
NVRAM	NO	Yes	Byte	unlimited	Expensive (SRAM + battery)	Fast

⇒ Overview of design process of embedded Systems :-

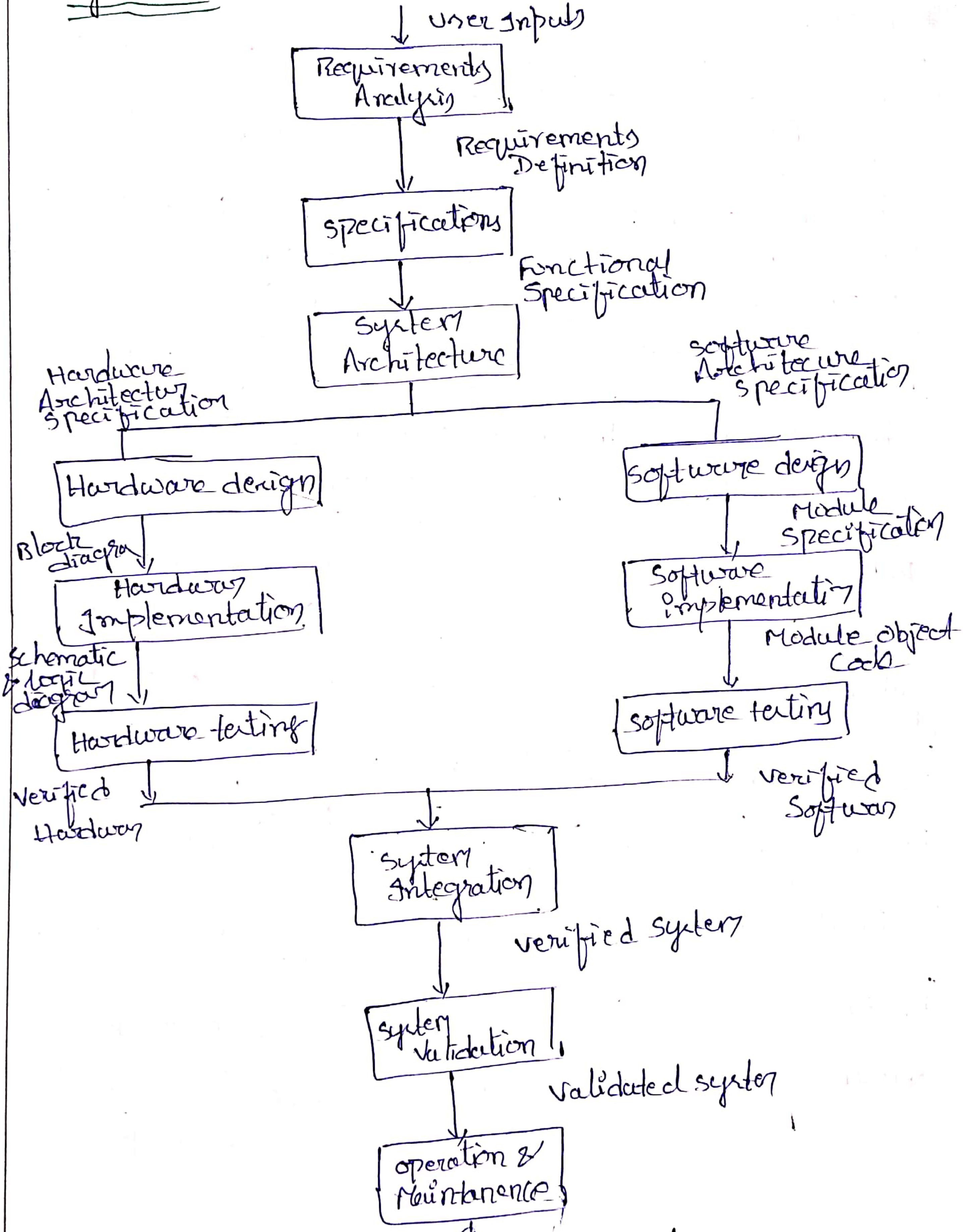


fig : Embedded System life cycle.

- The important ~~st~~steps in developing an embedded system are.

1- Requirements
2- Specifications
3- Architecture
4- Components
5- System Integration

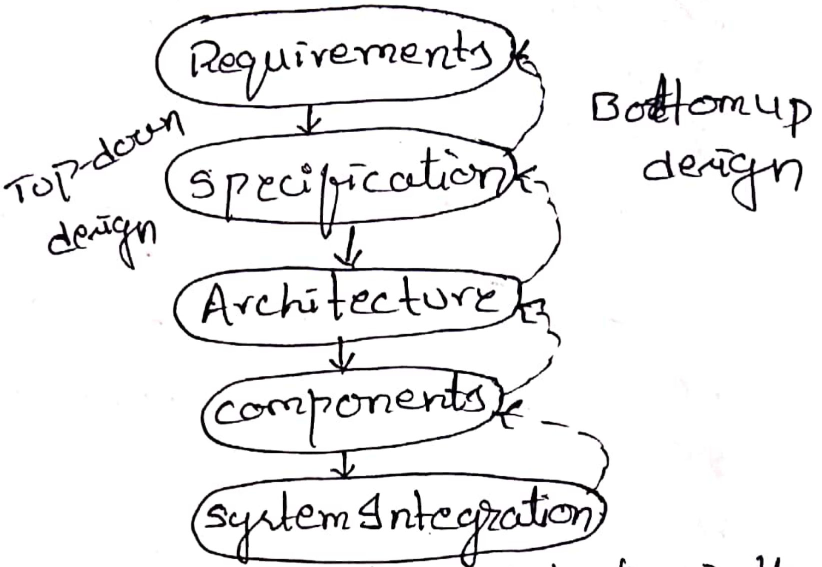


fig: major levels of abstraction in the design process.

- There is two kind of methodologys using it,
  1- Top-down design methodology [
  2- Bottom-up design methodology steps are shown in the fig as dashed-line arrows]

★ Top-down design:- Begin with the most abstract description of the system and conclude with concrete details

★ Bottom-up design:- Start with components to build a system, Ending up with perticular requirements.

- Most of the people prefor bottom-up ~~approa~~ design because we do not have perfect insight into how later stages of the design process will turn out.

★ Major goals of the design to be considered:
  - manufacturing cost
  - Performance (both overall speed and deadlines)
  - Power consumption.

★ Tasks which needs to be performed at each step:
  - We must analyze the design at each step to determine how we can meet the specifications.
  - We must refine the design to add detail.
  - We must verify the design to ensure that it still meets all system goals, such as cost, speed, & so on..

① Requirements:-

- Informal description gathered from customers is known as "requirements"
- Requirements can be of two types
  - Functional requirements
    - It need output as a function of input
  - Non-functional requirements.
- Some of the non-functional requirements

*** Power consumption:-**

- In the requirement stage power can be specified in terms of battery life.
- However, the allowable wattage can't be defined by the customer.

*** Physical size & weight:-**

- Depending on the application, the physical size & weight of the final system can vary a lot.
- If the application involves a handheld device then there are constraints on both the size & weight of the device.
- However, if it is an industrial control system then there is no constraints on the size & weight.

*** Performance:-**

- The cost and usability of the system are effected by its speed. The performance metrics can be combination of soft metrics and Hard metrics.

*** Cost:-**

- The system's purchase price or the target cost is very important. complete cost or simply cost includes the following two components
- Manufacturing cost : it is the cost of assembling and the components
- NRE costs [Non-Recurring Engineering]:- NRE costs are the cost of hiring personal and other design related costs.

* Requirements form or Requirements chart:-

- The requirement form/chart acts like a check list when the project is in initial stages.

Ex:- A sample form is given below

Name:
Purpose: what the system supposed to do
Inputs:
outputs:
Functions: functionality of the system
Performance:
manufacturing cost:
Power:
physical size & weight:

Fig: Sample requirements form.

- Name:- Naming not only describe the purpose of the machin but also helpful while commiting about the project.
- Purpose:- This should be a brief one or two line description of what the system is supposed to do.
- Inputs and outputs:- This field requires information about type of I/O devices, Data characteristics & types of data.
- Functions:- A detailed description about the functionality of the machine is described in this field.
- Performance:- In order to assume proper functionality, Performance requirements should be identified before hand and they must be measured carefully.
- Physical size & weight:- In order to take architectural decisions the approximate physical size and weight of the system is important.
- Power:- The rough idea about power consumption can very helpful. Decision about whether the system is battery based or non-battery is important here.
- Example: consider an example of a GPS moving map system sample requirement chart.

Name: GPS Moving map
purpose: Consumer-grade moving map for driving use

Inputs: 2 control buttons and Pawer buttons

outputs: Back-list LCD display with 400x600 pixels

Functions: User 5-receiver GPS system, 3 user-seletable resolutions & displays current longitude & lattitude.

Performance: updates screen within 0.25 seconds upon movement.

Manufacturing cost: RS 2000

Power: 100mw

Physical size & weight: Not more than 2"x 6" & 12 ounces

② Specification:

- Specification serves as the contract between the customers and the system architects.
- Specification is essential to create working systems with a minimum of designer effort.
- It must be specific, understandable and accurately reflect the customer's requirements.
- Example: considering the example of the GPS system, the specification would include details for several components.
  - Data receiver from the GPS satellite constellation
  - Map data
  - User interface
  - operations that must be performed to satisfy customer
  - Back ground actions.
- Differences b/w Requirements & Specifications:

Requirements	Specifications
① An informal description garthered from customer	① A contract between customer & system architects
② Requirement form is used to give a formal listing of requiremen	② UML (Unified Modeling Language) is used to give clear & proper specifications.
③ The basic needs to design a system are given by system requirements	③ The description of project is given by system specifications

(4) Less challenging Task	(4) More complex & challing task
(5) Requirements need not be perfect	(5) specifications should be perfect enough in order to develop correct application.
(6) Good psycological skills are required to produce good system requirements	(6) Good Engineering skills are required to produce good specifications

(3) Architecture Design:-

- The specification describes only the functions of the system; Implementation of the system is described by the Architecture.
- The architecture is a plan for the overall structure of the system
- It will be used later to design the components.

Example:

- A basic block diagram of the GPS system shows the major operations and the data flow among the blocks

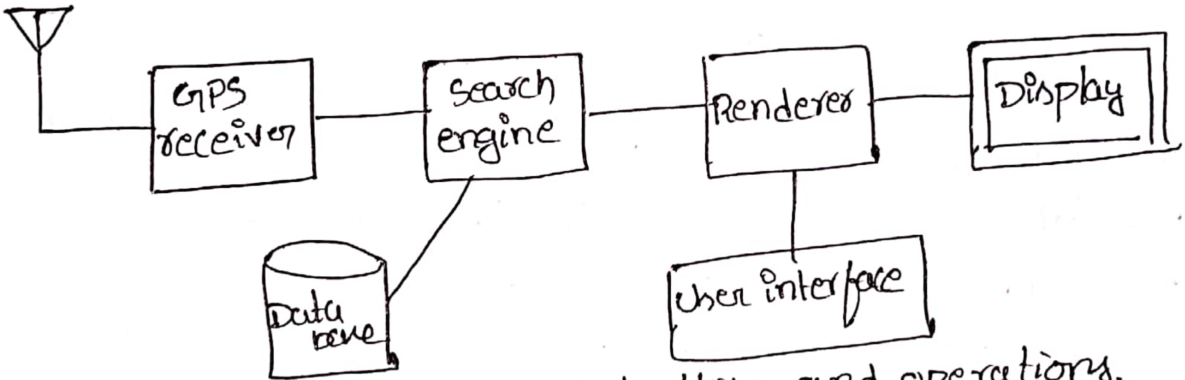


fig: GPS system data flow and operations.

- This block diagram is an initial architecture that is not based either on hardware or on software but combination of both.
- This block diagram explains about GPS navigating system where GPS receiver gets current position and the destination is taken from user, digital map for source to destination is found from database and displayed by the renderer.
- The system block diagram may be refined into two block diagrams - hardware and software.
- The refinement of Hardware and software architecture should begin only after the initial architecture is designed.
-

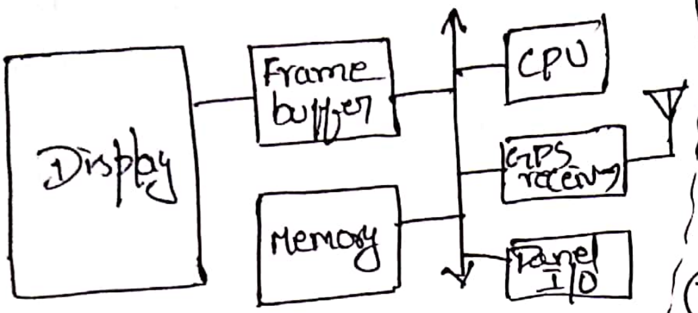


fig: GPS system Hardware

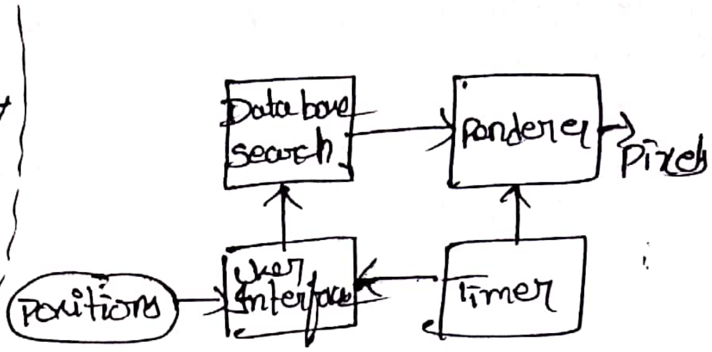


fig: GPS system Software.

- Hardware consists of one central CPU sorrounded by memory and I/O devices.
- We have chosen to use two memories that is frame buffer holds the pixels to be displayed and memory for program (or) data which will be used by CPU.
- A Bus is used to connect all these components
- The software block diagram is same as initial architecture but an additional timer is added. This timer is to control when we read the buttons on the user interface and render data onto the screen.
- Finally the Architectural descriptions must be designed ~~to~~ should
  - Satisfy user requirements [Functional & non-functional]
  - Include all the required functions.
  - Should meet speed, cost, Power requirements
  - meet all the specifications

④ Designing Hardware and Software components:-

- The architectural description tells us what components we need.
- The component design effort builds those components in conformance to the architecture and specification.
- The components will be in general includes both Hardware and software modules.
- Some of the components will be ready-made (example: CPU, memory chips)
- Some of the components must be designed by user's also which require lot of custom programs.

(5) System Integration:→

• The system integration deals with the integration or assembly of the components created.

• We know that Hardware and software made seperatlly and people who are working with Hardware/software ~~by~~ they made it perfectly worked out well. But when we are integration these things with simulator, simulator does not read the real time problems so that Bugs are typically found during system integration.

• However, good planning, phase level development, good tests running at each phase can assid in finding bugsquickly

• By debugging few simple bugs early, more complex or other bugs can be uncovered.

• System integration is difficult because it usually uncovers problems.

• The debugging facilities for Embedded systems are usually much more limited than the desktop systems.

• Careful attention to inserting appropriate debugging ~~to~~ facilities during design can help ease system integration problems.

⇒ 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.

★ Assembly language:-

- Little more than machine code translated into English.
- The instructions are written as words called mnemonics rather than binary values.
- 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.
- Most programming of small microcontrollers was done in 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.

★ C++:-

- An object-oriented language that is widely used for larger devices. A restricted set can be used for small microcontrollers bus 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. 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:-

★ Editor:- It enables users to write codes for high level as well as assembly language is computer.

• Different features like addition, deletion, copy, insertion are made available for easy writing.

• It saves the content in a file with user defined or default extension. User can make necessary modification of saved. files as and when required.

★ Compiler:- It takes the input of whole high level source code and converts it to machine readable object code.

• It may include functions, library routines etc. for compitation.

★ Interpreter:- It converts high-level codes to machine ~~reada~~ readable form line by line. Like compiler it may also include functions, library routines etc. for conversion.

★ Assembler: It is used for conversion of assembly language programs to executable binary files.

• It creates the list file which has address, source code and hexadecimal object codes. It is processor specific.

★ Cross assembler: Cross assembler assembles the assembly code of target processor as assembly code of the processor of the PC used in the system development. Later it provides the object codes for the target processor. These will be the final codes used for the developed system.

★ Simulator: It is the program which can simulate all the functions of an embedded system circuit including additional memory and peripherals. It is independent of a particular target system.

★ Stethoscope: This program is used to keep track of dynamic change in program variables and parameters. It can demonstrate the sequences of multiple processes, tasks, threads that execute and keeps entire time history

★ Trace Scope: It traces the change in module according to time. Accordingly list of actions to be initiated at desired time is also prepared.

★ IDE {Integrated Development Environment: Total software and hardware environment consist of simulator, compiler, assembler, cross assembler, logic analyser EPROM/EBPROM, application codes, burners defines the integrated development environment of the system.

★ Locator: Locator program uses cross-assembler output and a memory allocation map and provides locator program output

★ characteristics of a programming language:-

- A programming language must be simple, easy to learn and use, have good readability & human recognizable.
- Abstraction is a ~~momust~~-have characteristics for a programming language in which ability to define the complex structure and then its degree of usability comes.
- A portable programming language is always preferred.
- Programming language's efficiency must be high so that it can be easily converted into a machine code and executed consumes little space in memory.
- A programming language should be well structured and documented so that it is suitable for application development
- Necessary tools for development, debugging, testing, maintenence of a program must be provided by a programming language.
- A programming language ~~must be consistent~~ in ~~terms of~~ should provide single environment known as IDE.

EMBEDDED SYSTEM

UNIT-II

Embedded Processor Architecture

UNIT-II

Embedded Processor Architecture

⇒ CISC Vs RISC design philosophy:—

- The architecture of the CPU operates the capacity to function from "Instruction set architectorade" to where it was designed.
- Instruction set can be defined as the communication interface between the processor and the programmer.
- Every processor has its own indruction set implemented in the hardware to execute instructions.
- Programmers can either use any 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 architectures of instruction set implementation

① RISC [Reduced Instruction Set computer]

② CISC [Complex Instruction Set computer].

★ RISC:-

- A reduced instruction set computer is a computer which only uses simple commands that can be divided into several instructions which achieve low-level operation within a single CLK cycle, as ~~its~~ ~~n~~ [It is designed to reduce the Execution time by smplifying the instruction set computer].

★ CISC:-

- A complex instruction set computer is a computer where single instructions can perform numerous low-level operations like a load from memory, an arithmetic operation, and a memory store or are accomplished by multi-step processes or addressing modes in single instructions; ~~as its name proposes~~

[It is designed to minimize the number of instruction program]

★ Differences between RISC & CISC:→

RISC	CISC
① RISC Stands for Reduced Instruction Set computer.	① CISC stands for complex Instruction set Computer

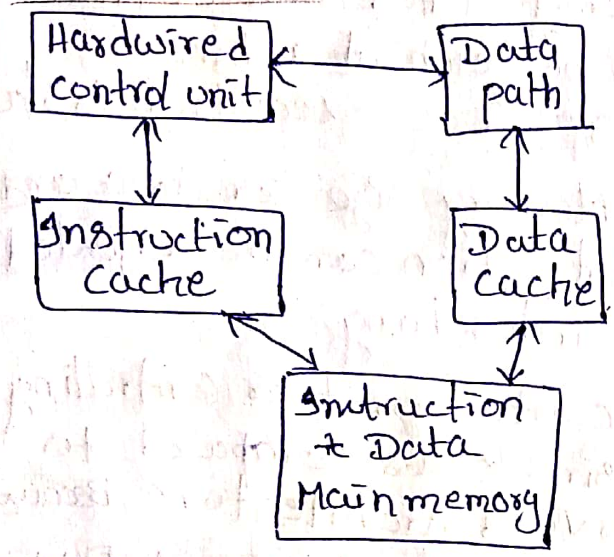
RISC	CISC
(2) RISC processors have simple instructions taking about one clock cycle. The average clock cycle per instruction is 1.5	(2) CSIC processor has complex instructions that take up multiple clocks for execution. The average clock cycle per instruction is in the range of 2 & 15.
(3) Performance is optimized with more focus on Software	(3) Performance is optimized with more focus on Hardware.
(4) It has no memory unit & uses a separate Hardware to implement intructions.	(4) It has a memory unit to implement complex intructions
(5) It has a Hardwired unit of programming.	(5) It has a microprogramming unit
(6) The instruction set is reduced i.e it has only a few intructions in the intruction set. Many of these intructions are very primitive.	(6) The instruction set has a variety of different intructions that can be used for complex operations.
(7) Complex Addressing modes are synthesized using the software	(7) CISC already supports complex Addressing modes
(8) Multiple register sets are ~~pres~~ present.	(8) Only has a single register set.
(9) RISC processors are highly pipelined.	(9) They are normally not pipe-lined or less pipelined
(10) The complexity of RISC lies with the compiler that Executes the program	(10) The complexity lies in the microprogram.
(11) Execution time is very less.	(11) Execution time is very high.
(12) Code expansion can be a problem	(12) Code expansion is not problem
(13) Decoding of instructions is Simple	(13) Decoding of instructions is complex.
(14) It does not require external memory for calculations	(14) It requires external memory for calculations.
(15) RISC Architecture is used in high end applications	(15) CISC Architecture is used in low end applications.

such as video processing, telecommunication and image processing.

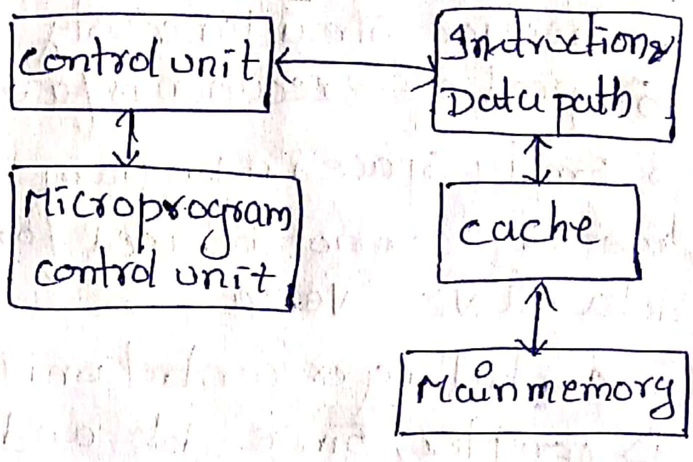
such as security system, home automation, etc.,

⇒ Von-Neumann Vs Harvard architecture:→

Architecture



Architecture:



⇒ Von-Neumann Vs Harvard Architecture :—

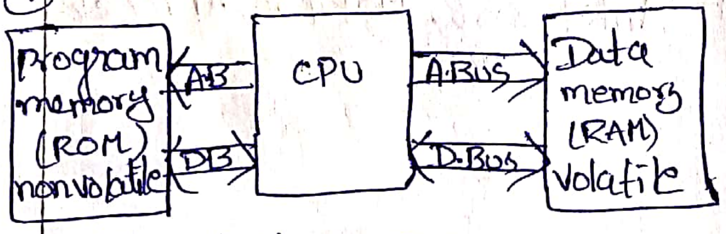
- Every CPU requires RAM and ROM, The arrangement of RAM & ROM / Data & program memory with respect to CPU that is known as the Architecture of the microcontroller and there are two techniques the CPU can connected to the data and program memory.

① Harvard Architecture

② Von-Neumann Architecture

Harvard Achitecture

①



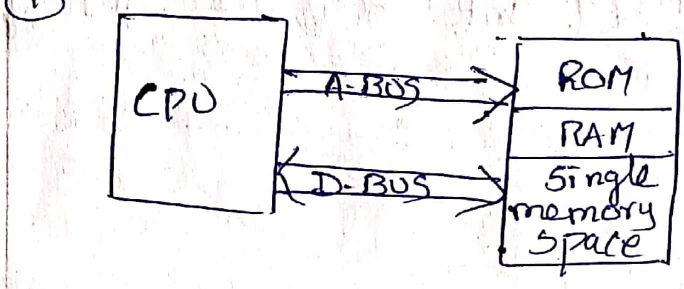
AB → Address BUS

DB → Dada BUS

① It has sperate data and program memory.

Von-Neumann Architecture

①



① It has common data and program memory.

(2) It requires more Hardware, Because seperate data and address bus for Each memory.	(2) It requires less hardware because as requires only one data and address bus
(3) It Requires more space	(3) It Requires less space
(4) Processor can fetch data and instruction simultaneously.	(4) Only either data or instruction can be fetch at a time
(5) Speed of Execution is fast	(5) Speed of Execution is slow
(6) Empty space in program memory cannot be used for data & vic versa	(6) Memory size for data and Instruction can be adjented or interchange
(7) Controlling or control unit is complex, Since data and instruction are fetch simultaniously	(7) control unit or controlling is simple as compared to Harvard Architecture, Because Single Address & Data Bus for fetch data & Instruction.
(8) Single set of clock cycle sufficient	(8) 2-set of clock cycles requiring 1 cycle for data fetch & 1 cycle for intruction feth.
(9) Pipelling is possible	(9) pipelling is not possible
(10) Difficult for program contents to be modified by the program itself	(10) program contents can be easily modified by itself
(11) Widely used in MicroController	(11) widely used in Microprocessor
(12) Eg: Intel 8051, PIC, ARM9 etc.	(12) Intel 8086, MSP430, ARM7 etc.

⟹ Introduction to ARM architecture and cortex – M series:

*Introduction:-

- The ARM-cortex microcontroller is a most popular microcontroller in the digital embedded system world
- It consists of enormous features to implement products with an advanced appearance
- The ARM stands for "Advanced RISC machine" and it is 32-bit reduced instruction set computer micro-controller.
- It was first introduced by the "Acron computers" organization in 1987.

* Features	Benefits to Embedded system
(1) High Performance	ensure 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 provide faster context switch in a multitasking Application.

- The ARM microcontroller architecture come with a few different versions and each one has its own advantage and disadvantages.

* ARM7: has three-stage (fetch, decode, execute) pipeline, Von-Neumann architecture where both address and data use the same but. It executes V4T instruction set. T stands for Thumb.

* ARM9: has five-stage {fetch, decode, execute, memory & write] pipeline with high performance, Harrvard architecture with seperate instruction and data bus. ARM9 executes V4T & V5TE instructions 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 instruction set. With the addition of vector floating point unit, it performs fast floating point operations.

- The ARM cortex microcontroller is a advanced microcontroller in the ARM family, which is developed by the ARMv7 architecture.

* A Basic architecture of the ARMv7:-

Block diagram shown in next page

- The Register Bank has sixteen general purpose registers (R0-R15) and a CPSR [Current Program status Register] 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~~ :

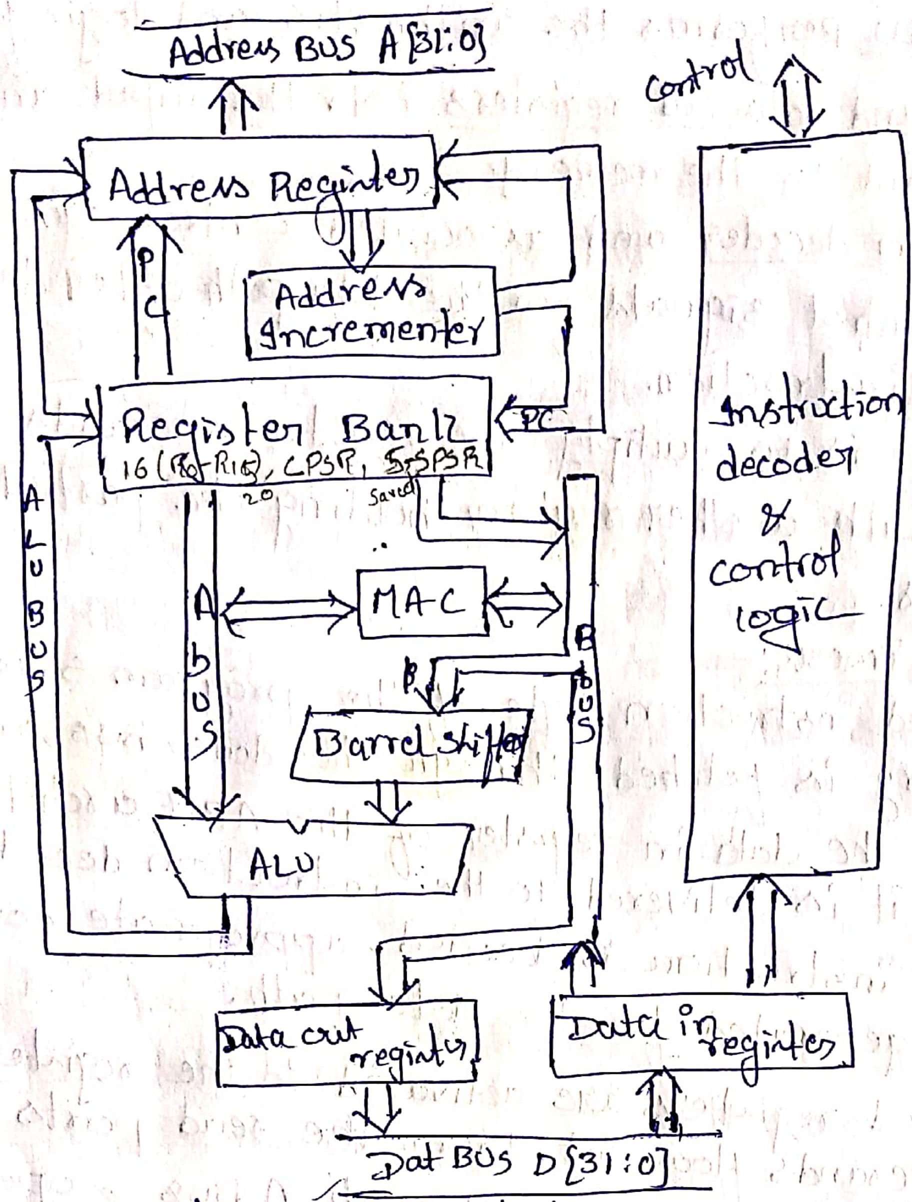


fig- ARM7 Architecture.

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 ~~stat~~ instruction set of the core or is incremented by 2 in Thumb instruction set 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 instuction decoder and associated control logic generates approprite control signals for the data path after decoding the fetched instruction.
- The MAC unit (Multiply & Accumulate) is to multiply two register operands and accumulate with another register holding the partial sum of the products.

→ ~~The execute~~ Process:-

- 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 the read ports of register bank to the ALU: operand1 through A-Bus & operand2 through B-Bus after preprocessing at barrel shifter.
- The result of operation at ALU is ~~retu~~ 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.

* The ARM cortex family divided into three sub-families such as: ① ARM-Cortex Ax-series [Application profile]
② ARM-Cortex Rx-series [Real-Time profile]
③ ARM-Cortex Mx-series [Microcontroller profile]

* ARM-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.
- 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 systems.
- Some cortex-A application products are smar phones, tablets, televisions and even high end computing servers.

* ARM-cortex-R :-

- Cortex-R series of architectures are designed for deeply embedded real time multitasking applications.
- They have low interrupt latency and predectability 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.
- ~~Ex~~ Typical application examples are: Hard disk drive controller, base band controller for mobile applications & engine management unit where high performance & reliability at very low interrupt latency and determinism are critical requirements.

★ ARM Cortex-M:-

- Cortex M series of architectures have v6-M as Cortex M0, M0+ & M1 & v7-M with cortex M3, M4 & other successors.
- This series of architectures developed for deeply embedded microcontroller profile, offer lowest gate count so smallest silicon area.
- There 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 at 8-bit price.

★ ARMv6-~~A~~ ~~cor~~Cortex-M:-

★ cortex M0:-

- The cortex-M0 core is optimized for small silicon die size and use in the lowest price chips
- Key features of the cortex-M0 ~~cor~~core are:.
  - which is developed by ARMv6-M architecture
  - It has 3-stage pipeline
  - Executes Instruction sets are
    - Thumb-1 [most, missing CBZ, CBNZ, IT]
    - Thumb-2 [some, only BL, DMB, DSB, ISB, MRS, MSR]
  - performs - 32 bit Hardware integer multiply with 32-bit result
  - 1 to 32 interrupts, plus NMI.

silicon options:
- Hardware integer multiply speed: 1 or 32 cycles

★ cortex M0+:-

- The cortex-M0+ is an optimized superset of the cortex-M0.
- The cortex M0+ has complete instruction set compatibility

with the cortex-M0 thus allowing the use of the same compiler and debug tools.

- key features of the cortex-M0+ core are:
  - Which is developed by ARMV6-M architecture
  - It has 2-stage pipeline (one fewer than cortexM0)
  - Executes Instruction sets same as cortex-M0
    - Thumb-1 {most,
    - Thumb-2 {some
  - performs 32-bit hardware integer multiply with 32-bit result
  - 1 to 32 interrupts, plus NMI

silicon options:
  - Hardware integer multiply speed: 1 or 32 cycles
  - It provides 8-region MPU [Memory protection unit].
  - Provides Vector table relocation
  - Provides single-cycle input/output port.
  - Additionally provide MTB [MicroTrace Buffer].

★ ARMv7 Cortex-M:-

- key features for ARMv7-M architectures are:
  - Enable implementations with industry excelling Power, performance and silicon area contraints 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 & Cortex M4, which are developed by ARMv7-M architecture with enhanced key features.

★ Cortex-M3:-

- cortex M3 is the first architecture of ARMv7-M profile)
- Cortex M3 is general purpose CPU, has optimized debug options for microcontroller applications.
- key features of the cortex-M3 core are:
  - Which is developed by ARMv7-M architecture.
  - It has 3-stage pipeline with branch speculation
  - Executes Instruction sets are
    - Thumb-1 (entire)
    - Thumb-2 (entire)
  - Performs 32-bit hardware integer multiply with 32 bit or 64-bit result, signed or unsigned, add or subtract after the multiply. 32bit multiply is 1 cycle, but 64-bit multiply and MAC instructions require extra cycles.
  - Performs 32-bit hardware integer divide [2-12 cycles].
  - It support saturation arithmetic.
  - 1 to 240 interrupts, plus NMI
  - It has 12 cycle interrupt latency.
  - It has integrated sleep modes.

silicon options optional memory protection unit [MPU]: 0 or 8 regions

★ Cortex-M4:-

- Conceptually the cortex-M4 is a cortex M3 + DSP instructions and optional Floating-Point Unit (FPU). A core with an FPU is known as cortex-M4F.
- key features of the cortex-M4 core are:
  - Which is developed by ARMv7E-M architecture
  - It has 3-stage pipeline with branch speculation
  - Executes Instruction sets are
    - Thumb-1 (entire)
    - Thumb-2 (entire)
  - Performs 32-bit hardware integer multiply with 32-bit or 64-bit result, signed or unsigned, add or subtract after the multiply. 32 bit Multiply and MAC are 1 cycle.

