SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SECA3019 - EMBEDDED PROCESSORS

Sathyabama Institute of Science and Technology

Department of Electronics and Communication Engineering

SECA3019	EMBEDDED PROCESSORS	L	T	P	Credits	Total marks
		3	0	0	3	100

Pre requisite: Nil	Co Requisite: Nil

Course Objectives

- To analyze the features of various embedded processors
- To analyze the On-chip peripherals
- To develop ARM processor-based applications
- To design innovative applications by interfacing the processors with real world
- To analyze various ARM cortex processors
- To develop firmware for ARM cortex processors

UNIT	CONTENTS	HOURS
I	**INTRODUCTION TO EMBEDDED PROCESSORS** Introduction to embedded processors– Compare Von Neumann architecture and Harvard architecture, RISC Vs CISC – System on Chip (SoC)-Introduction to SoC Architecture, An approach for SOC Design, System Architecture and Complexity. Processor Selection for SOC, Basic concepts in Processor Architecture, Overview of SOC external memory, Internal Memory, Scratchpads and Cache memory, SOC Memory System, Models of Simple Processor – memory interaction, SOC Standard Buses	9
II	**EMBEDDED PROCESSORS ON CHIP PERIPHERALS** Memory - Interrupts - I/O Ports-Timers & Real Time Clock (RTC), Watch dog timer - CCP modules - Capture Mode - Compare Mode-PWM Mode - Serial communication module - USART - SPI interface - I2C interface, Analog Comparator, Analog interfacing and data acquisition.	9
III	**ARM PROCESSOR** Architecture of ARM Controller – Registers, Pipeline organization 3 stage & 5 stage, Thumb mode of operation - D/A and A/D converter, sensors, actuators and their interfacing – Case study- Digital clock, Temperature sensing, Light sensing, Introduction to Internet of Things, smart home concepts	9
IV	**REAL WORLD INTERFACING USING ARM PROCESSOR** Interfacing the peripherals to LPC2148: GSM and GPS using UART, on-chip ADC using interrupt (VIC), EEPROM using I2C, SD card interface using SPI, on-chip DAC for waveform generation.	9
V	**ARM CORTEX PROCESSORS** Introduction to ARM CORTEX series, improvement over classical series and advantages for embedded system design. CORTEX A, CORTEX M, CORTEX R processors series, versions, features and applications, need of operating system in	9

	developing complex applications in embedded system, Firmware development for ARM Cortex, Survey of CORTEX M3 based controllers, its features and comparison	

Maximum Hours: 45

Course Outcomes

On completion of the course, the student will be able to

CO1-Analyze the architectures of different Embedded Processors

CO2-Identify an appropriate on chip peripherals for serial and parallel communication

CO3-Examine the functions of ARM processors

CO4-Develop real time applications using ARM processors

CO5-Develop a firmware for embedded applications

CO6-Develop innovative products using Embedded processors

TEXT / REFERENCE BOOKS

1. F. Vahid and T. Givargis, “Embedded System Design: A Unified Hardware/Software Introduction”, Wiley India Pvt. Ltd., 2002.
2. Michael J. Flynn and Wayne Luk, “Computer System Design System-on-Chip”, Wiley India Pvt. Ltd.
3. Steve Furber, “ARM System on Chip Architecture “, 2nd Edition, 2000, Addison Wesley Professional.
4. S. Pascricha and N. Dutt, Morgan Kaufmann, On-Chip Communication Architectures, System on Chip Interconnect, -Elsevier Publishers 2008
5. Mark Fisher, “ARM Cortex M4 Cookbook”, Packt Publishing, 2016.
6. Lyla B. Das, “Architecture, Programming and Interfacing of Low-power Processors ARM 7, Cortex-M”, Cengage, 1st Edition, 2017.
7. Joseph Yiu, “The Definitive Guide to the ARM Cortex-M3”, Newness, 2nd Edition, 2009

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – I INTRODUCTION TO EMBEDDED PROCESSOR - SECA3019

UNIT I: INTRODUCTION TO EMBEDDED PROCESSORS

[Introduction to embedded processors– Compare Von Neumann architecture and Harvard architecture, RISC Vs CISC – System on Chip (SoC)-Introduction to SoC Architecture, An approach for SOC Design, System Architecture and Complexity. Processor Selection for SOC, Basic concepts in Processor Architecture, Overview of SOC external memory, Internal Memory, Scratchpads and Cache memory, SOC Memory System, Models of Simple Processor – memory interaction, SOC Standard Buses]

1.1 Overview of Embedded Application Architecture

Embedded systems, an emerging area of computer technology, combine multiple technologies, such as computers, semiconductors, microelectronics, and the Internet, and as a result, are finding ever-increasing application in our modern world. With the rapid development of computer and communications technologies and the growing use of the Internet, embedded systems have brought immediate success and widespread application in the post-PC era, especially as the core components of the Internet of Things. They penetrate into every corner of modern life from the mundane, such as an automated home thermostat, to industrial production, such as in robotic automation in manufacturing. Embedded systems can be found in military and national defense, healthcare, science, education, and commercial services, and from mobile phones, MP3 players, and PDAs to cars, planes, and missiles.

This chapter provides the concepts, structure, and other basic information about embedded systems and lays a theoretical foundation for embedded application development, of which application development for Android OS is becoming the top interest of developers.

1.2 Introduction to Embedded Systems

Since the advent of the first computer, the ENIAC, in 1946, the computer manufacturing process has gone from vacuum tubes, transistors, integrated circuits, and large-scale integration (LSI), to very-large-scale integration (VLSI), resulting in computers that are more compact, powerful, and energy efficient but less expensive (per unit of computing power).

After the advent of microprocessors in the 1970s, the computer-using world witnessed revolutionary change. Microprocessors are the basis of microcomputers, and personal computers (PCs) made them more affordable and practical, allowing many private users to own them. At this stage, computers met a variety of needs: they were sufficiently versatile to satisfy various demands such as computing, entertainment, information sharing, and office automation. As the adoption of microcomputers was occurring, more people wanted to embed them into specific systems to intelligently control the environment. For example, microcomputers were used in machine tools in factories. They were used to control signals and monitor the operating state through the configuration of peripheral sensors. When microcomputers were embedded into such environments, they were prototypes of embedded systems.

As the technology advanced, more industries demanded special computer systems. As a result, the development direction and goals of specialized computer systems for specific environments and general-purpose computer systems grew apart. The technical requirement of general-purpose computer systems is fast, massive, and diversified computing, whereas the goal

of technical development is faster computing speed and larger storage capacity. However, the technical requirement of embedded computer systems is targeted more toward the intelligent control of targets, whereas the goal of technical development is embedded performance, control, and reliability closely related to the target system.

Embedded computing systems evolved in a completely different way. By emphasizing the characteristics of a particular processor, they turned traditional electronic systems into modern intelligent electronic systems. Figure 1-1 shows an embedded computer processor, the Intel Atom N2600 processor, which is 2.2 × 2.2 cm, alongside a penny.

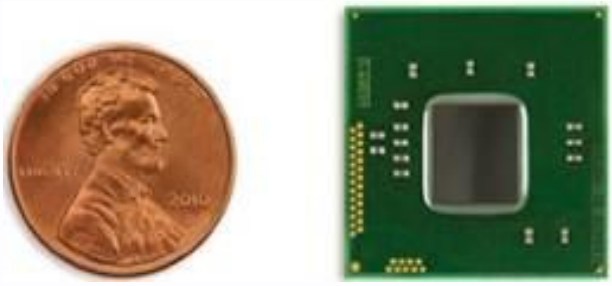


Figure 1.1: Comparison of an embedded computer chip to a US penny.

The emergence of embedded computer systems alongside general-purpose computer systems is a milestone of modern computer technologies. The comparison of general-purpose computers and embedded systems is shown in Table 1-1.

Today, embedded systems are an integral part of people's lives due to their mobility. As mentioned earlier, they are used everywhere in modern life. Smartphones are a great example of embedded systems.

Table 1-1. Comparison of General-Purpose Computers and Embedded Systems

Item	General-purpose computer systems	Embedded systems
Hardware	High-performance hardware, large storage media	Diversified hardware, single-processor solution
Software	Large and sophisticated OS	Streamlined, reliable, real-time systems
Development	High-speed, specialized development team	Broad development sectors

1.2.1 Mobile Phones

Mobile equipment, especially smartphones, is the fastest growing embedded sector in recent years. Many new terms such as *extensive embedded development* and *mobile development* have been derived from mobile software development. Mobile phones not only are pervasive but also have powerful functions, affordable prices, and diversified applications. In addition to basic telephone functions, they include, but are not limited to, integrated PDAs, digital cameras, game consoles, music players, and wearables.

1.2.2 Consumer Electronics and Information Appliances

Consumer electronics and information appliances are additional big application sectors for embedded systems. Devices that fall into this category include personal mobile devices and home/entertainment/audiovisual devices. Personal mobile devices usually include smart handsets such as PDAs, as well as wireless Internet access equipment like mobile Internet devices (MIDs). In theory, smartphones are also in this class; but due to their large number, they are listed as a single sector.

Home/entertainment/audiovisual devices mainly include network television like interactive television; digital imaging equipment such as digital cameras, digital photo frames, and video players; digital audio and video devices such as MP3 players and other portable audio players; and electronic entertainment devices such as handheld game consoles, PS2 consoles, and so on. Tablet PCs (tablets), one of the newer types of embedded devices, have become favorites of consumers since Apple released the iPad in 2010.

1.3 General Architecture of an Embedded System

Figure 1-2 shows a configuration diagram of a typical embedded system consisting of two main parts: embedded hardware and embedded software. The embedded hardware primarily includes the processor, memory, bus, peripheral devices, I/O ports, and various controllers. The embedded software usually contains the embedded operating system and various applications. Input and output are characteristics of any open system, and the embedded system is no exception. In the embedded system, the hardware and software often collaborate to deal with various input signals from the outside and output the processing results through some form.

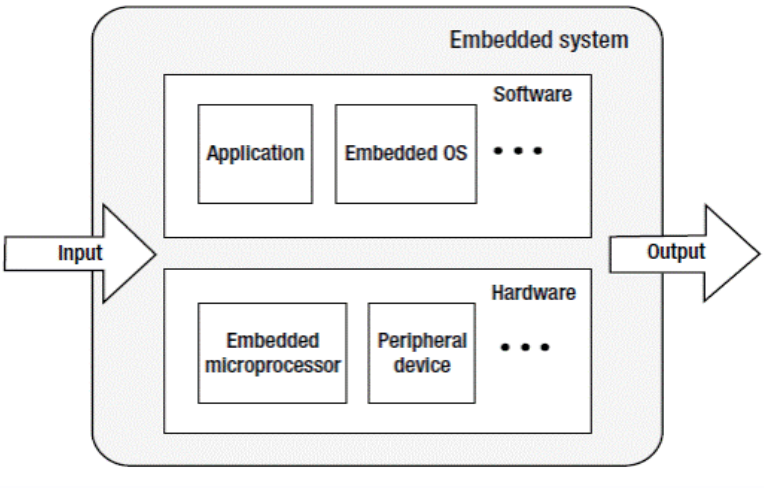


Figure 1.2: Basic architecture of an embedded system

The input signal may be an ergonomic device (such as a keyboard, mouse, or touch screen) or the output of a sensor circuit in another embedded system. The output may be in the form of sound, light, electricity, or another analog signal, or a record or file for a database.

The basic computer system components—microprocessor, memory, and input and output modules are interconnected by a system bus in order for all the parts to communicate and execute a program (see Figure 1-3).

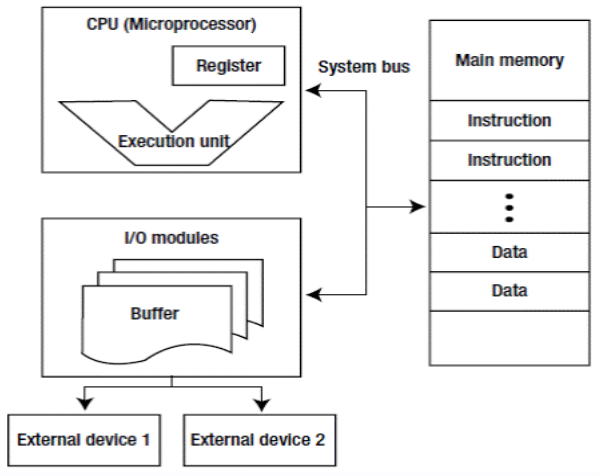


Figure 1.3: Hardware architecture of Embedded System

In embedded systems, the microprocessor's role and function are usually the same as those of the CPU in a general-purpose computer: control computer operation, execute instructions, and process data. In many cases, the microprocessor in an embedded system is also called the CPU. Memory is used to store instructions and data. I/O modules are responsible for the data exchange between the processor, memory, and external devices.

External devices include secondary storage devices (such as flash and hard disk), communications equipment, and terminal equipment. The system bus provides data and controls signal communication and transmission for the processor, memory, and I/O modules.

There are basically two types of architecture that apply to embedded systems: Von Neumann architecture and Harvard architecture. In a Von-Neumann architecture, the same memory and bus are used to store both data and instructions that run the program. Since you cannot access program memory and data memory simultaneously, the Von Neumann architecture is susceptible to bottlenecks and system performance is affected.

1.3.1 Von Neumann Architecture

Von Neumann architecture (also known as Princeton architecture) was first proposed by John von Neumann. The most important feature of this architecture is that the software and data use the same memory: that is, "The program is data, and the data is the program" (as shown in Figure 1-4).

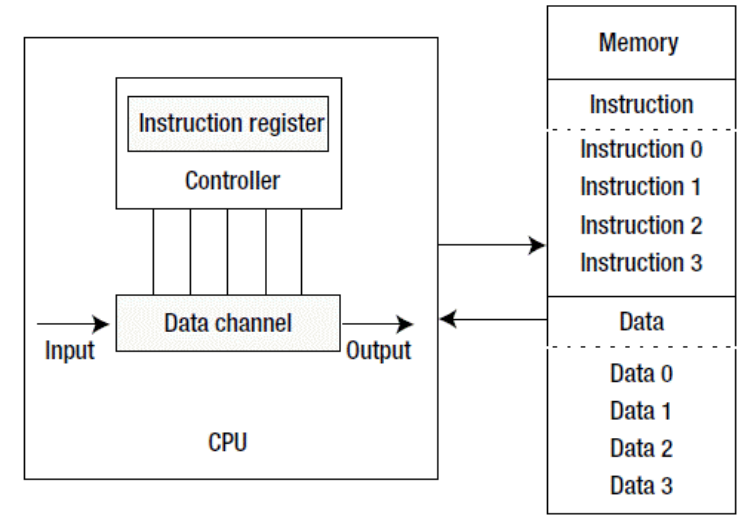


Figure 1.4: Von Neumann architecture

In the Von Neumann architecture, an instruction and data share the same bus. In this architecture, the transmission of information becomes the bottleneck of computer performance and affects the speed of data processing; so, it is often called the *Von Neumann bottleneck*. In reality, cache and branch-prediction technology can effectively solve this issue.

1.3.2 Harvard Architecture

The Harvard architecture was first named after the Harvard Mark I computer. Compared with the Von Neumann architecture, a Harvard architecture processor has two outstanding

features. First, instructions and data are stored in two separate memory modules; instructions and data do not coexist in the same module. Second, two independent buses are used as dedicated communication paths between the CPU and memory; there is no connection between the two buses. The Harvard architecture is shown in Figure 1-5.