- Performs 32-bit hardware integer divide [2-12 cycles]
- It support saturation arithmetic
- DSP extension: Single cycle 16/32-bit MAC, single cycle dual 16-bit MAC, 8/16-bit SIMD arithmetic [single instruction, multiple data]
- 1 to 240 interrupts, plus NMI.
- It has 12 cycle interrupt latency.
- It has integrated sleep modes.

silicon option - optional floating-point unit [FPU]: single-precision only IEEE-754 compliant, It is called the FPv4-SP extension.

- Optional memory protection unit (MPU): 0 or 8 regions.

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

→ TIVA TM4C123GH6PM Microcontroller:-

Block diagram (next page

• The TM4C microcontroller block diagram has six functional units: The ARM cortex M4F core, on-chip memory, system peripherals, serial peripherals, Analog peripherals, & motion control peripherals.

~~Features~~ • All of the TIVA C series, including the TM4C123GH6PM microcontroller. are designed around an ARM cortex-M processor core.

• The ARM Cortex-M processor provides the core for a high-performance, low-cost platform that meets the needs of minimal memory implementation, reduced pin count, and low power consumption, while delivering outstanding computational performance and exceptional system response to interrupts.

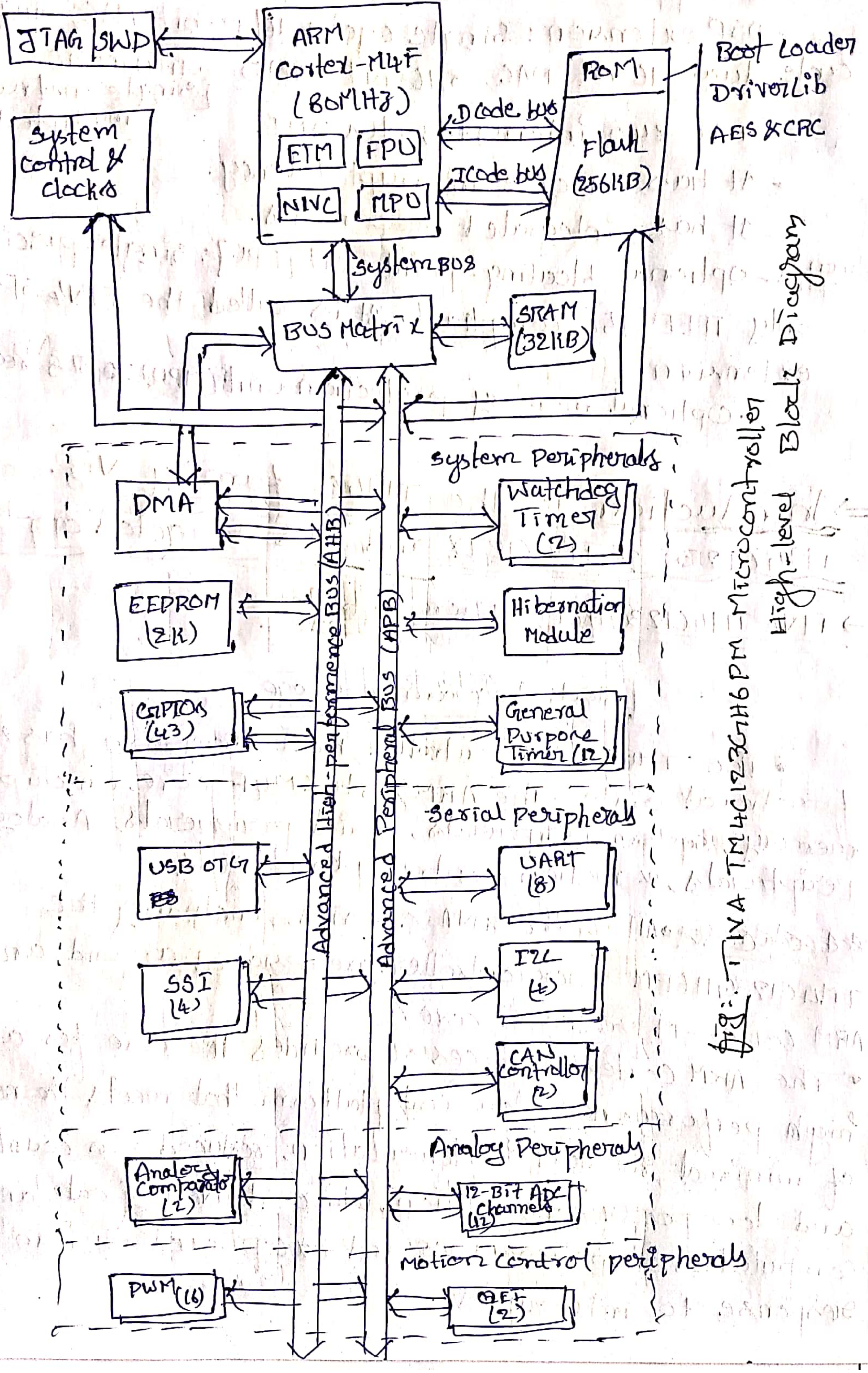


fig: TIVA TM4C123GH6PM-Microcontroller High-level Block Diagram

★ Features:-

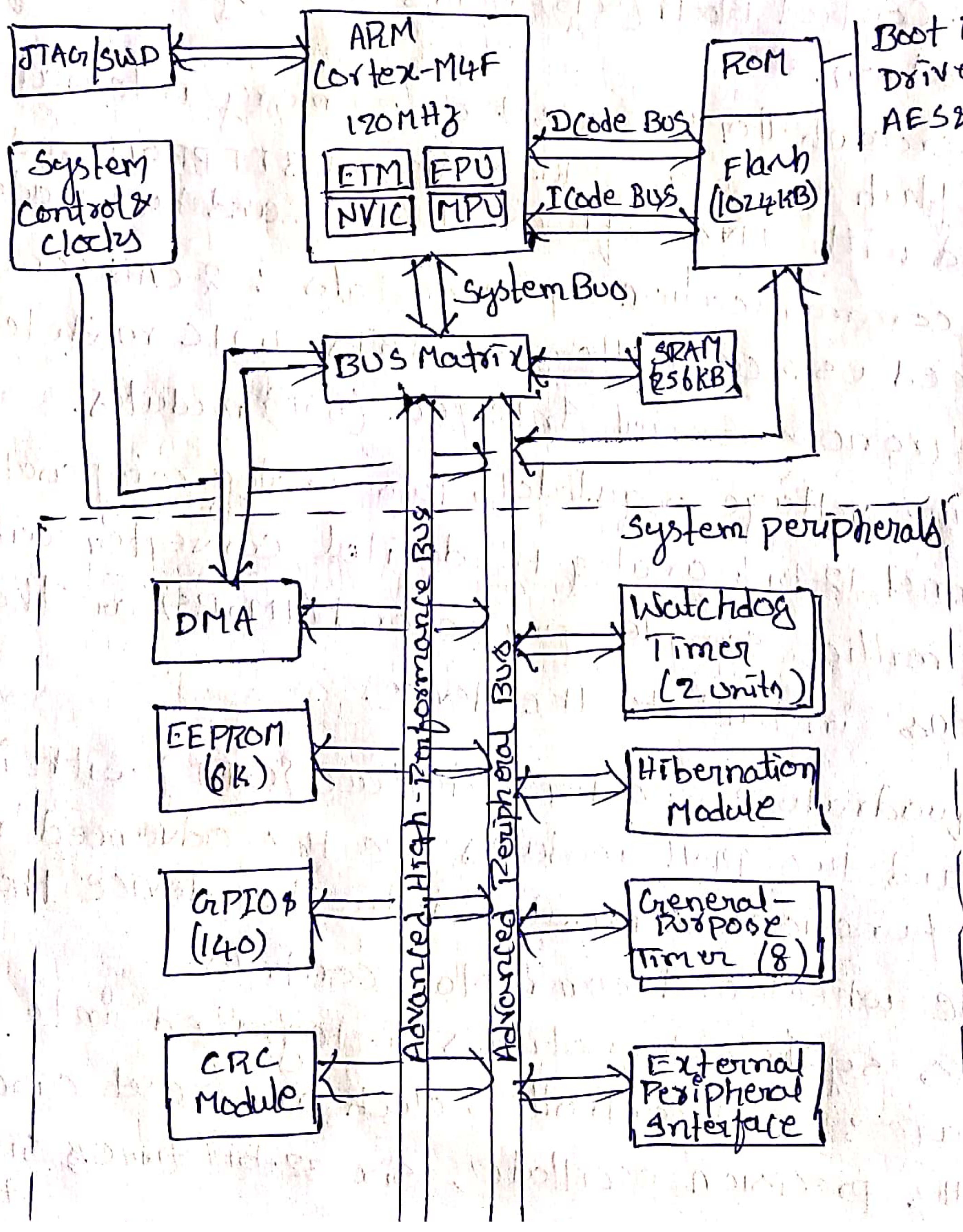
- 32-bit ARM Cortex-M4F architecture optimized for small-footprint embedded applications with 80MHz operation; 100 DMIPS performance
- Memory protection unit (MPU) to provide a privileged mode for protected operating system functionality.
- IEEE 754- compliant single-precision Floating-Point Unit (FPU).
- Serial Wire Debug (SWD) and Serial Wire Trace (SWD/T) reduce the number of pins required for debugging and tracing.
- 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 : {256 KB flash memory, 32 KB SRAM, 2KB EEPROM and ROM loaded with TIVA software library and boot loader.
- Serial communications peripherals : 2 CAN controllers, full speed USB controller, 8 UARTs, 4 IIC modules & 4 synchronous Serial Interface (SSI) modules.
- On chip voltage regulator, two analog-comparators and 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 ~~faci~~ facilitate wheel and ~~cor~~ motor controls.
- Various system functions integrated into the device are : DMA controller, clock and reset circuitry with 16MHz 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 modules 2 watchdog timers and 43 GPIOs.

★ Applications:

- Building automation system
- Lighting control system
- Data acquisition system
- Motion control
- IOT and sensor networks

→ TIVA TM4C129CNCZAD Microcontroller:→



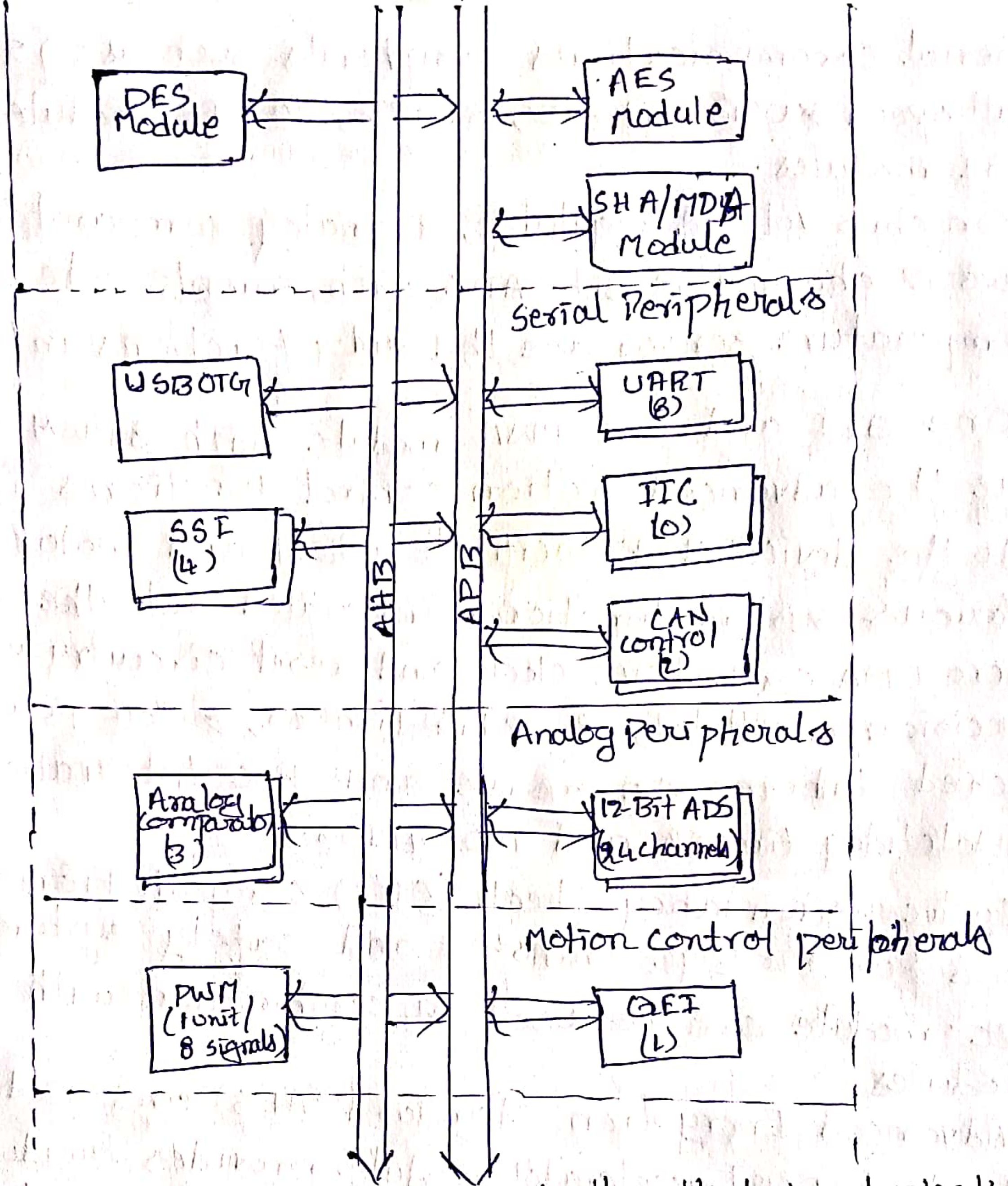


fig: TM4C129CNCZAD Microcontroller High-Level Block Diagram.

★ Features:

- TM4C129CNCZAD microcontroller has 32 bit ARM Cortex M4F processor core with 120 MHz clock rate & 150 DMIPS performance
- Memory protection unit (MPU) provides a privileged mode for protected operating system functionality
- IEEE 754- compliant single-precision Floating-point-Unit (FPU)
- NVIC reduces interrupt response latency and high performance interrupt handling for time critical applications.
- The MC has a set of memory integrated in it { 1 MB flash memory, 256 KB SRAM, 6 KB EEPROM & ROM loaded with TIVA software library & boot loader }

- Serial communications peripherals such as: {2CANControllers full speed USB controller, 8UARTs, 10 IIC modules & 4 SSI modules.
- On chip voltage regulator, 3 analog comparators and two 12 channel 12-bit ADC with sample rate 2 MSPS and temperature sensor are the analog functions in buit to the device
- One QEI and one PWM module with 8 PWM ~~of~~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 DMA controller, clock and reset circuitry with 16MHz precision oscillator, 8 32-bit timers, & 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 &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]:-

- The TM4C123GH6PM chip consists of a 256 KB of Flash memory and 32KB of SRAM.

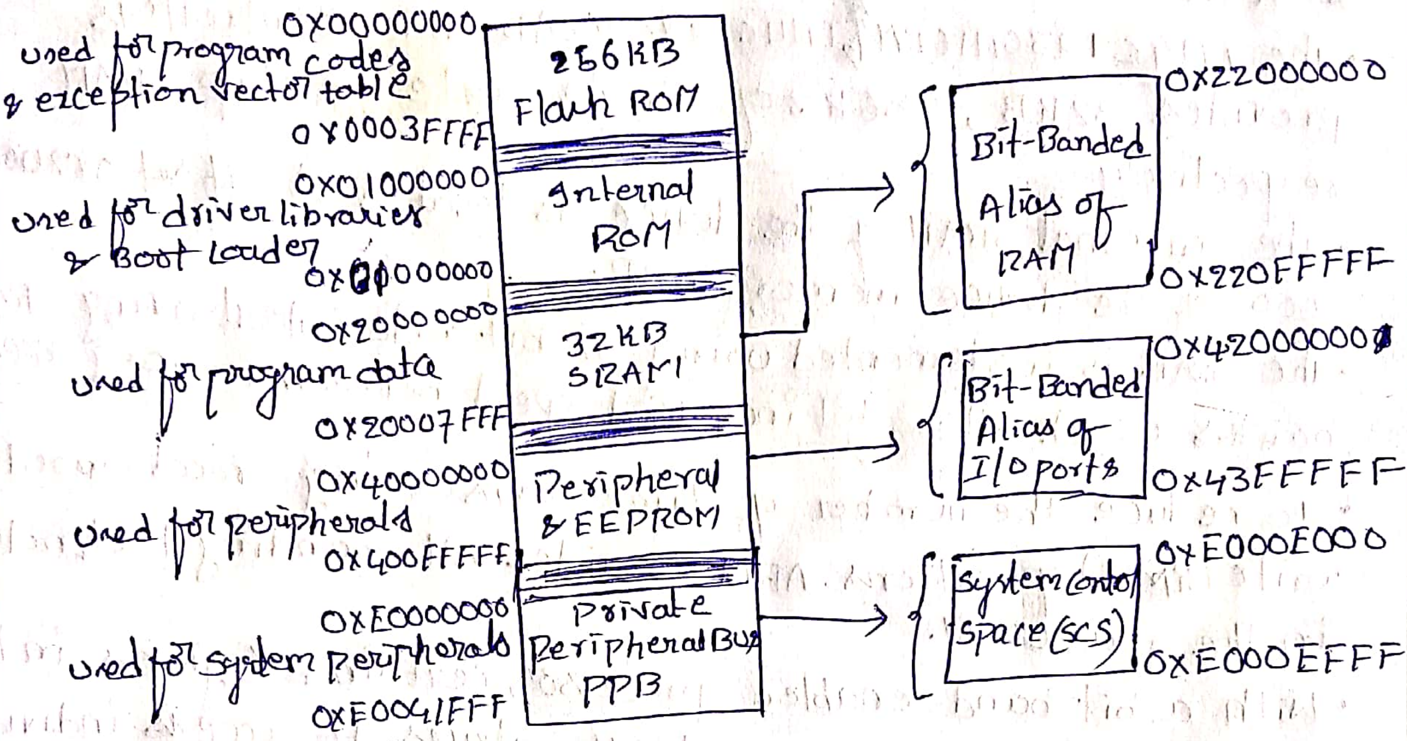


fig: Memory Mapping in TM4C123GH6PM Chip

★ Flash Memory:-

- The TM4C123GH6PM microcontroller provides 256KB of single-cycle on-chip flash memory.
- The Flash memory is organized as a set of 1-KB blocks that can be individually ~~written~~ erased.
- The TM4C129CNCZAD microcontroller provides 1024 KB of on-chip Flash memory. The Flash memory is configured as four banks of 16K X128 bits (4×256 KB total) which are two-way interleaved.
- The Memory blocks can be marked as read-only or execute-only, providing different levels of code protection.
- Read-only blocks cannot be erased or programmed, protecting the contents of those blocks from being modified.
- Execute-only blocks cannot be erased or programmed, and can only be read by the controller instruction fetch ~~mech~~ mechanism.

- Protecting the contents of those blocks from being read by either the controller or by a debugger.
- The Flash can also be accessed by the UDMA in Run mode

*SRAM:

- The TM4C123GH6PM & TM4C129CNCZAD microcontrollers provides 32KB, 256KB of single-cycle on-chip SRAM respectively
- The internal SRAM of the device is located at offset 0x2000.0000 of the device memory map.
- The SRAM is implemented using 32-bit wide interleaving SRAM banks which allow for increased speed between memory accesses
- To reduce the number of time consuming read-modify-write (RMW) operations, ARM provides bit-banding technology in the processor.
- With a bit band-enabled processor, certain regions in the memory map & can use address aliases to access individual bits in a single, atomic operation
- The bit-band alias is calculated by using the formula.

bit-band alias = bit-band base + (byte offset * 32) + (bit number * 4)

Ex: ~~for TM4C123GH6PM.~~

if bit 3 at address 0x2000.1000 is to be modified, the bit-band alias is calculated as:

0x2200.0000 + (0x1000 * 32) + (3 * 4) = 0x2202.000C

*Internal ROM:-

- The internal ROM of the TM4C µC device is located ~~d~~at address 0x0100.0000 of the device memory map.
- The ROM contains the following components.

(+) TivaWare Boot Loader and vector table

- The boot loader is used ~~as~~ ~~an~~ to download code to the Flash memory of a device without the use of a debug interface.

② TivaWare Peripheral Driver Library release for product-specific peripherals & interfaces

③ Advanced Encryption standard (AES) cryptography tables

④ Cyclic Redundancy check (CRC) error detection functionality

★ **Peripheral:** All peripheral devices, timers, and ADCs are mapped as MMIO in address space 0x4000000 to 0x400FFFFF. Since the number of supported peripherals is different among ICs of ARM families,

★ **Private Peripheral Bus (PPB):-** PPB is used for system peripheral like System Timer, NVIC, system control Block (SCB), MPU [Memory Protection Unit], & FPU [Floating Poin Unit

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.

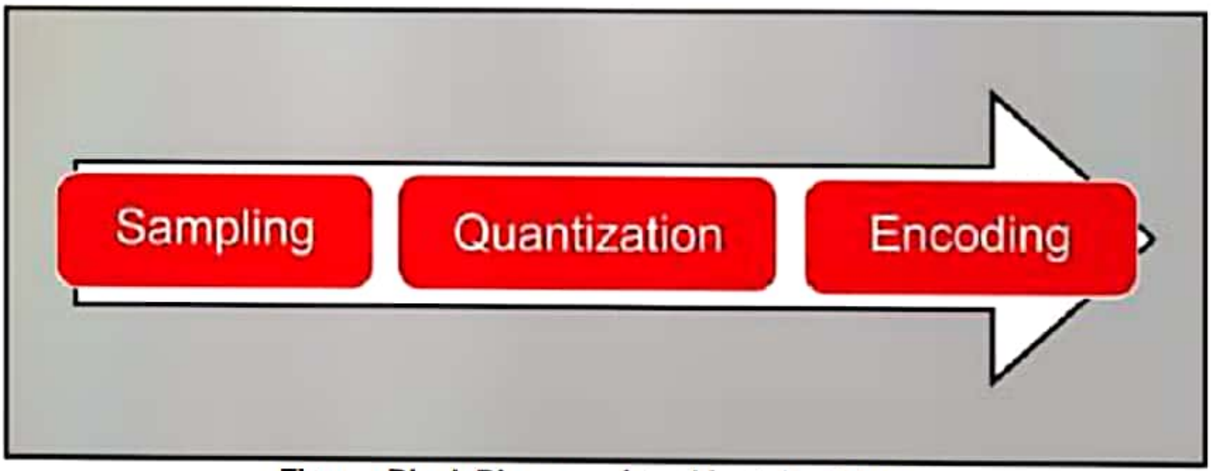


Figure: Block Diagram of working of an ADC

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.

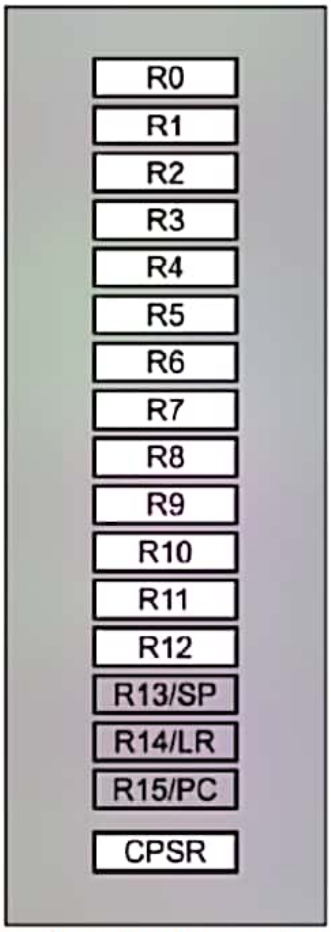


Figure: User Mode Register Set

Current Program Status Register (CPSR):

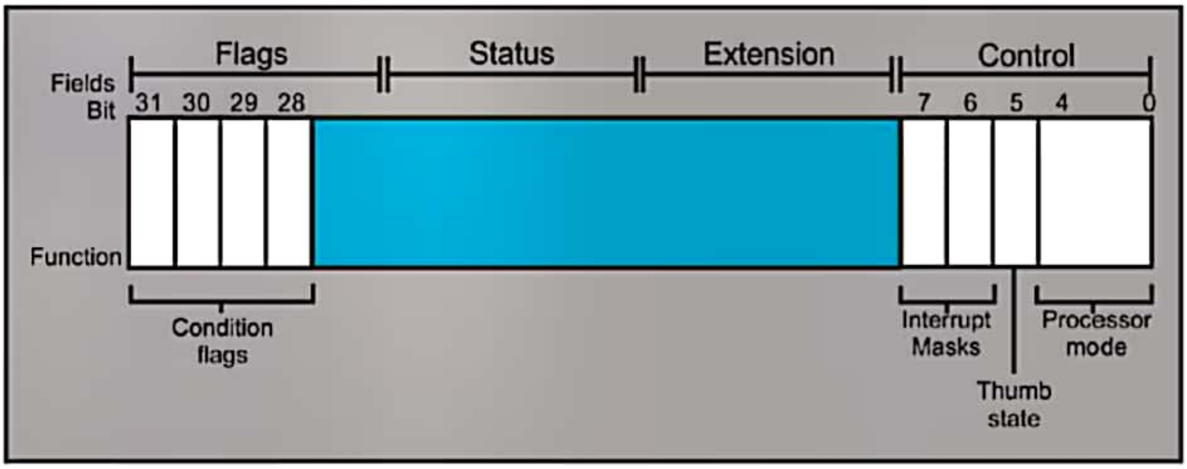


Figure: A Generic CPSR Format

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.

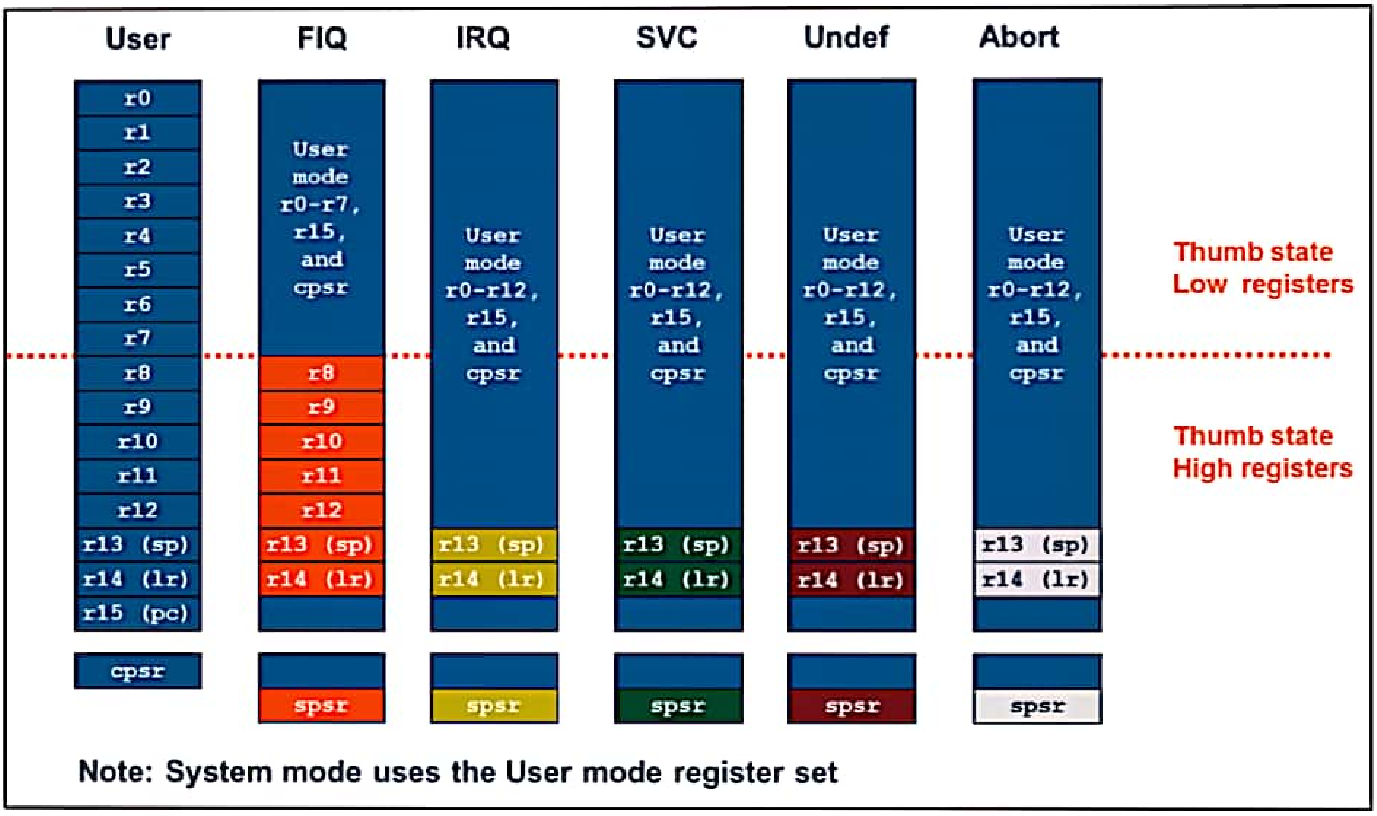

User
FIQ
IRQ
SVC
Undef
Abort
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
User mode r0-r7, r15, and cpsr
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
Thumb state Low registers
Thumb state High registers
Note: System mode uses the User mode register set

Figure: Complete Register Bank

⇒ Addressing Modes:-

- The way in which operands are specified in the instruction is called addressing modes.
- The ARM supports the following addressing modes.
  - Data Processing Instructions Addressing modes
    - ~~① Register Addressing~~
    - ① Immediate Addressing Mode
    - ② Register Addressing Mode
    - ③ Register Indirect Addressing Mode
    - ④ Relative Addressing Mode
    - ⑤ Register offset Addressing Mode
  - Load store Addressing Mode
    - ⑥ Register based with offset Addressing mode.
      - Pre-Indexed Addressing mode
      - Pre-Indexed with auto-Indexed A-M
      - Post - Indexed Addressing mode.

- The general syntax for Data Processing A-M is

<opcode>{condition}{S} <Rd>, <Rn>, <Shifter operand>.

Various A-Ms used to calculate the <shifter operand> in ARM data processing instruction.

① Immediate Addressing Mode:-

Immediate operand is a part of instruction. i.e operand2 is an immediate value.

Ex:
```
MOV  Ro, #0      // Put 0 to RO
SUB  Ro, Ro, #1  // Save (RO-1) to Ro
ADD  R3, R3, #2  // Save (R3+2) to R3
```

② Register Addressing Mode:-

In register addressing mode the operand is held in a register which is specified in the instruction.

Ex:- MOV R1, R2 // move operand from R2 to R1

SUB R0, R1, R2 // subtract operand of R2 from R1 and move the result to R0.

ADD R4, R3, R2 // Add the operand of R3 with the operand of R2 & move the result to R4.

(3) Register Indirect Addressing Mode:-

- In this addressing mode, the operand resides in the memory whose address is specified in the instruction.
- The 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 & put the result into R0.

(4) Relative Addressing Mode:-

- In this addressing mode, the 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.

(5) Register offset Addressing mode [(or) Shifted register operand Addressing mode]:-

- operand2 is in a register with some offset calculation.
- A shifter register operand value is the value of a register.
- shifted/rotated before it is used as the data processing operand.
- The operand2 first preprocessed after that operation will performs.

Syntax: <operand>{cond}{S} <Rd> <Rm>, shift/rotate #<immediate> (or) <Rn>

ARM7 defined several types of shift & rotate operation

i.e: - LSL - Logical shift Left

LSR - Logical shift Right

ASL - Arithmetic shift Left

ASR - Arithmetic shift Right

ROR - Rotate Right

RRX - Rotate Right with extended by 1 bit with carry bit

Ex: MOV R2, R0, LSL #2 // Shift R0 left by 2 positions & then copy the new value to R2.

ADD R9, R4, R5, LSL #3 // shift R5 left by 3 positions and then Add R4 with new value of R5 & store the result into R9. [R9 = R4 + R5 * 8]

AND R0, R1, R2 LSR R3 // shift R2 right by the operand of R3 positions and then logical AND between R1 & new value of R2 & Result stores into R0.

MOV R12, R4, ROR R3 // R4 Rotate right by operand of R3 positions & copy the new value of R4 into R12.

ADDEQ R9, R6, R5, LSL #4;

This intruction execute conditionally when z = 1; i.e shift R5 left by 4 positions and new value of R5 Add with R6 and Result store into R9 if zero flag = 1;

⑥ Register based with offset Addressing mode (or) Load store Instruction Addressing mode:-

- Effective memory address has to be calculated from a base address and an offset.
- Here offset can be an immediate offset, register offset or scaled register offset.

(a) Pre-Indexed Addressing Mode:- syntax. <opcode>{cond}{S} <Rd>, [<Rn>, <shift op>]

- Immediate offset A.M:-

Ex: LDR R4, [R2, #5] // R4 <= mem32[R2+5]

offset

// Loads the R4 reg with the value stored at memory address [R2+5].

-Register offset A.M:- (offset held in Register)

Ex: LDR R4, [R2,-R3] || R4 <= mem32[R2-R3]

//R4 loads with the operand stored at memory address R2-R3.

STR R1, [R0, R2] || R1 => mem32[R0+R2]

// stores R1 operat to an address locaticon pointed by R0+R2.

- Scaled register offset A.M:-

• In this mode ~~first operand~~ operand2 pregister has to be preprocessed & then Effect Address has to be calculated

Ex: • LDR R3, [R1, R2, LSR#3]

•//. First R2 shift right by 3 position & new value of R2 add with R1 for calculation of E-A]. then R3 will load operan stored at E-A i.e ~~memory~~

• STR R0, [R1, R2, LSL #2].

//stores R0 to an ~~R0~~ address equal to som of R1 & four times of R2 | R0 => mem32[R1+(R2*4)]

(B) Pre-indexed with auto-indexing A.M:-

This ~~first~~ mode used for Address pointer to access an array type of data.

Syntax: <opcode>{cond}{S} <Rd>, [<Rm>, <shifted operand>]!

base offset

~~In~~: '!' symbol indicates that the instruction saves the calculated address in the base address register.

-Immediate pre indexed:-

Syntax: <opcode>{cond}{S} <Rd>, {<Rm>, #immediate]!

Ex: STR R0, [R1, #2]!

• use (R1+2) as address & stored the R0 operand to new address location pointed by R1+2. and then update R1 by R1+2.

- LDR R0, [R1, #4]! //use (R1+4) as address and load the data from the address to R0 & updata R1 by (R1+4)

- Register Pre-indexed AM:-

Syntax: <opcode>{cond}{S} <Rd> [<Rm>, <Rn>]!

Ex: LDR R4, [R2, R3]! // R4 ⇐ mem32 [R2+R3]
update R2 = R2+R3.

STR R1, [R2, R0]! // R1 ⇒ mem32[R2+R0] &
update R2 = R2+R0.

- Scaled Register Pre-indexed AM:-

syntax: <opcode>{cond}{S} <Rd> [<Rm>, <Rn> shift/rotate #immediate].

Ex: LDR R4, [R2, R3, LSL #2]! // R4 <= mem32[R2+(R3*4)].
update R2 = R2+(R3*4).

STR R3, [R1, R2 LSL #4]! // R3 ⇒ mem32[R1+(R2*16)]
update R1 = R1+(R2*16).

(B) Post-Indexed AM (or) Post-Indexed with auto-indexed AM:-

Syntax: <opcode>{cond}{S} <Rd>, [<Rm>], <shifted operand>.

- Immediate post-Indexed AM:-

Syntax: <opcode>{cond}{S} <Rd>, [<Rm>], #immediate.

Ex: LDR R0, [R1], #4 // R0 ⇐ mem32[R1] ~~R0 ⇐ [R1]~~ then update R1 by R1+4.

Load the data pointed by R1 address location to R0 and then update R1 by R1+4.

- STR R1, [R3], #2 // R1 ⇒ mem32[R3] then R3 = R3+2

store the data R1 to address location pointed by R3 and then R3 update by R3+2

- Register post-indexed A.M:-

Syntax: <opcode>{cond}{S} <Rd>, [<Rm>], <Rn> :

Ex: STR R1, [R3], R4 // ~~R1 ⇒ mem[R3]~~ R1 ⇒ mem32[R3] then update R3 by R3+R4.

// store the data in R1 to the memory location pointed by R3 and then update R3 by R3+R4.

- Scaled Register Post-indexed A.M:-

Syntax: <opcode>{cond}{S} <Rd>, {<Rm>], <Rn> shift/rotate #imm
Immediate

Ex: ~~LDR {P~~
LDR R2, {R0}, -R3, LSL #4 // R2 ⇐ mem32[R0] & then R0 = R0-(R3*16)

// load the data from the address pointed to by R0 to R2 and then updata R0 by R0-(R3 * 16).

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

Embedded Hardware and Various Building Blocks:

The basic hardware components of an embedded system shown in a block diagram in below figure.

These include the processing unit, sensors and actuators, ADC, DAC, I/O unit and the memory block. The processing unit could be a microprocessor, a microcontroller, FPGA i.e. field programmable gate array or ASIC (Application Specific IC) depending on the application requirements.

Sensors such as sound sensor, ambient temperature sensor, motion sensor etc. are generally analog in nature since they sense the data from outside world. This data is converted from analog to digital and sent to a processing unit, post which required action is performed by actuators.

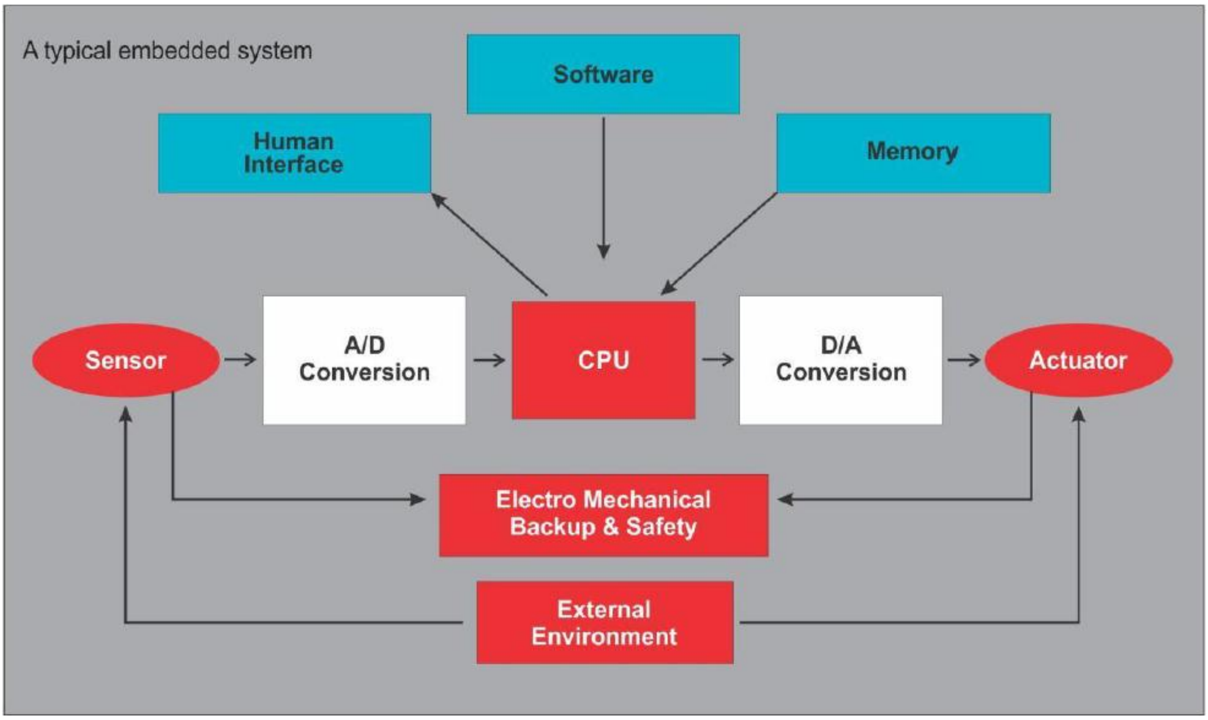


Figure: Basic block diagram of an embedded system

Processor Selection for an Embedded System:

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.

This processing unit executes the application program that is saved in the program memory ROM (read only memory). The RAM (random access memory) is used as the data memory to hold the system stack and the variables used in the program.

Stack is a portion in the RAM reserved to hold back the status of the program when the control is transferred by a branch instruction. To make a system interactive, input-output (I/O) unit is required. The memory block and the I/O units communicate with the processing unit through the system bus.

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.

Interfacing Processor:

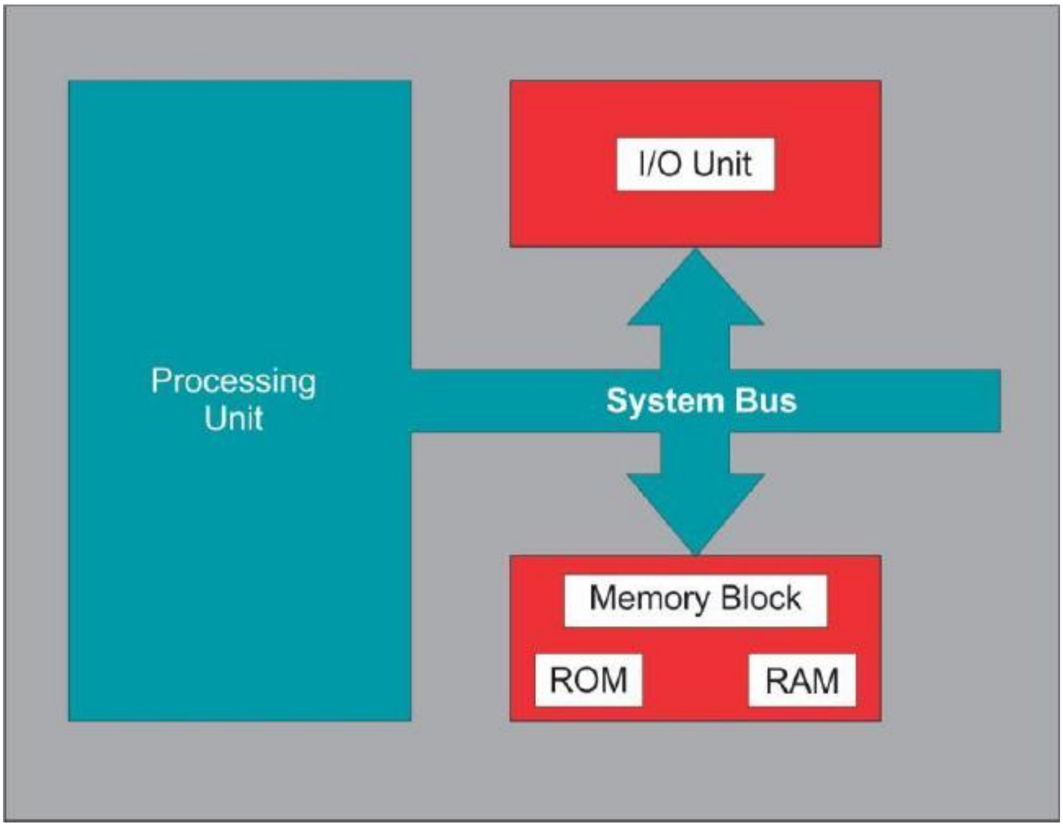


Figure: Processing Unit and System Bus

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

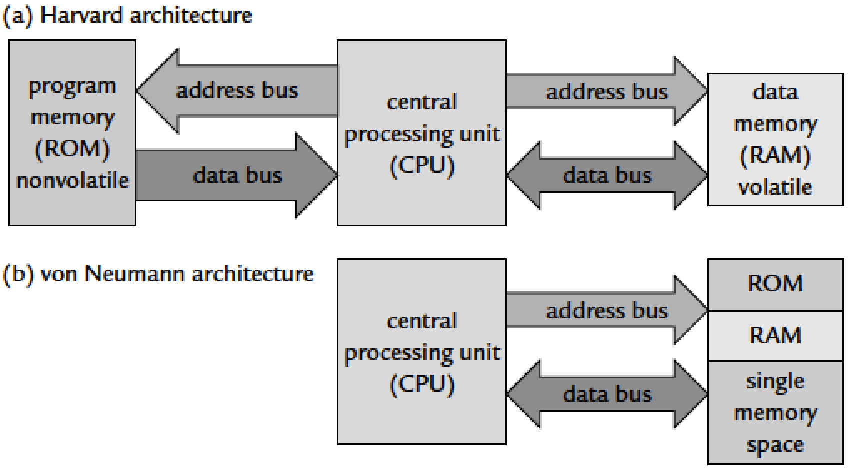
Field programmable gate arrays (FPGA) have programmable macro cells and their interconnects are configured based on the design. They are used in embedded systems when it is required to enhance the computational capability of the existing system or to make a system reprogrammable and reconfigurable when the need arises.

Memories and I/O Devices:

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. Various ROM and RAM devices are used in embedded systems based on the applications.


(a) Harvard architecture
program memory (ROM) nonvolatile
address bus
data bus
central processing unit (CPU)
address bus
data bus
data memory (RAM) volatile
(b) von Neumann architecture
central processing unit (CPU)
address bus
data bus
ROM
RAM
single memory space

Harvard and von Neumann architectures for memory.

ROM

Read only memory (ROM) is non-volatile i.e. it retains the contents even after power goes off. It is used as the 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

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.

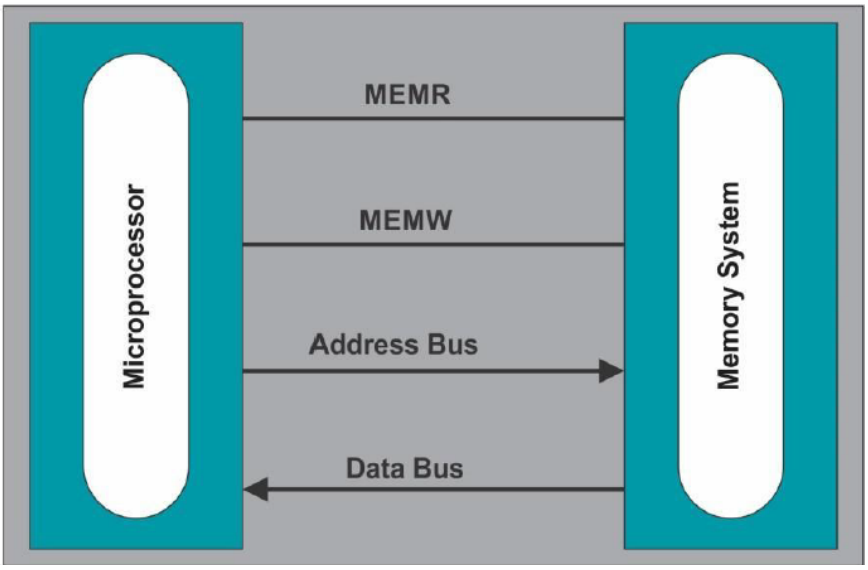


Figure: Interconnection of RAM with Microprocessor

I/O Devices:

Embedded systems have to interact with the external environment through the input/output devices.

Input Device

Embedded systems receive user commands from input devices such as keypad, switch or a touch screen device at the input port. The processing unit executes software instructions to process these inputs to make decisions that further guide the operation of the system. A port is a termination point that gives connectivity between the processing unit and the peripherals.

Output Device

Output devices are used to display results from the system or to sending data to another connected system at the output port. Some examples of output devices are: light emitting diodes (LEDs), liquid crystal diodes (LCDs), printers etc.

I/O Interfacing Concepts:

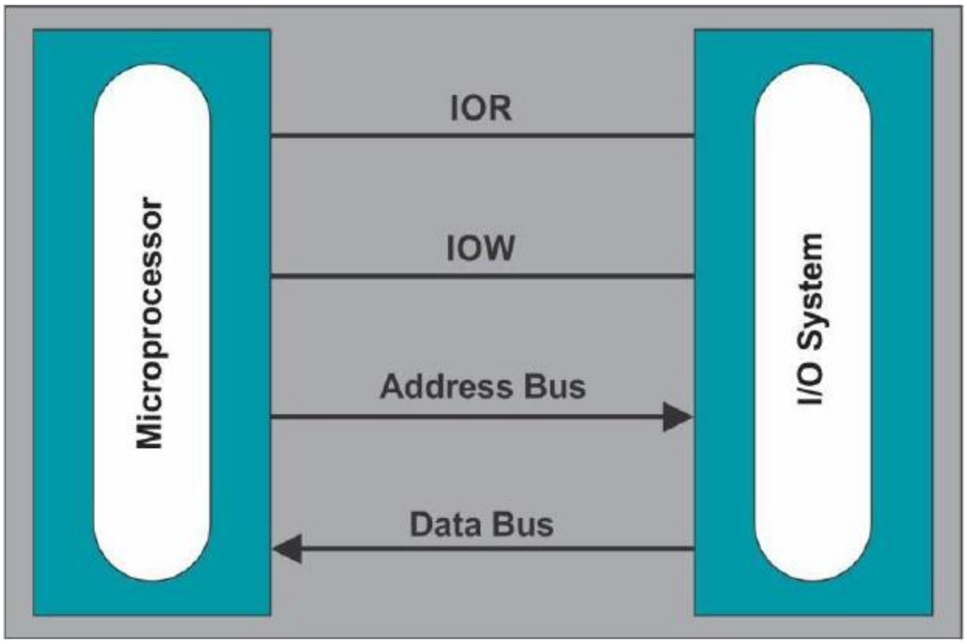


Figure: Interconnection of external devices with Microprocessor

I/O Communication Bus

I/O communication buses and protocols are used to communicate with the slower I/O devices. There are two communication methods used in any system: serial communication and parallel communication. Some of the communication protocols are: universal serial bus (USB), inter-integrated circuit (I2C), serial peripheral interface (SPI), peripheral component interconnect (PCI), IBM standard architecture (ISA) etc. Each protocol defines a standard way of communication between the devices. The features and usage of these protocols have been explained in subsequent chapters.

Sensors & Actuators

Sensors and electromechanical actuators are input and output devices used in real time embedded systems to exchange real time data between the system and the external environment. Sensors measure physical parameters such as temperature, pressure acceleration, proximity etc. being connected at the system input ports through analog to digital converters (ADCs). Some of the actuators used in embedded systems are: motor speed controllers, stepper motor controllers, relays and power drivers etc. Actuators are connected at the system output ports through the digital to analog converters (DACs).

Timer and Counting Devices:

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.

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.

The GPT Module is one timing resource available on the Tiva™ C Series microcontrollers. Other timer resources include the System Timer (SysTick) and the PWM timer in PWM modules.

The General-Purpose Timer Module (GPTM) blocks with the following functional options:

- ❑ **16/32-bit operating modes:**
  - ⇨ 16- or 32-bit programmable one-shot timer
  - ⇨ 16- or 32-bit programmable periodic timer
  - ⇨ 16-bit general-purpose timer with an 8-bit prescaler
  - ⇨ 32-bit Real-Time Clock (RTC) when using an external 32.768-KHz clock as the input
  - ⇨ 16-bit input-edge count- or time-capture modes with an 8-bit prescaler
  - ⇨ 16-bit PWM mode with an 8-bit prescaler and software-programmable output inversion of the PWM signal
- ❑ **32/64-bit operating modes:**
  - ⇨ 32- or 64-bit programmable one-shot timer
  - ⇨ 32- or 64-bit programmable periodic timer

    - ⇨ 32-bit general-purpose timer with a 16-bit prescaler
    - ⇨ 64-bit Real-Time Clock (RTC) when using an external 32.768-KHz clock as the input
    - ⇨ 32-bit input-edge count- or time-capture modes with a16-bit prescaler
    - ⇨ 32-bit PWM mode with a 16-bit prescaler and software-programmable output inversion of the PWM signal
- ❑ Count up or down
- ❑ Twelve 16/32-bit Capture Compare PWM pins (CCP)
- ❑ Twelve 32/64-bit Capture Compare PWM pins (CCP)
- ❑ Daisy chaining of timer modules to allow a single timer to initiate multiple timing events
- ❑ Timer synchronization allows selected timers to start counting on the same clock cycle
- ❑ ADC event trigger
- ❑ User-enabled stalling when the microcontroller asserts CPU Halt flag during debug (excluding RTC mode)
- ❑ Ability to determine the elapsed time between the assertion of the timer interrupt and entry into the interrupt service routine
- ❑ Efficient transfers using Micro Direct Memory Access Controller (µDMA)
    - ⇨ Dedicated channel for each timer
    - ⇨ Burst request generated on timer interrupt

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.

The following figure provides a schematic representation of the Design Cycle in the Development Phase for an Embedded System.

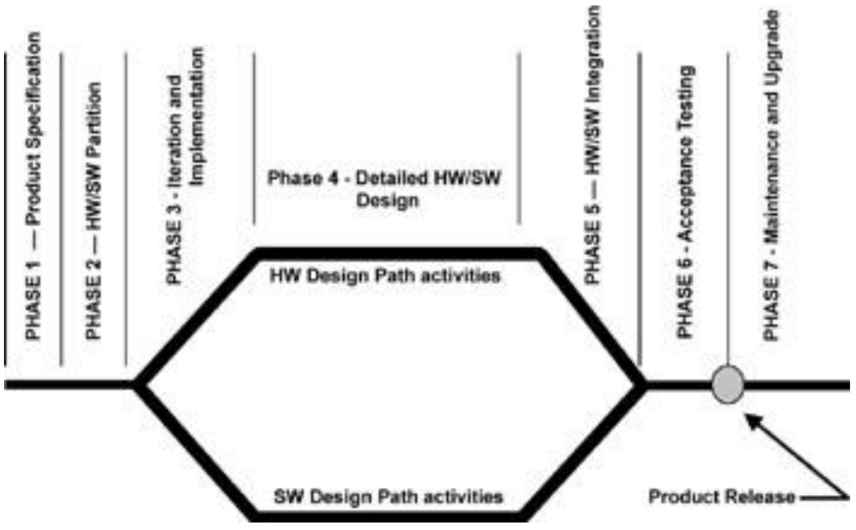

PHASE 1 — Product Specification
PHASE 2 — HW/SW Partition
PHASE 3 - Iteration and Implementation
Phase 4 - Detailed HW/SW Design
HW Design Path activities
SW Design Path activities
PHASE 5 — HW/SW Integration
PHASE 6 - Acceptance Testing
PHASE 7 - Maintenance and Upgrade
Product Release

Figure: Embedded Design Life Cycle Diagram

A Phase representation of the Embedded Design Life Cycle

Time flows from the left and proceeds through seven phases:

Product specification:

- ⇨ Partitioning of the design into its software and hardware components
- ⇨ Iteration and refinement of the partitioning
- ⇨ Independent hardware and software design tasks
- ⇨ Integration of the hardware and software components
- ⇨ Product testing and release
- ⇨ On-going maintenance and upgrading

The embedded design process is not as simple as above figure depicts. A considerable amount of iteration and optimization occurs within phases and between phases. Defects found in later stages often cause you to go back to square 1. For example, when product testing reveals performance deficiencies that render the design non-competitive, you might have to rewrite algorithms, redesign custom hardware such as Application-Specific Integrated Circuits (ASICs) for better performance speed up the processor, choose a new processor, and so on.

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

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.[1] 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.

UNIT-4

MICROCONTROLLER FUNDAMENTALS FOR BASIC PROGRAMMING

4.1. Introduction:

The I/O pin configurations for the TM4C123 microcontrollers. The regular function of a pin is to perform parallel I/O. Most of the pins have an alternative function. Joint Test Action Group (**JTAG)** is a standard test access port used to program and debug the microcontroller board. Each microcontroller uses five port pins for the JTAG interface.

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

- UART Universal asynchronous receiver/transmitter
- SSI Synchronous serial interface
- I2C 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.

I2C 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.

4.2. Tiva TM4C123 LaunchPad I/O pins:

Pins on the TM4C family can be assigned to as many as eight different I/O functions. Pins can be configured for digital I/O, analog input, timer I/O, or serial I/O. For example PA0 can be digital I/O or serial input. There are two buses used for I/O. The digital I/O ports are connected to both the advanced peripheral bus and the advanced high-performance bus. Because of the multiple buses, the microcontroller can perform I/O bus cycles simultaneous with instruction fetches from flash ROM. The TM4C123GH6PM adds up to 16 PWM outputs. There are 43 I/O lines. There are twelve ADC inputs; each ADC can convert up to 1M samples per second.

Each pin has one configuration bit in the GPIOAMSEL register. We set this bit to connect the port pin to the ADC or analog comparator. For digital functions, each pin also has four bits in the GPIOPCTL register, which we set to specify the alternative function for that pin (0 means regular I/O port). Not every pin can be connected to every alternative function.

Pins PC3 – PC0 were left off because these four pins are reserved for the JTAG debugger and should not be used for regular I/O. Notice, most alternate function modules (e.g., U0Rx) only exist on one pin (PA0). While other functions could be mapped to two or three pins (e.g., CAN0Rx could be mapped to one of the following: PB4, PE4, or PF0.)

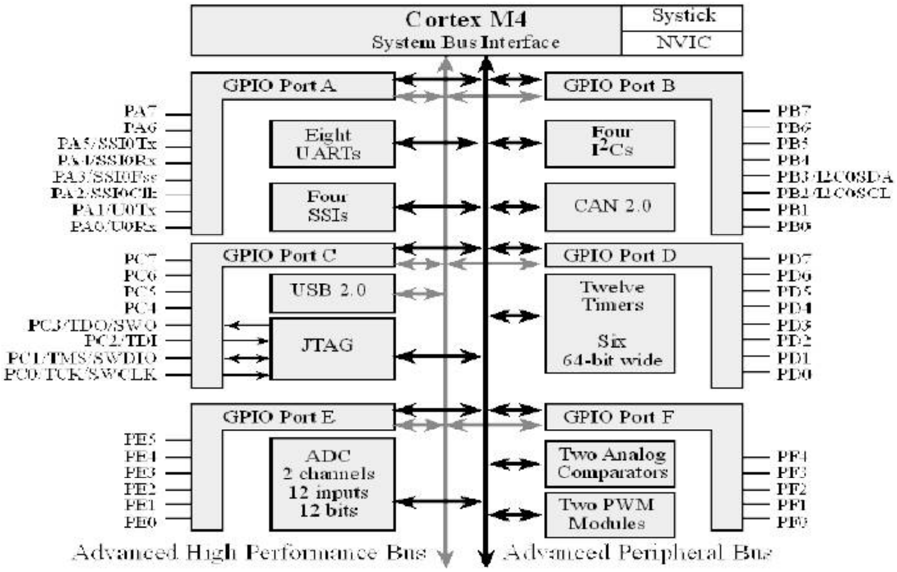
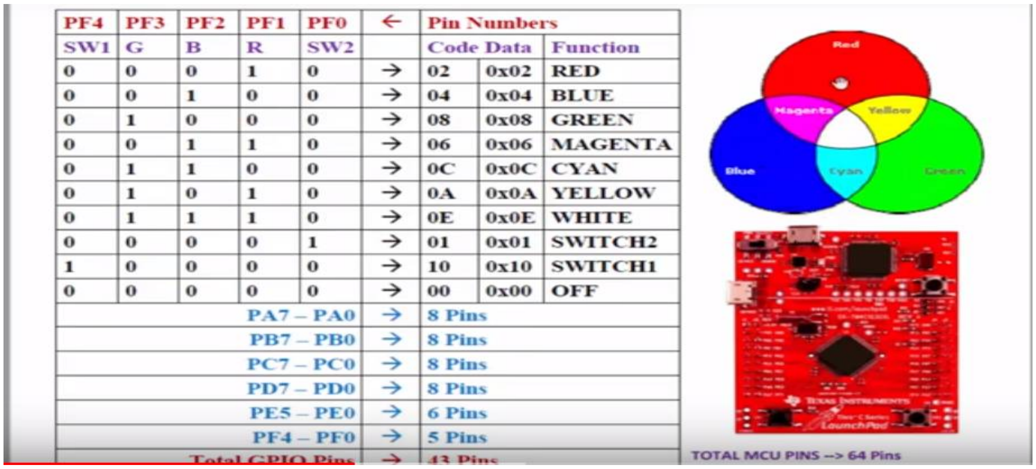


Figure : I/O port pins for the TM4C123GH6PM microcontrollers.

The microcontroller board provides an integrated In-Circuit Debug Interface (ICDI), which allows programming and debugging of the onboard TM4C123 microcontroller. One USB cable is used by the debugger (ICDI), and the other USB allows the user to develop USB applications (device). The user can select board power to come from either the debugger (ICDI) or the USB device (device) by setting the Power selection switch.

Pins PA1 – PA0 create a serial port, which is linked through the debugger cable to the PC. The serial link is a physical UART as seen by the TM4C and mapped to a virtual COM port on the PC. The USB device interface uses PD4 and PD5. The JTAG debugger requires pins PC3 – PC0. The LaunchPad connects PB6 to PD0, and PB7 to PD1. If you wish to use both PB6 and PD0 you will need to remove the R9 resistor. Similarly, to use both PB7 and PD1 remove the R10 resistor.

The Tiva LaunchPad evaluation board has two switches and one 3-color LED. See Figure The switches are negative logic and will require activation of the internal pull-up resistors. In particular, you will set bits 0 and 4 in **GPIO_PORTF_PUR_R** register. The LED interfaces on PF3 – PF1 are positive logic. To use the LED, make the PF3 – PF1 pins an output. To activate the red color, output a one to PF1. The blue color is on PF2, and the green color is controlled by PF3. The 0-Ω resistors (R1, R2, R11, R12, R13, R25, and R29) can be removed to disconnect the corresponding pin from the external hardware.



PF4	PF3	PF2	PF1	PF0	←	Pin Numbers		
SW1	G	B	R	SW2		Code Data		Function
0	0	0	1	0	→	02	0x02	RED
0	0	1	0	0	→	04	0x04	BLUE
0	1	0	0	0	→	08	0x08	GREEN
0	0	1	1	0	→	06	0x06	MAGENTA
0	1	1	0	0	→	0C	0x0C	CYAN
0	1	0	1	0	→	0A	0x0A	YELLOW
0	1	1	1	0	→	0E	0x0E	WHITE
0	0	0	0	1	→	01	0x01	SWITCH2
1	0	0	0	0	→	10	0x10	SWITCH1
0	0	0	0	0	→	00	0x00	OFF
PA7 – PA0					→	8 Pins		
PB7 – PB0					→	8 Pins		
PC7 – PC0					→	8 Pins		
PD7 – PD0					→	8 Pins		
PE5 – PE0					→	6 Pins		
PF4 – PF0					→	5 Pins		
Total GPIO Pins					→	43 Pins		

TOTAL MCU PINS --> 64 Pins

4.3. GPIOs :

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. We will see this later in this chapter when we use a switch to force the system wake from hibernation. GPIO peripherals vary quite widely. In some cases, they can exist as a group of pins that can be switched as a group to either input or output. In others, each pin can be set up adaptable to either accept or act as a source for different logic voltages, with configurable drive strengths and pull ups. Pin states of the GPIOs can be accessed using software instructions. These instructions can be represented by one or more types of interfaces. Memory mapped peripheral or a dedicated I/O port instruction can be used in this regard.

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.

4.3.1. 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-F 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.

4.3.2. 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

4.3.3. TM4C123 GPIO Programming:

The TI LaunchPad uses the TM4C123GH6PM microcontroller, which has 256K bytes (256KB) of onchip Flash memory for code, 32KB of on-chip SRAM for data, and a large number of on-chip peripherals.

The ARM Cortex-M4 has 4GB (Giga bytes) of memory space. It uses memory mapped I/O, which means that the I/O peripheral ports are mapped into the 4GB memory space.

Allocated size Allocated address

- Flash 256 KB 0x0000.0000 To 0x0003.FFFF
- SRAM 32 KB 0x2000.0000 To 0x2000.7FFF
- I/O All the peripherals 0x4000.0000 to 0x400F.FFFF

The General Purpose I/O ports (GPIO) on TM4C123GXL LaunchPad are designated to port A to port F. The address range assigned to each GPIO port is shown as follows:

- Port A: 0x4000.4000 to 0x4000.4FFF
- Port B: 0x4000.5000 to 0x4000.5FFF
- Port C: 0x4000.6000 to 0x4000.6FFF
- Port D: 0x4000.7000 to 0x4000.7FFF
- Port E: 0x4002.4000 to 0x4002.4FFF
- Port F: 0x4002.5000 to 0x4002.5FFF

The 4K bytes of memory space is assigned to each of the GPIO. The reason is that each GPIO has a large number of special function registers associated with it, and furthermore GPIO Data Register supports bit-specific addressing, which allows collective access to 1 to 8 bits in a data port.

<u>To initialize an I/O port for general use seven steps need to be performed.</u>

1. Activate the clock for the port in the Run Mode Clock Gating Control Register 2 (**RCGC2**).
2. Unlock the port (**LOCK** = **0x4C4F434B**). This step is only needed for pins **PC0-3**, **PD7** and **PF0** on TM4C123GXL LaunchPad.
3. Disable the analog function of the pin in the Analog Mode Select register (**AMSEL**), because we want to use the pin for digital I/O. If this pin is connected to the ADC or analog comparator, its corresponding bit in **AMSEL**must be set as 1. In our case, this pin is used as digital I/O, so its corresponding bit must be set as 0.
4. Clear bits in the port control register (**PCTL**) to select regular digital function. Each GPIO pin needs four bits in its corresponding PCTL register. Not every pin can be configured to every alternative function. Figure 2.2 shows which pin can be used as what kind of alternate functions.
5. Set its direction register (**DIR**). A DIR bit of 0 means input, and 1 means output.
6. Clear bits in the alternate Function Select register (**AFSEL**).
7. Enable digital port in the Digital Enable register (**DEN**).

4.4. Peripheral and Memory Address:

A 32-bit processor can have 4 GB (=232) of address spaces. It depends on the architecture of the CPU how these address spaces are segregated, among the memory and peripherals.

4.4.1. 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).

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.

4.4.2. Memory Mapped Peripherals:

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	265 KB	0x0000.0000 to 0x0003.FFFF
SRAM	32 KB	0x2000.0000 to 0x2000.7FFF
I/O	All the peripherals	0x4000.0000 to 0x400F.FFFF

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. Bit band alias are computed according to following formula.

bitband alias= bitband base + byte offset *32 + bit number *4 (2.1)

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 TM Boot Loader and vector table.
- TivaWare TM Peripheral Driver Library (DriverLib) release of product-specific peripheral and interface.
- Advanced Encryption Standard (AES) cryptography tables.
- Cyclic Redandancy 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.

4.5. Programming System Registers:

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 configuring 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.

Data Direction Operation: In Tiva C series Launchpad, the GPIO Direction (GPIODIR) register is used 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.

4.5. 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).

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 & 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

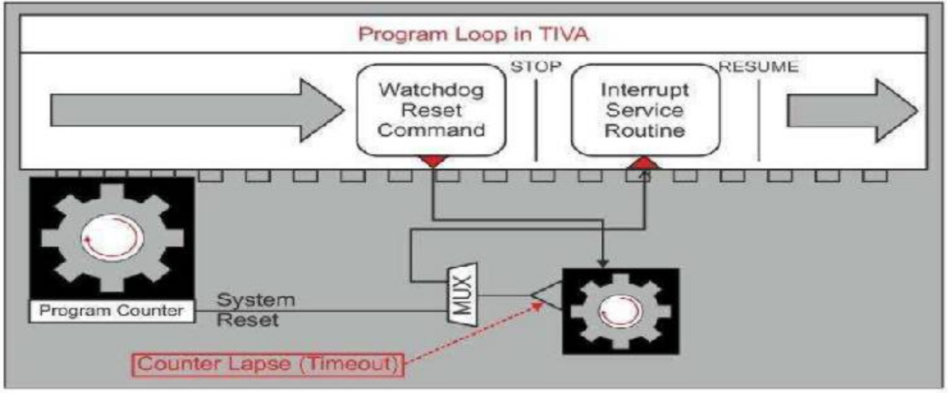


Fig : Operation of Watchdog Timer

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.

4.6. Low Power Microcontroller:

4.6.1. 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. Let us consider an example where we are to design a system to monitor water level in a tank. When the water level reduces below a particular level, water should be pumped in. There are many ways to go about this design.

4.6.2. Hibernation Module on TivaTM Microcontrollers:

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:

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

- 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.