To efficiently perform memory reads/writes, the processor is not directly connected to the main memory, but to the cache. Commonly, the only difference between the Harvard architecture and the Von Neumann architecture is single or dual L1 cache. In the Harvard architecture, the L1 cache is often divided into an instruction cache (I cache) and a data cache (D cache), but the Von-Neumann architecture has a single cache.

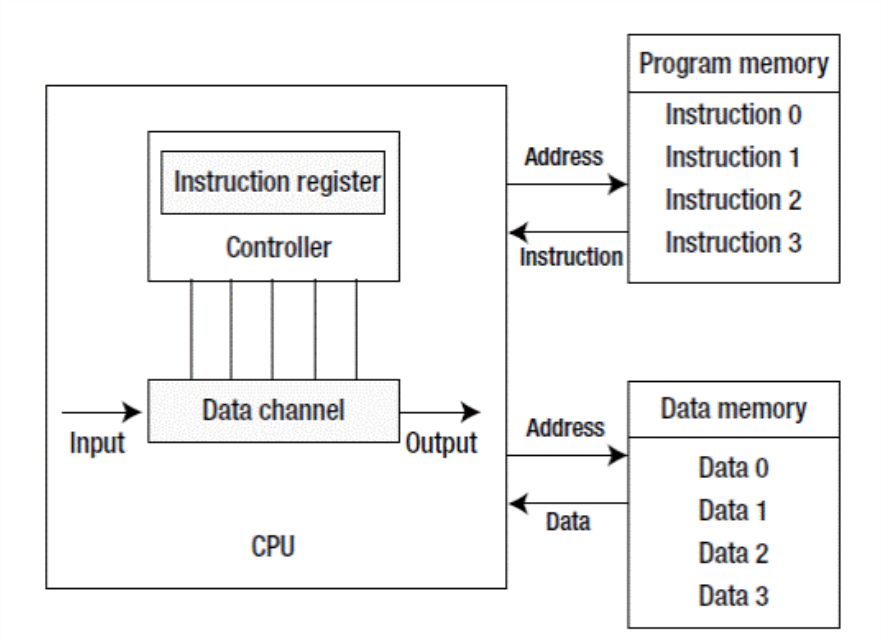


Figure 1.5: Harvard architecture

Because the Harvard architecture has separate program memory and data memory, it can provide greater data-memory bandwidth, making it the ideal choice for digital signal processing. Most systems designed for digital signal processing (DSP) adopt the Harvard architecture. The Von Neumann architecture features simple hardware design and flexible program and data storage and is usually the one chosen for general-purpose and most embedded systems.

1.4. Microprocessor Architecture for Embedded Systems

A microprocessor is the CPU of the computer fabricated on a single chip. The microprocessor is the core in embedded systems. By installing a microprocessor into a special circuit board and adding the necessary peripheral circuits and expansion circuits, a practical embedded system can be created. The microprocessor architecture determines the instructions,

supporting peripheral circuits, and expansion circuits. There are wide ranges of microprocessors: 8-bit, 16-bit, 32-bit and 64-bit, with clock performance from MHz to GHz, and ranging from a few pins to thousands of pins.

In general, there are two types of embedded microprocessor architecture: reduced instruction set computer (RISC) and complex instruction set computer (CISC). The RISC Nprocessor uses a small, limited, simple instruction set. Each instruction uses a standard word length and has a short execution time, which facilitates the optimization of the instruction pipeline. To compensate for the command functions, the CPU is often equipped with a large number of general-purpose registers. The CISC processor features a powerful instruction set and different instruction lengths, which facilitates the pipelined execution of instructions.

Currently, microprocessors used in most embedded systems have five architectures: RISC, CISC, MIPS, PowerPC, and SuperH. The details follow.

1.4.1 CISC Architecture

The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. Computers based on the CISC architecture are designed to decrease the memory cost (figure 1.6).

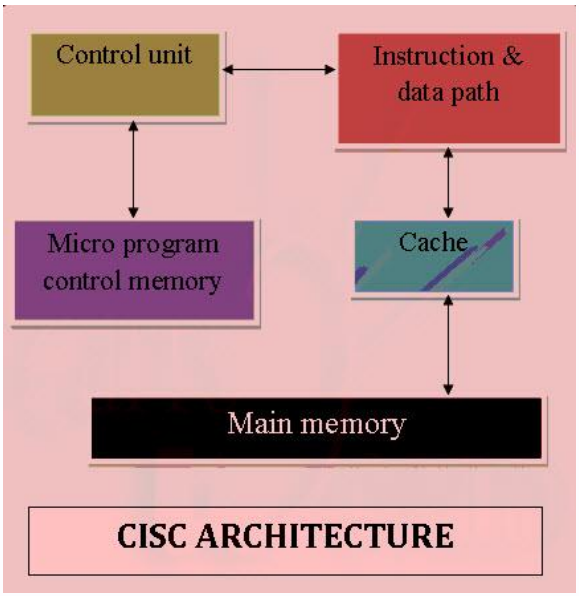


Figure 1.6 CISC Architecture

Because, the large programs need more storage, thus increasing the memory cost and large memory becomes more expensive. To solve these problems, the number of instructions per program can be reduced by embedding the number of operations in a single instruction, thereby making the instructions more complex.

Characteristics of CISC processor

- MUL loads two values from the memory into separate registers in CISC.
- CISC uses minimum possible instructions by implementing hardware and executes operations.
- Instruction-decoding logic will be Complex.
- One instruction is required to support multiple addressing modes.
- Less chip space is enough for general purpose registers for the instructions that are operated directly on memory.
- Various CISC designs are set up two special registers for the stack pointer, handling interrupts, etc.
- MUL is referred to as a "complex instruction" and requires the programmer for storing functions.

Note: Instruction Set Architecture is a medium to permit communication between the programmer and the hardware. Data execution part, copying of data, deleting or editing is the user commands used in the microprocessor and with this microprocessor the Instruction set architecture is operated.

Examples of CISC PROCESSORS

- **IBM 370/168** – It was introduced in the year 1970. CISC design is a 32 bit processor and four 64-bit floating point registers.
- **VAX 11/780** – CISC design is a 32-bit processor and it supports many numbers of addressing modes and machine instructions which is from Digital Equipment Corporation.
- **Intel 80486** – It was launched in the year 1989 and it is a CISC processor, which has instructions varying lengths from 1 to 11 and it will have 235 instructions.

1.4.2 RISC Architecture

RISC (Reduced Instruction Set Computer) processors take simple instructions and are executed within a clock cycle. The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy which has become known as RISC. Certain design features have been characteristic of most RISC processors:

- *one cycle execution time*: RISC processors have a CPI (clock per instruction) of one cycle. This is due to the optimization of each instruction on the CPU and a technique called pipelining.
- *pipelining*: A techique that allows for simultaneous execution of parts, or stages, of instructions to more efficiently process instructions;
- *large number of registers*: the RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory

RISC is used in portable devices due to its power efficiency. For Example, Apple iPod and Nintendo DS. RISC is a type of microprocessor architecture that uses highly-optimized set of instructions. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program Pipelining is one of the unique feature of RISC. It is performed by overlapping the execution of several instructions in a pipeline fashion. It has a high performance advantage over CISC.

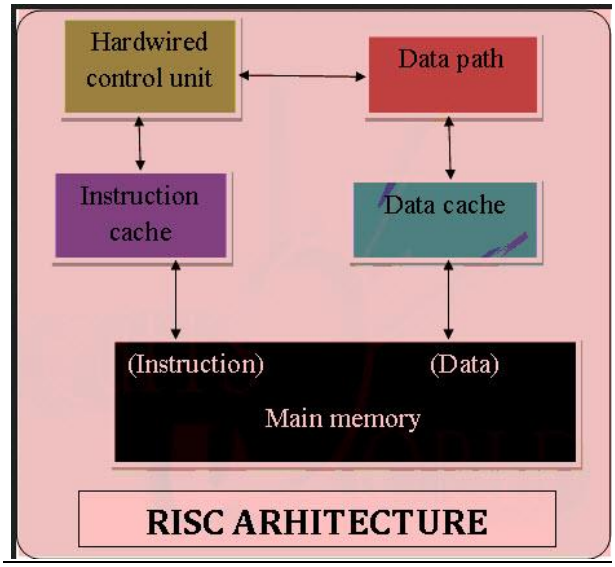


Figure 1.7: RISC Architecture

RISC Architecture Characteristics

- Simple Instructions are used in RISC architecture.
- RISC helps and supports few simple data types and synthesize complex data types.
- RISC utilizes simple addressing modes and fixed length instructions for pipelining.
- RISC permits any register to use in any context.

- One Cycle Execution Time
- The amount of work that a computer can perform is reduced by separating “LOAD” and “STORE” instructions.
- RISC contains Large Number of Registers in order to prevent various number of interactions with memory.
- In RISC, Pipelining is easy as the execution of all instructions will be done in a uniform interval of time i.e. one click.
- In RISC, more RAM is required to store assembly level instructions.
- Reduced instructions need a less number of transistors in RISC.
- RISC uses Harvard memory model means it is Harvard Architecture.
- A compiler is used to perform the conversion operation means to convert a high-level language statement into the code of its form.

A comparison of RISC and CISC is given in Table 1-2.

Table 1-2. Comparison of RISC and CISC

	RISC	CISC
Instruction system	Simple and efficient instructions. Realizes uncommon functions through combined instructions.	Rich instruction system. Performs specific functions through special instructions; handles special tasks efficiently.
Memory operation	Restricts the memory operation and simplifies the controlling function.	Has multiple memory operation instructions and performs direct operation.
Program	Requires a large amount of memory space for the assembler and features complex programs for special functions.	Has a relatively simple assembler and features easy and efficient programming of scientific computing and complex operations.
Interruption	Responds to an interrupt only at the proper place in instruction execution.	Responds to an interruption only at the end of execution.
CPU	Features fewer unit circuits, small size, and low power consumption.	Has feature-rich circuit units, powerful functions, a large area, and high power consumption.
Design cycle	Features a simple structure, a compact layout, a short design cycle, and easy application of new technologies.	Features a complex structure and long design cycle.
Usage	Features a simple structure, regular instructions, simple control, and easy learning and application.	Features a complex structure, powerful functions, and easy realization of special functions.
Application scope	Determines the instruction system per specific areas, which is more suitable for special machines.	Becomes more suitable for general-purpose machines.

RISC and CISC have distinct characteristics and advantages, but the boundaries between RISC and CISC begin to blur in the microprocessor sector. Many traditional CISCs absorb RISC advantages and use a RISC-like design. Intel x86 processors are typical of them. They are considered as CISC architecture. These processors translate x86 instructions into RISC-like instructions through a decoder and comply with the RISC design and operation to obtain the benefits of RISC architecture and improve internal operation efficiency.

1.5 System on Chip (SoC) Processor

With the development of integrated circuit design and manufacturing technology, integrated circuit design has gone from transistor integration, to logic-gate integration, to the current IP integration or system on chip (SoC). The SoC design technology integrates popular circuit modules on a single chip. SoC usually contains a large number of peripheral function modules such as microprocessor/microcontroller, memory, USB controller, universal asynchronous receiver/transmitter (UART) controller, A/D and D/A conversion, I2C, and Serial Peripheral Interface (SPI). Figure 1-8 is an example structure of SoC-based hardware for embedded systems.

A System on Chip or an SoC is an integrated circuit that incorporates a majority of components present on a computer. As the name suggests, it is an entire system fabricated on a silicon chip. SoC also includes software and an interconnection structure for integration. The hardware-software integration approach makes the SoC smaller in size, allows for less power consumption, and more reliable than a standard multi-chip system.

1.5.1 Components of an SoC

SoCs can be identified as the following types: built around a microcontroller, build around a microprocessor, built for specific applications, and programmable SoCs (PSoC). The integral parts of an SoC include a processor, primary and secondary memory storage and input/output ports. The other vital components include a graphics processor unit (GPU), a WiFi module, Digital Signal Processor (DSP), and various peripherals such as USB, Ethernet, SPI (Serial Peripheral Interface), ADC, DAC, and even FPGAs. Usually, it has multiple cores. Depending on various deciding factors and preferences, the core can be a microcontroller, microprocessor, DSP, or even an ASIP (Application Specific Instruction- set Processor). ASIPs have instruction sets based on a particular application. Usually, SoCs use ARM architecture, which is a family of RISC (Reduced Instruction Set Computing), which requires less digital design, thereby making it compatible for embedded system use. The ARM architecture is much more power-efficient than processors like the 8051 because, in contrast to processors using the

CISC architecture, processors with RISC architecture require fewer transistors. This also reduces heat dissipation and the cost.

The following diagram shows an example of an SoC block diagram.

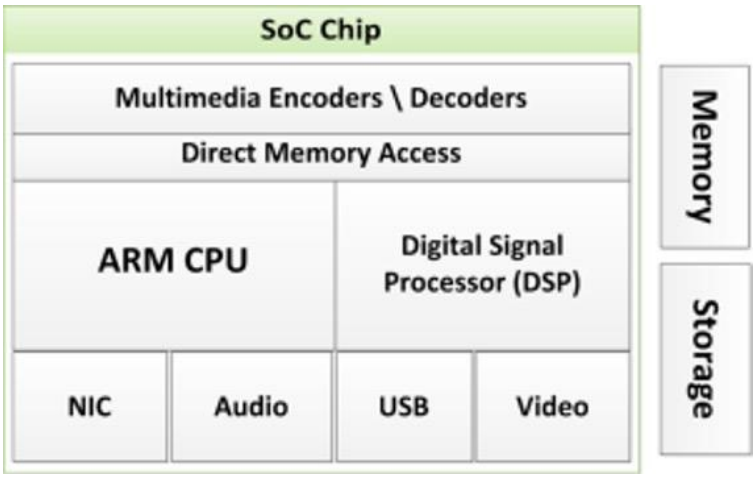


Figure 1.8: Example of an SoC block diagram.

1.5.2 Processor architecture/Models for SoC

At the heart of the SoC is its Processor. It usually has multiple processor cores. Multiple cores allow different processes to run at the same time, which increases the speed of the system as it enables your computer to perform multiple operations at the same time. The operating system sees the multiple cores as multiple CPUs, which increases performance. As multiple cores are fitted onto the same chip, there is less latency, which is because of faster communication between the cores.

1.5.2.1 Simple Sequential Processor

Sequential processors directly implement the sequential execution model. These processors process instructions sequentially from the instruction stream. The next instruction is not processed until all execution for the current instruction is complete and its results have been committed. The semantics of the instruction determines the sequence of actions that must be performed to produce the specified result. These actions include

1. fetching the instruction into the instruction register (IF),
2. decoding the opcode of the instruction (ID),
3. generating the address in memory of any data item residing there (AG),

4. fetching data operands into executable registers (DF),

5. executing the specified operation (EX), and

6. writing back the result to the register file (WB).

A simple sequential processor model is shown in Figure 1.9. During execution, a sequential processor executes one or more operations per clock cycle from the instruction stream. An instruction is a container that represents the smallest execution packet managed explicitly by the processor. One or more operations are contained within an instruction. The distinction between instructions and operations is crucial to distinguish between processor behaviors. Scalar and superscalar processors consume one or more instructions per cycle, where each instruction contains a single operation. Although conceptually simple, executing each instruction sequentially has significant performance drawbacks: A considerable amount of time is spent on overhead and not on actual execution. Thus, the simplicity of directly implementing the sequential execution model has significant performance costs.