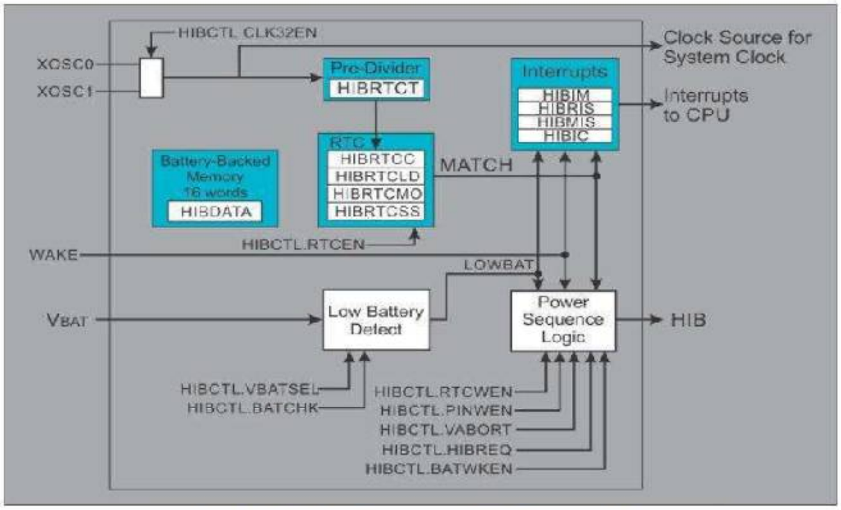


Fig : Block diagram of Hibernation module

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

4.6.3. Active Vs Standby Current Consumption:

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
Peripheral	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

4.6.4. 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.

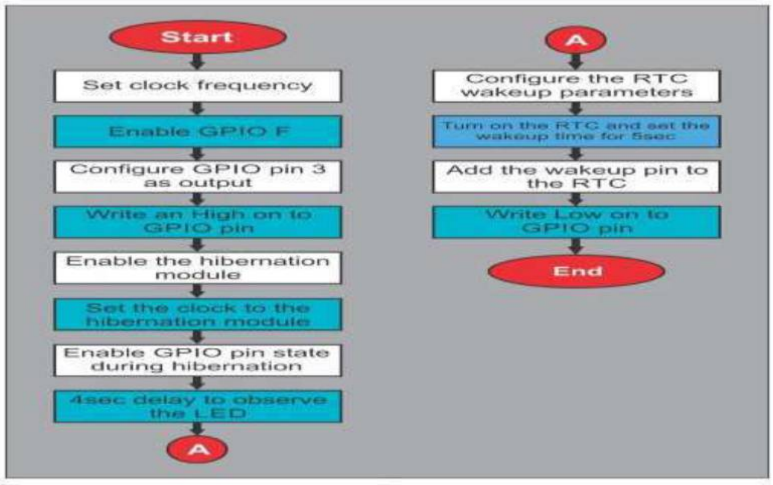

Start
Set clock frequency
Enable GPIO F
Configure GPIO pin 3 as output
Write an High on to GPIO pin
Enable the hibernation module
Set the clock to the hibernation module
Enable GPIO pin state during hibernation
4sec delay to observe the LED
A
A
Configure the RTC wakeup parameters
Turn on the RTC and set the wakeup time for 5sec
Add the wakeup pin to the RTC
Write Low on to GPIO pin
End

Fig : Flowchart for programming hibernation module

4.7. Introduction to Interrupts:

The reader is aware that a microprocessor is connected to several input and output devices. It is important at this point for us to know how a microprocessor manages these devices efficiently.

4.7.1. 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.

4.7.2. 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. The below 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 onchip 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 mamory 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 Interrupt (interrupts generated fron peripherals and software)	0x0000.0040 to 0x0000.03FC

On system reset, the vector table is fixed at address 0x0000.0000.

Table:Interrupt Vector Table for ARM Cortex M4

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 above Table, "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 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.

4.7.3.Interrupt Programming:

Aim: To understand how exceptions/interrupts work

```
#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_M
AIN);
// 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
// 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);}}
```

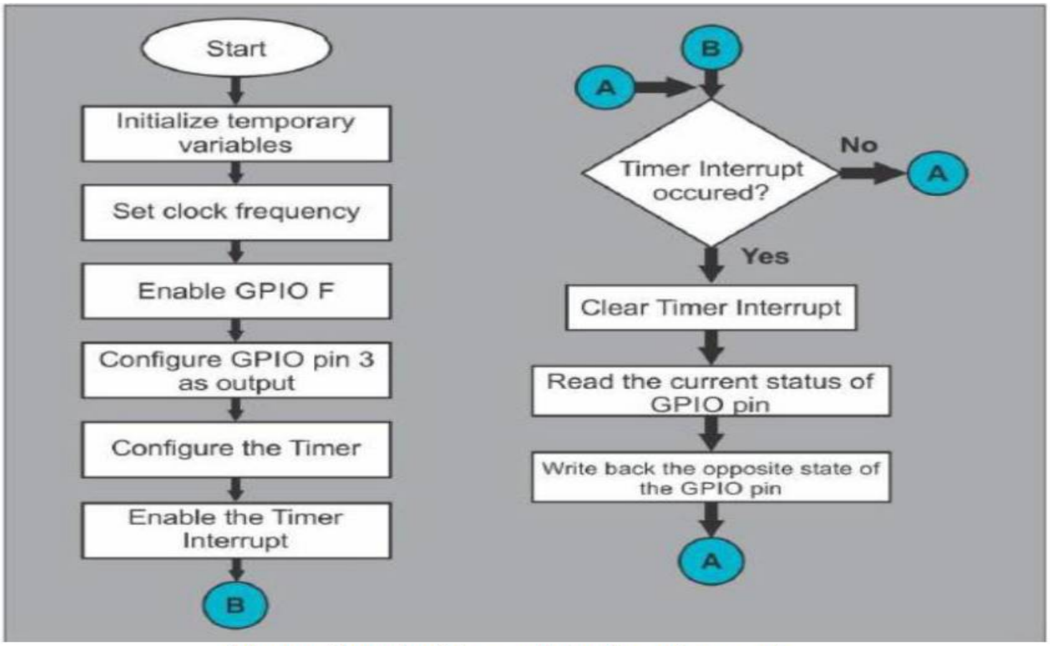

Start
Initialize temporary variables
Set clock frequency
Enable GPIO F
Configure GPIO pin 3 as output
Configure the Timer
Enable the Timer Interrupt
B
B
A
Timer Interrupt occured?
No
A
Yes
Clear Timer Interrupt
Read the current status of GPIO pin
Write back the opposite state of the GPIO pin
A

Figure: Flowchart for servicing timer interrupts

4.8. 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. 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.

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.

The GPT Module is one timing resource available on the Tiva™ C Series microcontrollers. Other timer resources include the System Timer (SysTick) and the PWM timer in PWM modules

The General-Purpose Timer Module (GPTM) blocks with the following functional options:

16/32-bit operating modes:

1. 16- or 32-bit programmable one-shot timer
2. 16- or 32-bit programmable periodic timer
3. 16-bit general-purpose timer with an 8-bit prescaler
4. 32-bit Real-Time Clock (RTC) when using an external 32.768-KHz clock as the input
5. 16-bit input-edge count- or time-capture modes with an 8-bit prescaler
6. 16-bit PWM mode with an 8-bit prescaler and software-programmable output inversion of
7. the PWM signal

32/64-bit operating modes:

1. 32- or 64-bit programmable one-shot timer
2. 32- or 64-bit programmable periodic timer
3. 32-bit general-purpose timer with a 16-bit prescaler
4. 64-bit Real-Time Clock (RTC) when using an external 32.768-KHz clock as the input
5. 32-bit input-edge count- or time-capture modes with a16-bit prescaler
6. 32-bit PWM mode with a 16-bit prescaler and software-programmable output inversion of
7. the PWM signal

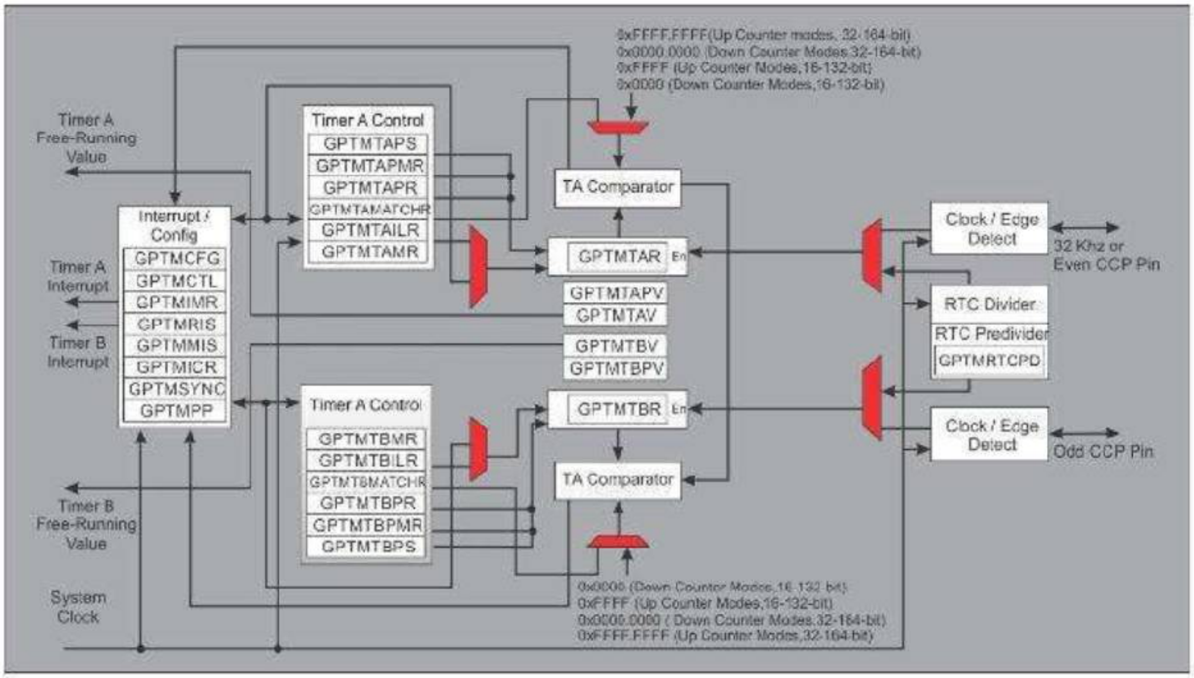


Fig : GPTM block diagram

- Count up or down
- Twelve 16/32-bit Capture Compare PWM pins (CCP)
- Twelve 32/64-bit Capture Compare PWM pins (CCP)
- Daisy chaining of timer modules to allow a single timer to initiate multiple timing events
- Timer synchronization allows selected timers to start counting on the same clock cycle
- ADC event trigger
- User-enabled stalling when the microcontroller asserts CPU Halt flag during debug (excluding RTC mode)
- Ability to determine the elapsed time between the assertion of the timer interrupt and entry into the interrupt service routine
- Efficient transfers using Micro Direct Memory Access Controller (μDMA)
  1. Dedicated channel for each timer
  2. Burst request generated on timer interrupt

4.8.1. Basic Timer:

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 of timer). For instance, we cannot get 1us resolution and a 1sec maximum period using a 16-bit timer. If we want 1us resolution we are restricted to about 65ms maximum period. If we want 1sec maximum period, we are bounded to about 16us resolution. The pre-scaler allows us to manage resolution and maximum period to fit your needs.

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 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 humanfriendly value such as e.g. 1 millisecond or 1 microsecond, or any other design specified units.

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 vent 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 event based on preset counts being achieved.

For instance, the compare registers can be 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.

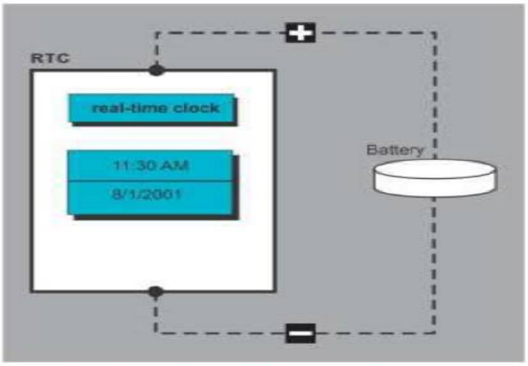
Another notable use of a compare register can be to generate a pulse with variable width. Set an output high/low when the timer is at 0, configure the compare register with value of pulse width, and on the compare event set the output low/high. We may use a second compare register with a larger value, to set the pulse interval by retuning the timer on compare.

4.8.2.Real Time Clock (RTC):

The RTC module is designed to keep wall time. 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



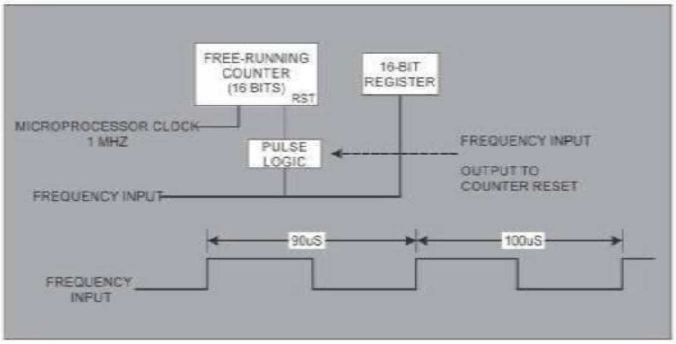


Fig: Real Time Clock with external power source ***Fig: Frequency Based Measurement System***

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 super capacitor.

4.9.Motion Control Peripherals PWM Module & Quadrature Encoder Interface (QEI):

4.9.1. Pulse Width Modulation Module (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.

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.

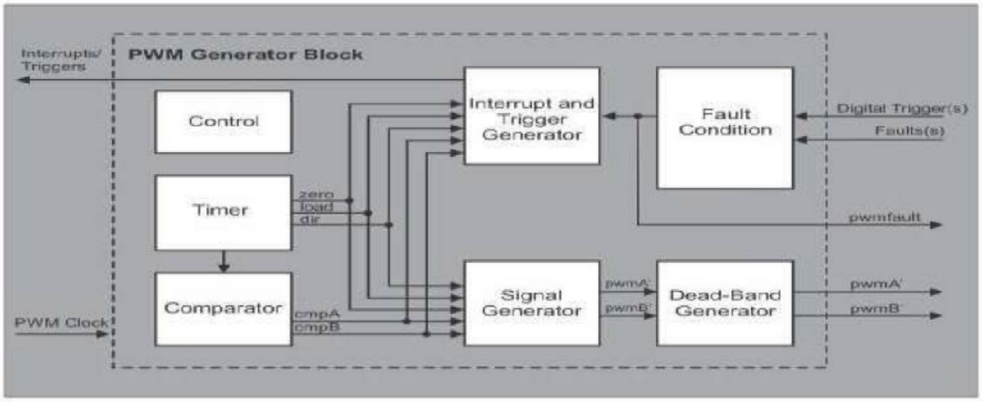


Fig 3.14. PWM Module Block Diagram

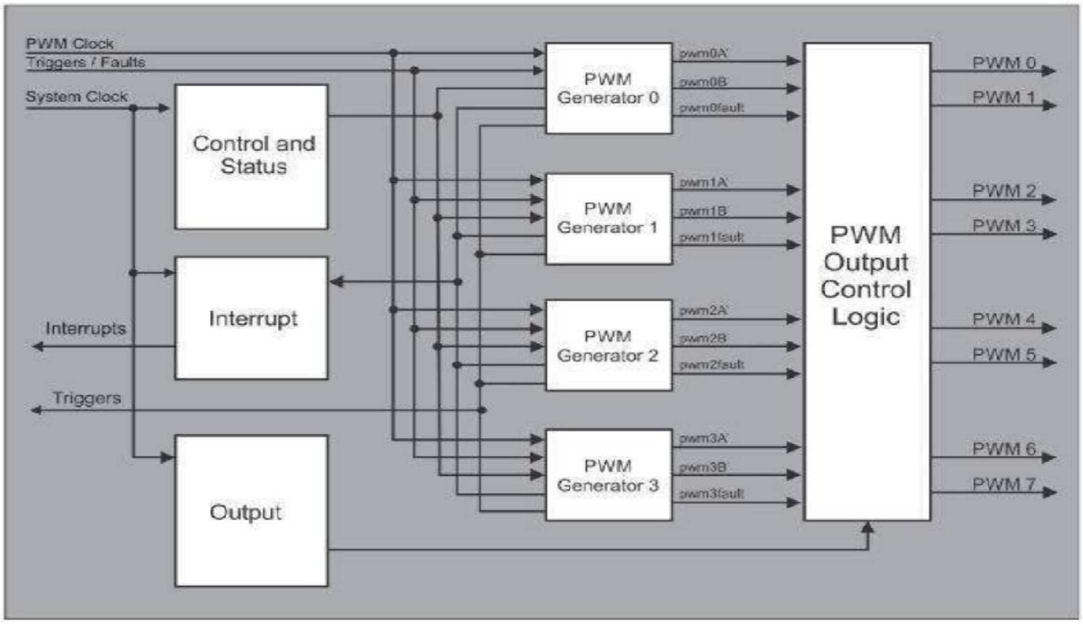


Fig 3.15. PWM Generator Block Diagram

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 deadband 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 deadband delays inserted.

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.

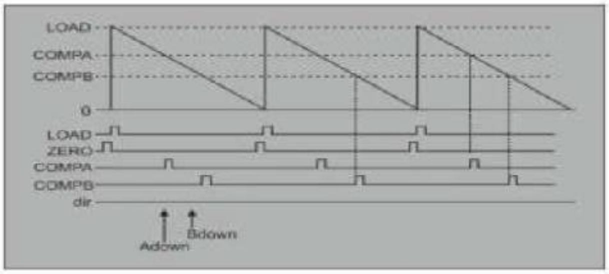
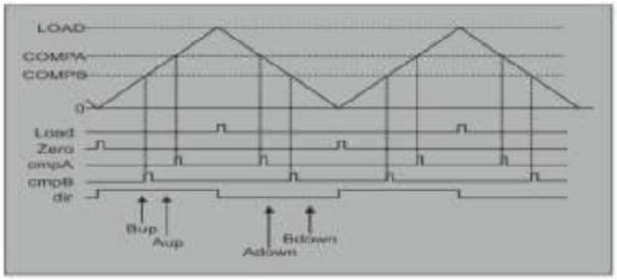


Figure (a): PWM Count- Down Mode



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

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.

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 deadband delays preventing shoot-through current from damaging the power electronics.

4.9.2. 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.

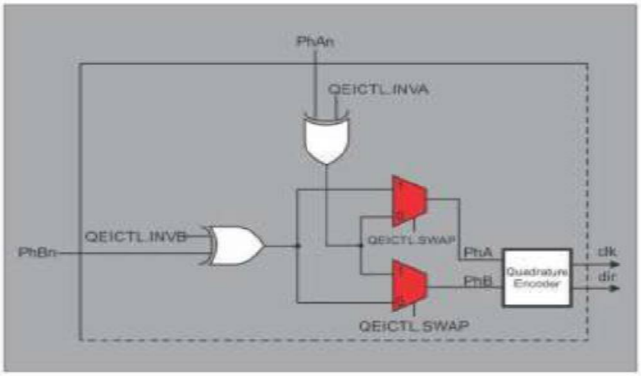


Fig: QEI Input Signal Logic

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
    - 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 to change the meaning of forward and backward and to correct for misfiring of the system. Alternatively, the phase signals can be interpreted as a clock and direction signal as output by some encoders.

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.

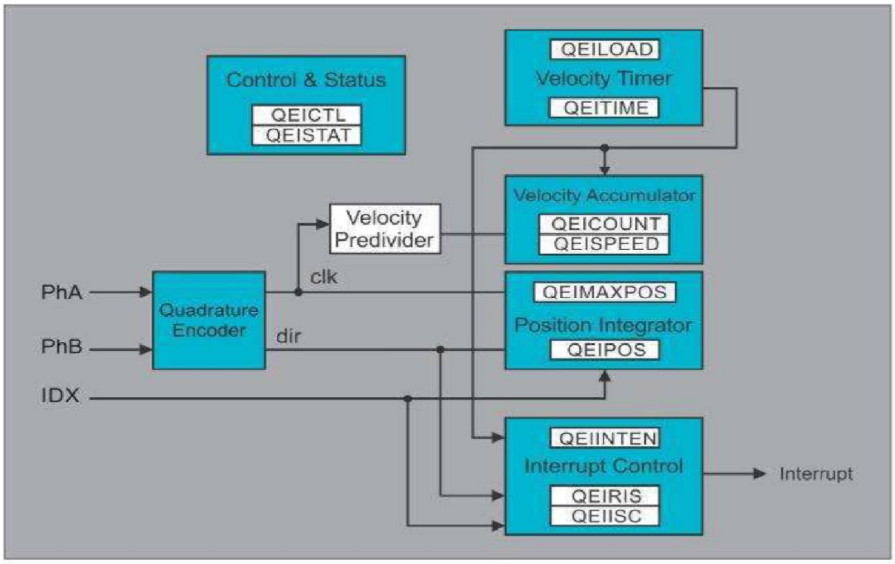


Fig : QEI Block Diagram

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.

- When RESMODE is set, the positional counter is reset when the index pulse is sensed. This mode limits the positional counter to the values [0: N-1], where N is the number of phase edges in a full revolution of the encoder wheel. The QEI Maximum Position (QEIMAXPOS) register must be programmed with N-1 so that the reverse direction from position 0 can move the position counter to N-1. In this mode, the position register contains the absolute position of the encoder relative to the index (or home) position once an index pulse has been seen.
- When RESMODE is clear, the positional counter is constrained to the range [0: M], where M is the programmable maximum value. The index pulse is ignored by the positional counter in this mode. Velocity capture uses a configurable timer and a count register. The timer counts the number of phase edges (using the same configuration as for the position integrator) in a given time period.

The edge count from the previous time period is available to the controller via the QEI Velocity (QEISPEED) register, while the edge count for the current time period is being accumulated in the QEI Velocity Counter (QEICOUNT) register. As soon as the current time period is complete, the total number of edges counted in that time period is made available in the QEISPEED register (overwriting the previous value), the QEICOUNT register is cleared, and counting commences on a new time period. The number of edges counted in a given time period is directly proportional to the velocity of the encoder.

UNIT-V

Embedded Communications Protocols and Internet of Things

COMMUNICATION:

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 we need to know when building most electronics projects. In this series of articles, we will discuss the basics of the three most common protocols: SPI, I2C and UART.

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.

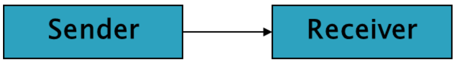
Example: printers and hard disk

Serial Communication Basics:

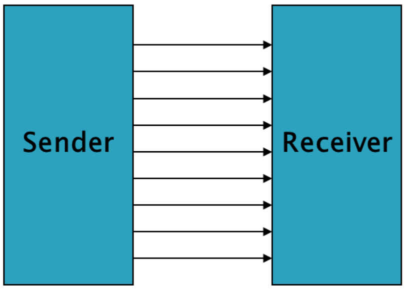
- In serial communication the data bits are transmitted serially one by one i.e. bit by bit on single communication line
- It requires only one communication line rather than n lines to transmit data from sender to receiver.
- Thus all the bits of data are transmitted on single lines in serial fashion.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

Example: Telephone.

Serial Transfer



Parallel Transfer



Serial communication uses two methods:

- Asynchronous.
- Synchronous.

Asynchronous:

- ⇨ Transfers single byte at a time.
- ⇨ No need of clock signal
  - ❖ Example: UART (universal asynchronous receiver transmitter)

Synchronous:

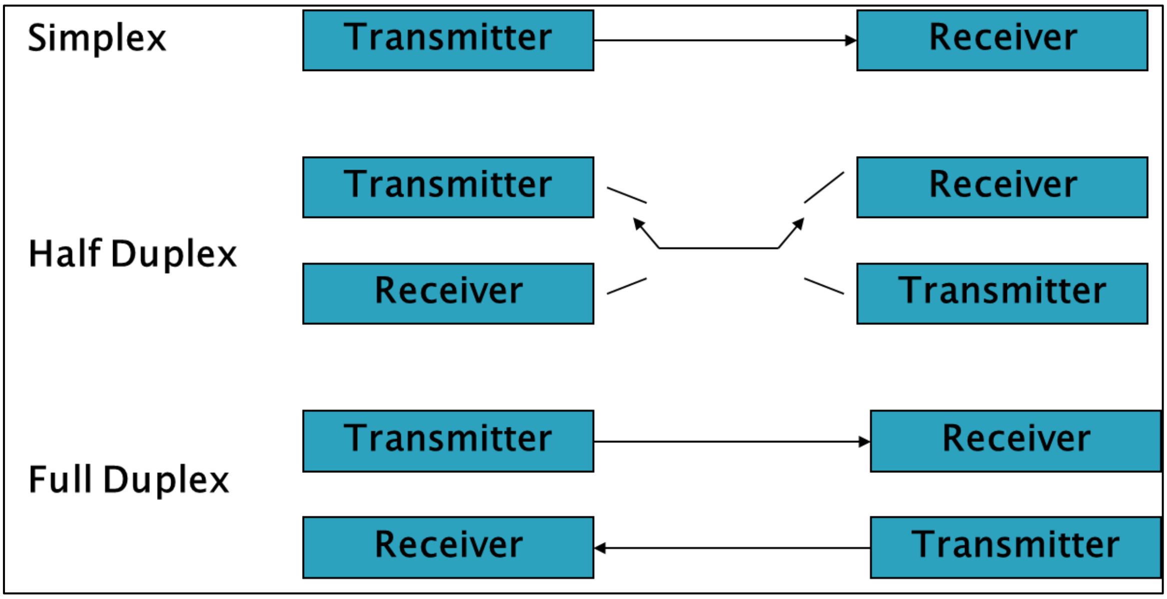
- ⇨ Transfers a block of data (characters) at a time.
- ⇨ Requires clock signal
  - ❖ Example: SPI (serial peripheral interface),
    I2C (inter integrated circuit).

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

Table: Various Serial Communication Protocols

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):

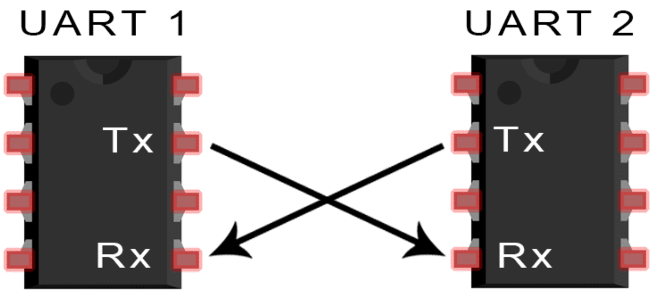
Serial communication protocols can be categorized as Synchronous and Asynchronous protocols. In synchronous communication, data is transmission and receiving is a continuous stream at a constant rate. Synchronous communication requires the clock of transmitting device and receiving device synchronized. 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), 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^{2}C 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 90ʺs 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. In the following sections, we discuss fundamentals of the above mentioned communication protocols.

UART COMMUNICATION

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.

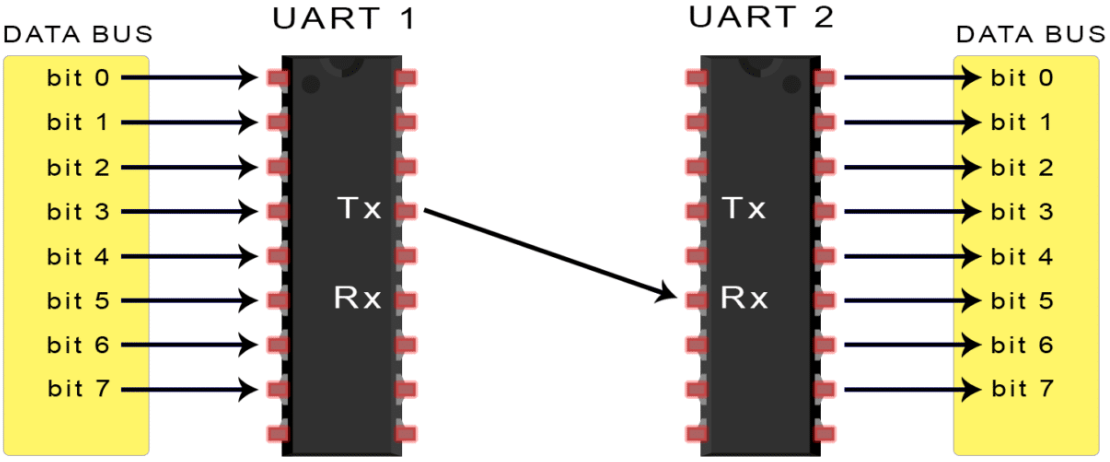
When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the *baud rate*. Baud rate is a measure of the speed of data transfer, expressed in bits per second (bps). Both UARTs must operate at about the same baud rate. The baud rate between the transmitting 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.

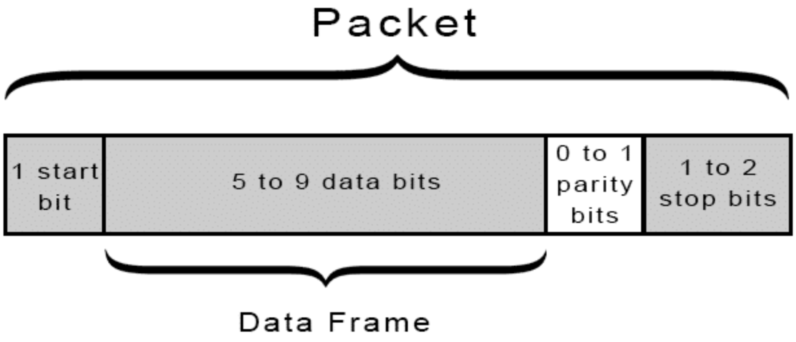
Wires Used	2
Maximum Speed	Any speed up to 115200 baud, usually 9600 baud
Synchronous or Asynchronous?	Asynchronous
Serial or Parallel?	Serial
Max # of Masters	1
Max # of Slaves	1

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 voltage level when it’s 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

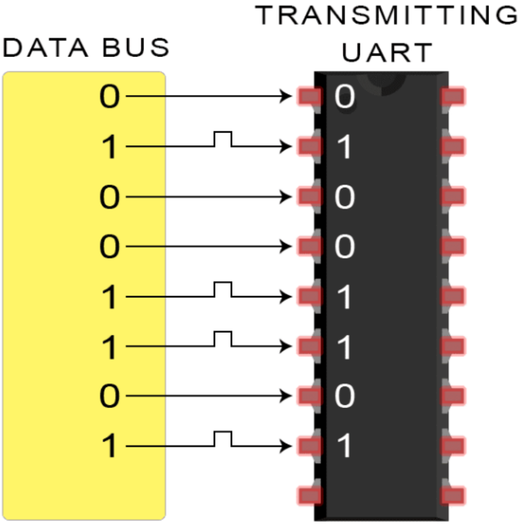
Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers. After the receiving UART reads the data frame, 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

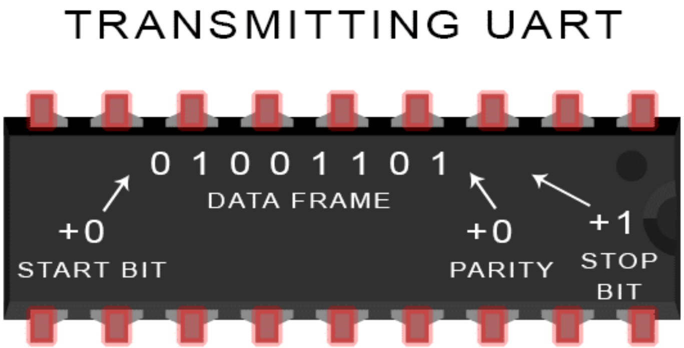
The Stop Bit, as the name suggests, marks the end of the data packet. It is usually two bits long but often only on bit is used. In order to end the transmission, the UART maintains the data line at high voltage (1).

STEPS OF UART TRANSMISSION

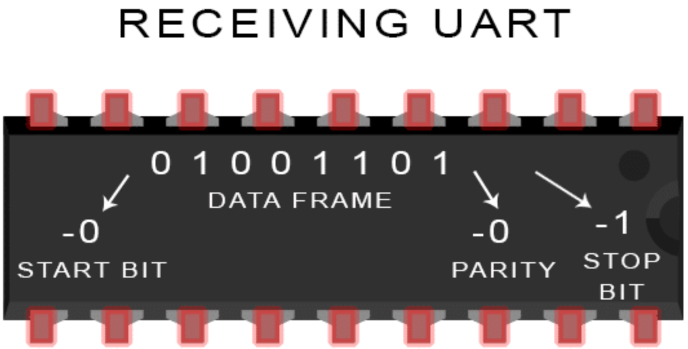
1. The transmitting UART receives data in parallel from the data bus:



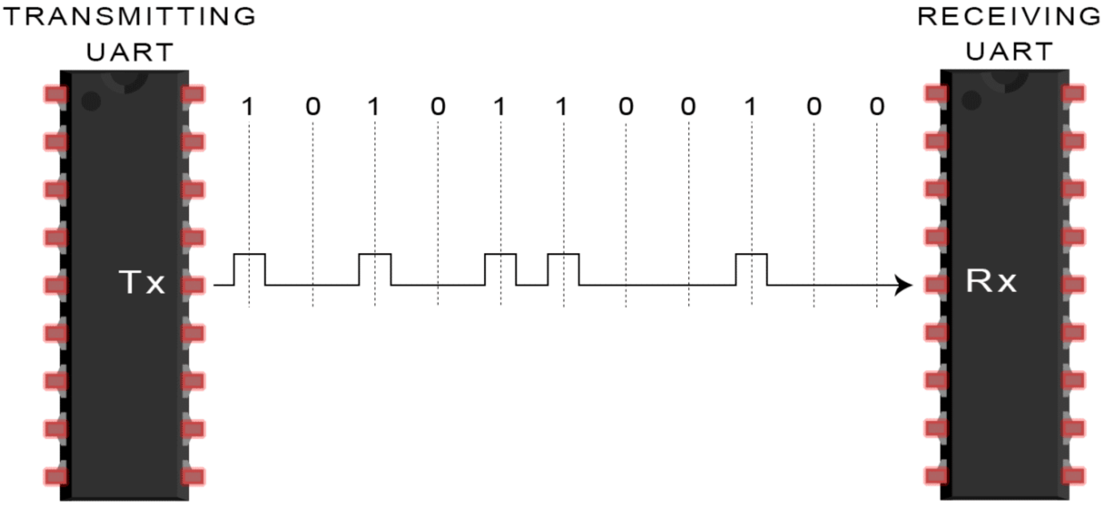
2. The transmitting UART adds the start bit, parity bit, and the stop bit(s) to the data frame:



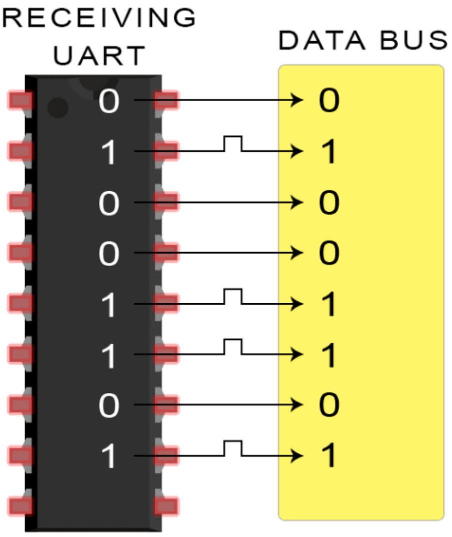
3. The entire packet is sent serially from the transmitting UART to the receiving UART. The receiving UART samples the data line at the pre-configured baud rate:



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.

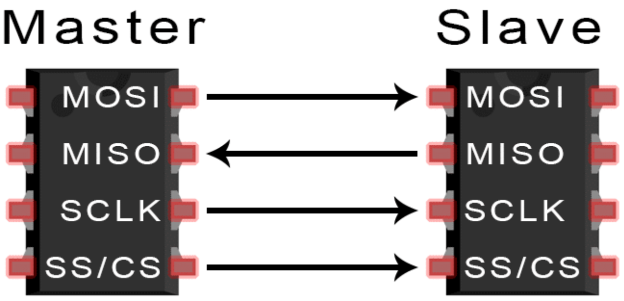
UART is one of the most simple and most commonly used Serial Communication techniques. Today, UART is being used in many applications like GPS Receivers, Bluetooth Modules, GSM and GPRS Modems, Wireless Communication Systems, RFID based applications etc.

SPI COMMUNICATION PROTOCOL

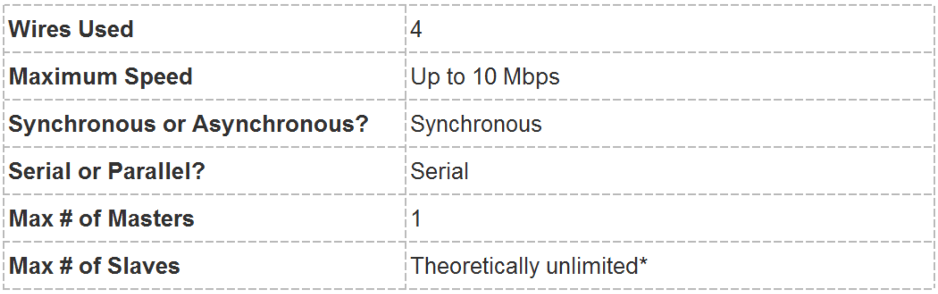
SPI is a common communication protocol used by many different devices. For example, SD card modules, RFID card reader modules, and 2.4 GHz wireless transmitter/receivers all use SPI to communicate with microcontrollers.

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, in UART communication, both sides are set to a pre-configured baud rate that dictates the speed and timing of data transmission.

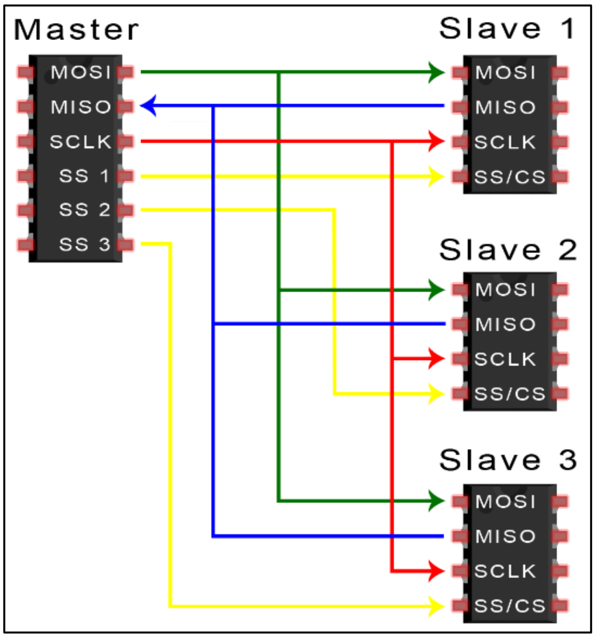
The clock signal in SPI can be modified using the properties of *clock polarity* and *clock phase*. These two properties work together to define when the bits are output and when they are sampled. Clock polarity can be set by the master to allow for bits to be output and sampled on either the rising 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

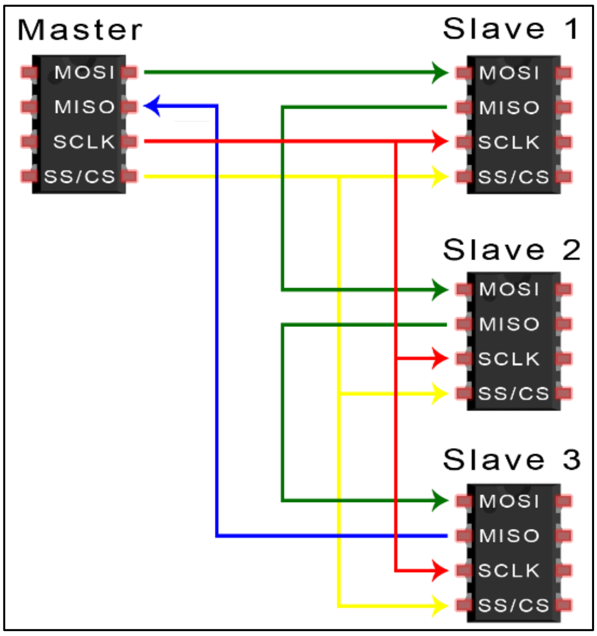
The master can choose which slave it wants to talk to by setting the slave’s CS/SS line to a low voltage level. In the idle, non-transmitting state, the slave select line is kept at a high voltage level. Multiple CS/SS pins may be available on the master, which allows for multiple slaves to be wired 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:



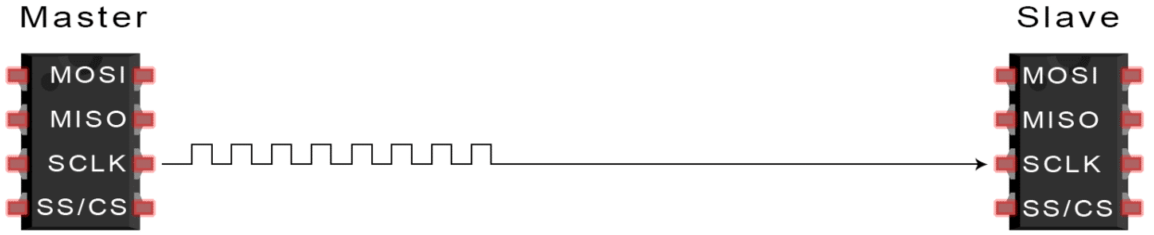
MOSI AND MISO

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.

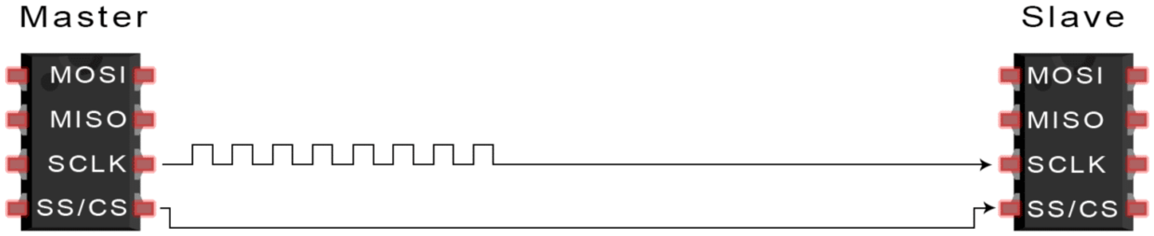
The slave can also send data back to the master through the MISO line in serial. The data sent from the slave back to the master is usually sent with the least significant bit first.

STEPS OF SPI DATA TRANSMISSION

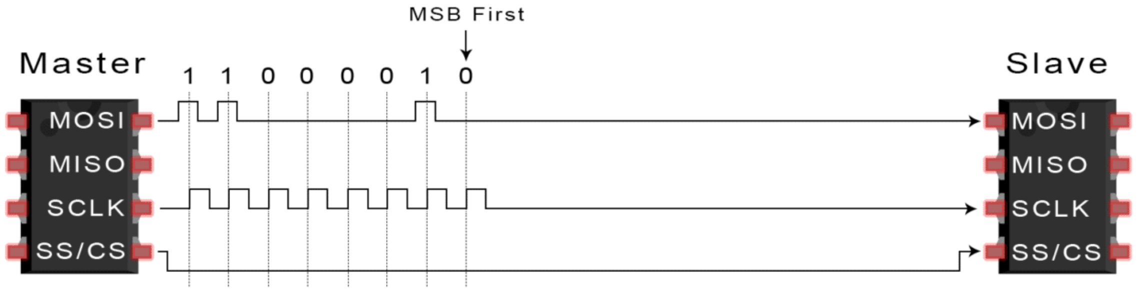
1. The master outputs the clock signal:



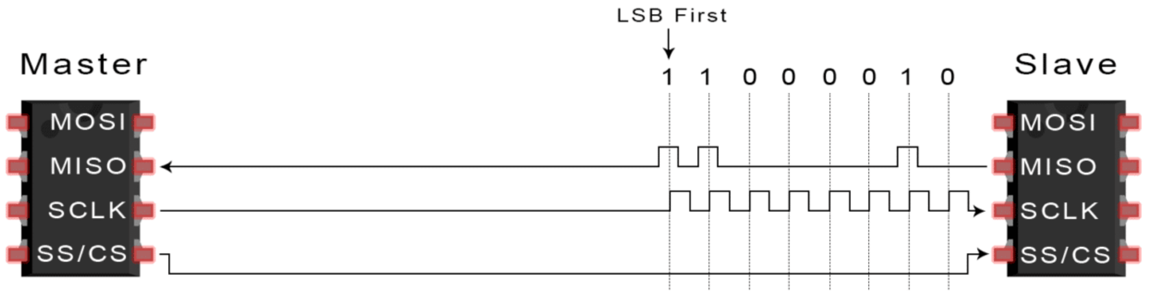
2. The master switches the SS/CS pin to a low voltage state, which activates the slave:



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 AND DISADVANTAGES OF SPI

There are some advantages and disadvantages to using SPI, and if given the choice between different communication protocols, you should know when to use SPI according to the requirements of your project:

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

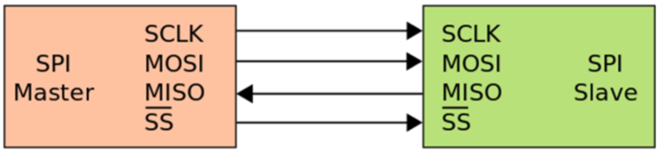


Fig: SPI Master connected to a single slave

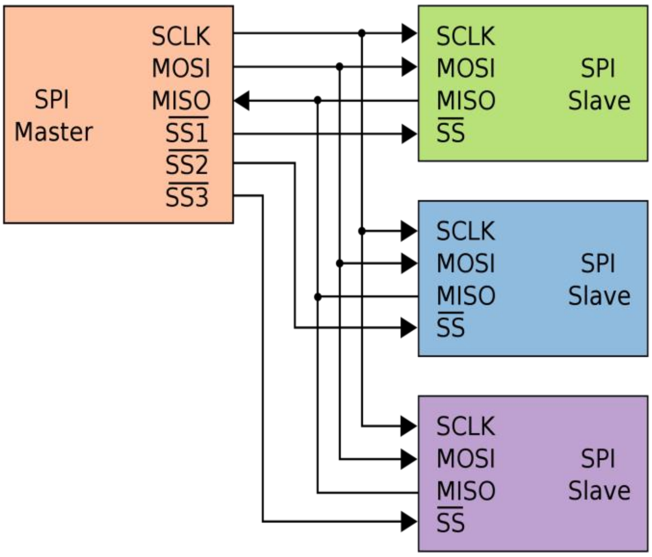


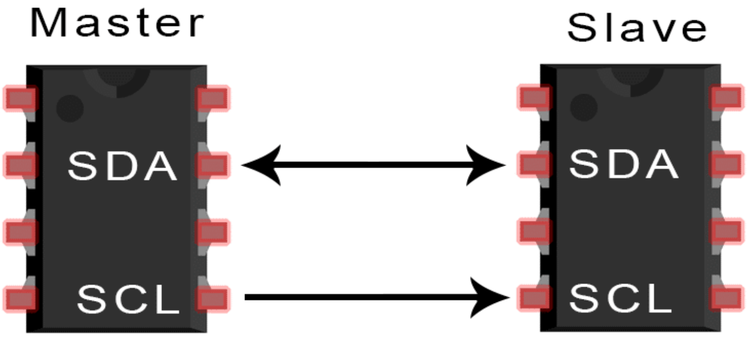
Fig: SPI master connected to multiple slaves

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

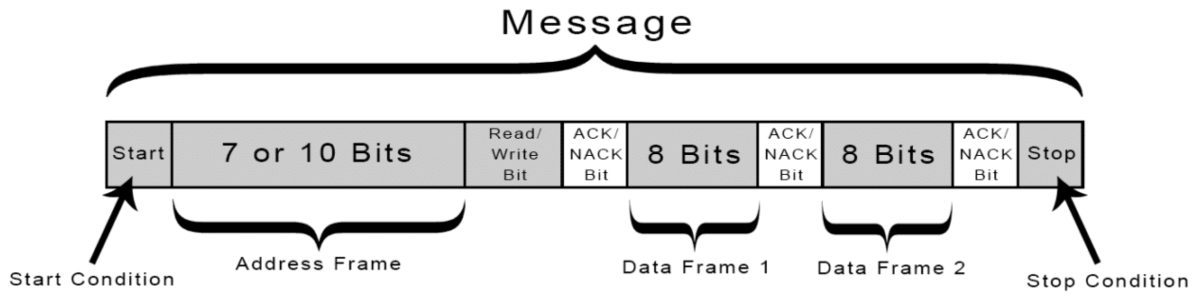
GENERAL ELECTRICAL CHARACTERISTICS OF I2C

To implement I2C (For TIVA series microcontrollers or for most of the microcontrollers) a 4.7kilo ohm pull-up resistor for each line is needed. This is required to implement wired-AND logic in IIC.

More than 100 devices can be connected to I2C bus theoretically. It is better to restrict to 15 devices for better performance of the network. Each device is called as node. Nodes which generates clock are called Master nodes and devices which work based on the clock generated by master node are called Slave nodes. Generally, master nodes initiate and terminate the transmission. The four possible modes of operation are: master transmitter, master receiver, slave transmitter and slave receiver.

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:



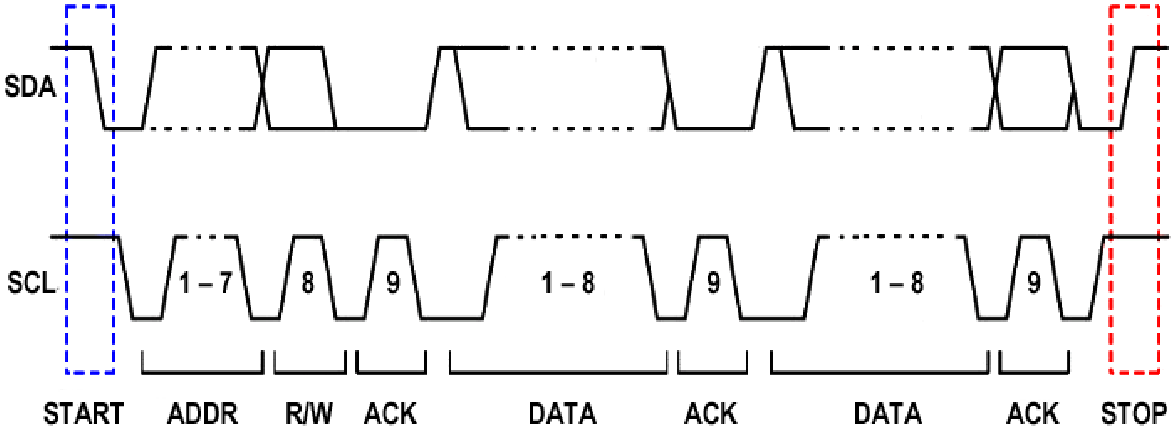
Start Condition: The SDA line switches from a high voltage level to a low voltage level *before* the SCL line switches from high to low.

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 lines like SPI, so it needs another way to let the slave 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 master. If the address doesn't 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

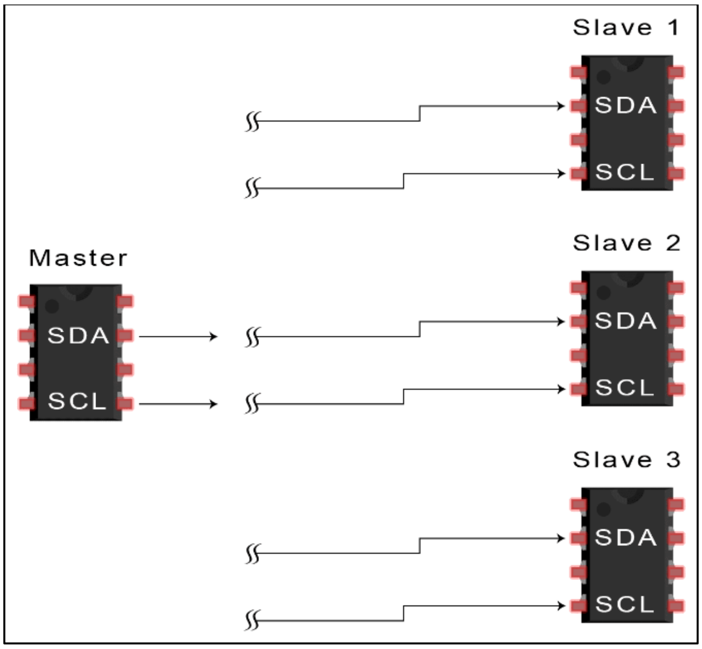
After the master detects the ACK bit from the slave, the first data frame is ready to be sent.

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.

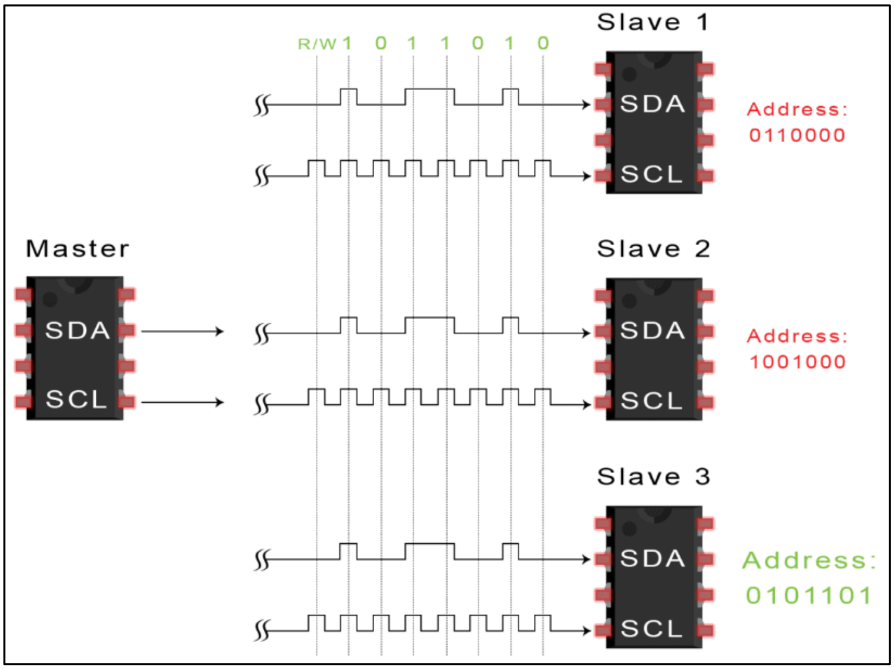
After all of the data frames have been sent, the master can send a stop condition to the slave to halt the transmission. The stop condition is a voltage transition from low to high on the SDA line after a low to high transition on the SCL line, with the SCL line remaining high.

STEPS OF I2C DATA TRANSMISSION

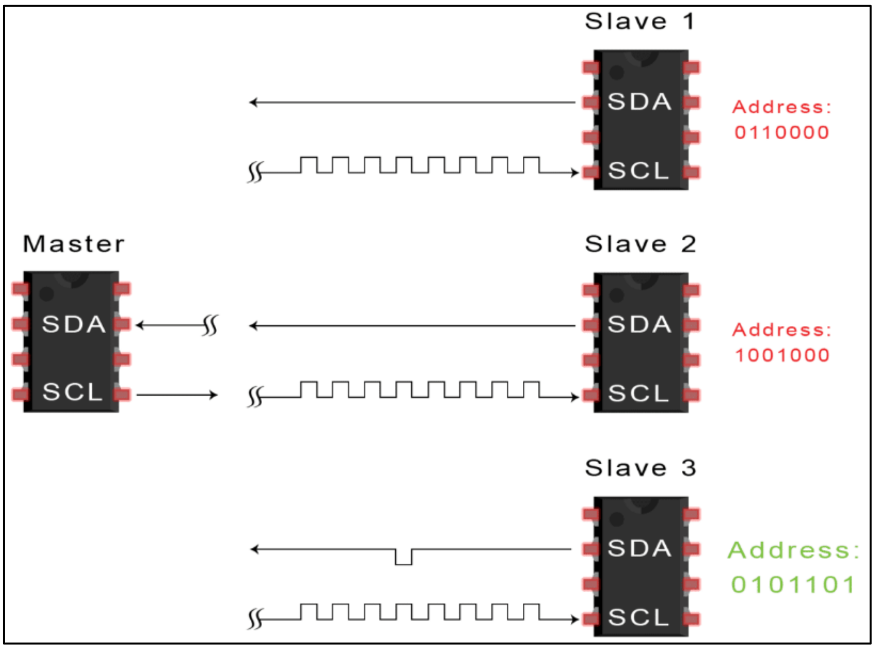
1. The master sends the start condition to every connected slave by switching the SDA line from a high voltage level to a low voltage level *before* switching the SCL line from high to low:



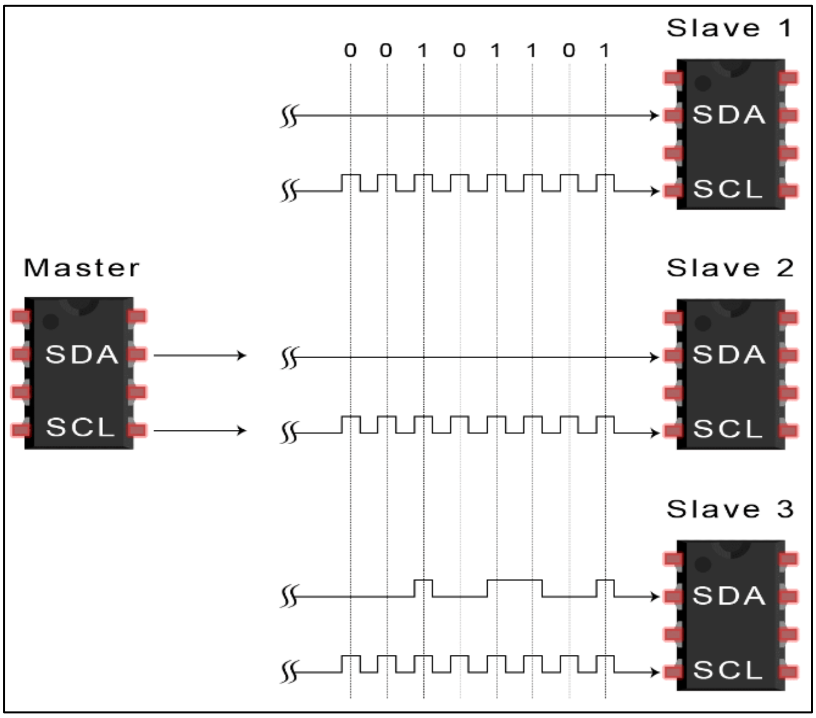
2. The master sends each slave the 7 or 10 bit address of the slave it wants to communicate with, along with the read/write bit:



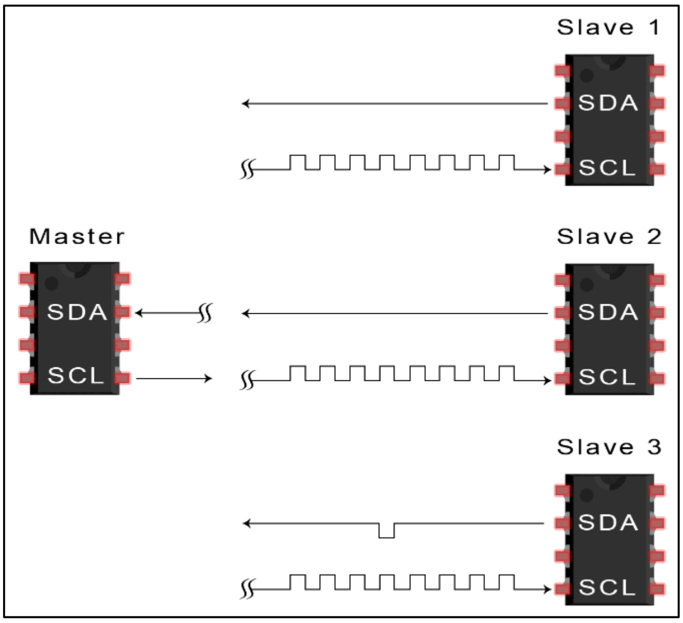
3. Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave's own address, the slave leaves the SDA line high.



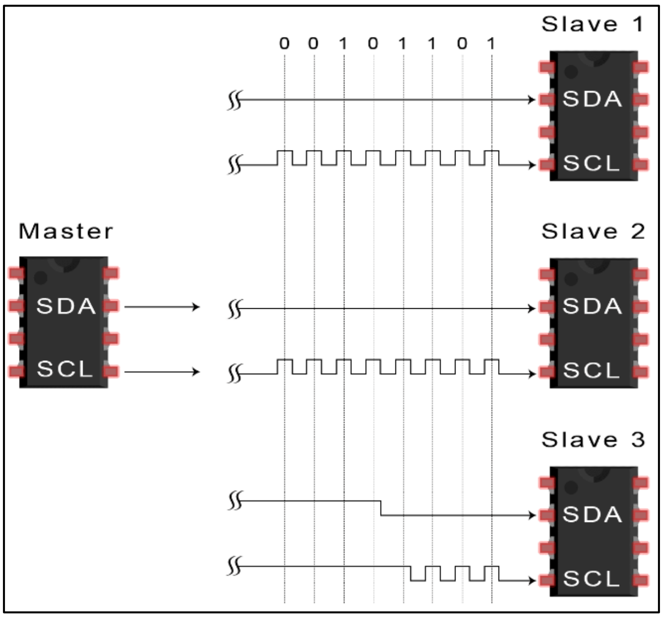
4. The master sends or receives the data frame:



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:

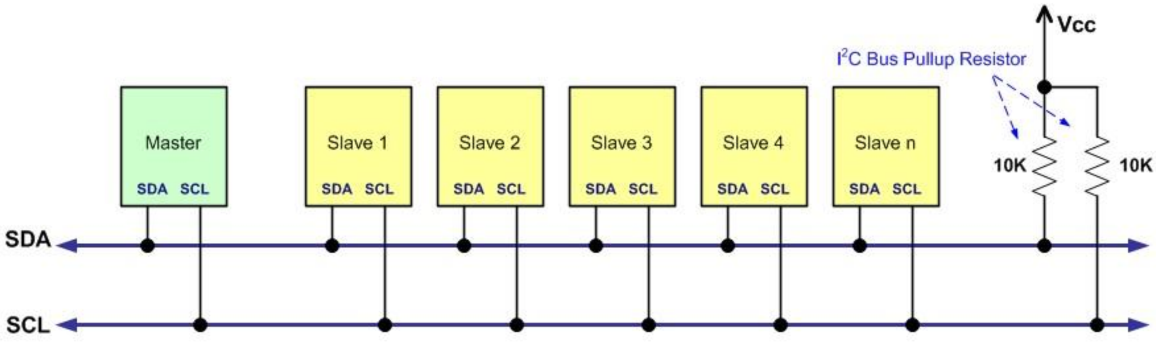
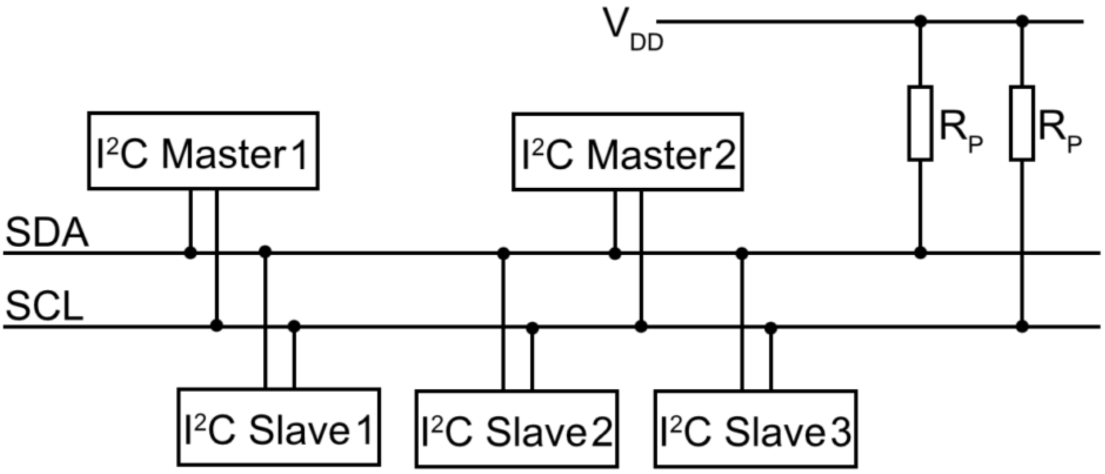


Figure: I2C Bus Connection

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 the message. To connect 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 AND DISADVANTAGES OF I2C

There is a lot to I2C that might make it sound complicated compared to other protocols, but there are some good reasons why you may or may not want to use I2C to connect to a particular device:

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 also allows hot swapping. The "hot-swapping" means that the devices can be plugged and unplugged without rebooting the computer or turning off the device. That means, when plugged in, everything configures automatically. Once the user is finished, they can simply unplug the cable out; the host will detect its absence and automatically unload the driver. This makes the USB a plug-and-play interface between a computer and add-on devices.

USB is now the most used interface to connect devices like mouse, keyboards, PDAs, game-pads and joysticks, scanners, digital cameras, printers, personal media players, and flash drives to personal computers.

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: 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. The USB 2.0 specification was released in April 2000 and was standardized at the end of 2001.

Supporting three speed modes (1.5, 12 and 480 Mbps), USB 2.0 supports low-bandwidth devices such as keyboards and mice, as well as high-bandwidth ones like high-resolution Web-cams, scanners, printers and high-capacity storage systems.

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: USB 3.0 is the latest version of USB release. It is also called as Super-Speed USB having a data transfer rate of 4.8Gbps (600 MB/s). That means it can deliver over 10x the speed of today's Hi-Speed USB connections.

The USB 3.0 specification was released by Intel and its partners in August 2008. Products using the 3.0 specifications are come out in 2010.

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.

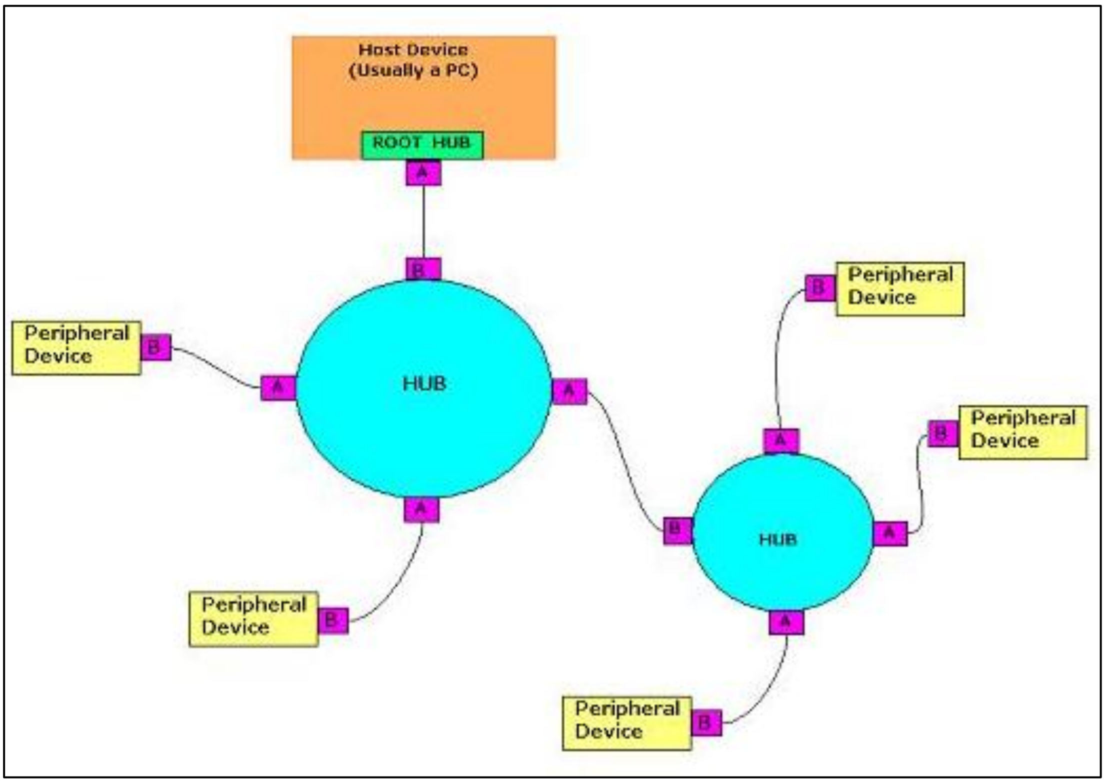


Fig: The USB "tiered star" topology

USB connectors:

Connecting a USB device to a computer is very simple -- you find the USB connector on the back of your machine and plug the USB connector into it. If it is a new device, the operating system auto-detects it and asks for the driver disk. If the device has already been installed, the computer activates it and starts talking to it.

The USB standard specifies two kinds of cables and connectors.

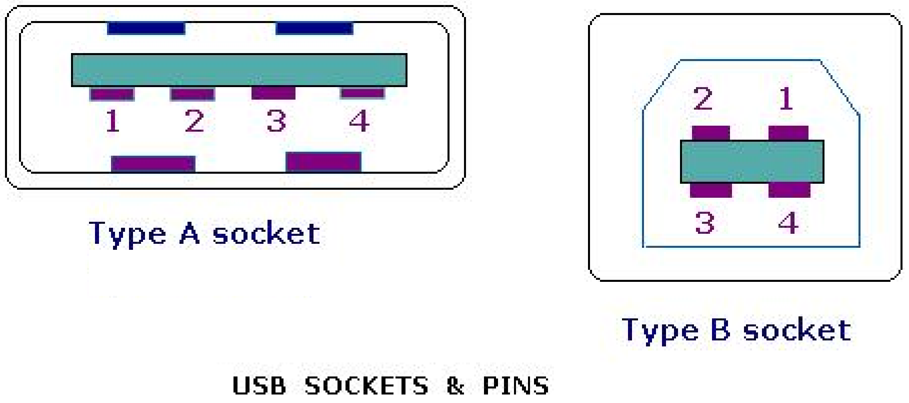


Fig: USB Type A & B Connectors

The USB standard uses "A" and "B" connectors mainly to avoid confusion:

1. "A" connectors head "upstream" toward the computer.