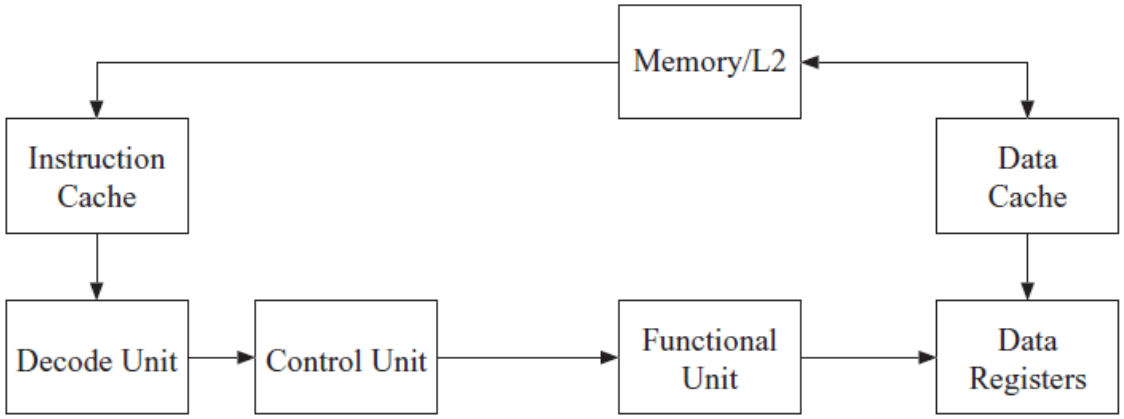


Figure1.9: Sequential Processor Model

1.5.2.2 Pipelined Processor

Pipelining is a straightforward approach to exploiting parallelism that is based on concurrently performing different phases (instruction fetch, decode, execution, etc.) of processing an instruction. Pipelining assumes that these phases are independent between different operations and can be overlapped — when this condition does not hold, the processor stalls the downstream phases to enforce the dependency. Thus, multiple operations can be processed simultaneously with each operation at a different phase of its processing. Figure 1.10 illustrates the instruction timing in a pipelined processor, assuming that the instructions are independent.

For a simple pipelined machine, there is only one operation in each phase at any given time; thus, one operation is being fetched (IF); one operation is being decoded (ID); one operation is generating an address (AG); one operation is accessing operands (DF); one operation is in execution (EX); and one operation is storing results (WB). Figure 1.10 illustrates the general form of a pipelined processor.

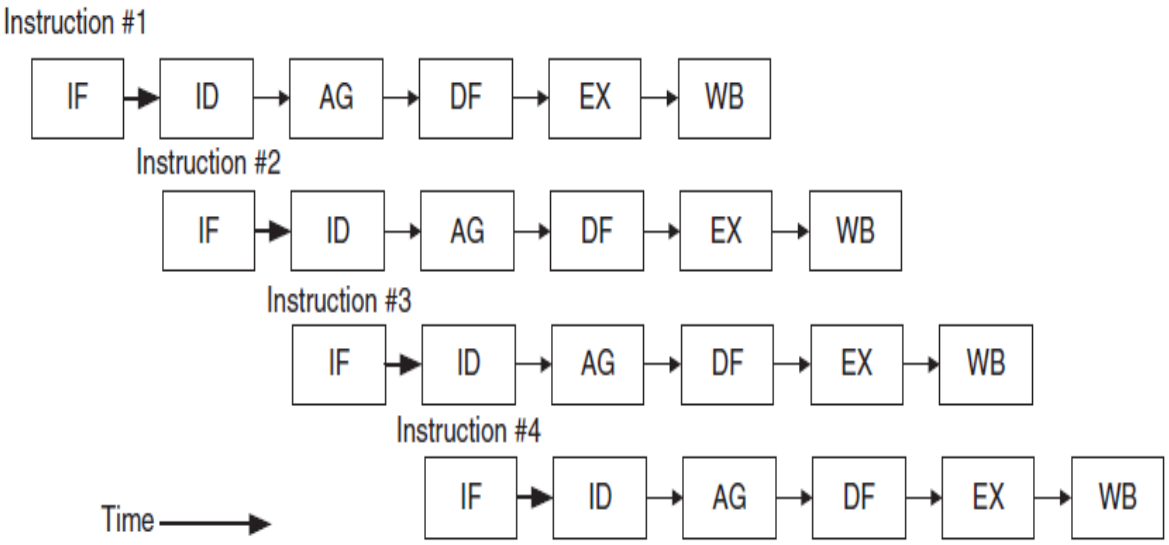


Figure 1.10: Instruction Execution in a Pipelined Processor

The most rigid form of a pipeline, sometimes called the static pipeline, requires the processor to go through all stages or phases of the pipeline whether required by a particular instruction or not. A dynamic pipeline allows the bypassing of one or more pipeline stages, depending on the requirements of the instruction. The more complex dynamic pipelines allow instructions to complete out of (sequential) order, or even to initiate out of order. The out - of - order processors must ensure that the sequential consistency of the program is preserved.

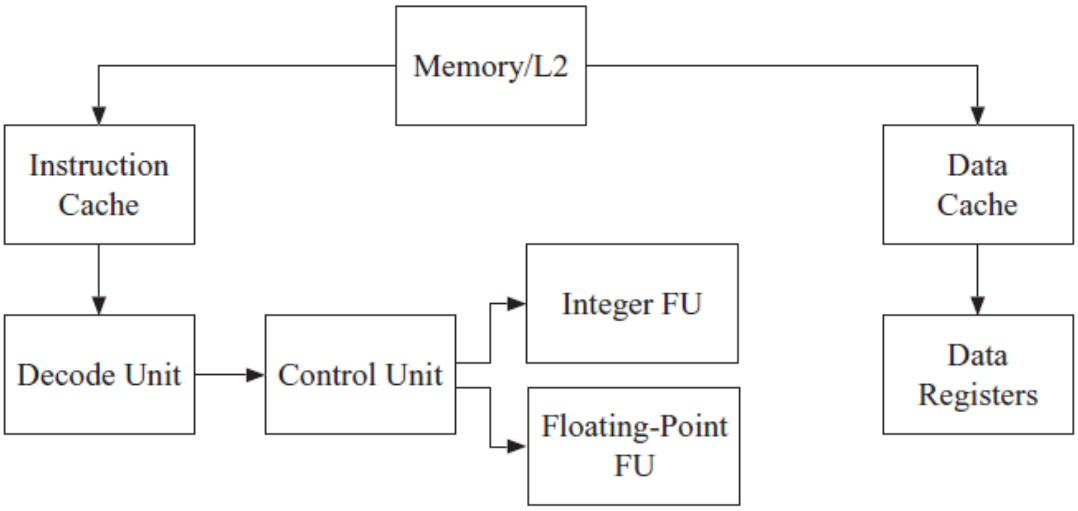


Figure 1.11 : Pipelined processor model.

Two architectures that exploit ILP (Instruction level parallelism) are *superscalar* and *VLIW* processors. They use different techniques to achieve execution rates greater than one operation per cycle. A superscalar processor dynamically examines the instruction stream to determine which operations are independent and can be executed. A VLIW processor relies on the compiler to analyze the available operations (OP) and to schedule independent operations into wide instruction words, which then execute these operations in parallel with no further analysis.

1.5.2.3 Superscalar Processors

Dynamic pipelined processors remain limited to executing a single operation per cycle by virtue of their scalar nature. This limitation can be avoided with the addition of multiple functional units and a dynamic scheduler to process more than one instruction per cycle (Figure 1.12). These superscalar processors can achieve execution rates of several instructions per cycle (usually limited to two, but more is possible depending on the application). The most significant advantage of a superscalar processor is that processing multiple instructions per cycle is done transparently to the user, and that it can provide binary code compatibility while achieving better performance.

Compared to a dynamic pipelined processor, a superscalar processor adds a scheduling instruction window that analyses multiple instructions from the instruction stream in each cycle. Although processed in parallel, these instructions are treated in the same manner as in a pipelined

processor. Before an instruction is issued for execution, dependencies between the instruction and its prior instructions must be checked by hardware.

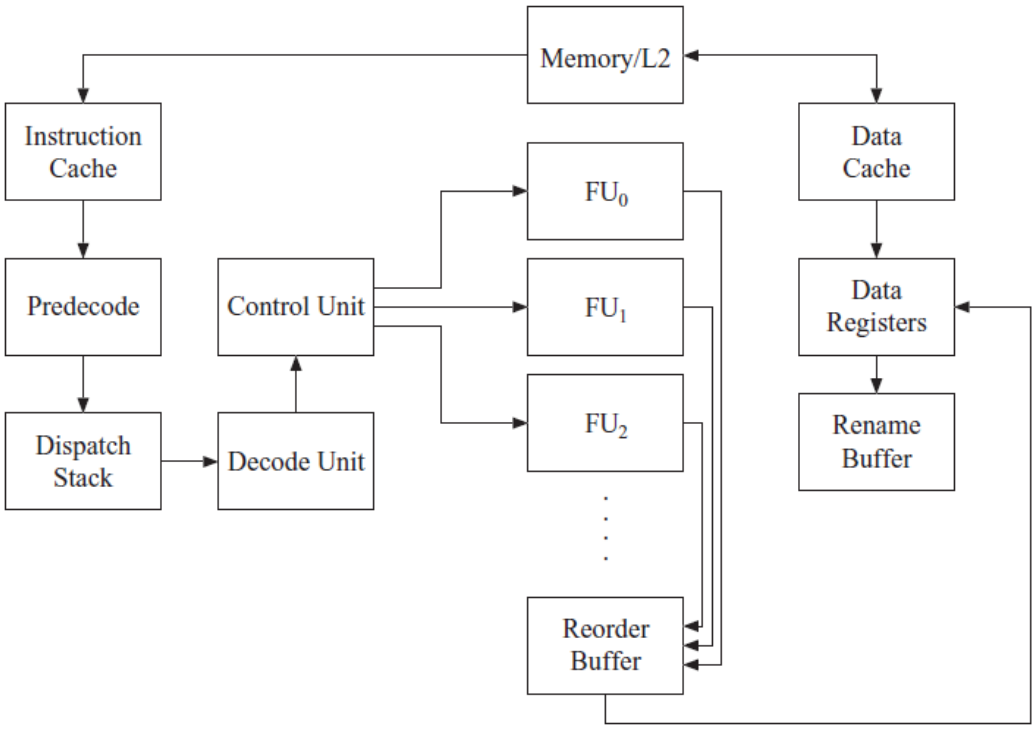


Figure 1.12 Superscalar processor model.

Because of the complexity of the dynamic scheduling logic, high – performance superscalar processors are limited to processing four to six instructions per cycle. Although superscalar processors can exploit ILP from the dynamic instruction stream, exploiting higher degrees of parallelism requires other approaches.

1.5.2.4 VLIW Processors

In contrast to dynamic analyses in hardware to determine which operations can be executed in parallel, VLIW processors (Figure 1.13) rely on static analyses in the compiler. VLIW processors are thus less complex than superscalar processors and have the potential for higher performance. A VLIW processor executes operations from statically scheduled instructions that contain multiple independent operations. Because the control complexity of a VLIW processor is not significantly greater than that of a scalar processor, the improved performance comes without the complexity penalties. VLIW processors rely on the static analyses performed by the compiler and are unable to take advantage of any dynamic execution characteristics. For applications that can be scheduled statically to use the processor resources

effectively, a simple VLIW implementation results in high performance. Unfortunately, not all applications can be effectively scheduled statically. In many applications, execution does not proceed exactly along the path defined by the code scheduler in the compiler.

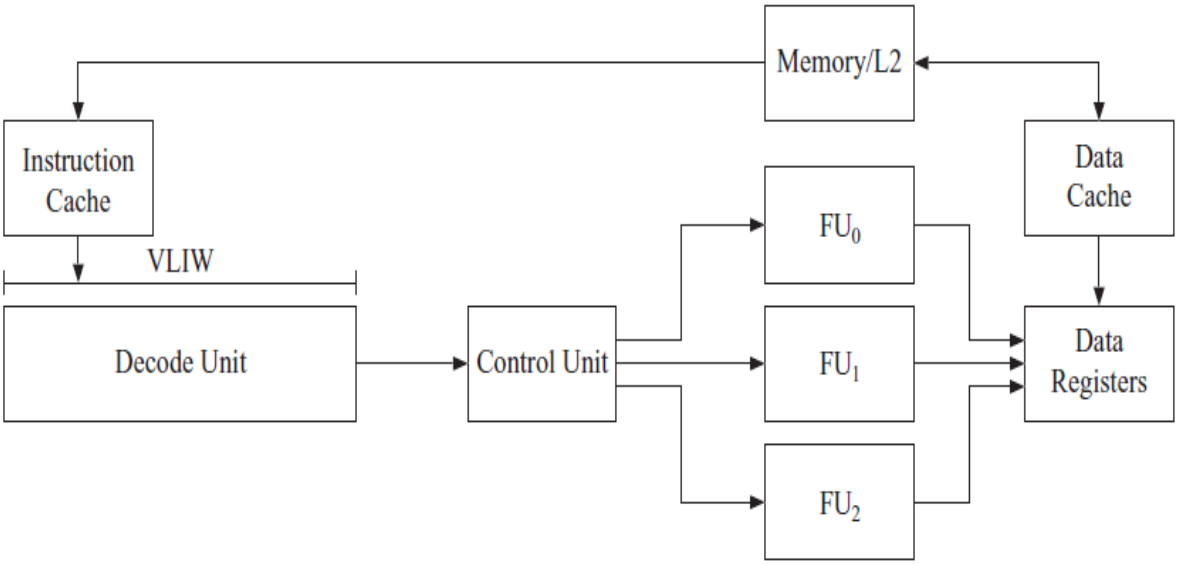


Figure 1.13 VLIW processor model.

Two classes of execution variations can arise and affect the scheduled execution behavior:

1. delayed results from operations whose latency differs from the assumed latency scheduled by the compiler and
2. interruptions from exceptions or interrupts, which change the execution path to a completely different and unanticipated code schedule.

Although stalling the processor can control a delayed result, this solution can result in significant performance penalties. The most common execution delay is a data cache miss. Many VLIW processors avoid all situations that can result in a delay by avoiding data caches and by assuming worst - case latencies for operations. However, when there is insufficient parallelism to hide the exposed worst - case operation latency, the instruction schedule has many incompletely filled or empty instructions, resulting in poor performance.

1.5.3 Digital Signal Processor (DSP)

Digital Signal Processor (DSP) is a chip optimized for operations for digital signal processing. This includes operations for sensors, actuators, data processing, and data analysis. It can be used for image decoding. The use of DSP saves CPU cycles for other processing tasks,

which increases performance. Dedicated DSPs are more power-efficient, which makes them befitting for use in SoCs. The instruction set used for DSP cores is SIMD (Single Instruction, Multiple Data) and VLIW (Very Long Instruction Word). The use of this architecture allows for parallel processing of instructions and superscalar execution. DSPs are used to perform operations like Fast Fourier Transform, convolution, multiply-accumulate.

1.5.4 Memories on SoC

SoCs have memories based on the application. The memories are semiconductor memory blocks for computation purposes. Semiconductor memory usually refers to Metal Oxide Semiconductor memory cells, which are fabricated on a single silicon chip. The types of memories are:

- Volatile memories: Memories that lose data after power off. In other words, they need a constant power source to retain information. Volatile memories are faster and cheaper, which is why they are chosen frequently.

RAM is a type of volatile memory. The most common RAM used are SRAM (Static RAM) and DRAM (Dynamic RAM). SRAM is made of memory cells which consist of either 1,3 or 6 transistors (MOSFETs). In contrast, DRAM has only one MOSFET and a capacitor which is charged and discharged according to the state of the FET. However, DRAM is prone to capacitor leakage currents. One significant advantage of DRAM is that its cheaper than SRAM. If an SoC has a cache hierarchy, SRAM is used for cache and DRAM is used for the main memory. This is because cache requires a faster type of memory as compared to the main memory.