2. "B" connectors head "downstream" and connect to individual devices.

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
  - 5, 6, 7, or 8 data bits
  - Even, odd, stick, or no-parity bit generation/detection
  - 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)
  - 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.

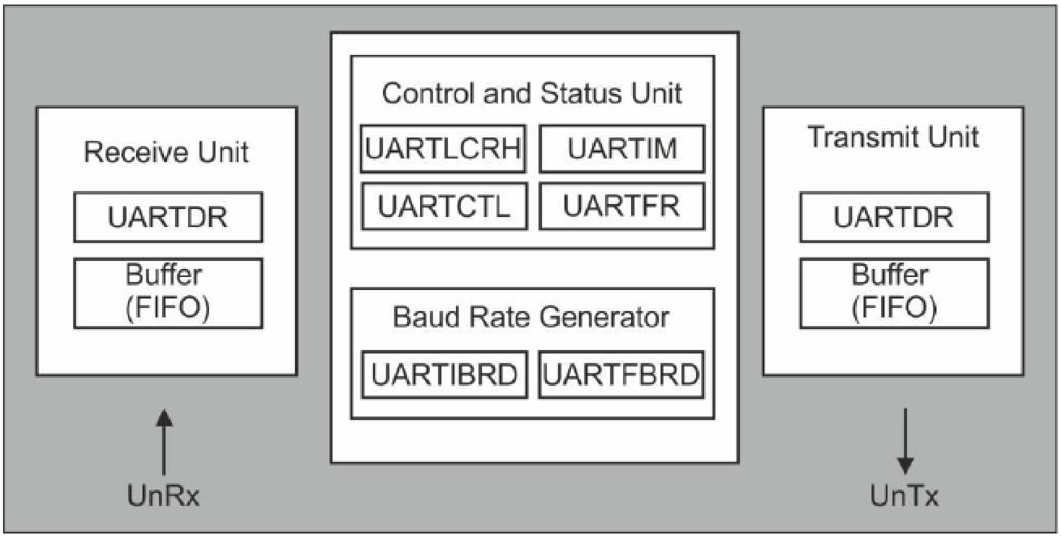


Figure: Simplified block diagram of UART

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 below:

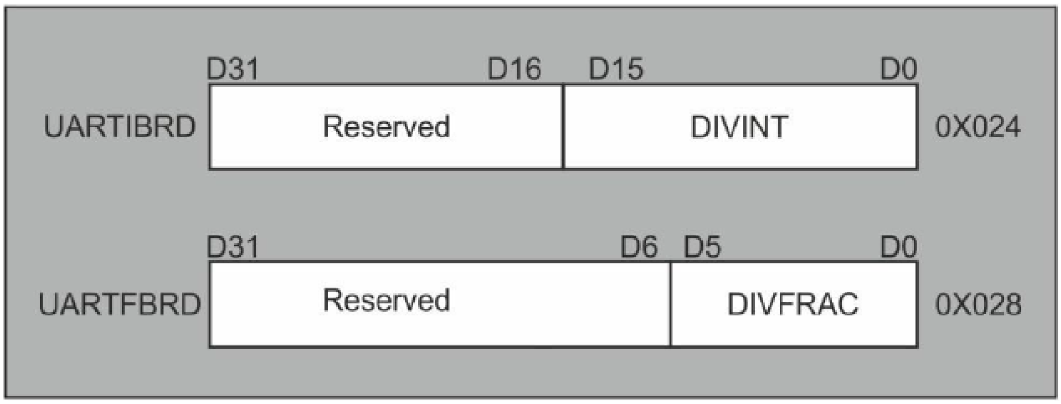


Figure: Baud Rate Registers

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:

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

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.

BRD = BRDI + BRDF = UARTSysClk / (ClkDiv * 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:

Baud Rate = 16MHz / (16 × ClkDiv) = 1MHz / 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

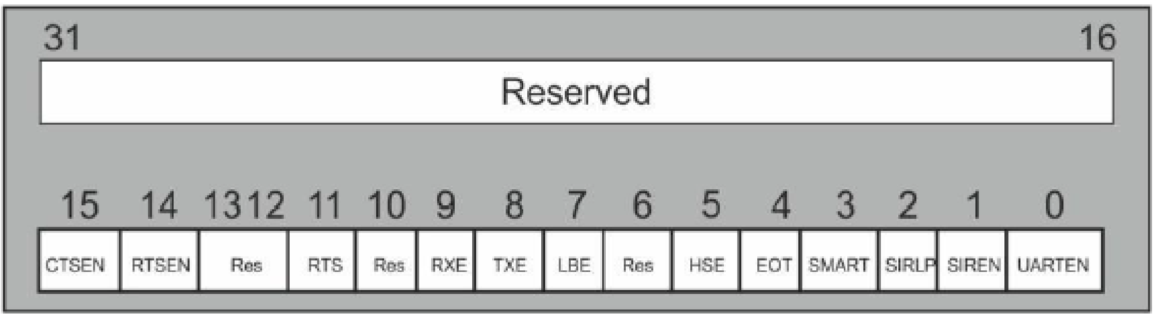
Serial IR (SIR):

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.



31 16
Reserved

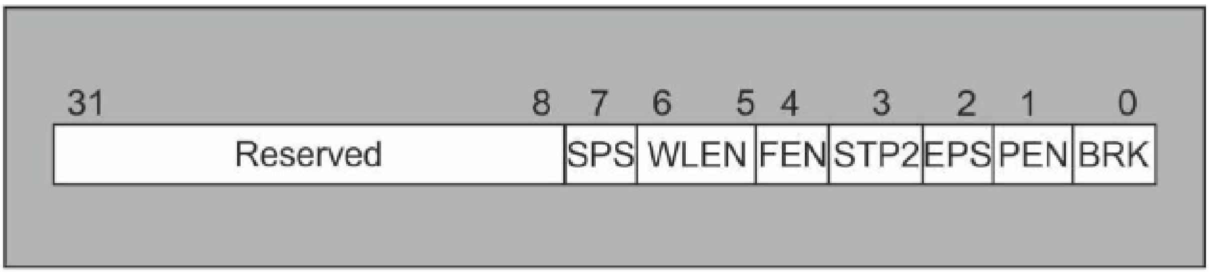
15	14	1312	11	10	9	8	7	6	5	4	3	2	1	0
CTSEN	RTSEN	Res	RTS	Res	RXE	TXE	LBE	Res	HSE	EOT	SMART	SIRLP	SIREN	UARTEN

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)

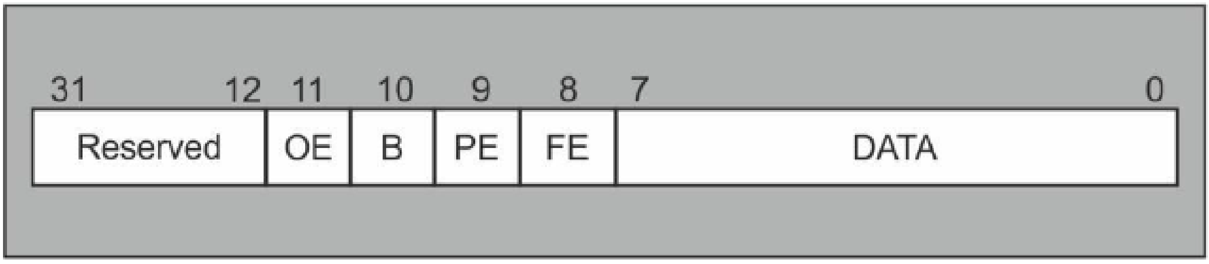
This register is used to set the length of data. The bits per character in a frame and number of stop bits are also decided.



31 8	7	6 5	4	3	2	1	0
Reserved	SPS	WLEN	FEN	STP2	EPS	PEN	BRK

- 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):



31 12	11	10	9	8	7 0
Reserved	OE	B	PE	FE	DATA

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.

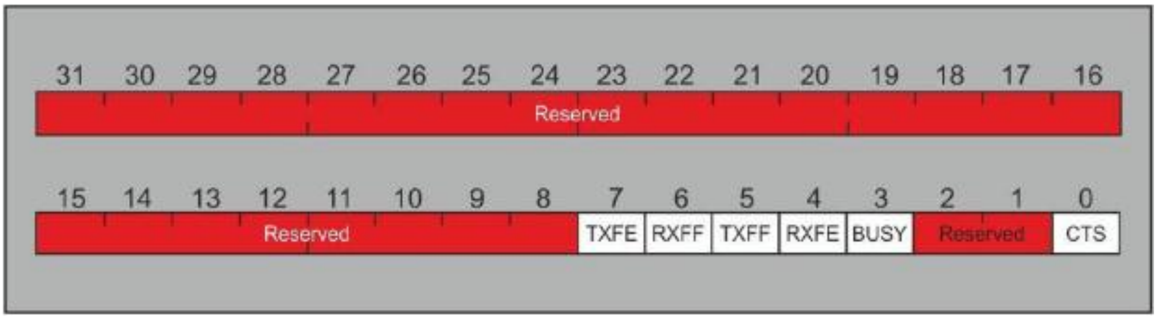


Figure: UART Flag Register (UARTFR)

- 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 */
void UART0Tx(char c)
{while((UART0->FR & 0x20) != 0);          /* wait until Tx buffer not full */
UART0->DR = c;                             /* before giving it another byte */
}
```

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 */
return c;                           /* and return it */
}
```

Implementing and Programming I2C:

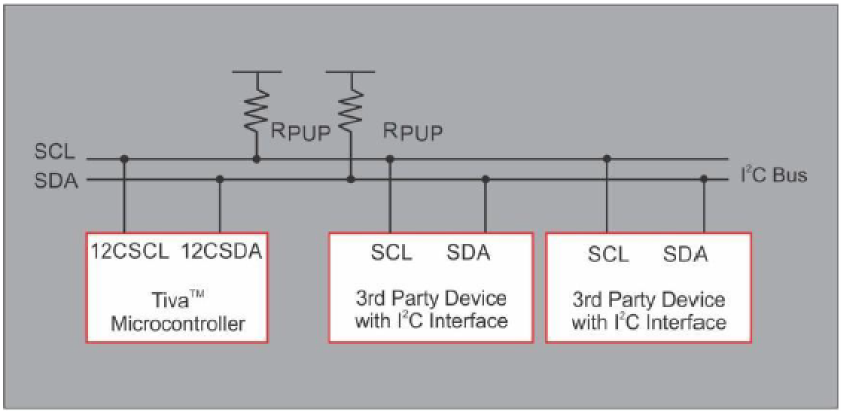
The TM4C123GH6PM controller includes four I2C modules with the following features:

- 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
  - Master transmit
  - Master receive
  - Slave transmit
  - Slave receive

- Four transmission speeds:
  - Standard (100 Kbps)
  - Fast-mode (400 Kbps)
  - Fast-mode plus (1 Mbps)
  - 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: I^{2}C 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

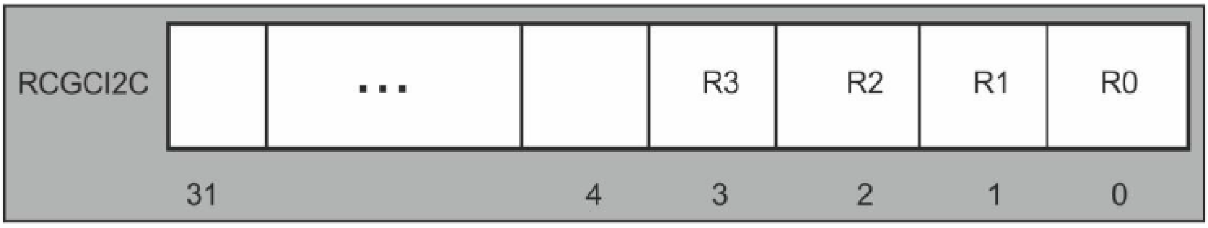


Figure: RunMode Clock Gating Control Register (RCGCI2C)

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.

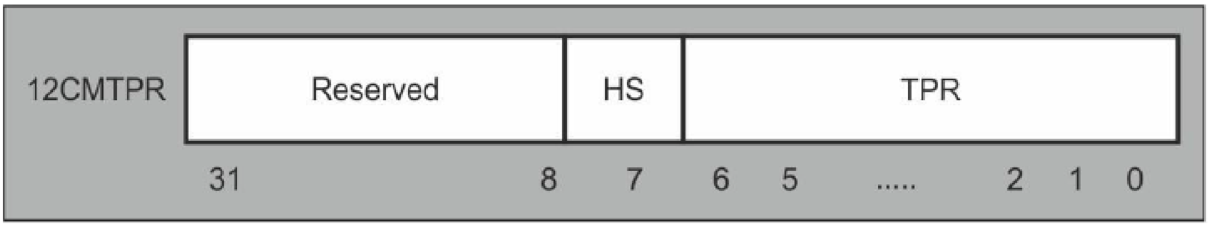
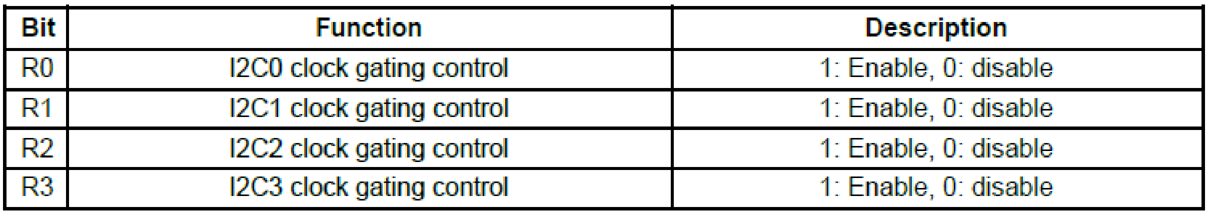


Figure: I2C Master Time Period Register

Table: RCG12C Register Description



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

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:

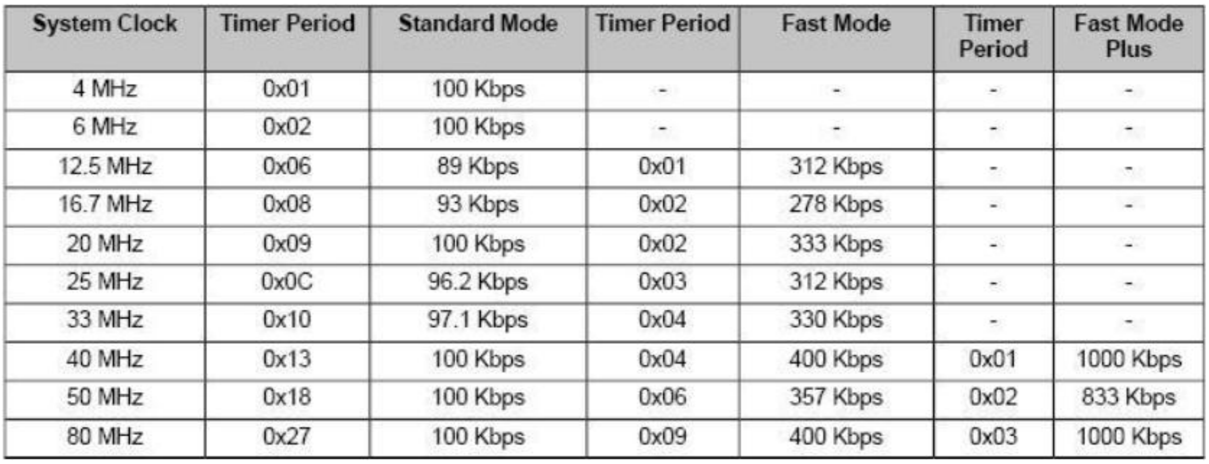
TPR = ((System clock frequency x SCL_PERIOD)/20) - 1)

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:

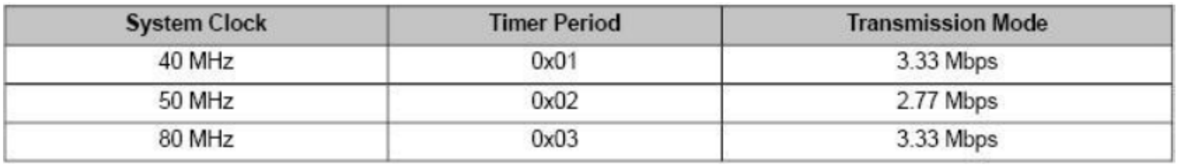
Table: TPR Values for I^{2}C modes



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:

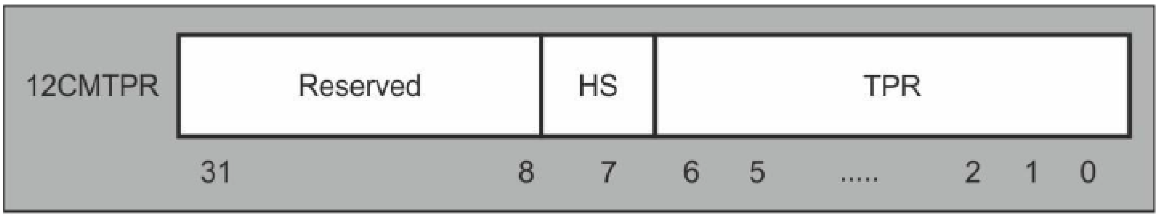
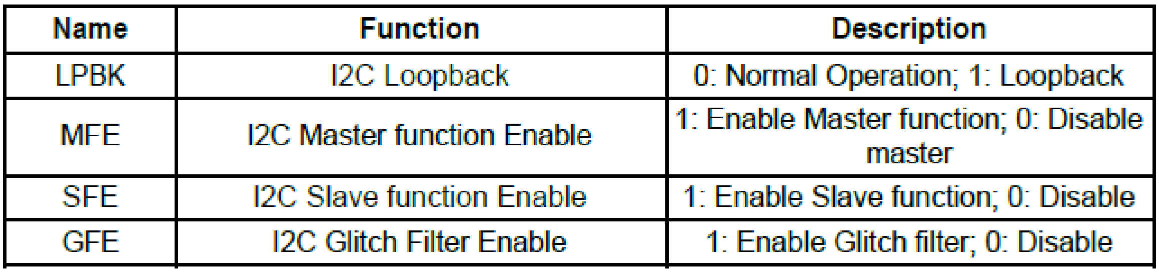


Figure: I2C Master Configuration Register

Table: I2CMCR Register Description



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

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.

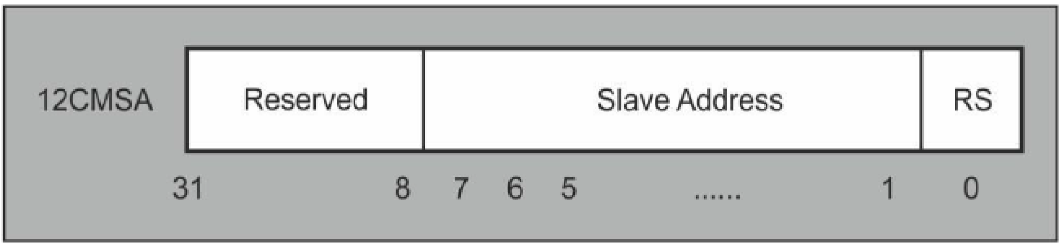


Figure: I2C Master Slave Address Register

Data Register:

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

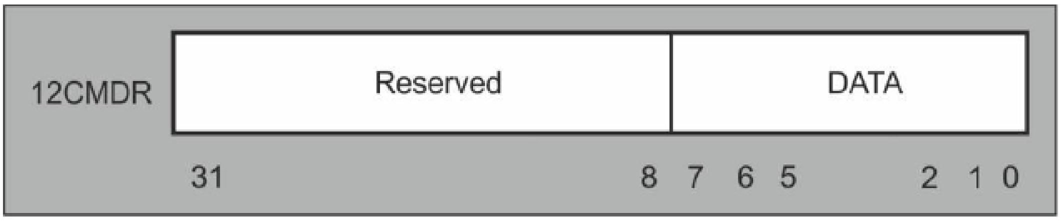


Figure: I2C Master Data Register

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.

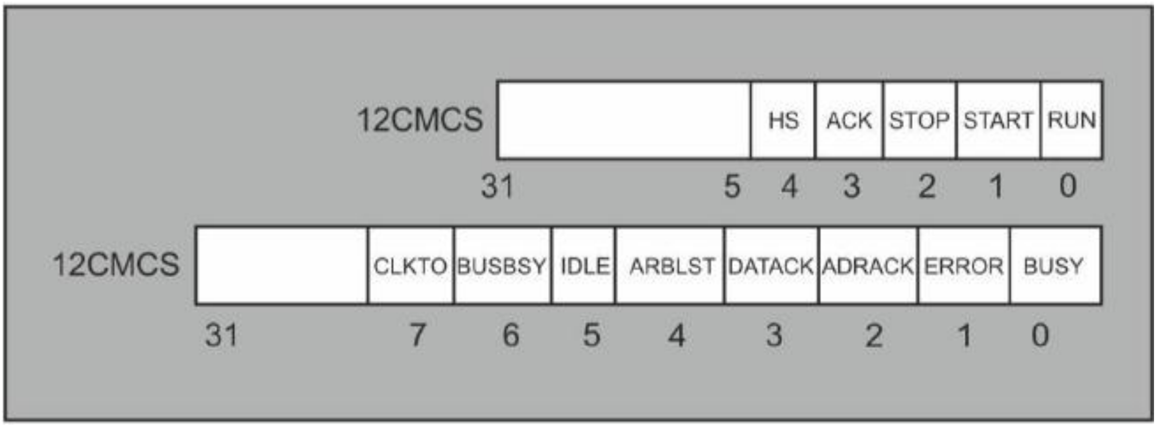


Figure: I2C Master Control/Status Register

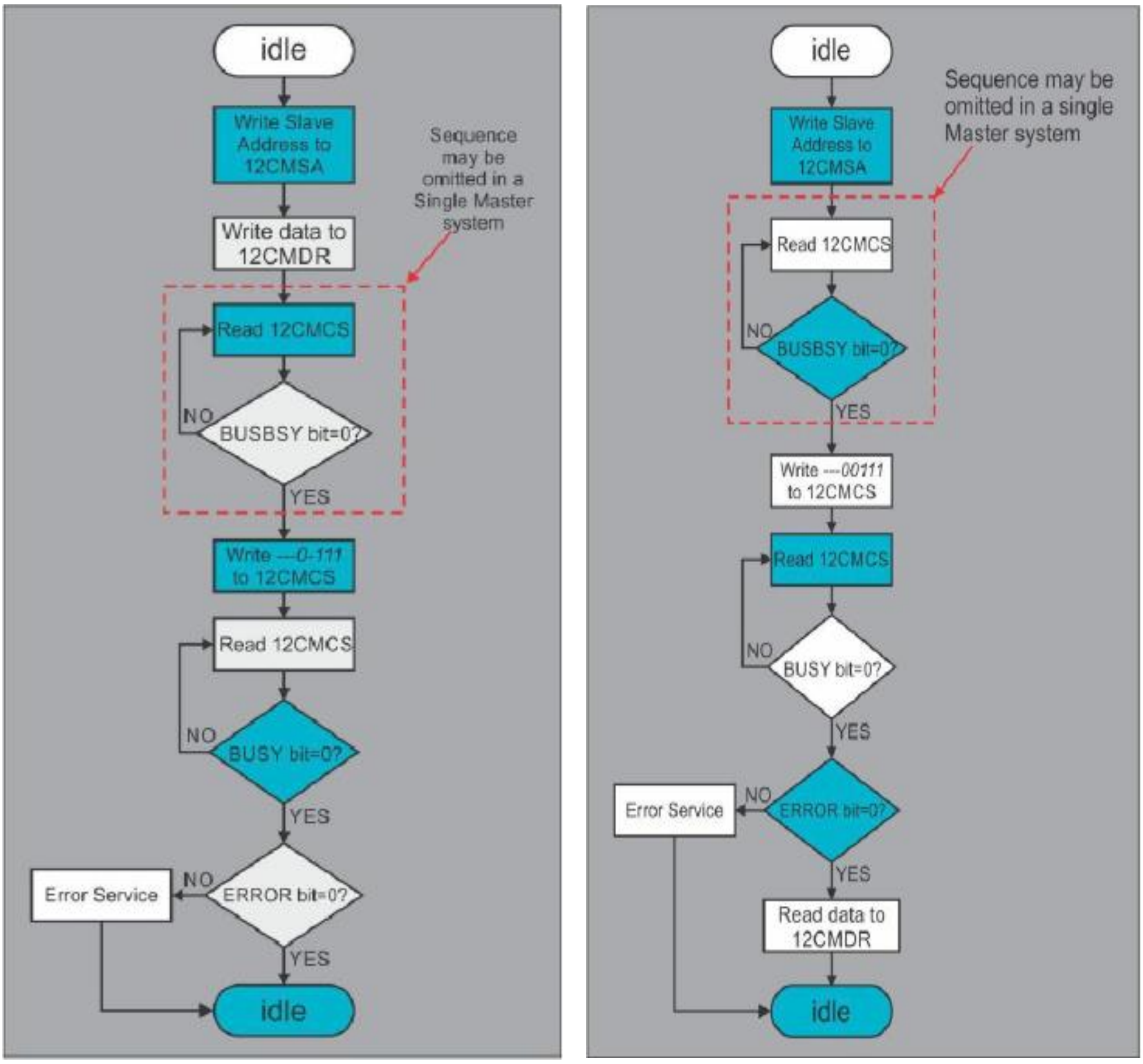
Table: I2C MCS Register Description

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

Configuring GPIO for I^{2}C Network:

GPIO pins are configured for I^{2}C 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.



(a) (b)

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.

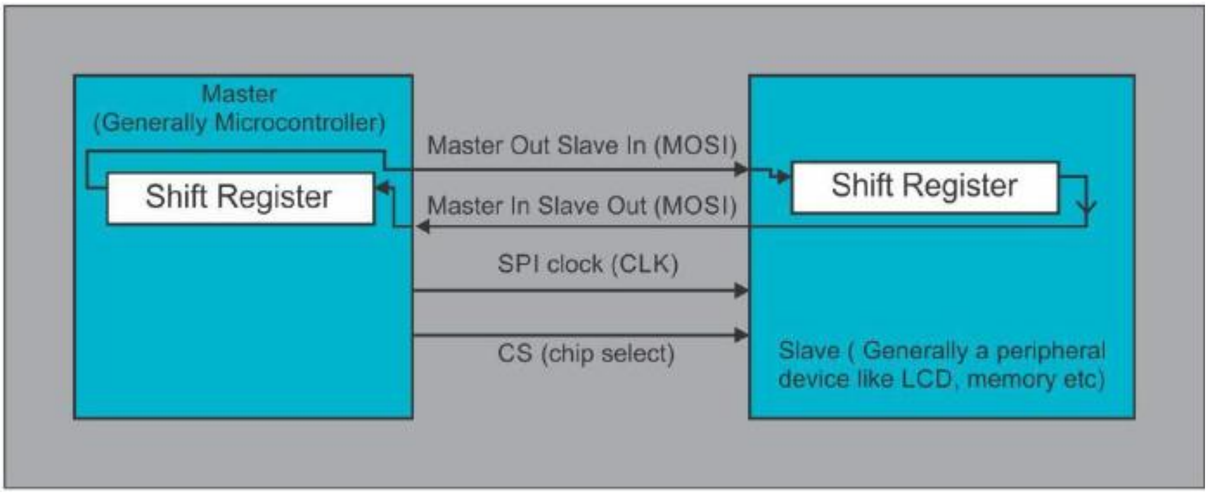

Master
(Generally Microcontroller)
Shift Register
Master Out Slave In (MOSI)
Master In Slave Out (MOSI)
Shift Register
SPI clock (CLK)
CS (chip select)
Slave (Generally a peripheral
device like LCD, memory etc)

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.

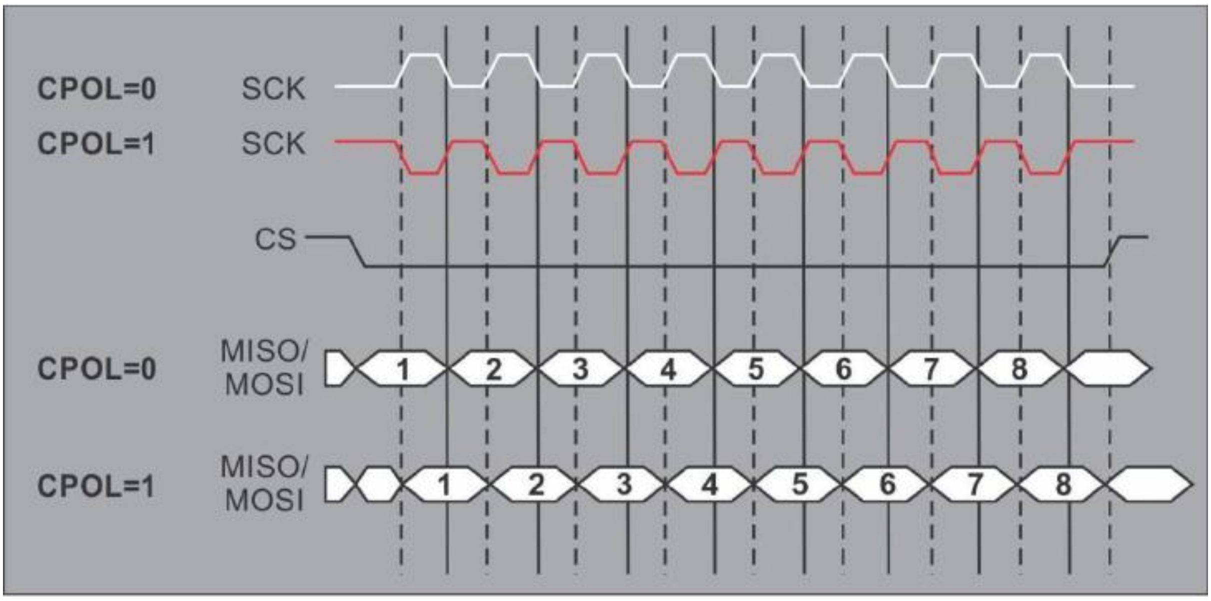
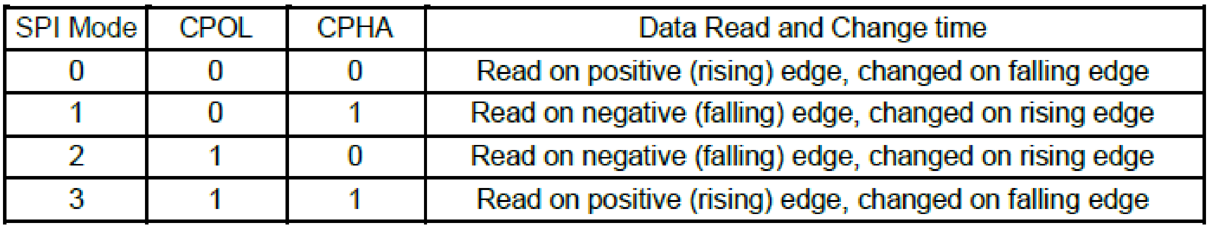


Figure: SPI Timing Diagram

Table: SPI Modes



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

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:

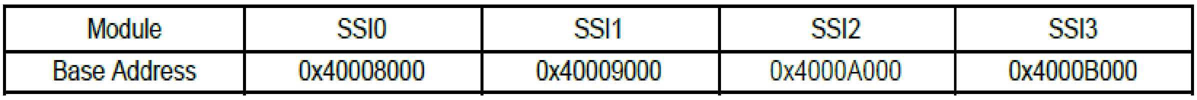
- Texas Instruments synchronous serial
- Freescale SPI
- Microwire

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.

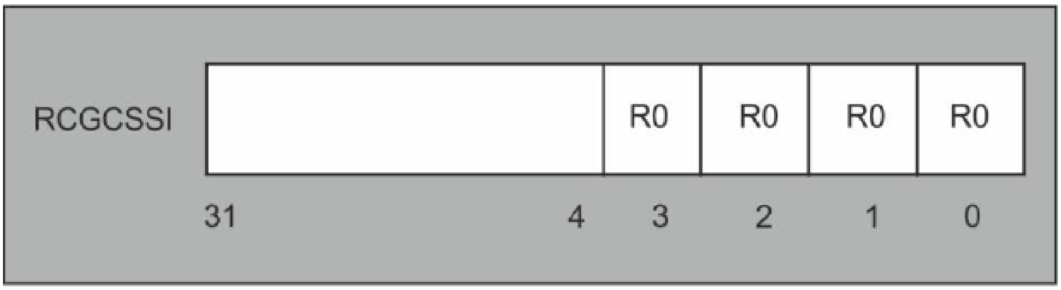


Figure: Synchronous Serial Interface Run Mode Clock Gating Control CRCG (SSI) Register

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.

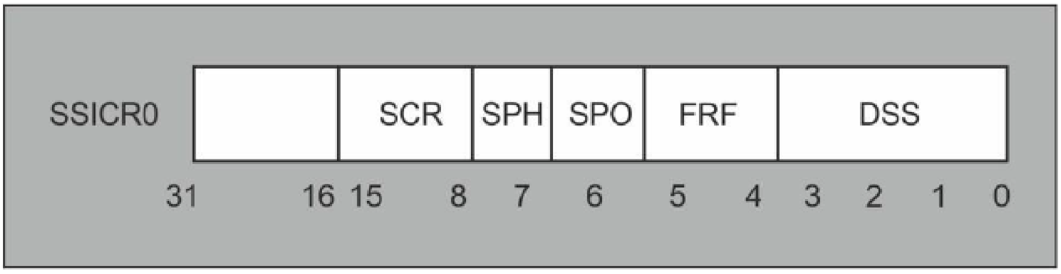


Figure: SSI Control O Register

Table: SSICRO Register Description

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))

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): $BR=SysClk/(CPSDVSR \times (1 + SCR))$

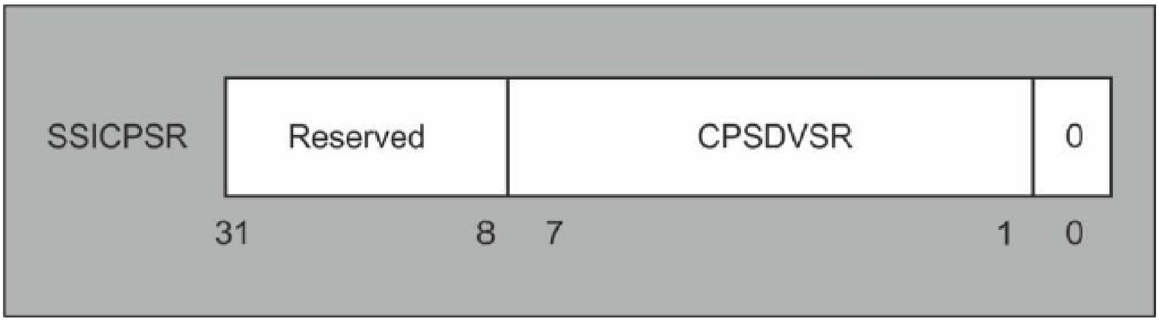


Figure: SSI Clock Prescaler Register

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.