There are RAM types designed for non-volatile function as well. These are FRAM (Ferroelectric RAM), MRAM (Magneto-resistive random-access memory), which stores data in magnetic states, PRAM (Parameter Random Access Memory), which is used in Macintosh computers to store system settings including the display and time-zone settings. Other than these, there is RRAM (Resistive Random Access Memory), which has a component called memristor. A memristor is a resistor whose voltage varies as per the applied voltage.

- Non-volatile memories: Memories that retain information even in the absence of a power source. ROM (Read Only Memory) is a kind of non-volatile memory. Types of ROM include EPROM (Erasable Programmable Read-Only Memory), which is an array of floating-gate transistors. UVROM (Ultra-Violet Erasable Programmable Read-Only Memory), which is erased using UV light and reprogrammed with data, EEPROM (Electrically Erasable Programmable ROM) and flash.

The type of memory selected depends upon the design specifications and application.

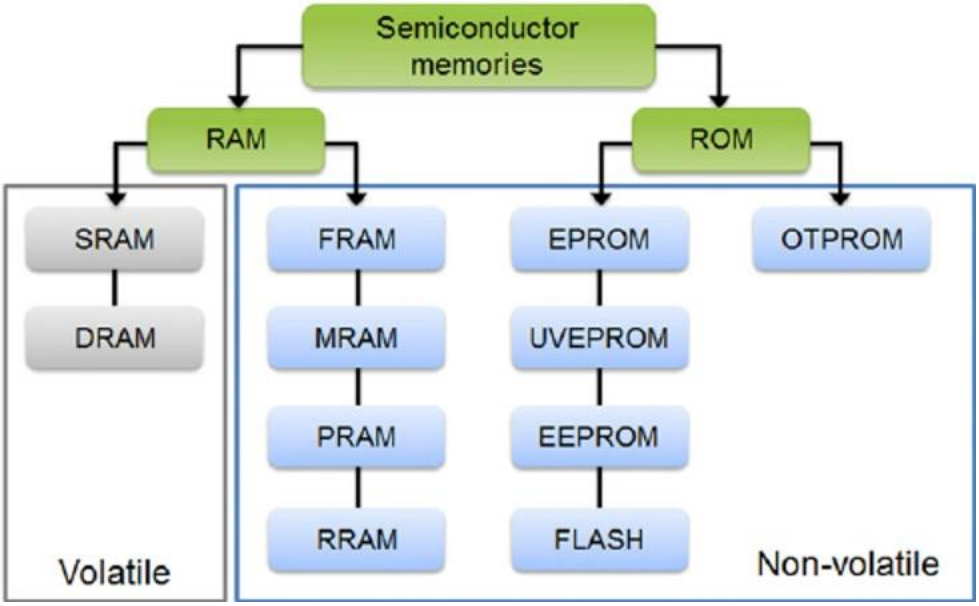


Figure 1.14: Classification of semiconductor memories used SoC.

1.5.5 SYSTEM - LEVEL INTERCONNECTION

SOC technology typically relies on the interconnection of predesigned circuit modules (known as intellectual property [IP] blocks) to form a complete system, which can be integrated onto a single chip. In this way, the design task is raised from a circuit level to a system level. Central to the system – level performance and the reliability of the finished product is the method of interconnection used. A well - designed interconnection scheme should have vigorous and efficient communication protocols, unambiguously defined as a published standard. This facilitates interoperability between IP blocks designed by different people from different organizations and encourages design reuse. It should provide efficient communication between different modules maximizing the degree of parallelism achieved. SOC interconnect methods can be classified into two main approaches:

- buses and
- network - on - chip

1.5.5.1 Bus - Based Approach

With the bus - based approach, IP blocks are designed to conform to published bus standards such as ARM ’ s Advanced Microcontroller Bus Architecture (AMBA)

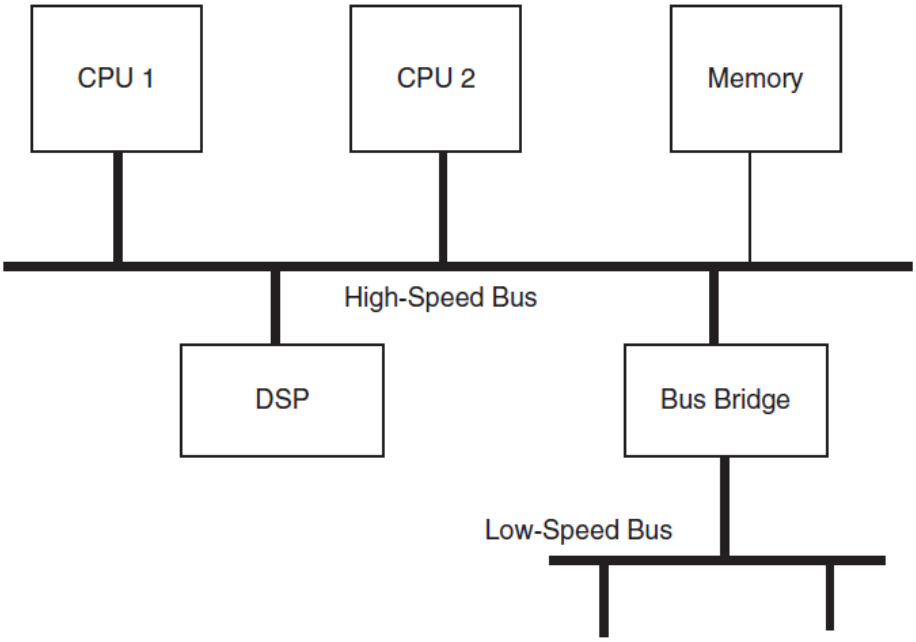


Figure 1.15: System - level interconnection: bus - based approach.

or IBM's CoreConnect. Communication between modules is achieved through the sharing of the physical connections of address, data, and control bus signals. This is a common method used for SOC system – level interconnect. Usually, two or more buses are employed in a system, organized in a hierarchical fashion. To optimize system - level performance and cost, the bus closest to the CPU has the highest bandwidth, and the bus farthest from the CPU has the lowest bandwidth.

1.5.5.2 Network - on - Chip Approach

A network - on - chip system consists of an array of switches, either dynamically switched as in a crossbar or statically switched as in a mesh. The crossbar approach uses asynchronous channels to connect synchronous modules that can operate at different clock frequencies. This approach has the advantage of higher throughput than a bus - based system while making integration of a system with multiple clock domains easier. In a simple statically switched network (Figure 1.16), each node contains processing logic forming the core, and its own routing logic. The interconnect scheme is based on a two - dimensional mesh topology. All communications between switches are conducted through data packets, routed through the router interface circuit within each node. Since the interconnections between switches have a fixed distance, interconnect - related problems such as wire delay and cross talk noise are much reduced.

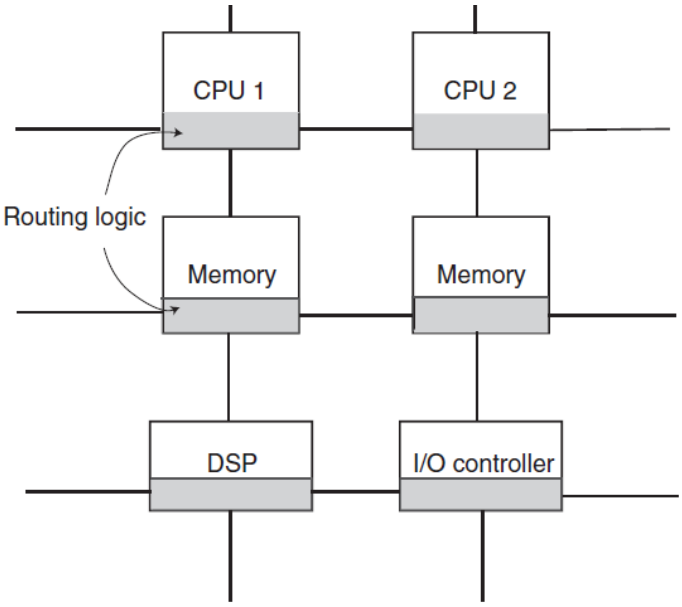


Figure 1.16: SOC interconnection: Network - on - Chip approach.

The Network-On-Chip employs system-level network techniques for on-chip traffic management. The NOC is a homogeneous, scalable switch fabric network that is used to transport multi-purpose data packets. This architecture is layered in nature with user-defined technology. The communication takes place over a three-layer communication scheme, namely Transaction, Transport and Physical.

The aim of a NOC interconnect fabric is to reduce the wire routing congestion on-chip, better timing closure, a standardized way to make changes various IPs to the SOC design. NOC architectures have proven to be more power-efficient and can match throughput requirements.

1.5.6 External interfaces

SOC interfaces defer as per the intended application. The external interfaces are commonly based on communication protocols such as WiFi, USB, Ethernet, I2C, SPI, HDMI. If required, analog interfaces may be added for interfacing with sensors and actuators.

1.5.7 Other components

Other components necessary for a fully functioning SOC are timing sources like clocks, timers, oscillators, phase lock loop systems, voltage regulators, and power management units.

1.5.6 Advantages & disadvantages of SoC

The main aim of an SoC is to minimize external components. Hence, it has the following advantages over a Single Board Computer:

- Size: The SoC is the size of a coin. Due to the rapidly decreasing size of MOS technology, SOCs can be made very small while being able to perform complex tasks. The size does not impact the features of the chip.
- Decreased power consumption: An SoC is optimized for low-power devices like cell-phones. Low power consumption results in higher battery capacity in cell-phones.
- Flexibility: SoCs are easily reprogrammable, which makes them flexible. They so allow the reuse of IPs.
- Reliability: SoCs offer high circuit security and reduced design complexity.
- Cost Efficient: Mainly due to fewer physical components and design reuse
- Faster circuit operation

SoCs pose some disadvantages as well:

1. **Time Consuming:** The entire process from design to fabrication can take between 6 months to 1 year. Hence, the time to market demand is very high.
2. Design Verification requirements are very high and consume 70% of the total time. DV is tedious due to the increasing complexity of SoC design.
3. Availability and compatibility of IPs play a very significant role, which can add to the time to market.
4. Exponentially increasing fabrication costs.
5. For low volume products, SoC may not be the best option.

1.5.7 Applications

The most common application of SOCs today is in mobile applications, including smartphones, smartwatches, tablets. Other applications include signal speech processing, PC interfaces, data communication. SoCs are being applied to personal computers as well due to the integration of communication modules like LTE and wireless networks onto the chip.

The most popular SoCs in the market today are manufactured by Qualcomm Technologies for smartphones, smartwatches, and the upcoming 5G network compatibility. Other manufacturers include Intel Technology, Samsung Inc, Apple Inc., among many others.

1.6 Software Development process for embedded system

Because machine code is the only language the hardware can directly execute, all other languages need some type of mechanism to generate the corresponding machine code. This mechanism usually includes one or some combination of *preprocessing*, *translation*, and *interpretation*. Depending on the language, these mechanisms exist on the programmer's ***host*** system (typically a non-embedded development system, such as a PC), or the ***target*** system (the embedded system being developed). See Figure 1.17.

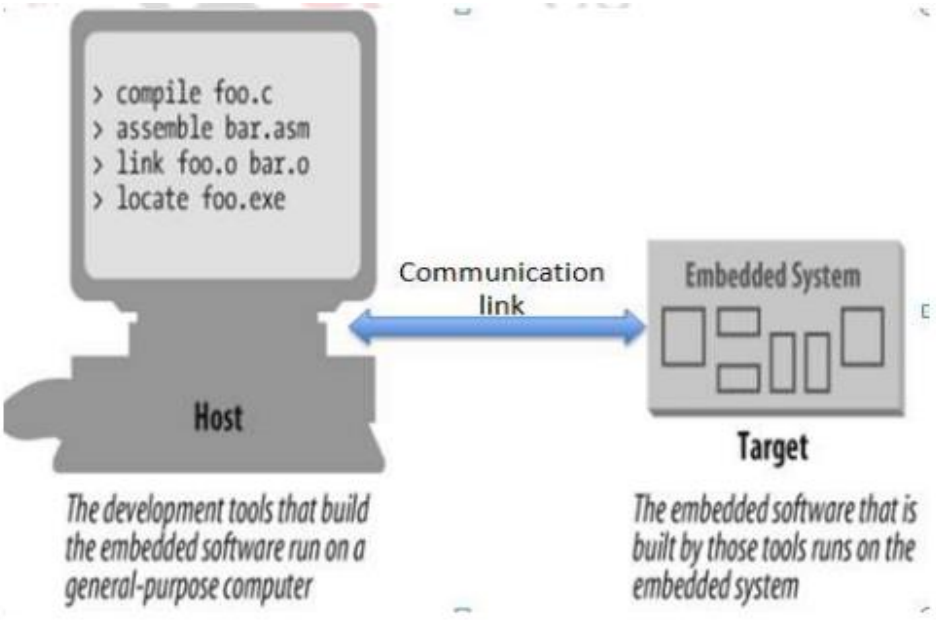


Figure 1.17: Host and target system diagram

Preprocessing is an optional step that occurs before either the translation or interpretation of source code, and whose functionality is commonly implemented by a ***preprocessor***. The preprocessor's role is to organize and restructure the source code to make translation or interpretation of this code easier. As an example, in languages like C and C++, it is a preprocessor that allows the use of named code fragments, such as *macros*, that simplify code development by allowing the use of the macro's name in the code to replace fragments of code. The preprocessor then replaces the macro name with the contents of the macro during preprocessing. The preprocessor can exist as a separate entity, or can be integrated within the translation or interpretation unit.

1.6.1 Compiler

Many languages convert source code, either directly or after having been preprocessed through use of a ***compiler***, a program that generates a particular target language such as machine

code and Java byte code from the source language as depicted in Figure 1.18. A compiler typically "translates" all of the source code to some target code at one time. As is usually the case in embedded systems, compilers are located on the programmer's host machine and generate target code for hardware platforms that differ from the platform the compiler is actually running on. These compilers are commonly referred to as ***cross-compilers***. In the case of assembly language, the compiler is simply a specialized cross-compiler referred to as an ***assembler***, and it always generates machine code. The language name plus the term "compiler, "such as" Java compiler and C compiler, commonly refer to other high-level language compilers.

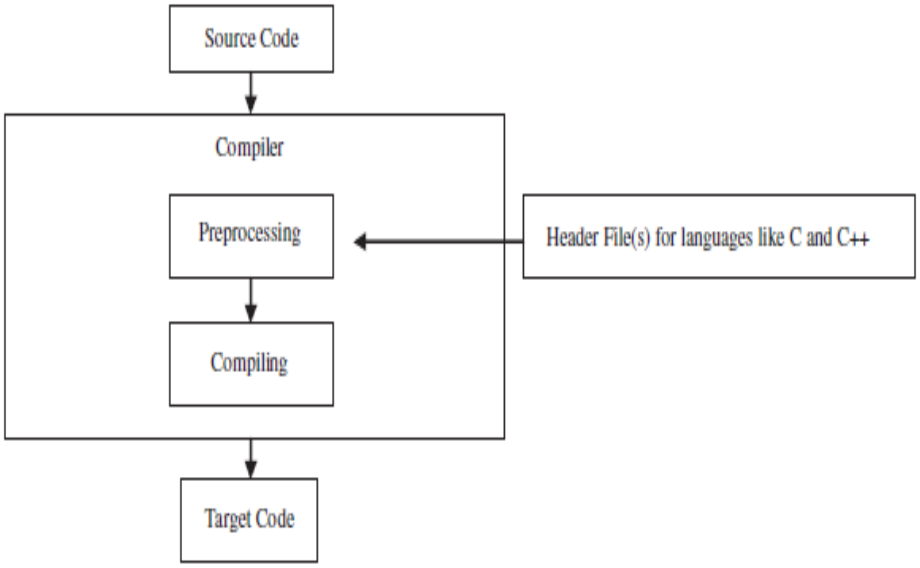


Figure 1.18 General functions of an Embedded software

High-level language compilers vary widely in terms of what is generated. Some generate machine code, while others generate other high-level code, which then requires what is produced to be run through at least one more compiler or interpreter, as discussed later in this section. Other compilers generate assembly code, which then must be run through an assembler. After all the compilation on the programmer's host machine is completed, the remaining target code file is commonly referred to as an ***object file***, and can contain anything from machine code to Java byte code (discussed later in this section), depending on the programming language used. As shown in Figure 1.13, after linking this object file to any system libraries required, the object file, commonly referred to as an ***executable***, is then ready to be transferred to the target embedded system's memory.

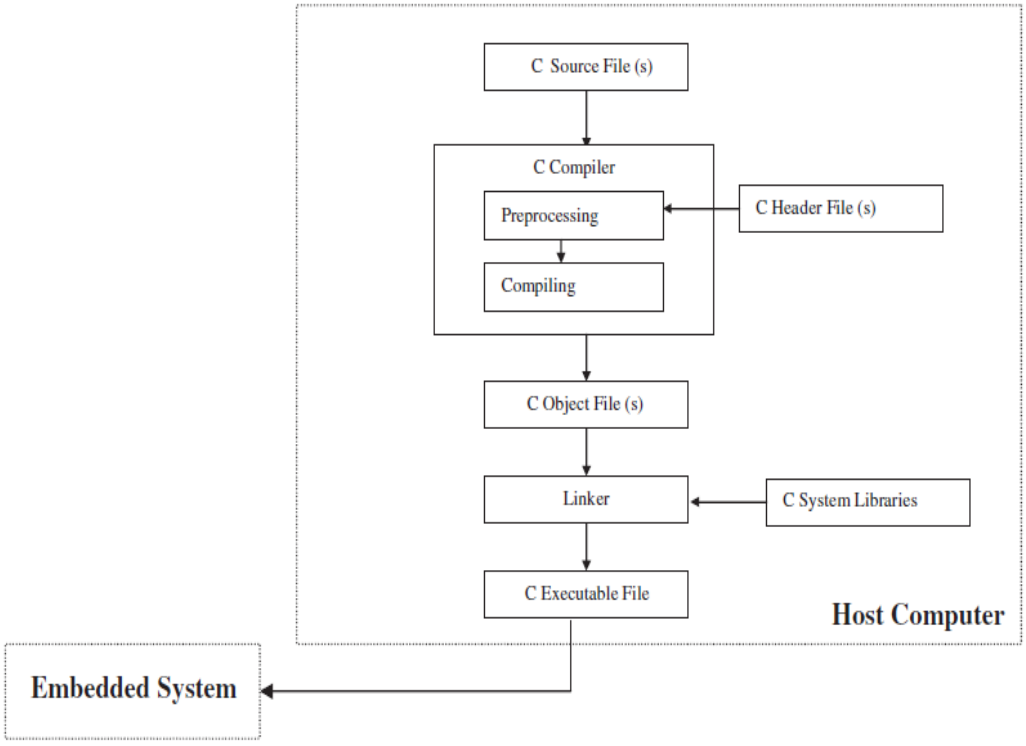


Figure 1.19: C Example compilation/linking steps and object file results

References

[1] F. Vahid and T. Givargis, "Embedded System Design: A Unified Hardware/Software Introduction", Wiley India Pvt. Ltd., 2002.

[2] Michael J. Flynn and Wayne Luk, "Computer System Design System-on-Chip", Wiley India Pvt. Ltd.

[3] Steve Furber, "ARM System on Chip Architecture ", 2nd Edition, 2000, Addison Wesley Professional.AAAXZX

[4] Pascricha and N. Dutt, Morgan Kaufmann, On-Chip Communication Architectures, System on Chip Interconnect, -Elsevier Publishers 2008

Exercise Questions

1. List the important considerations when selecting a processor for embedded system design.
2. Categorize the different types of computing devices used to design embedded systems.
3. List the merits and de-merits of Von Neumann processor architecture.
4. Mention the key characteristics of RISC processors.
5. Identify the scenarios that creates a bottleneck for pipelined instruction execution.
6. Contrast superscalar and VLIW processor architectures with respect to compiler design.
7. Outline the reasons for using CISC architecture based processors for desktop computers.
8. Compare SoC processor and application specific integrated circuits.
9. Mention the reason for the widespread use of Dynamic RAMs for main memory in spite of being slower than Static RAMs.
10. Distinguish scratch pads and cache memory.
11. Recall the two types of interconnect architectures used in SoC processors.
12. Give examples of commercial embedded processors with RISC architectures.
13. Illustrate the key aspects of Von-Neumann and Harvard architectures used in the design of computers.
14. Explain with suitable examples, the process of instruction execution in CISC and RISC processors.
15. Illustrate the basic architecture of a System on Chip processor and summarize the importance of each functional unit.
16. Classify the types of on-chip memories used in SoC processors.
17. Demonstrate with suitable examples that a superscalar processor can improve the efficiency of instruction level of parallelism.
18. Examine the architecture of VLIW processor model and give your opinion on how it leads to lower hardware complexity compared to superscalar model.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – II – SECA3019- IO PERIPHERALS

[Memory - Interrupts - I/O Ports-Timers & Real Time Clock (RTC), Watch dog timer - CCP modules - Capture Mode - Compare Mode-PWM Mode - Serial communication module - USART - SPI interface - I2C interface, Analog Comparator, Analog interfacing and data acquisition]

2.1 Basics of ATmega328P

ATmega328P is one of the high performances AVR technology microcontroller with a large number of pins and features. It is a an 8-bit microcontroller based on RSIC architecture, which enhances its performance and power efficiency. Its power consumption is reduced by auto sleep mode and internal temperature sensor. This **ATmega328P** IC comes with internal protections and multiple programming methods, which helps the engineers to priorities this controller for different situations. The IC allows multiple modern era communications methods for other modules and microcontrollers itself, which is why the microcontroller ATmega328P usage has been increasing every day.

General Features

- High performance, low power AVR® 8-bit microcontroller
- Advanced RISC architecture
- 131 powerful instructions – most single clock cycle execution
- 32 × 8 general purpose working registers
- Fully static operation
- Up to 16MIPS throughput at 16MHz
- On-chip 2-cycle multiplier
- High endurance non-volatile memory segments
- 32K bytes of in-system self-programmable flash program memory
- 1Kbytes EEPROM
- 2Kbytes internal SRAM
- Write/erase cycles: 10,000 flash/100,000 EEPROM
- Optional boot code section with independent lock bits
- In-system programming by on-chip boot program
- True read-while-write operation
- Programming lock for software security

Peripheral features

- Two 8-bit Timer/Counters with separate prescaler and compare mode
- One 16-bit Timer/Counter with separate prescaler, compare mode, and capture mode
- Real time counter with separate oscillator
- Six PWM channels
- 8-channel 10-bit ADC in TQFP and QFN/MLF package
- Temperature measurement
- Programmable serial USART
- Master/slave SPI serial interface
- Byte-oriented 2-wire serial interface (Phillips I2 C compatible)
- Programmable watchdog timer with separate on-chip oscillator
- On-chip analog comparator

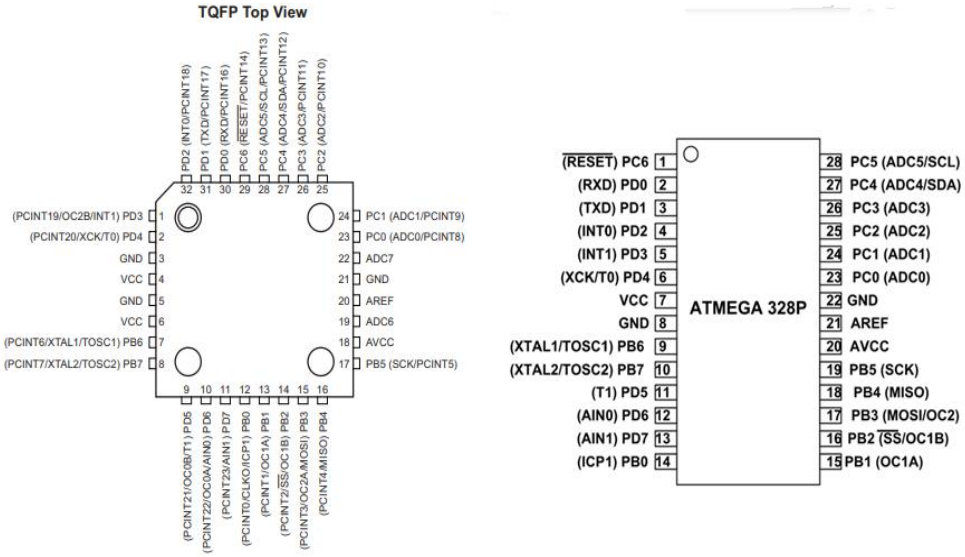


Figure 2.1 Pin details of ATMEGA328 in DIP and TQFP Package

2.2 Pin Description of ATMEGA328

ATMEGA328 comes in different packages as illustrated in figure 2.1 with 32pins and 28 pins. The functions of each pin of the controller is described in this section.

2.2.1 Digital Input/Output Pins

This microcontroller has three digital ports (B, C, D) such as PORTB, PORTC, and PORTD. All these pins can be used as digital input/output. On top of that, each port can be used for other purposes. To use them as output/input or for any other function it should be defined first otherwise there won't be any default function by all I/O pins.

2.2.2 Interrupt Pins

Most of the electrical functions required an interrupt system to operate like AC dimmer, etc. ATmega328P gives the support of 2 interrupts within the controller which can be used to get the attention of the CPU at any instant. Interrupt pins of ATmega328P are given below:

- IN0 – GPIO4
- IN1 – GPIO5

2.2.3 UART Communication in ATmega328P

Although there are multiple kinds of communication systems within the devices and modules but the most common one is USART. It is one of the simplest and easiest method for implement and understanding by most of the developers and systems. In this method, two wires used to send and receive the data. The USART pins of microcontroller ATmega328P are:

- RX – GPIO2
- TX – GPIO3
- XCK – GPIO6

The data can be sent by specified the sending rate within the controllers but it can also use the external clock pin to keep the data sync.

2.2.4 SPI Communication in ATmega328P

It one of the best serial communication systems in the case of multiple peripherals. SPI protocol allows multiple devices to use the same channel for communication. It consists of four wires, two for data sending and one for clock but the fourth wire is used to select the peripherals knows as a select slave. In the case of multiple peripherals number of the select slave, pins will be increased. The SPI pins of the microcontroller are:

- MOSI – GPIO17
- MISO – GPIO18
- SS – GPIO16
- SCK – GPIO19

2.2.5 I^2C Communication Module

Most of the peripherals come with the I^2C communication method which is one way at a specific time. I^2C protocol only uses one data wire and one clock wire. Data wire will transfer and receive the data and clock wire will send the clock pulse to keep the data sync. The wires on the microcontroller are:

- SDA – GPIO27
- SCL – GPIO28

2.2.6 Timers Modules

ATtiny328P has two internal timers. We can use these timers to make counters and to generate pulses. Both of these timers are dependent on an oscillator. Both timers can use the internal and external clock to operate, but they also have an internal pin which can be used to count according to the external pulses. All of these pins in microcontroller ATmega328P are given below:

- T0 – GPIO6
- T1 – GPIO11
- TOSC1 – GPIO9
- TOSC2 – GPIO10
- ICP1 – GPIO

ICP1 is an input capture pin which can be used to capture the external pulse at a specific interval of time. When an input pulse will occur on this pin then it will generate a timestamp which can tell when the external signal was received.

2.2.7 System Clock

The internal clock and external clock pulses can be divided by the Prescaler and their value can be received at an external pin. The external pin for divided clock pulses will be:

- CLKO – GPIO14

2.2.8 Comparator Module

The microcontroller has internal comparator modules for analog signal. This module takes the input in inverting and non-inverting form which can be used further for any internal purpose

or it can also be used to generate the output signals. Comparator pins of the microcontroller are listed below:

- AN0 (Positive) – GPIO12
- AN1 (Negative) – GPIO13

2.2.9 Capture/Compare/PWM Channels

There are six capture/compare/PWM pins are used to generate the desired time pulse-based signal. It uses a Prescaler to divide the time pulse. All of these pins in ATmega328P are:

- OC0B – GPIO11
- OC0A – GPIO12
- OC1A – GPIO15
- OC1B – GPIO16
- OC2A – GPIO17
- OC2B – GPIO5

2.2.10 Analog to Digital Converter Channels

In ATmega328P there are 6 ADC channels that can be used to convert the analog signal to digital. The analog converter needs to be activated first by its power pin (AVCC). The ADC channels use power supply voltage as a reference to differentiate the different levels of the analog signal. The analog pins of the controller are:

- ADC0 – GPIO23
- ADC1 – GPIO24
- ADC2 – GPIO25
- ADC3 – GPIO26
- ADC4 – GPIO27
- ADC5 – GPIO28
- AVCC – Pin20

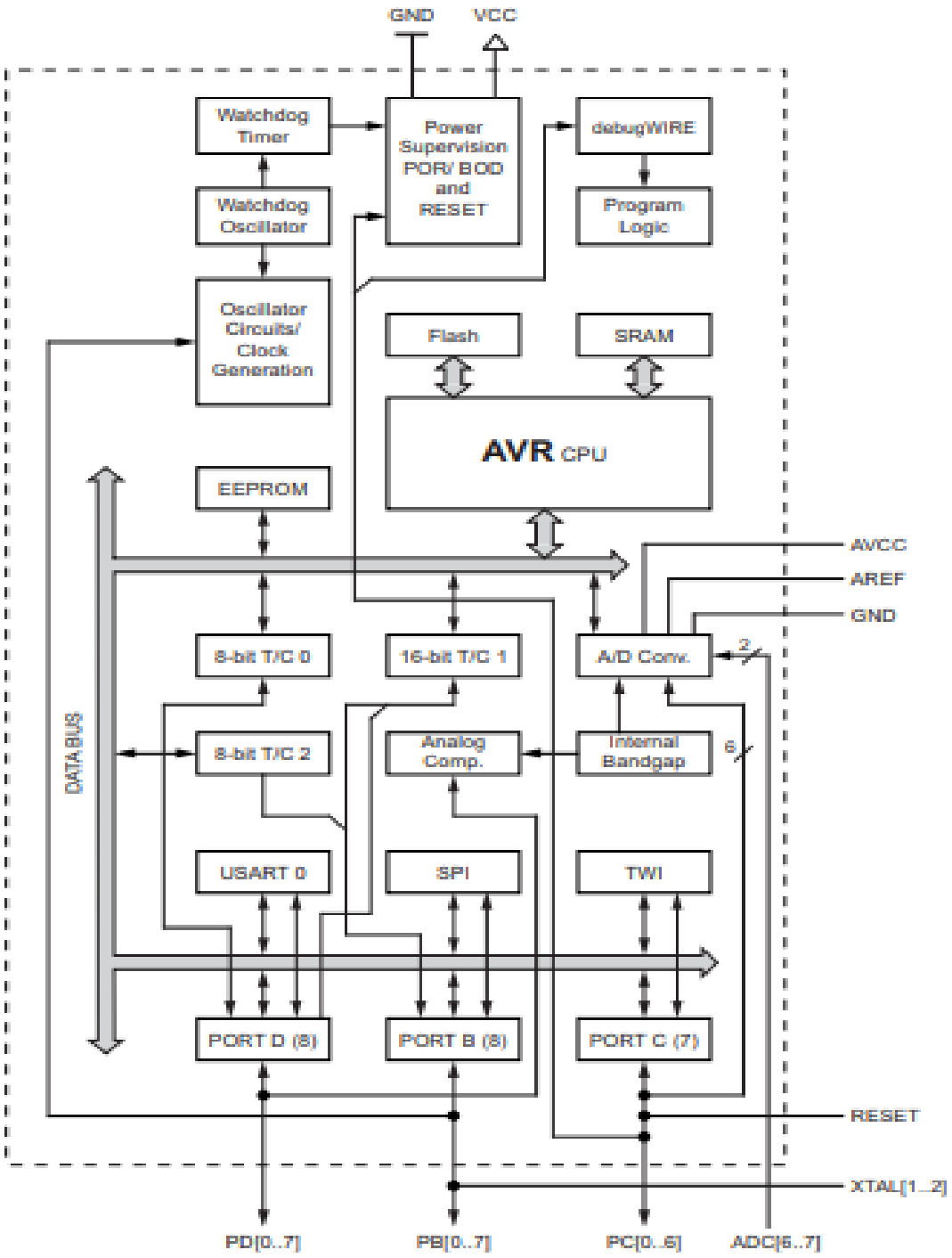


Figure 2.2: Block diagram of ATMEGA328 Microcontroller

2.3 ATMega328 I/O Register Configuration

In this section, the registers related to port configuration and input/output pin control of AVR/Atmel controllers is discussed.