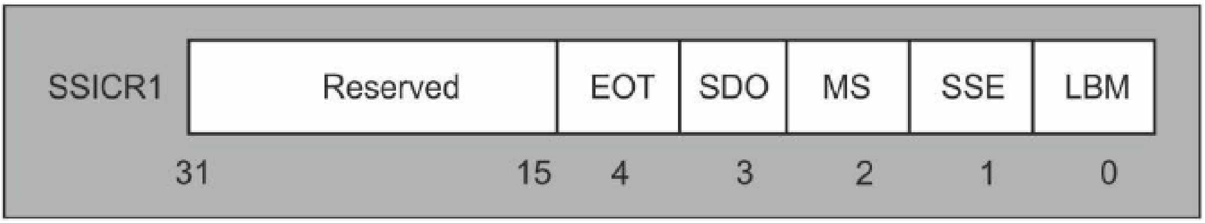


Figure: SSI Control 1 Register

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.

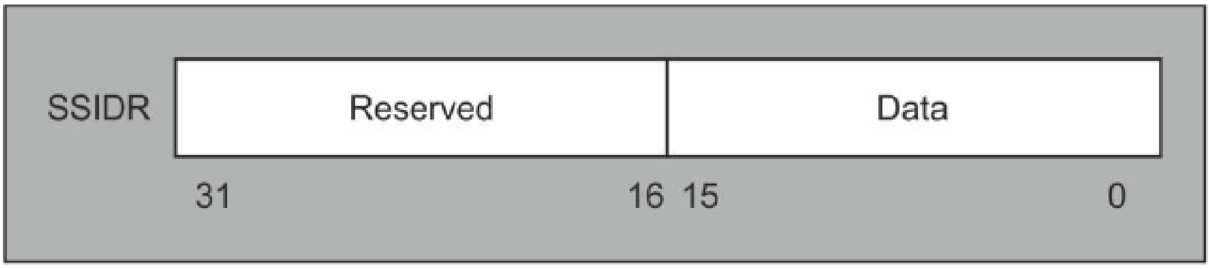


Figure: SSI Data Register

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

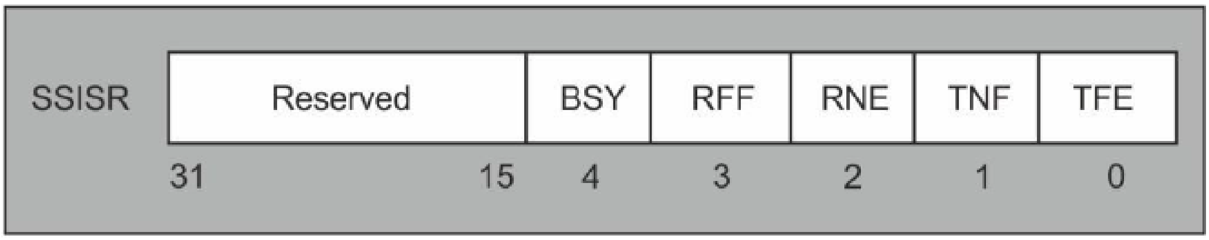
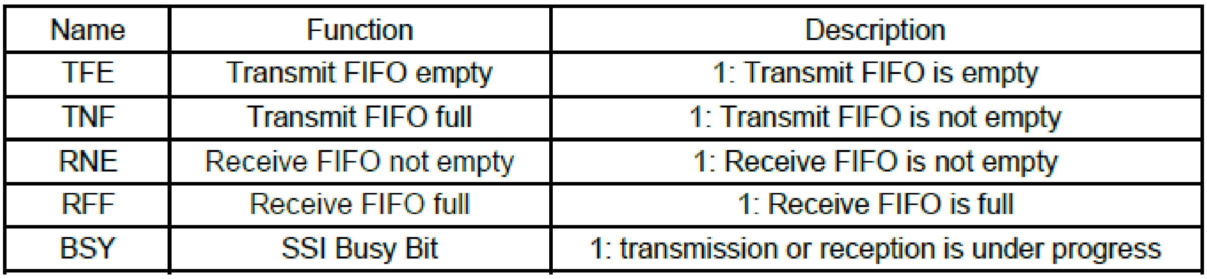


Figure: SSI Status Register

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.

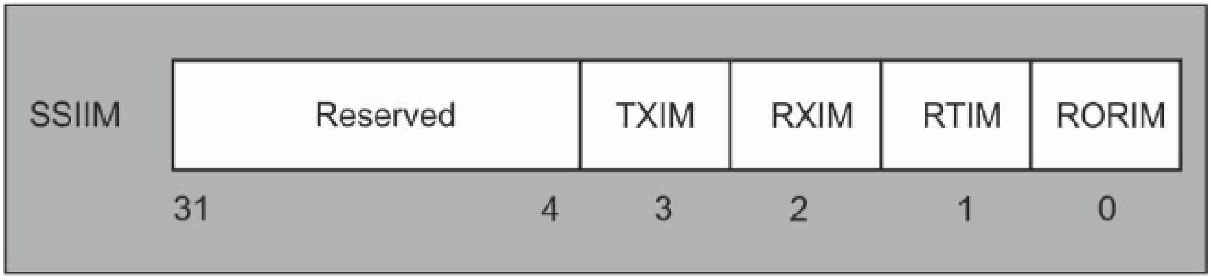


Figure: SSI Interrupt Mask Register

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 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.

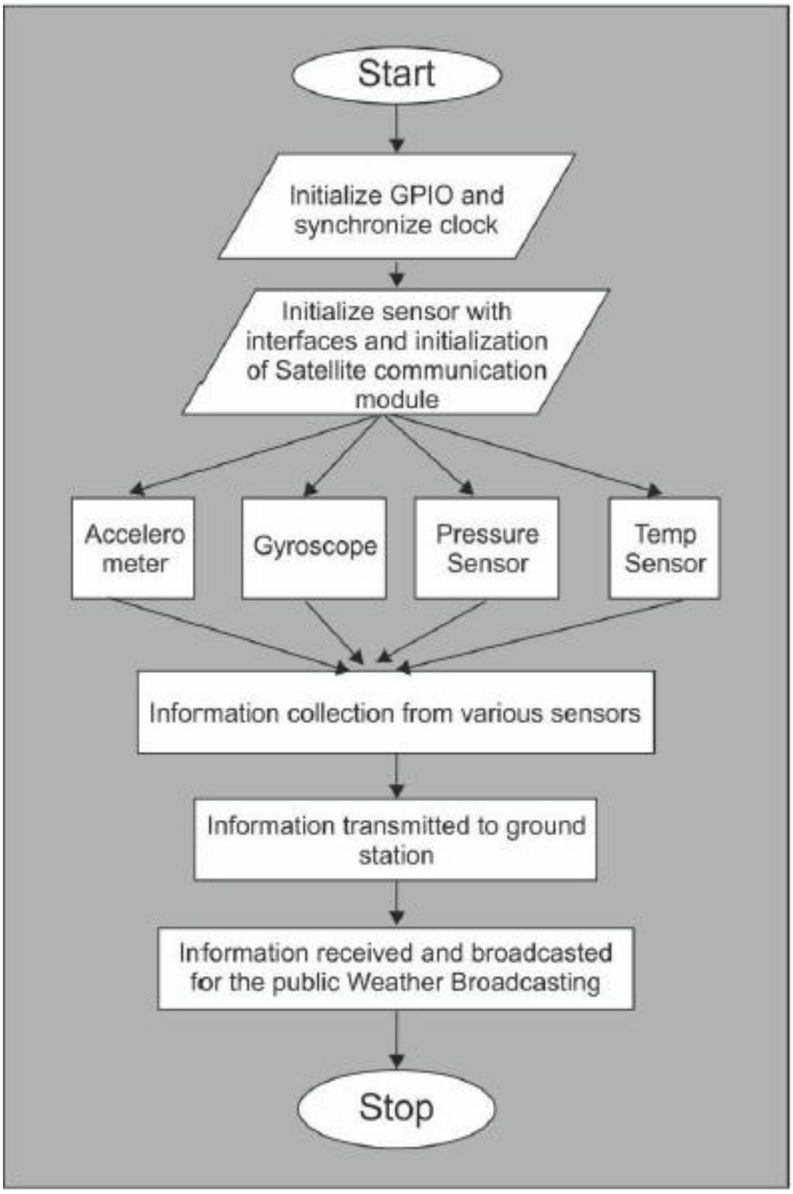

Start
Initialize GPIO and synchronize clock
Initialize sensor with interfaces and initialization of Satellite communication module
Accelero meter
Gyroscope
Pressure Sensor
Temp Sensor
Information collection from various sensors
Information transmitted to ground station
Information received and broadcasted for the public Weather Broadcasting
Stop

Figure: Flowchart for Interfacing TIVA with Sensor Hub Booster Pack

Embedded Networking Fundamentals:

Introduction:

Embedded networking technologies such as ZigBee, NFC, Bluetooth, Wi-Fi etc. are key elements in designing internet enabled applications. For example, in a residential set-up, these enable control of all devices remotely, even if there is no one physically present in the house. Such a „Smart home' allows the owner to monitor and control all smart equipment including power controls, security devices such as surveillance camera, etc. remotely. That is possible by using Wi-Fi technology, gateway solutions that provide connection to Cloud and of course the Internet to access the devices. Other typical application areas are monitoring, smart Grid, Smart Transport, smart plug, wearable devices, health monitoring etc.

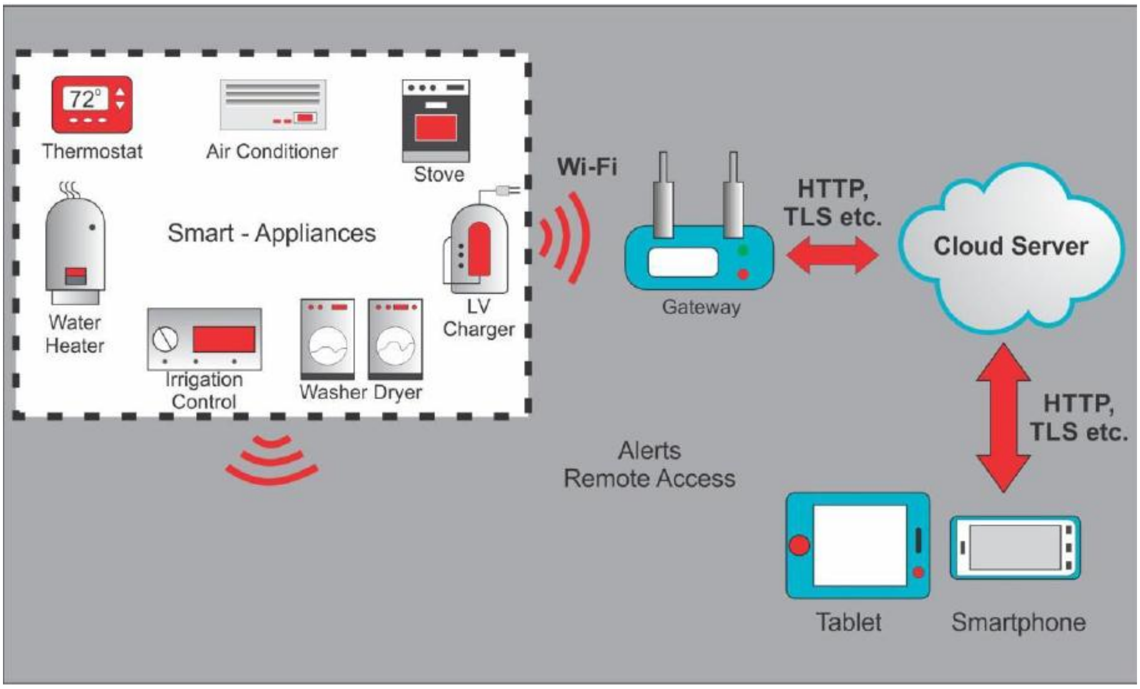


Figure: Embedded Network

This chapter covers wireless sensor networks as well as different wireless protocols, which provide connectivity between smart devices and gateway solutions. Readers will also learn about the CC3100 wireless module and how its architecture that provides wireless connectivity can be interfaced with TIVA C Series. The chapter also discusses the configuration of this module in access point mode with the TIVA Launchpad for use in typical IoT applications.

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. The above figure shows different sensors and systems involved in a typical smart-home application. 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.

Smart Grid
Automated Lighting
Pool Sensor
Garage Door Opener
Smart Appliances
Smart Thermostat
Security System
Gas/Water Meter
Smart e-meter
WLAN/HAN Gateway
Sprinkler System

Figure: Embedded Network for Smart Home Application

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 the below figure TCP/IP protocol can be used over Ethernet to provide Internet connectivity to the outside world. As shown in figure, 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.

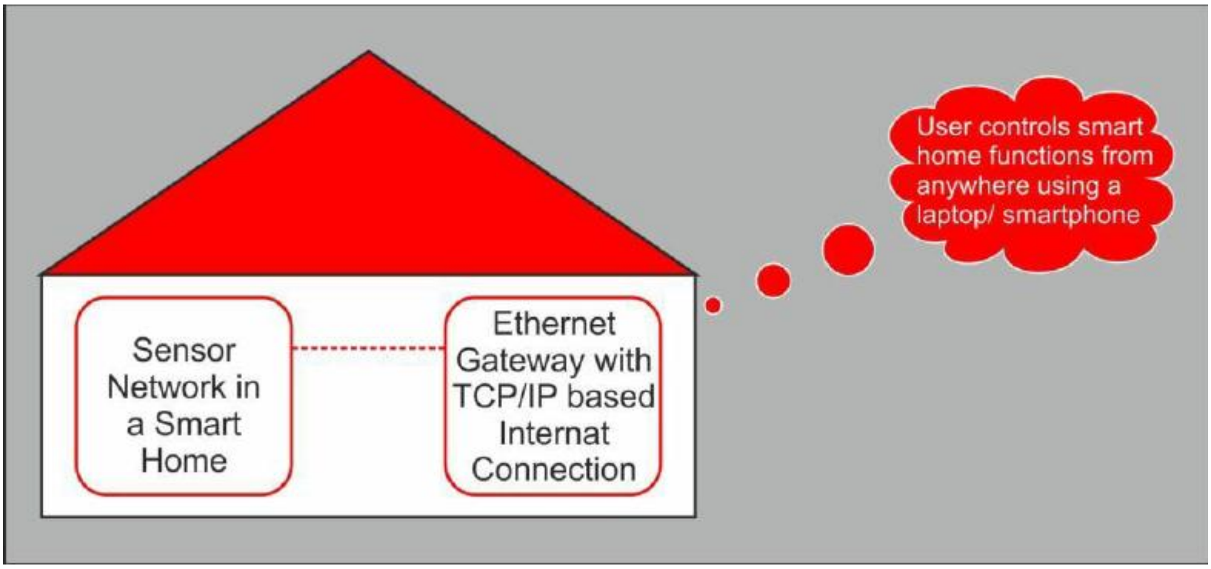

User controls smart home functions from anywhere using a laptop/ smartphone
Sensor Network in a Smart Home
Ethernet Gateway with TCP/IP based Internat Connection

Figure: Smart Home Architecture with TCP/IP connectivity to the Internet

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: Due to the limits of the IPv4 address space, the current Internet had to adopt a stopgap solution to face its unplanned expansion: 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. While this technique allows single stakeholders to mount large applications, it becomes completely unmanageable if the same end-points are to be used by many different stakeholders; this would occur in an IoT deployment where the same sensors are to be used by multiple, independent, stakeholders. Secondly the mechanism cannot be used to access specific end-points from the Internet.

- Multi-Stakeholder Support: IPv6 provides for end devices to have multiple addresses and an even more distributed routing mechanism than the IPv4 Internet. This allows different stakeholders to assign IoT end-device addresses that are consistent with their own application and network practices. Thus 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. Since then, IoT has generated a lot of interest, and it is expected that the number of „things‟ connected to IoT will grow from 20 billion things in 2015 to an estimated 200 billion by 2020. It refers to a scenario in which 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. 5.7 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.

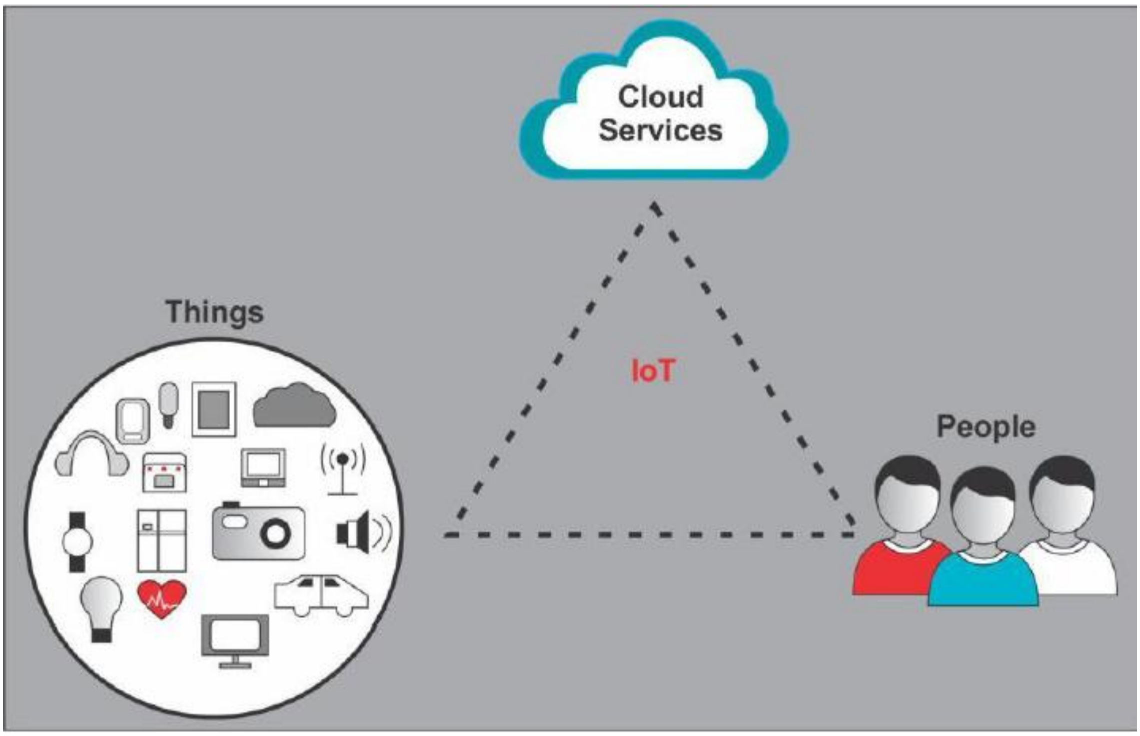


Figure: Main components of IoT

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's capabilities.

Anywhere: Any conceived wireless connectivity framework should be abstract enough to run from any location – both geographically and from a network topology perspective. The former refers to Internet-based ubiquity; the latter, refers to the ability to clone the framework into intranet environments where Internet access is limited or undesired. Acknowledging the structure of the Internet beyond the public domain is important to enable the expansion of the IoT paradigm.

Anyone: Currently, not all things are connected to the IoT. But an IoT ecosystem that is easy to use and secure is not that far away. This will make the IoT accessible to anyone. Anyone will be able to connect their product to the Internet, and also customize it to their personal preferences.

Applications of IOT:

With the industry's broadest IoT-ready portfolio of wired and wireless connectivity technologies, microcontrollers, processors, sensors and analog signal chain and power solutions, TI offers cloud ready system solutions. From high-performance home, industrial and automotive applications to battery-powered wearable and portable electronics or energy-harvested wireless sensor nodes, TI makes developing applications easier with hardware, software, tools and support to get anything connected as an IoT device.

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.

The IoT paradigm discussed may be encountered in a wide variety of venues that span across various activity circles throughout the day using different kinds of devices. 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 systems 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's. 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. In some cases can be used to carry out actions requested by clients, either users or things.

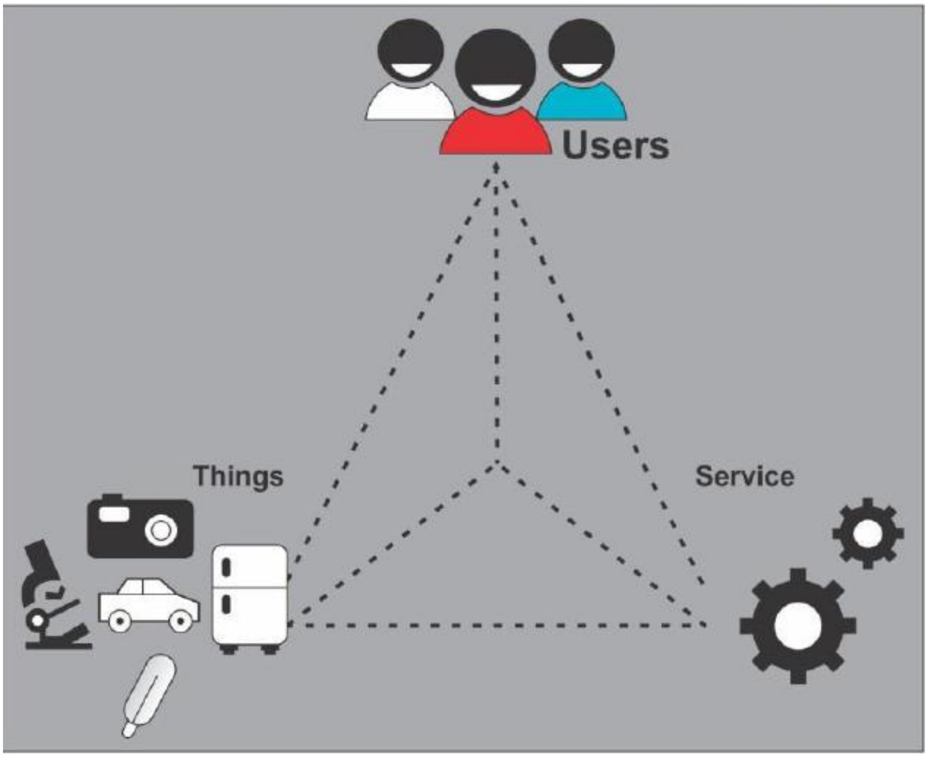


Figure: The IOT Players

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.

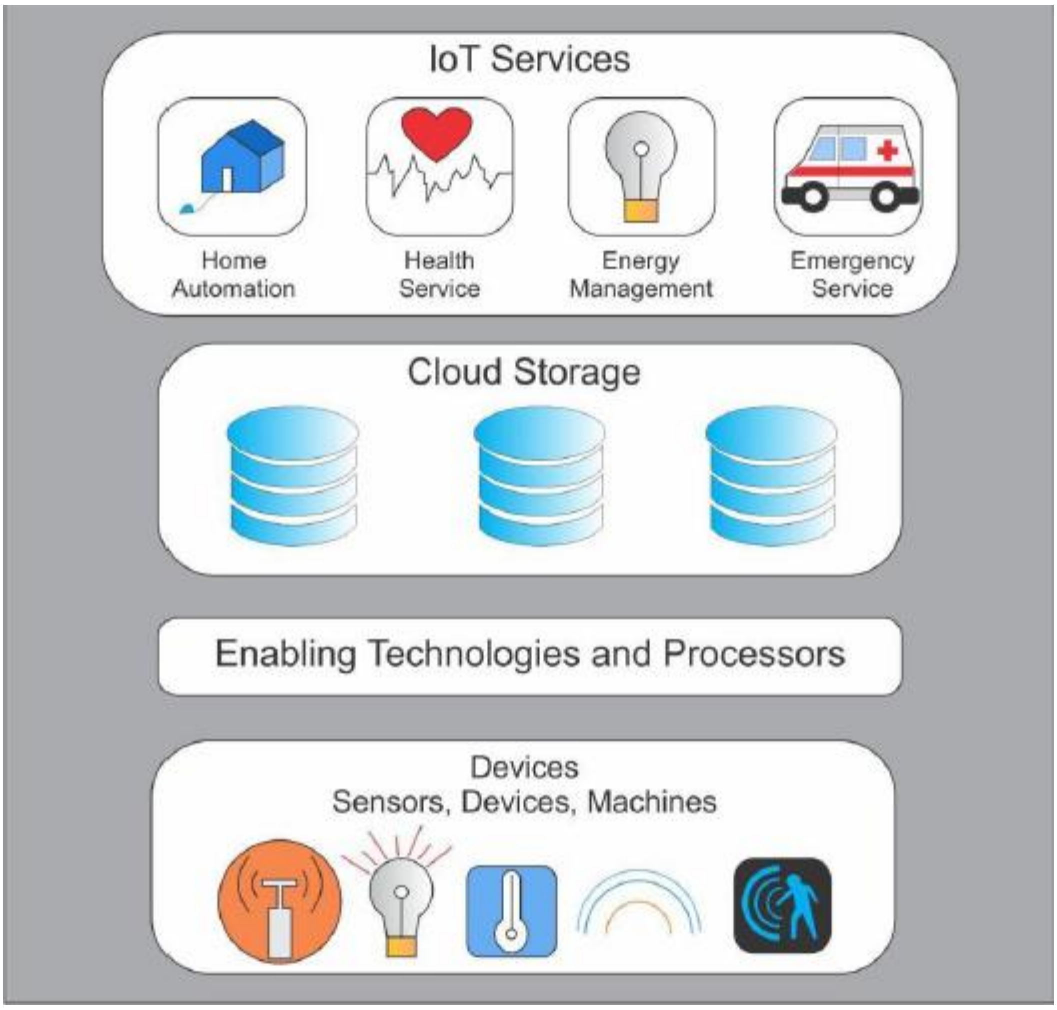


Figure: Architecture of IoT

Challenges of IOT:

Preparing the lowest layers of technology for the horizontal nature of the IoT requires manufacturers to deliver on the most fundamental challenges, including:

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 everyday and the industry is still in its naissance. The challenge facing the industry is the unknown; unknown devices, unknown applications, unknown use cases. Given this, there needs to be flexibility in all facets of development. Processors and microcontrollers that range from 16–1500 MHz to address the full spectrum of applications from a microcontroller (MCU) in a small, energy-harvested wireless sensor node to high-performance, multi-core processors for IoT infrastructure. A wide variety of wired and wireless connectivity technologies are needed to meet the various needs of the market. Last, a wide selection of sensors, mixed-signal and power-management technologies are required to provide the user interface to the IoT and energy-friendly designs.

- There are several fundamental features that a “thing” has to encompass to be a good IoT solution. Among these, the most important features are energy efficiency, security, data handling and simplicity.

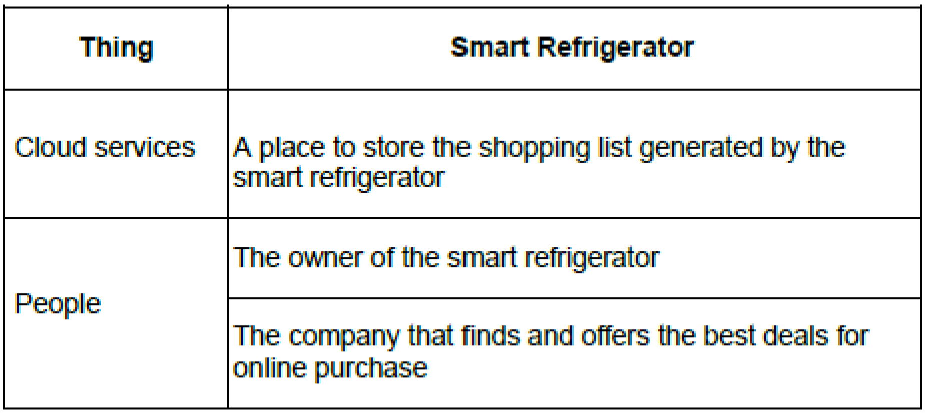
Energy Efficiency: As the number of devices grows, even small amounts of excessive power are a noticeable waste. When it comes to power, the challenge is to ensure that adding Internet connectivity does not impose a change to the power supply. In other words, ideally it should fit within the existing power budget headroom. The TIVA Launchpad, being an ultra-low power MCU ensures that the IoT application takes minimal power.

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. The TI wireless modules provide easy interfacing with the TIVA Launchpad to provide connectivity that suits the need of the IoT application.

Overview of Wireless Sensor Networks and Design Examples:

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.



Thing	Smart Refrigerator
Cloud services	A place to store the shopping list generated by the smart refrigerator
People	The owner of the smart refrigerator
	The company that finds and offers the best deals for online purchase

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 Figure 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.

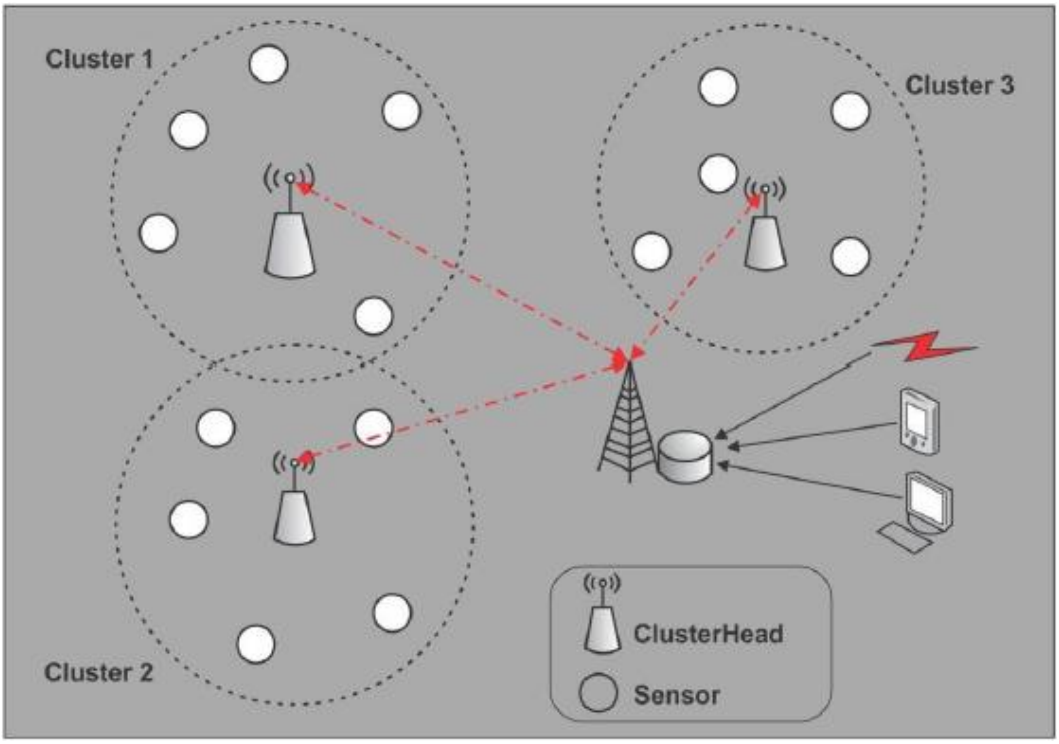


Figure: A Typical Sensor Network Architecture

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. Before learning more detail about these wireless communication technologies, a brief overview of the Open System Interconnection (OSI) model used for communication between two entities is given below:

OSI Model for Communication:

OSI model is a conceptual model that is used to organize the various functions of a communication system by arranging them into seven different layers as shown in below figure. The function performed by each layer in implementing an end-to-end communication system is described below:

Physical Layer:

This layer specifies the physical medium used to transmit bits between communicating systems. In wired systems, the physical layer may specify the use of copper wires or fiber optic cable for wired systems. Similarly for wireless technology like ZigBee, the physical layer specifications mention the use of 2.4 GHz ISM frequency band as one of the options for communication.

Data Link Layer:

When two or more nodes try to use the physical media simultaneously for data transfer, the data packets may collide and, the nodes need to try again for access to the media. In this case, data link layer acts as a local traffic cop to regulate the medium access by the nodes of the network. Another important role of the data link layer can be to detect and correct the errors that may occur when data is transferred on the medium.

Network Layer:

The primary function of network layer is to forward data packets (received from higher layers) from one point to another over the network. The data packets may travel across many different networks, guided on the way by gateway and router devices, to their final destination.

Transport Layer:

This layer provides a reliable end-to-end connection oriented data transfer along with error and flow control services. Transmission Control Protocol (TCP) is the most common transport layer protocol used on Internet.

Session Layer:

The reliable end-to-end connection provided by transport layer is used to set-up an interactive session between the two communicating computers or end-user applications. The session layer protocols are responsible to open, manage and close these sessions to support effective data communication.

Presentation Layer:

This layer ensures that the data presented to the application layer is in proper format and ready to be used. For example, data transmitted in EBCDIC-code by the sender may be converted at the receiver end by presentation layer to ASCII code format used by the application layer.

Application layer:

The protocols used in this layer define the user interface that finally displays the information to the user.

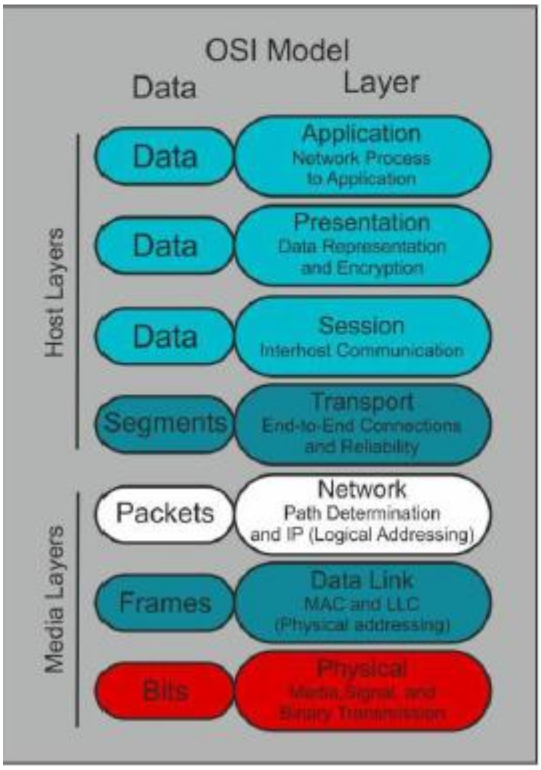

OSI Model
Data
Layer
Host Layers
Data
Application
Network Process to Application
Data
Presentation
Data Representation and Encryption
Data
Session
Interhost Communication
Segments
Transport
End-to-End Connections and Reliability
Media Layers
Packets
Network
Path Determination and IP (Logical Addressing)
Frames
Data Link
MAC and LLC (Physical addressing)
Bits
Physical
Media, Signal, and Binary Transmission

Figure: Protocol stack of OSI

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 capability to the Microcontroller:

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). A functional block diagram of CC3100 module is shown in the below figure.