2.3.1 GPIO Registers

The basic and important feature of any controllers is the number of gpio's available for connecting the peripherals. Atmega32 has 32-gpio's grouped into four 8-bit ports namely PORTA-PORTD as shown. Many I/O pins have 2-3 functions. If a pin is used for other function then it may not be used as a gpio. Though the gpio pins are grouped into 8-bit ports they can still be configured and accessed individually.

Each Port is associated with 3 registers for direction configuration(Input/Output), read and write operation as shown in Table 2.1.

Table 2.1 Registers for GPIO configuration

Register	**Description**
DDRx	Used to configure the respective PORT as output/input
PORTx	Used to write the data to the Port pins
PINx	Used to Read the data from the port pins

Note: Here 'x' could be A,B,C,D so on depending on the number of ports supported by the controller.

DDRx: Data Direction Register

Before reading or writing the data from the ports, their direction needs to be set. Unless the PORT is configured as output, the data from the registers will not go to controller pins.This register is used to configure the PORT pins as Input or Output. Writing 1's to DDRx will make the corresponding PORTx pins as output. Similarly writing 0's to DDRx will make the corresponding PORTx pins as Input.

1. DDRB = 0xff; *// Configure PORTB as Output.*
2. DDRC = 0x00; *// Configure PORTC as Input.*
3. DDRD = 0x0F; *// Configure lower nibble of PORTD as Output and higher nibble as Input*
4. DDRD = (1<<PD0) | (1<PD3) | (1<<PD6); *// Configure PD0,PD3,PD6 as Output and others as Input*

PORTx:

This register is used to send the data to port pins. Writing 1's to PORTx will make the corresponding PORTx pins as HIGH. Similarly writing 0's to PORTx will make the corresponding PORTx pins as LOW.

1. PORTB = 0xff; *// Make all PORTB pins HIGH.*
2. PORTC = 0x00; *// Make all PORTC pins LOW..*
3. PORTD = 0x0F; *// Make lower nibble of PORTD as HIGH and higher nibble as LOW*
4. PORTD = (1<<PD0) | (1<PD3) | (1<<PD6); *// Make PD0,PD3,PD6 HIGH,*

PINx: PORT Input Register

This register is used to read the data from the port pins. Before reading the data from the port pins, the ports needs to be configured as Inputs.

1. DDRB = 0x00; *// Configure the PORTB as Input.*
2. value = PINB; *// Read the data from PORTB.*
3. DDRB = 0x00; *// Configure PORTB as Input*
4. DDRD = 0xff; *// Configure PORTD as Output*
5. PORTD = PINB; *// Read the data from PORTB and send it to PORTD.*

2.3.1 Enabling Internal Pull Up Resistors

Making the DDRx bits to 0 will configure the PORTx as Input. Now the corresponding bits in PORTx register can be used to enable/disable pull-up resistors associated with that pin. To enable pull-up resistor, set bit in PORTx to 1, and to disable set it to 0.

1. DDRB = 0x00; *// Configure the PORTB as Input.*
2. PORTB = 0xFF; *// Enable the internal Pull Up resistor of PORTB.*
3. DDRD = 0xff; *// Configure PORTD as Output*
4. PORTD = PINB; *// Read the data from PORTB and send it to PORTD.*

2.4 Led Blinking Example

After knowing how to configure the GPIO ports, its time to write a simple program to blink the Leds. Below points needs to be considered for this example.

- Include the io.h file as it has the definitions for all the PORT registers.

- Include delay.h file to use the delay functions.
- Configure the PORT as Output before writing any data to PORT pins.

2.4.1 Program for GPIO Control in ATMEGA328 Controller

```
#include <avr/io.h>
#include <util/delay.h>
int main()
{
   DDRC = 0xff;          // Configure PORTC as output
   while(1)
   {
      PORTC = 0xff;       // Turn ON all the Leds connected to PORTC
      _delay_ms(100);     // Wait for some time
      PORTC = 0x00;       // Turn OFF all the Leds connected to PORTC
      _delay_ms(100);     // Wait for some time
   }
   return 0;
}
```

2.5 TIMER Registers in ATMEGA328

The ATmega328P is equipped with two 8-bit timer/counters and one 16-bit counter. These Timer/Counters let us do the following tasks.

- Turn on or turn off an external device at a programmed time.
- Generate a precision output signal (period, duty cycle, frequency). For example, generate a complex digital waveform with varying pulse width to control the speed of a DC motor
- Measure the characteristics (period, duty cycle, frequency) of an incoming digital signal
- Count external events

2.5.1 Timer Terminologies

Frequency : The number of times a particular event repeats within a 1-s period. The unit of frequency is Hertz, or cycles per second. For example, a sinusoidal signal with a 60-Hz frequency means that a full cycle of a sinusoid signal repeats itself 60 times each second, or every 16.67 ms. For the digital waveform shown in figure 2.2, the frequency is 2 Hz.

Period: The flip side of a frequency is a period. If an event occurs with a rate of 2 Hz, the period of that event is 500 ms. To find a period, given a frequency, or vice versa, we simply need to remember their inverse relationship, F = 1/T where F and T represent a frequency and the corresponding period, respectively.

Duty Cycle: In many applications, periodic pulses are used as control signals. A good example is the use of a periodic pulse to control a servo motor. To control the direction and sometimes the speed of a motor, a periodic pulse signal with a changing duty cycle over time is used.

Duty cycle is defined as the percentage of one period a signal is ON. The periodic pulse signal shown in the Figure is ON for 50% of the signal period and off for the rest of the period. Therefore, we call the signal in a periodic pulse signal with a 50% duty cycle. This special case is also called a square wave.

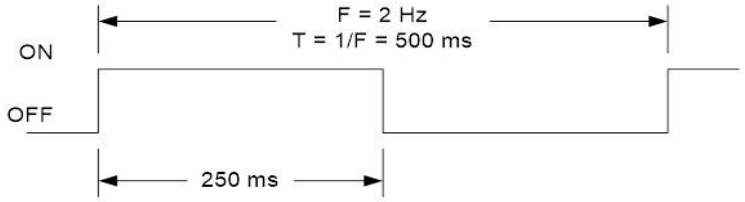


Figure 2.3: A 50% Duty Cycle square wave signal

2.5.2 Timer Modes

The simplest AVR Timer mode of operation is the Normal mode. Waveform Generation Mode for Timer/Counter 1 (WGM1) bits 3:0 = 0. These bits are located in Timer/Counter Control Registers A/B (TCCR1A and TCCR1B).

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	-	-	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 2.4: Timer/Counter Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 2.5: Timer/Counter Control Register B

- In this mode the Timer/Counter 1 Register (TCNT1H:TCNT1L) counts up (incrementing), and no counter clear is performed. The counter simply overruns when it passes its maximum 16-bit value 0xFFFF and then restarts 0x0000.
- There are no special cases to consider in the Normal mode, a new counter value can be written anytime.

Bit	7	6	5	4	3	2	1	0	
(0x85)	TCNT1[15:8]								TCNT1H
(0x84)	TCNT1[7:0]								TCNT1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 2.6: Timer/Counter 1 Register

- In normal operation the Timer/Counter Overflow Flag (TOV1) bit located in the Timer/Counter1 Interrupt Flag Register (T1FR1) will be set in the same timer clock cycle as the Timer/Counter 1 Register (TCNT1H:TCNT1L) becomes zero. The TOV1 Flag in this case behaves like a 17th bit, except that it is only set, not cleared.

Bit	7	6	5	4	3	2	1	0	
0x16 (0x36)	–	–	ICF1	–	–	OCF1B	OCF1A	TOV1	TIFR1
Read/Write	R	R	R/W	R	R	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 2.7: Timer/Counter 1 Interrupt Flag Register

2.5.3 Timer/Counter-1 Prescalar

The clock input to Timer/Counter 1 (TCNT1) can be pre-scaled (divided down) by 5 preset values (1, 8, 64, 256, and 1024).

Table 2.2: Timer Clock Frequency Selection Bits Configuration

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped).
0	0	1	$clk_{I/O}$/1 (No prescaling)
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

Clock Select Counter/Timer 1 (CS1) bits 2:0 are located in Timer/Counter Control Registers B [yellow].

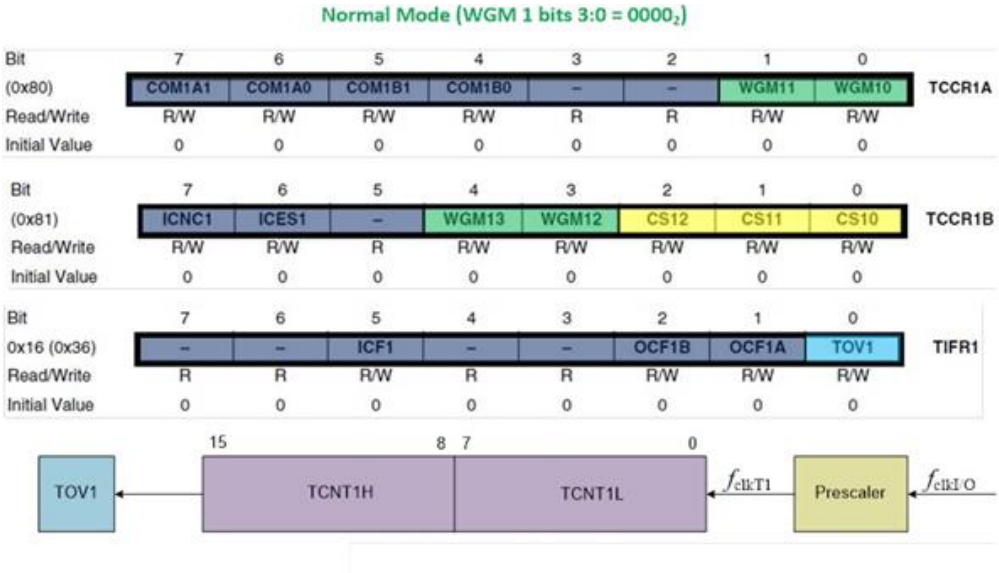


Figure 2.8: Timer-1 register configuration for normal mode

2.6 Timer programming Example

In this design example, we want to write a 250 msec delay routine assuming a system clock frequency of 16.000 MHz and a prescale divisor of 64. The first step is to discover if our 16-bit Timer/Counter 1 can generate a 250 ms delay as shown in figure 2.3.

Variable Definitions

- t_{clk_T1} : period of clock input to Timer/Counter1
- f_{clk} : AVR system clock frequency
- $f_{Tclk\_I/O}$: AVR Timer clock input frequency to Timer/Counter Waveform Generator

Calculating Maximum Delay (Normal Mode)

The largest time delay possible is achieved by setting both TCNT1H and TCNT1L to zero, which results in the overflow flag TOV1 flag being set after 2^{16} = 65,536 tics of the Timer/Counter1 clock.

$f_{T1} = f_{Tclk_{I/O}}/64$, given $f_{Tclk_{I/O}} = f_{clk}$ then $f_{T1} = 16.000MHz/64 = 250KHz$

and therefore $T_{1max} = 65536tics/250KHz = 262.14msec$

Clearly, Timer 1 can generate a delay of 250 msec. Our next step is to calculate the TCNT1 load value needed to generate a 250 ms delay.

Steps to Calculate Timer Load Value (Normal Mode)

1. Divide desired time delay by tclkT1 where

   tclkT1 = 64/fclkI/O = 64 / 16.000 MHz = 4 µsec/tic

   250msec / 4 µs/tic = 62,500 tics

2. Subtract 65,536 – step 1

   65,536 – 62,500 = 3,036

3. Convert step 2 to hexadecimal.

   3,036=0x0BDC

   For our example, TCNT1H = 0x0B and TCNT1L = 0xDC

4. Check Answer

   3,036ticsx4µs/tic=12.14msec

   262.14 msec – 250 msec = 12.14 msec √

Code Snippet for Timer Delay

```
void T1Delay()
{
```

```
  while (!(TIFR & (1<  TIFR = 1<


  TCNT1H = 0x0B;
  TCNT1L = 0xDC;
}
```

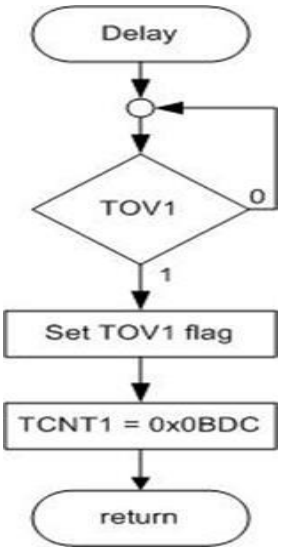


Figure 2.9: Workflow of timer based delay generation

2.7 Application Design with ATMEGA328

A microcontroller requires power supply, crystal and a power-on reset circuit for its functionality. Figure 2.10 shows the basic circuit design with ATMEGA328 controller with LEDs connected to port-C for flashing. A 16MHz crystal is used to provide clock for the Atmega32 microcontroller and 22pF capacitors are used to stabilize the operation of crystal. The 10µF capacitor and 10KΩ resistor is used to provide Power On Reset (POR) to the device. When the power is switched-ON, voltage across capacitor will be zero so the device resets (since reset is active low), then the capacitor charges to VCC and the reset will be disabled. 30th pin (AVCC) of Atmega32 should be connected to VCC if you are using PORTA, since it is the supply voltage pin for PORT A.

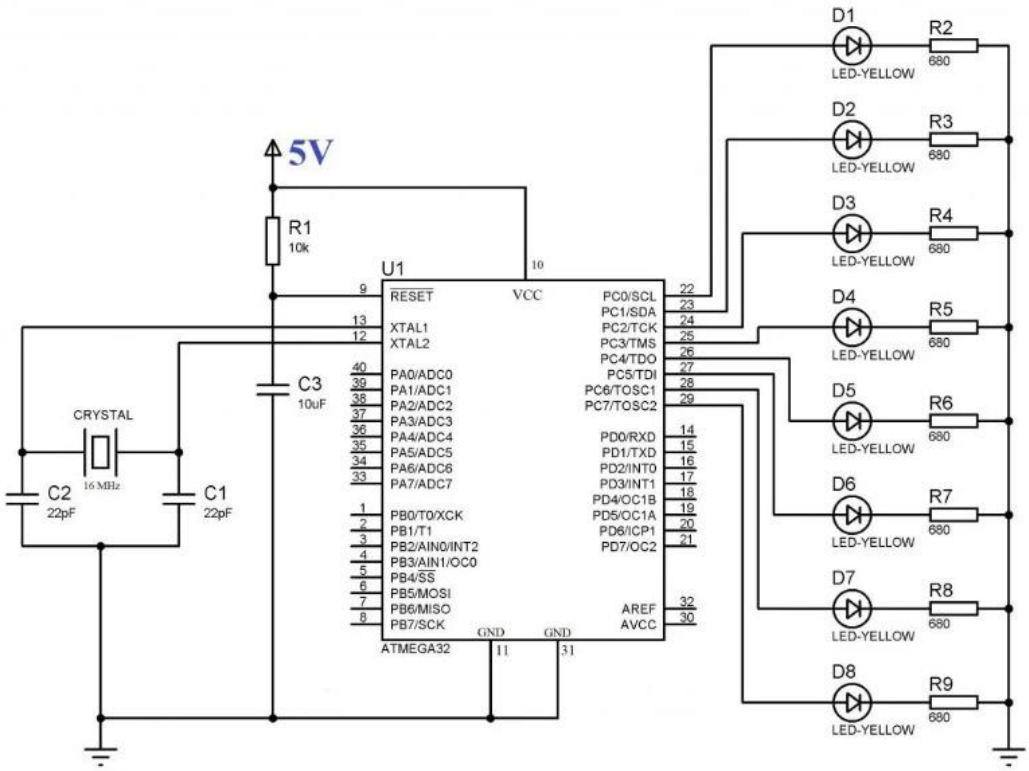


Figure 2.10: Circuit design with ATMEGA328 controller

2.7.1 Program for flashing LEDS

```
#ifndef F_CPU
#define F_CPU 16000000UL // 16 MHz clock speed
#endif

#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
 DDRC = 0xFF; //Nakes PORTC as Output
 while(1) //infinite loop
 {
  PORTC = 0xFF; //Turns ON All LEDs
  _delay_ms(1000); //1 second delay
  PORTC= 0x00; //Turns OFF All LEDs
  _delay_ms(1000); //1 second delay
 }
}
```

Description of the code

- DDRC = 0xFF makes all pins on PORTC as output pins
- PORTC = 0xFF makes all pins on PORTC Logic High (5V)
- PORTC = 0x00 makes all pins on PORTC Logic Low (0V)
- _delay_ms(1000) provides 1000 milliseconds delay.
- while(1) makes an infinite loop

You have seen that PORT registers are used to write data to ports. Similarly to read data from ports PIN registers are used. It stand for Port Input Register (eg : PIND, PINB). You may like to set or reset individual pins of PORT or DDR registers or to know the status of a specific bit of PIN register. There registers are not bit addressable, so we can't do it directly but we can do it through program. To make 3^{ed} bit (PC2) of DDRC register low we can use *DDRC &= ~(1<<PC2).* (1<<PC2) generates the binary number 00000100, which is complemented 11111011 and ANDed with DDRC register, which makes the 3^{ed} bit 0. Similarly *DDRC |= (1<<PC2)* can be used set the 3^{ed} bit (PC2) of DDRC register and to read 3^{ed} bit (PC2) we can use *PINC & (1<<PC2).* Similarly we can set or reset each bit of DDR or PORT registers and able to know the logic state of a particular bit of PIN register.

2.8 Arduino Development Board

The Arduino Uno is an open-source microcontroller board that is based on the Microchip ATmega328P (for Arduino UNO R3) or Microchip ATmega4809 (for Arduino UNO WIFI R2) micro-controller by Atmel and was the first USB powered board developed by Arduino. Atmega 328P based Arduino UNO pinout and specifications are given in figure below. Both Atmega328 and ATmega4809 have a built-in bootloader, which makes it very convenient to flash the board with our code. Like all Arduino boards, we can program the software running on the board using a language derived from C and C++. The easiest development environment is the Arduino IDE.

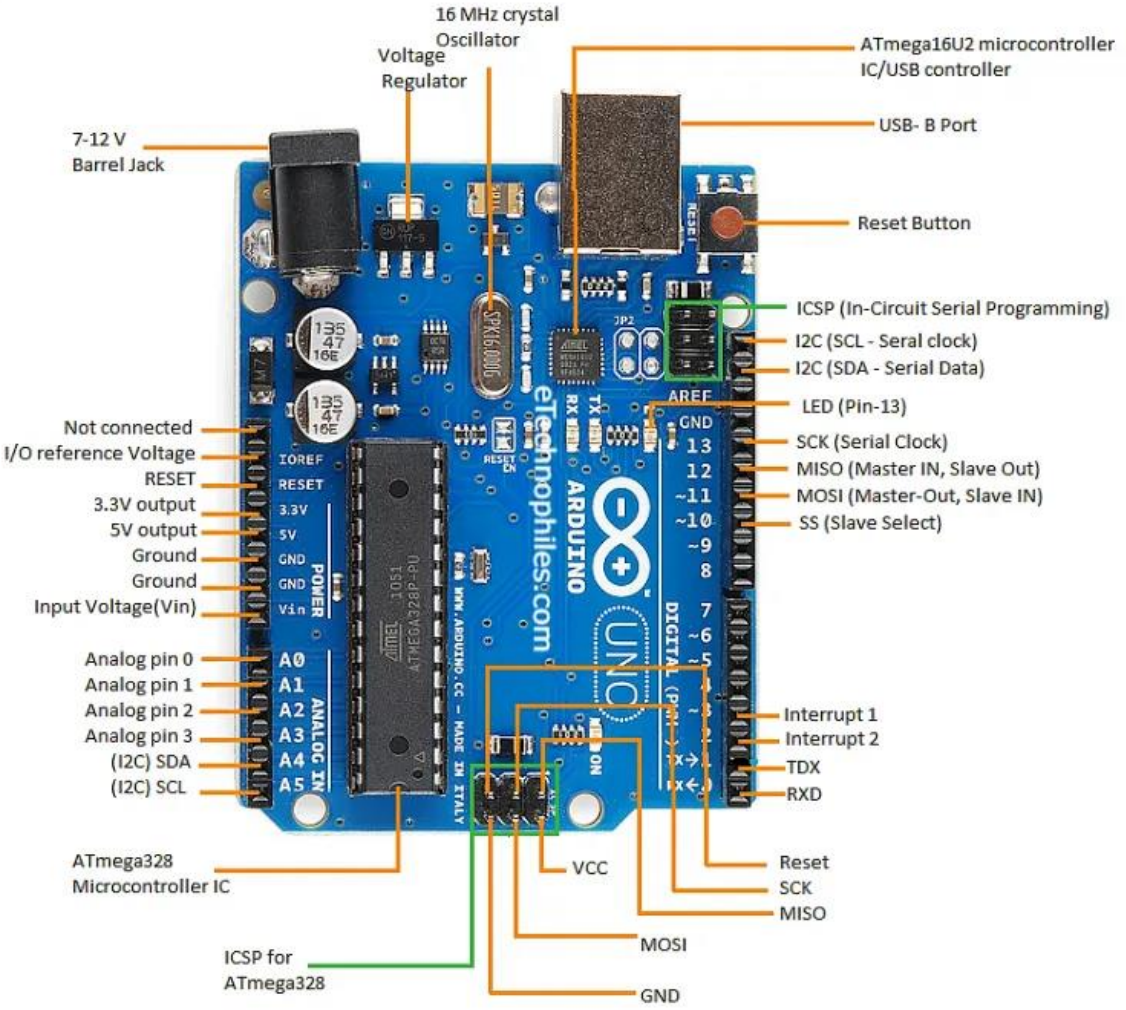


Figure 2.11: Pin details of Arduino Uno Board

Analog pins

The Arduino UNO board has six analog input pins A0 through A5. These pins can read the signal from an analog sensor like the humidity sensor or temperature sensor and convert it into a digital value that can be read by the microprocessor.

ICSP pin

Mostly, ICSP (12) is an AVR, a tiny programming header for the Arduino consisting of MOSI, MISO, SCK, RESET, VCC, and GND. It is often referred to as an SPI (Serial Peripheral Interface), which could be considered as an "expansion" of the output. Actually, you are slaving the output device to the master of the SPI bus.

Digital I/O

The Arduino UNO board has 14 digital I/O pins [numbered 0-13] of which 6 provide PWM (Pulse Width Modulation) output. These pins can be configured to work as input digital pins to read logic values (0 or 1) or as digital output pins to drive different modules like LEDs, relays, etc. The pins labeled "~" can be used to generate PWM.

AREF

AREF stands for Analog Reference. It is sometimes, used to set an external reference voltage (between 0 and 5 Volts) as the upper limit for the analog input pins.

2.9 Temperature measurement with LM35 and Arduino Uno

LM35 is a temperature sensor which can measure temperature in the range of -55°C to 150°C.It is a 3-terminal device that provides analog voltage proportional to the temperature. Higher the temperature, higher is the output voltage. The output analog voltage can be converted to digital form using ADC so that a microcontroller can process it.

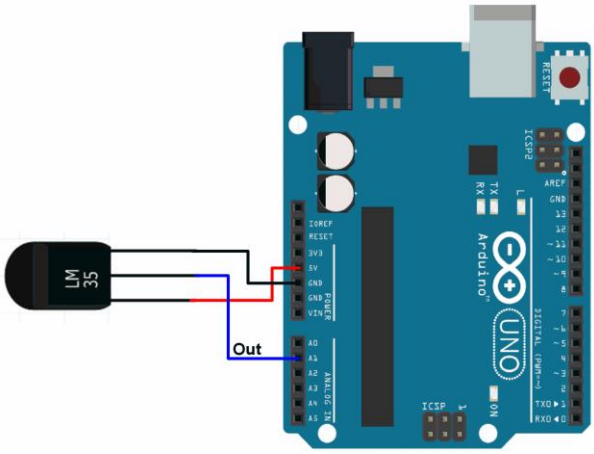


Figure 2.12: Connecting LM35 with Arduino Board

Program for measuring temperature using LM35 sensor

```
const int sensor=A1; // Assigning analog pin A1 to variable 'sensor'
float tempc;  //variable to store temperature in degree Celsius
```

```
float tempf;  //variable to store temperature in Fahreinheit
float vout;  //temporary variable to hold sensor reading
void setup()
{
pinMode(sensor,INPUT); // Configuring pin A1 as input
Serial.begin(9600);
}
void loop()
{
vout=analogRead(sensor);
vout=(vout*500)/1023;
tempc=vout; // Storing value in Degree Celsius
tempf=(vout*1.8)+32; // Converting to Fahrenheit
Serial.print("in DegreeC=");
Serial.print("\t");
Serial.print(tempc);
Serial.println();
Serial.print("in Fahrenheit=");
Serial.print("\t");
Serial.print(tempf);
Serial.println();
delay(1000); //Delay of 1 second for ease of viewing
}
```

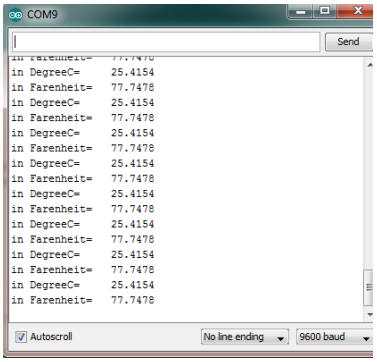


Figure 2.13: Serial monitor screenshot showing temperature

2.10 Humidity and Temperature measurement with DHT11 sensor

The DHTxx sensors have four pins, VCC, GND, data pin and a not connected pin which has no usage. A pull-up resistor from 5K to 10K Ohms is required to keep the data line high and in order to enable the communication between the sensor and the Arduino Board. There are some versions of these sensors that come with a breakout boards with built-in pull-up resistor and they have just 3 pins. The DHTXX sensors are digital sensors and have their own single wire protocol used for transferring the data suing single line(data pin). This protocol requires precise timing and the timing diagrams for getting the data from the sensors can be found from the datasheets of the sensors.

The DHT22 is the more expensive version which obviously has better specifications. Its temperature measuring range is from -40 to +125 degrees Celsius with +-0.5 degrees accuracy, while the DHT11 temperature range is from 0 to 50 degrees Celsius with +-2 degrees accuracy. Also the DHT22 sensor has better humidity measuring range, from 0 to 100% with 2-5% accuracy, while the DHT11 humidity range is from 20 to 80% with 5% accuracy.

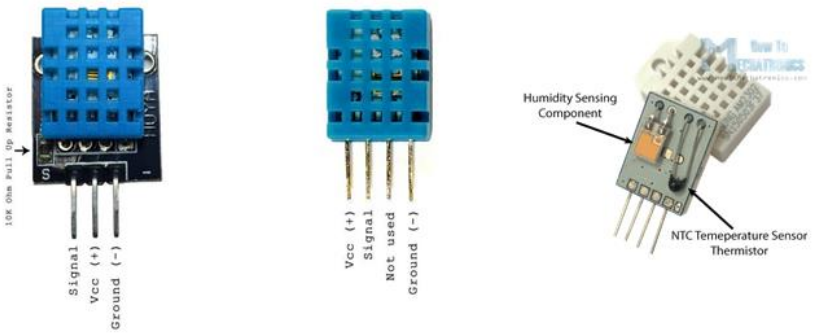


Figure 2.13 DHT11/12 sensor- pinout and internal view

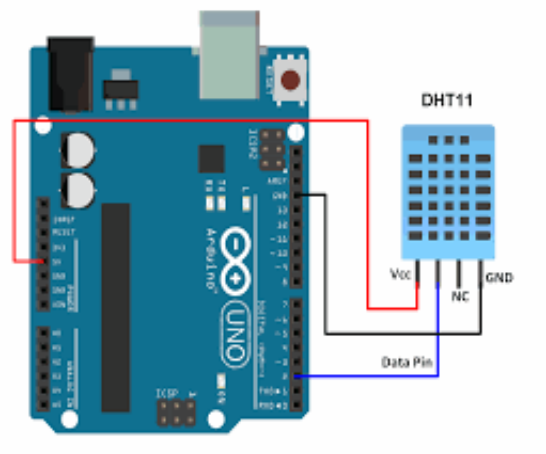


Figure 2.14: Connecting DHTxx sensor with Arduino board

2.10.1 Programming for DHTxx sensors

First we need to included the DHT library which can be found from the Arduino official website, then define the pin number to which our sensor is connected and create a DHT object. In the setup section, we need to initiate the serial communication because we will use the serial monitor to print the results. Using the read22() function we will read the data from the sensor and put the values of the temperature and the humidity into the t and h variables. If you use the DHT11 sensor you will need to you the read11() function. At the end, we will print the temperature and the humidity values on the serial monitor.

Program for reading data from DHTxx sensors with Arduino board

```
/* DHT11/ DHT22 Temperature and Humidity Sensor
#include <dht.h>
#define dataPin 2 // Defines pin number to which the sensor is connected
dht DHT; // Creats a DHT object
void setup() {
Serial.begin(9600);
}
void loop() {
int readData = DHT.read22(dataPin); // Reads the data from the DHT22 sensor
float  t = DHT.temperature; // Gets the values of the temperature
float  h = DHT.humidity; // Gets the values of the humidity
// Printing the results on the serial monitor
```

```
Serial.print("Temperature = ");
Serial.print(t);
Serial.print(" *C ");
Serial.print(" Humidity = ");
Serial.print(h);
Serial.println(" % ");
delay(2000); // Delays 2 seconds, as the DHT22 sampling rate is 0.5Hz
}
```

Note: Install DHT11/12 library in Arduino IDE, before executing the code

2.11 Serial Communication Protocols

This section compares **UART vs SPI vs I2C** interfaces and mentions **difference between UART, SPI and I2C** in tabular format. It provides comparison between these interfaces based on various factors which include interface diagram, pin designations, data rate, distance, communication type, clock, hardware and software complexity, advantages, disadvanatages etc.