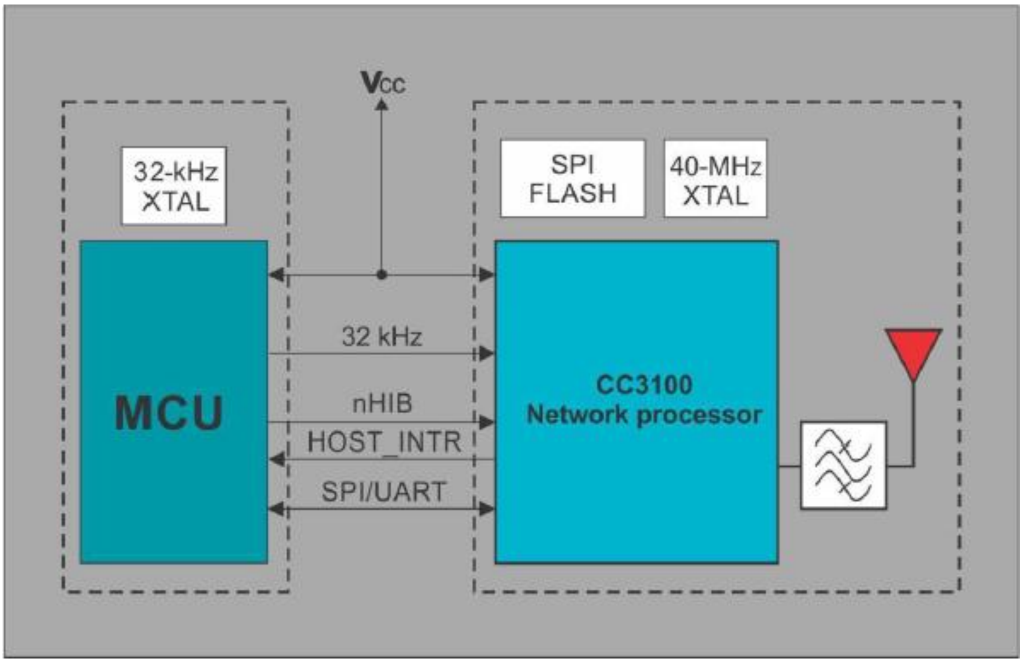


Figure: Functional diagram of SimpleLink Wi-Fi CC3100 Module

Figure: CC3100 Booster Pack (SimpleLink Wi-Fi) mounted on TIVA Launchpad

Embedded Wi-Fi:

It is important to understand the hardware and software architecture of any device before using it in a design. Figure 5.17 shows the hardware architecture for SimpleLink 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

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 pre-programmed Wi-Fi driver and multiple Internet protocols
3) TCP/IP Stack – supports communication with Figure Hardware Architecture for CC3100 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.

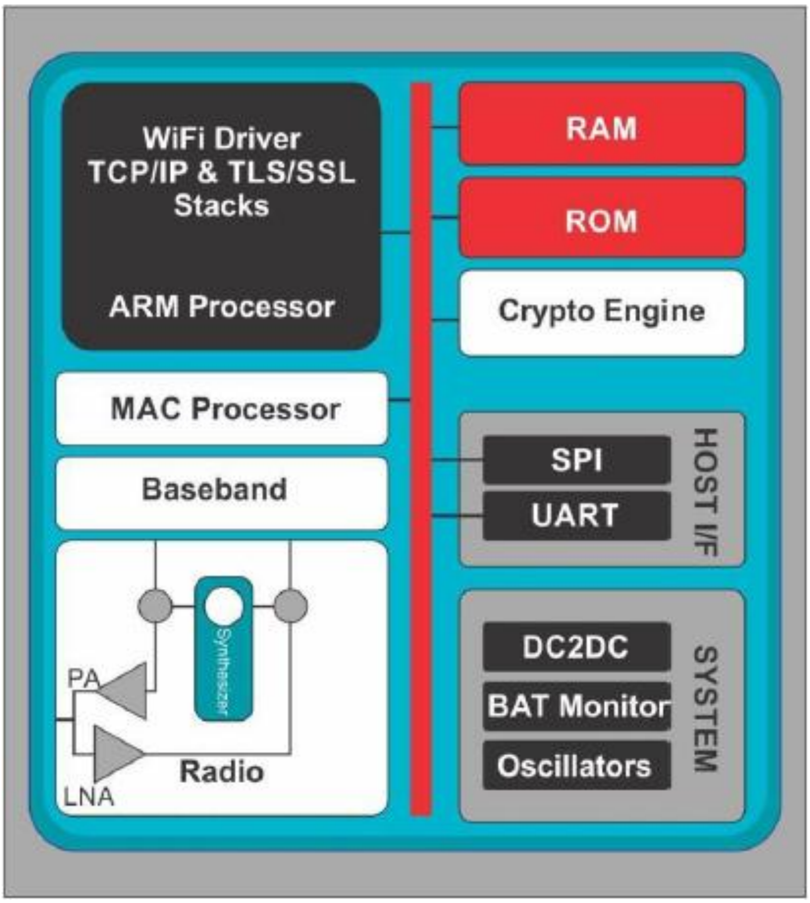


Figure: Hardware Architecture for CC3100

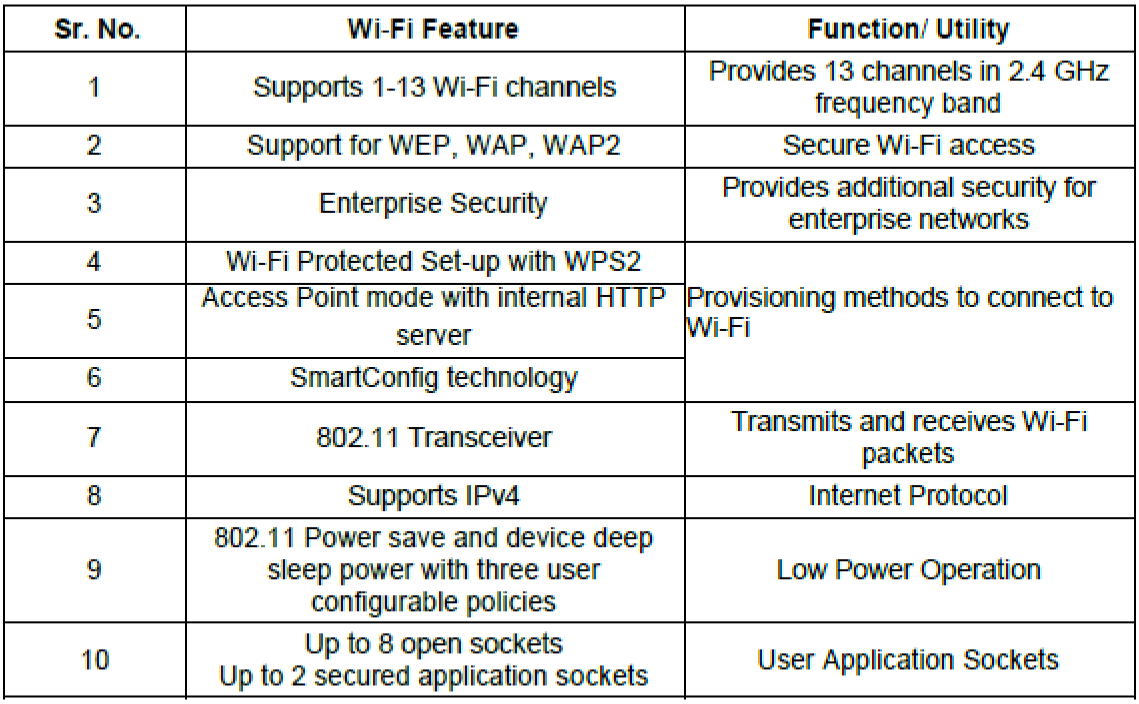
Power Management Subsystem:

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 below Table.

Table: 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 for Wireless and Networking Applications:

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:User APIs for Wireless and Networking applications.

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.

Building IoT applications using CC3100 user API:

Get whether application using CC3100:

This application demonstrates how to connect to openweathermap.org server and request for weather details of a city. The application opens a TCP socket w/ the server and sends a HTTP Get request to get the weather details. The received data is processed and displayed on the console window as shown below.

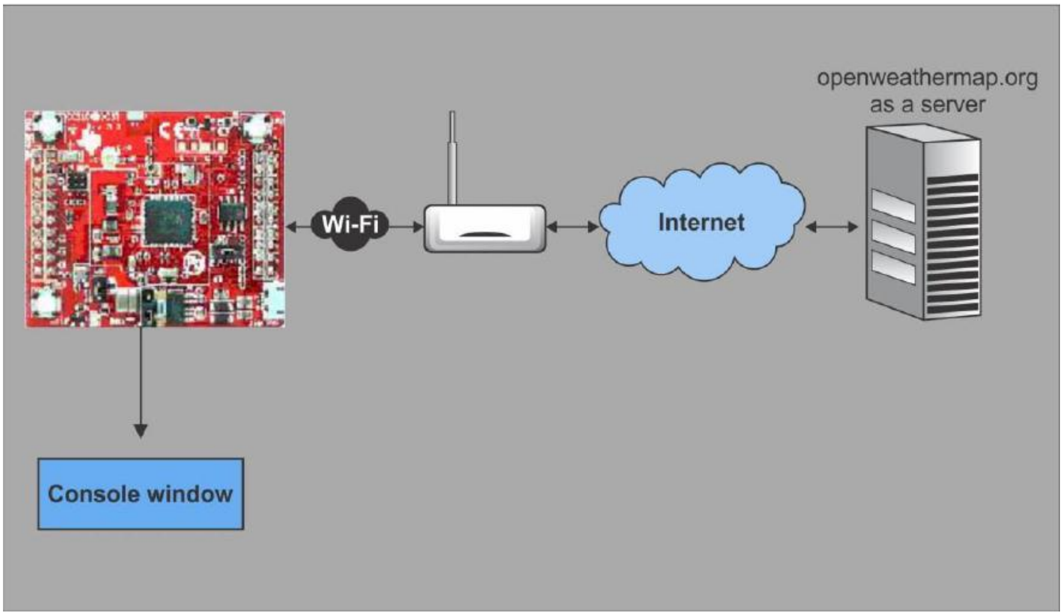
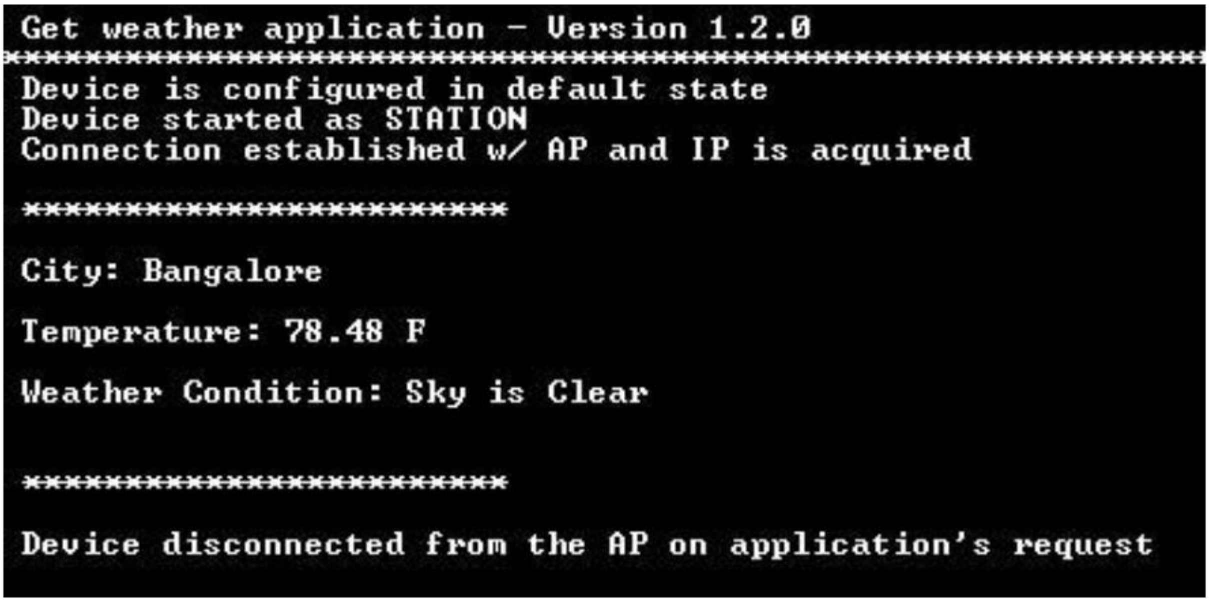
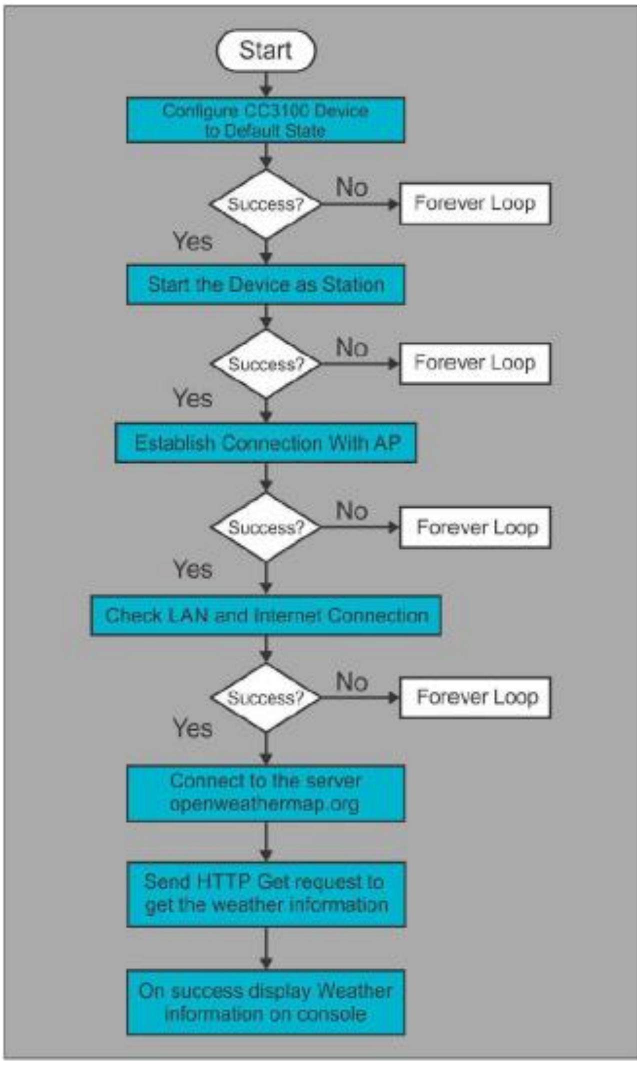


Figure: Block diagram of Get Weather application



```
Get weather application - Version 1.2.0
**********************************************************************
Device is configured in default state
Device started as STATION
Connection established w/ AP and IP is acquired

************************

City: Bangalore

Temperature: 78.48 F

Weather Condition: Sky is Clear


************************

Device disconnected from the AP on application's request
```

Figure: Get Weather Application Console Window



Start
Configure CC3100 Device to Default State
Success?
No
Forever Loop
Yes
Start the Device as Station
Success?
No
Forever Loop
Yes
Establish Connection With AP
Success?
No
Forever Loop
Yes
Check LAN and Internet Connection
Success?
No
Forever Loop
Yes
Connect to the server openweathermap.org
Send HTTP Get request to get the weather information
On success display Weather information on console

Figure: Flow Chart of getting weather application

To perform this application, we need to set an IP address for the device CC3100 with TIVA Launchpad. We can set IP address for the device CC3100 statically or dynamically as we discussed in the session. The below steps demonstrates the configuration of a static IP address for CC3100 TIVA Launchpad. Here the device connects to the Access Point (APwith the configured static IP. The static IP address is stored inside the non-volatile memory of CC3100.The basic steps for assigning IP address to a CC3100 device are given in the flowchart shown in figure.

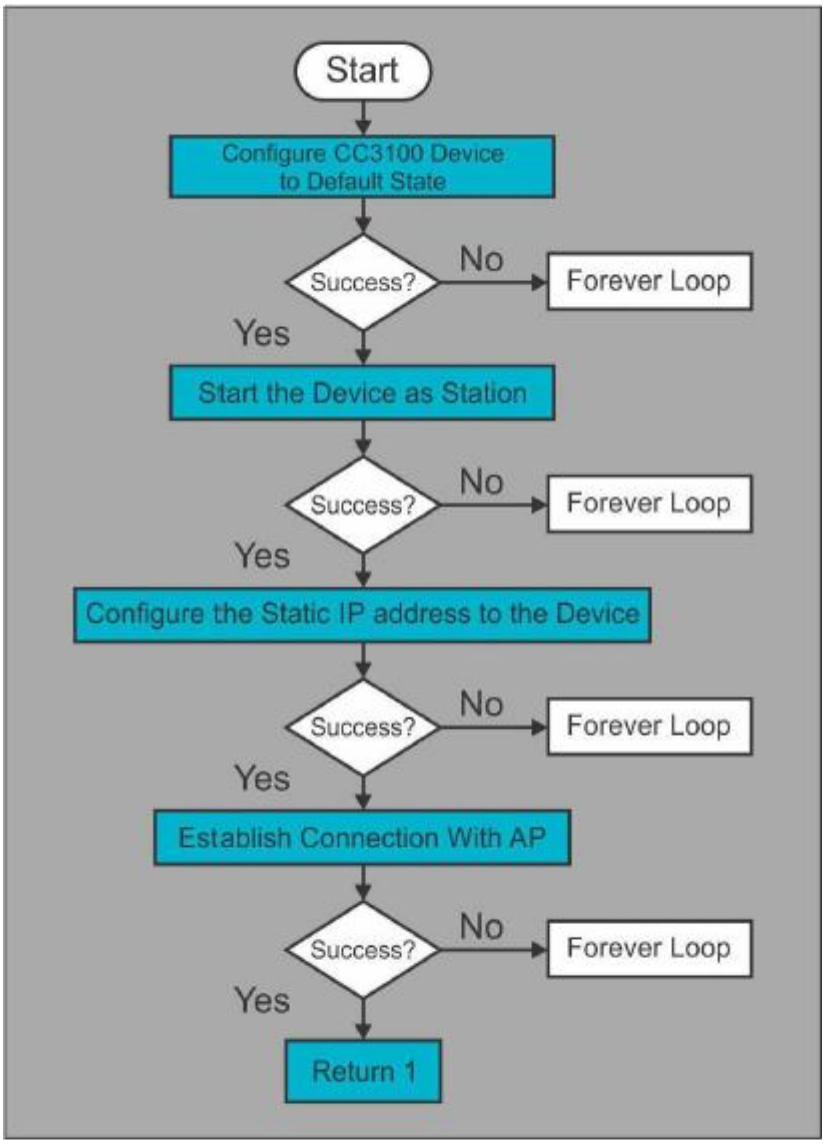

Start
Configure CC3100 Device to Default State
Success?
No
Forever Loop
Yes
Start the Device as Station
Success?
No
Forever Loop
Yes
Configure the Static IP address to the Device
Success?
No
Forever Loop
Yes
Establish Connection With AP
Success?
No
Forever Loop
Yes
Return 1

Figure: Flowchart for configuring a static IP address for CC3100 module

In this case study the module CC3100 is configured as a Wireless Local Area Network (WLAN) Station to connect to the internet and open weather.org as a server. A wireless local area network (WLAN) is a wireless computer network that connects two or more devices without wires within a confined area for example within a building. This facilitates the users to stay connected without physical wiring constraints and also access Internet. Wi-Fi is based on IEEE 802.11 standards including IEEE 802.11a and IEEE802.11b.

All nodes that connect over a wireless network are referred to as stations (STA). Wireless stations can be categorized into Wireless Access Points (AP) or clients. Access Points (AP) work as the base station for a wireless network. The Wireless clients could be any device such as computers, laptops, mobile devices, smartphones etc. The flowchart for this case study is shown in figure.

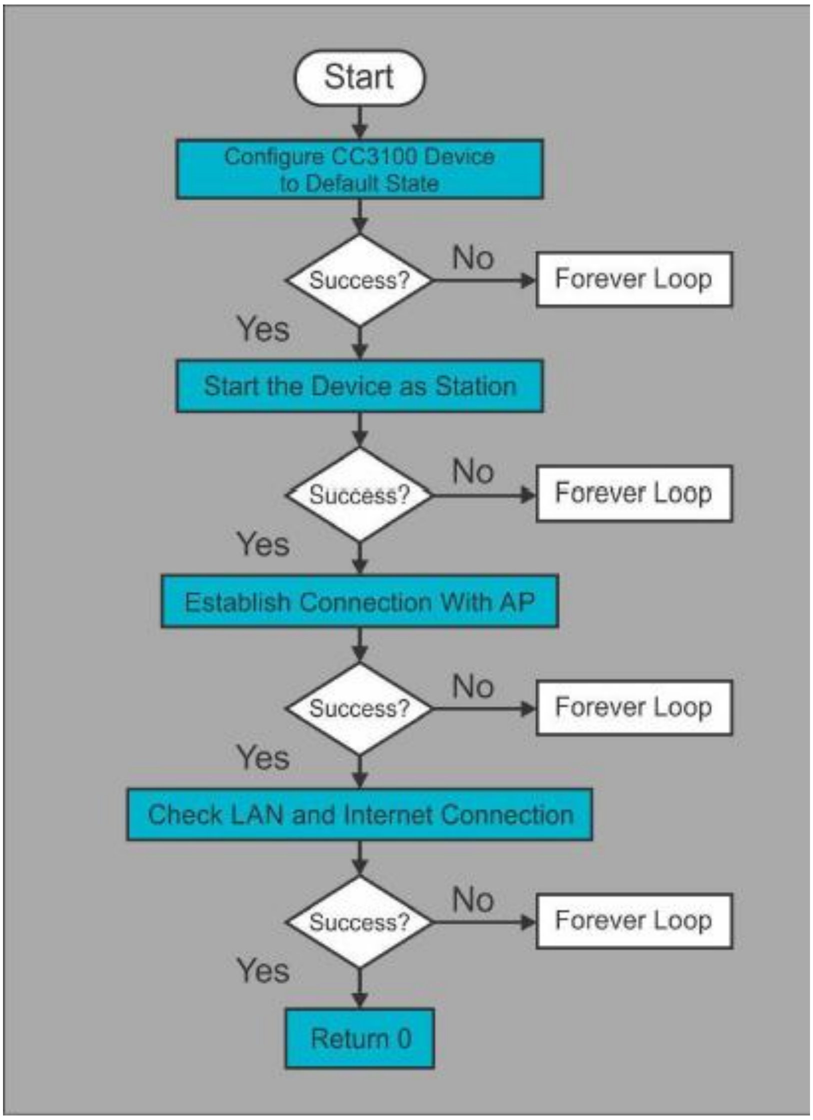

Start
Configure CC3100 Device to Default State
Success?
No
Forever Loop
Yes
Start the Device as Station
Success?
No
Forever Loop
Yes
Establish Connection With AP
Success?
No
Forever Loop
Yes
Check LAN and Internet Connection
Success?
No
Forever Loop
Yes
Return 0

Figure: Flowchart for using CC3100 as a WLAN Station

We can also make CC3100 module as a HTTP server with TIVA Launchpad. HTTP is an acronym for Hyper Text Transfer Protocol. HTTP is a client/server protocol used to deliver hypertext resources (HTML web pages, images, query results, and so forth) to the client side. HTTP works on top of a predefined TCP/IP. (Transmission Control Protocol / Internet Protocol). HTTP web server allows endusers to remotely communicate with the CC3100 by using a standard web browser. The HTTP web server enables the following functions:

- Device configuration
- Device status and diagnostic
- Application-specific functionality

The HTTP server handles the HTTP request by listening on the HTTP socket id which is by default 80. Based on the request type, such as HTTP GET or HTTP POST, the server handles the request URI resource and content. The server then composes the appropriate HTTP response and returns it to the client. The server communicates with the serial flash file system, which hosts the web page files. The files are saved in the serial flash with their individual filenames.

If we configure CC3100 as a server then it will be in Access Point (AP) mode with a pre-defined SSIDNAME and uses the sample HTML pages stored in Flash which can be accessed by the clients. Clients can connect to CC3100 and request for web-pages using the IP of device from any standard web browser. There are pre-programmed html pages already residing on the flash and new HTML pages can be downloaded on serial-flash of CC3100 using CCS_UniFlash utility using a separate tool EMU-BOOST. The scope of this study will be to use the existing html pages already pre-programmed in the flash by default. The flowchart for using CC3100 device as a HTTP server is given in below figure.

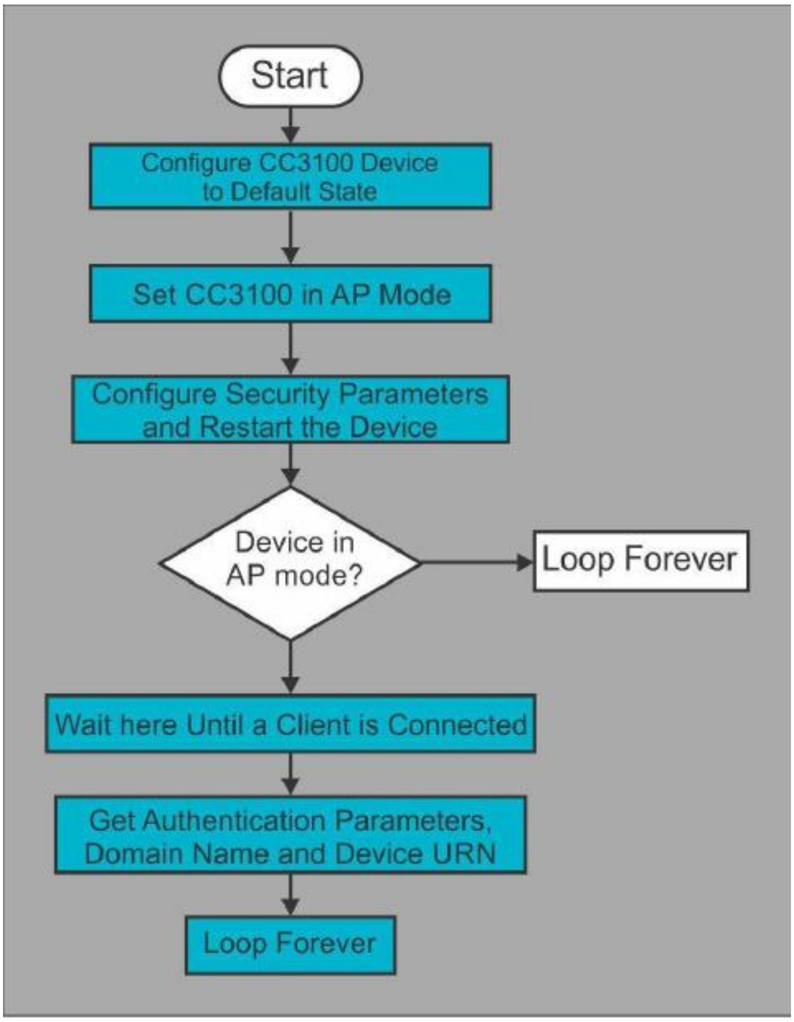

Start
Configure CC3100 Device to Default State
Set CC3100 in AP Mode
Configure Security Parameters and Restart the Device
Device in AP mode?
Loop Forever
Wait here Until a Client is Connected
Get Authentication Parameters, Domain Name and Device URN
Loop Forever

Figure: Flowchart for configuring CC3100 as a HTTP Server

Case Study: Tiva based Embedded Networking Application: *"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.

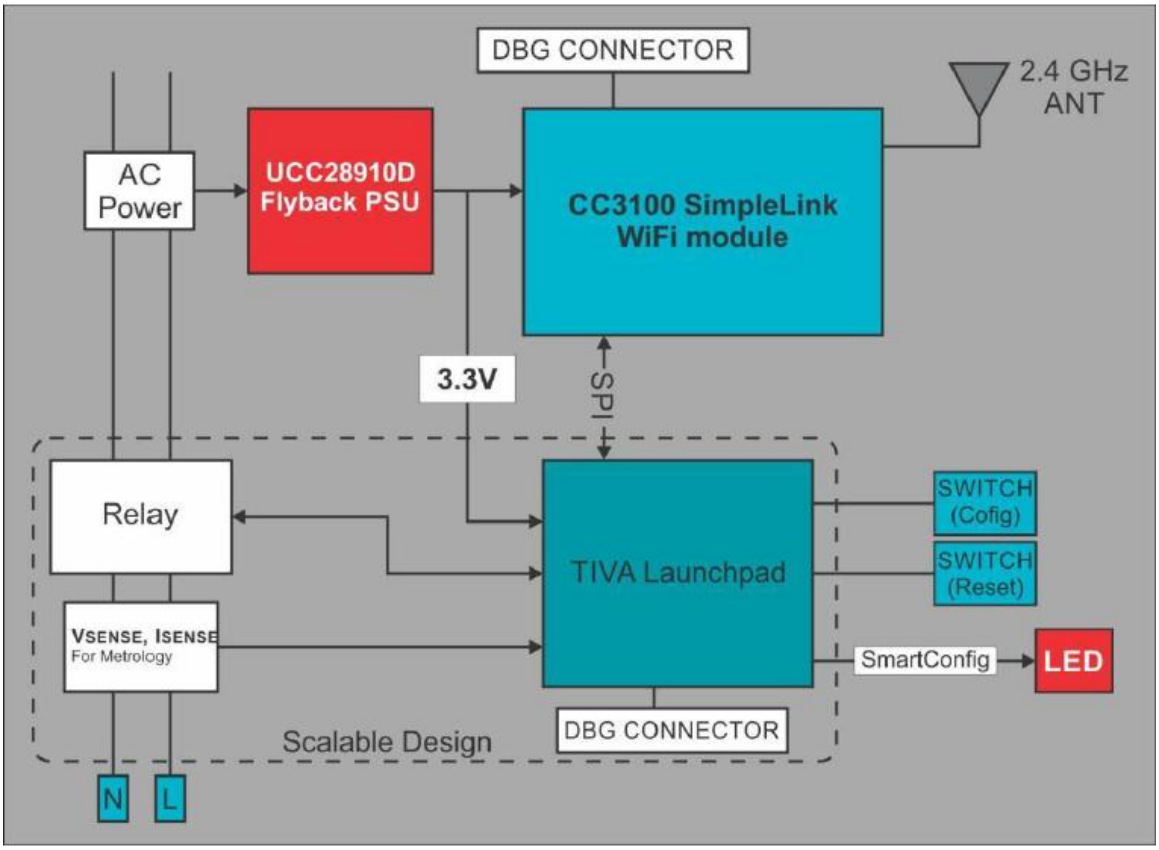


Figure: Block diagram of Smart Plug with WiFi connectivity

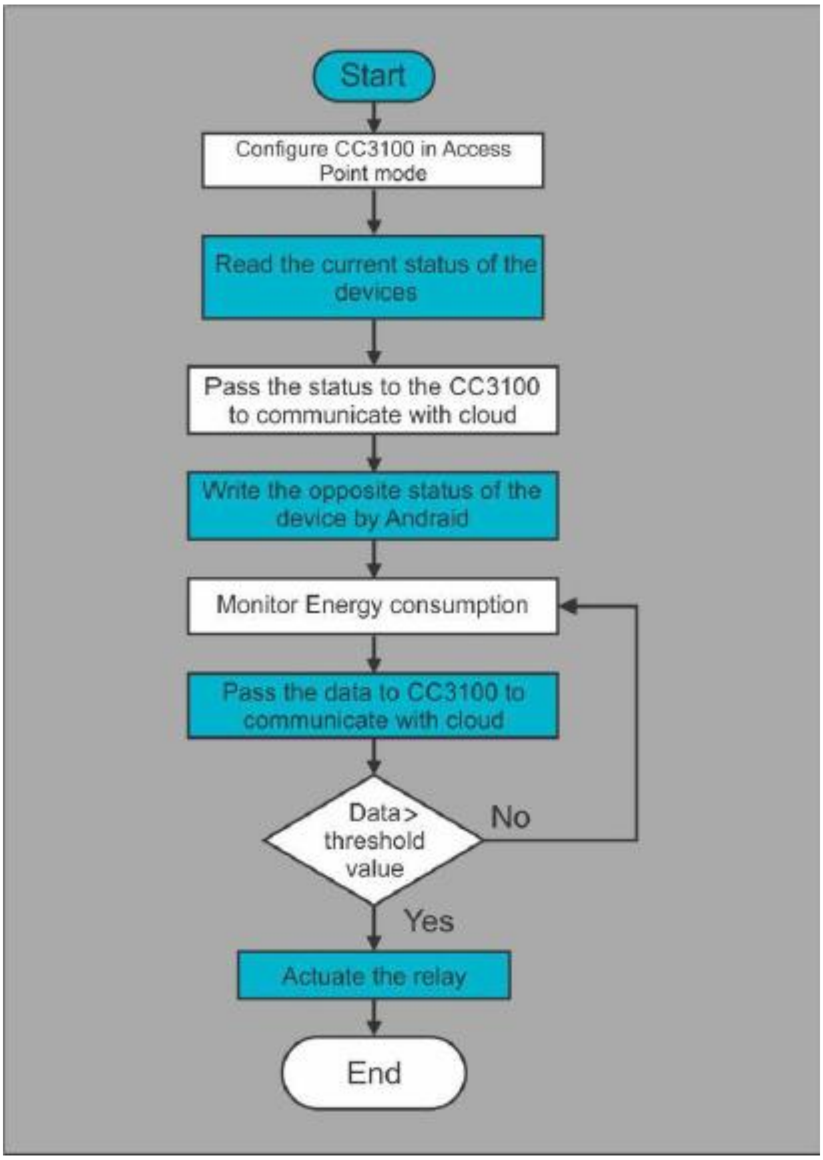

Start
Configure CC3100 in Access Point mode
Read the current status of the devices
Pass the status to the CC3100 to communicate with cloud
Write the opposite status of the device by Andraid
Monitor Energy consumption
Pass the data to CC3100 to communicate with cloud
Data> threshold value
No
Yes
Actuate the relay
End

Figure: Flow chart of Smart Plug with Wi-Fi connectivity