2.12 UART Interface

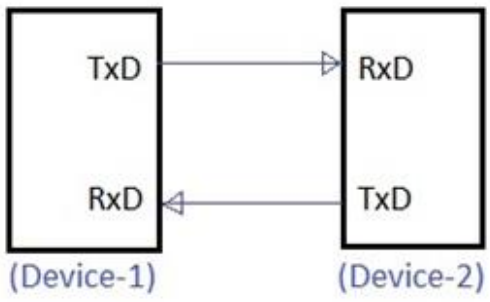


Figure 2.15: USART Interface

Features of UART interface

- **UART** supports lower data rate.
- Receiver need to know baudrate of the transmitter before initiation of reception i.e. before communication to be established.
- UART is simple protocol, it uses start bit (before data word), stop bits (one or two, after dataword), parity bit (even or odd) in its base format for data formatting. Parity bit helps in one bit error detection.
- UART Packet = 1 start bit(low level), 8 data bits including parity bit, 1 or 2 stop bit(high level).

- Data is transmitted byte by byte.
- UART generates clock internally and synchronizes it with data stream with the help of transition of start bit.
- It is also referred as RS232 .
- For long distance communication, 5V UART is converted to higher voltages viz. +12V for logic 0 and -12V for logic 1.
- Figure 2.15 depicts UART interface between two devices.

2.13 SPI Interface

SPI stands for Serial Peripheral Interface and has four lines for communication namely MOSI, MISO,SCLK and slave select (SS). The functions of the four lines are outlined below.

- MOSI - Master Output Slave Input, it is used to transfer data from master device to slave device.
- MISO - Master Input Slave Output, it is used to transfer data from slave device to master device.
- SCLK - Serial Clock, it is clock output from master and used for synchronization.
- SS - Slave Select, it is used by master device to select one slave out of multiple slaves. It inserts active low signal to select the particular slave device.

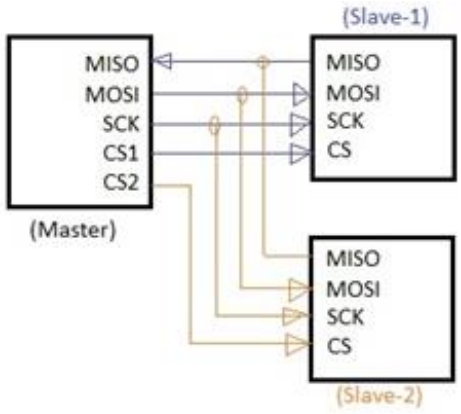


Figure 2.16: SPI Interface

As shown in the figure 2.16, one slave is connected with one master device. Clock is generated by master device for synchronization of data transfer. It is also possible to connected more than one slave device with single master for communication. SPI interface operates in either

half or full duplex mode. SPI is the short form of Serial Peripheral interface. The figure-2 depicts SPI interface between master and slave devices.

2.14 I2C Interface

I^2C stands for "inter-IC bus". It is also used as I2C for simplicity. I2C is a low speed and two wire serial data connection bus used in IC (Integrated Circuit). It is used to run signals between ICs mounted on the same PCB (Printed Circuit Board). The figure 2.17 depicts I2C interface between master and slave devices.

- It uses only two lines between multiple masters and multiple slaves viz. SDA (Serial Data) and SCL (Serial Clock).
- I2C supports various data rates as per versions from 100 Kbps, 400 Kbps, 1 Mbps to 3.4 Mbps
- It is synchronous communication like SPI and unlike UART. Hence there is common clock signal between masters and slaves.
- It uses start and stop bits and ACK bit for every 8 bits of data transfer.

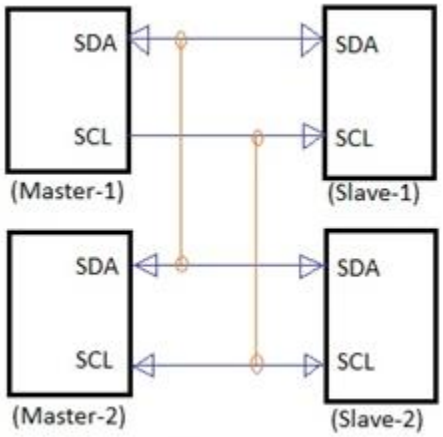


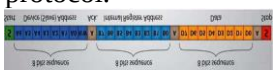
Figure 2.17: I2C Interface

2.15 Difference between UART, SPI and I2C

Let us compare UART vs SPI vs I2C and summarize difference between UART, SPI and I2C.

Table 2.2 Comparison between UART, SPI and I2C.

Features	UART	SPI	I2C
Full Form	Universal Asynchronous Receiver/Transmitter	Serial Peripheral Interface	Inter-Integrated Circuit
Pin names	TxD: Transmit Data RxD: Receive Data	SCLK: Serial Clock MOSI: Master Output, Slave Input MISO: Master Input, Slave Output SS: Slave Select	SDA: Serial Data SCL: Serial Clock
Data rate	As this is is asynchronous communication, data rate between two devices wanting to communicate should be set to equal value. Maximum data rate supported is about 230 Kbps to 460kbps.	Maximum data rate limit is not specified in SPI interface. Usually supports about 10 Mbps to 20 Mbps	I2C supports 100 kbps, 400 kbps, 3.4 Mbps. Some variants also supports 10 Kbps and 1 Mbps.
Distance	Lower about 50 feet	highest	Higher
Type	Asynchronous	Synchronous	Synchronous
Number of masters	One to one Communication only	One	One or more than One

Clock	No Common Clock signal is used. Both the devices will use there independent clocks.	There is one common serial clock signal between master and slave devices.	There is common clock signal between multiple masters and multiple slaves.
Hardware complexity	lesser	less	more
Protocol	For 8 bits of data one start bit and one stop bit is used.	Each company or manufacturers have got their own specific protocols to communicate with peripherals. Hence one needs to read datasheet to know read/write protocol for SPI communication to be established. For example we would like SPI communication between microcontroller and EPROM. Here one need to go through read/write operational diagram in the EPROM data sheet.	It uses start and stop bits. It uses ACK bit for each 8 bits of data which indicates whether data has been received or not. Figure depicts the data communication protocol. 
Software addressing	As this is one to one connection between two devices, addressing is not needed.	Slave select lines are used to address any particular slave connected with the master. There will be 'n' slave select lines on master device for 'n' slaves.	There will be multiple slaves and multiple masters and all masters can communicate with all the slaves. Upto 27 slave devices can be

			connected/addressed in the I2C interface circuit.
Merits	• It is simple communication and most popular which is available due to UART support in almost all the devices with 9 pin connector. It is also referred as RS232 interface.	•It is simple protocol and hence so not require processing overheads. •Supports full duplex communication. •Due to separate use of CS lines, same kind of multiple chips can be used in the circuit design. •SPI uses push-pull and hence higher data rates and longer ranges are possible. •SPI uses less power compare to I2C	•Due to open collector design, limited slew rates can be achieved. •More than one masters can be used in the electronic circuit design. •Needs fewer i.e. only 2 wires for communication. •I2C addressing is simple which does not require any CS lines used in SPI and it is easy to add extra devices on the bus. •It uses open collector bus concept. Hence there is bus voltage flexibity on the interface bus. •Uses flow control.
De-merits	• They are suitable for communication between only two devices. • It supports fixed data rate	• As number of slave increases, number of CS lines increases, this results in hardware complexity as	•Increases complexity of the circuit when number of slaves and masters increases. •I2C interface is half

	agreed upon between devices initially before communication otherwise data will be garbled.	number of pins required will increase. • To add a device in SPI requires one to add extra CS line and changes in software for particular device addressing is concerned. •Master and slave relationship cannot be changed as usually done in I2C interface. •No flow control available in SPI.	duplex. •Requires software stack to control the protocol and hence it needs some processing overheads on microcontroller/micro processor.

2.11 Arduino Portable Weather Station Design

A weather station is a system that measures atmospheric parameters such as temperature, pressure, humidity, gas content in air, etc. Here is a design example that measures temperature, pressure and humidity and display the values in a graphical display device along with time stamp.

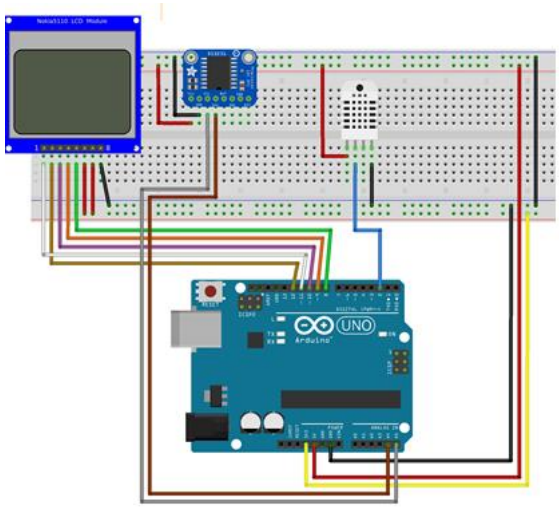


Figure 2.18: Connecting RTC,GLCD and sensors with Arduino board

TEXT / REFERENCE BOOKS

[1] Thomas Grace, " Programming & Interfacing Atmel AVR Microcontrollers" , 1st Edition,Cengage Learning , 2015

[2] Francis Perea, Arduino Essentials, 1st Edition, Packet Publishers, 2015

[3] Lyla B. Das, “Architecture, Programming and Interfacing of Low-power Processors ARM 7, Cortex-M”, Cengage, 1st Edition, 2017.

[4] Joseph Yiu, “The Definitive Guide to the ARM Cortex-M3”, Newness, 2nd Edition, 2009

Exercise Questions

1. List the registers associated with I/O pin configuration in ATMEGA328p micron roller.
2. Compare and contrast compiler and cross compiler
3. Identify two real time applications that require watch dog timers.
4. Calculate the resolution of a 10-bit A/D converter, if Vref pin is kept at 5V and 2V.
5. Identify the communication protocol used in real-time clock module DS1307.
6. A microcontroller operates at 5V DC and uses PWM to control the speed of a DC meter. Determine the required duty cycle of the PWM signal to provide an average voltage of 1.25V to the DC motor.
7. Identify a suitable communication protocol for an embedded system application that has to collect data 5 different sensors using minimum number of I/O pins.
8. It is required to introduce delay in tens of seconds in a system. Which timer will you choose for this use case in ATMEGA328?
9. Identify the right choice of embedded processor and its word size for designing a i) smart lighting system for home and ii) face recognition based authentication system.
10. Design a circuit with ATMEGA328 controller, write a C program to measure temperature using LM35 sensor connected to portB and serially transmit the atmospheric temperature value in Centigrade at 9600 baudrate.
11. Discuss about the registers involved in IO interfacing in ATMEGA328 microcontroller with examples.
12. Design a circuit with ATMEGA328 microcontroller and develop a C program to control the speed and direction of two DC motors connected to portC.
13. Design a real-time digital clock with a LCD module that displays current time and date in the following format:
    i) DD/MM/YY in first row of the LCD
14. ii) HH/MM/SS in second row of the LCD
15. Develop the system level model of a real-time data acquisition system that records atmospheric temperature, humidity, pressure with time stamp and stores the data in a SD card.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – V SECA3019 – ARM CORTEX PROCESSORS

[Introduction to ARM CORTEX series, improvement over classical series and advantages for embedded system design. CORTEX-A, CORTEX-M, CORTEX-R processors series, versions, features and applications, need of operating system in developing complex applications in embedded system, Firmware development for ARM Cortex, Survey of CORTEX-M3 based controllers, its features and comparison]

3.1 Introduction to ARM processors

ARM was formed in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer, Acorn Computer Group, and VLSI Technology. In 1991, ARM introduced the ARM6 processor family, and VLSI became the initial licensee. Subsequently, additional companies, including Texas Instruments, NEC, Sharp and ST Microelectronics, licensed the ARM processor designs, extending the applications of ARM processors into mobile phones, computer hard disks, personal digital assistants (PDAs), home entertainment systems, and many other consumer products. The ARM microcontroller architecture come with a few different versions such as ARMv1, ARMv2 etc and each one has its own advantage and disadvantages.

3.2 ARM Architecture Versions

The evolution of features and enhancements to the processors over time has led to successive versions of the ARM architecture. Note that architecture version numbers are independent from processor names. For example, the ARM7TDMI processor is based on the ARMv4T architecture (the T is for Thumb® instruction mode support).

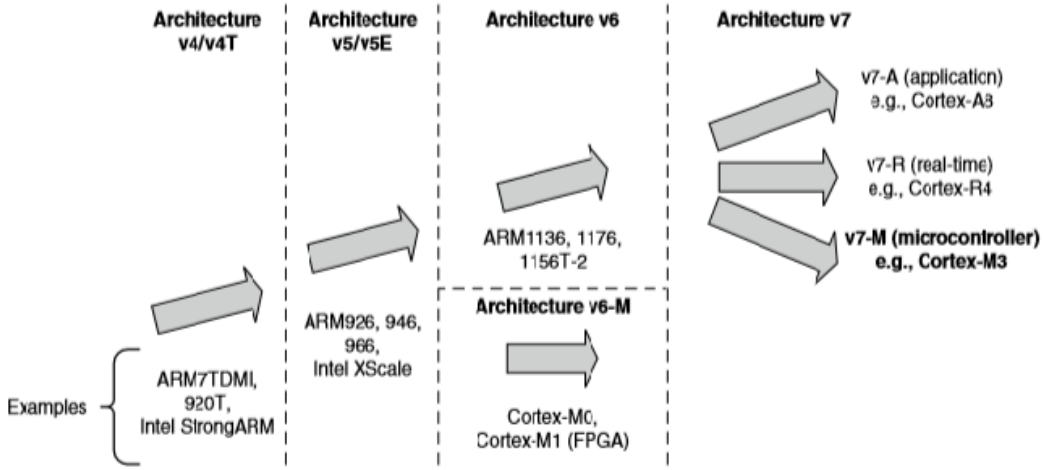


Figure 3.1 : The Evolution of ARM Processor Architecture.

The ARMv5E architecture was introduced with the ARM9E processor families, including the ARM926E-S and ARM946E-S processors. This architecture added "Enhanced" Digital Signal Processing (DSP) instructions for multimedia applications. With the arrival of the ARM11 processor family, the architecture was extended to the ARMv6. New features in this architecture included memory system features and Single Instruction–Multiple Data (SIMD) instructions. Processors based on the ARMv6

architecture include the ARM1136J(F)-S, the ARM1156T2(F)-S, and the ARM1176JZ(F)-S

Over the past several years, ARM extended its product portfolio by diversifying its CPU development, which resulted in the architecture version 7 or v7. In this version, the architecture design is divided into three profiles:

- **A-profile** is designed for high-performance open application platforms.
- **R-profile** is designed for high-end embedded systems in which real-time performance is needed.
- **M-profile** is designed for deeply embedded microcontroller-type systems.

The Cortex processor families are the first products developed on architecture v7, and the Cortex-M3 processor is based on one profile of the v7 architecture, called ARM v7-M, an architecture specification for microcontroller products.

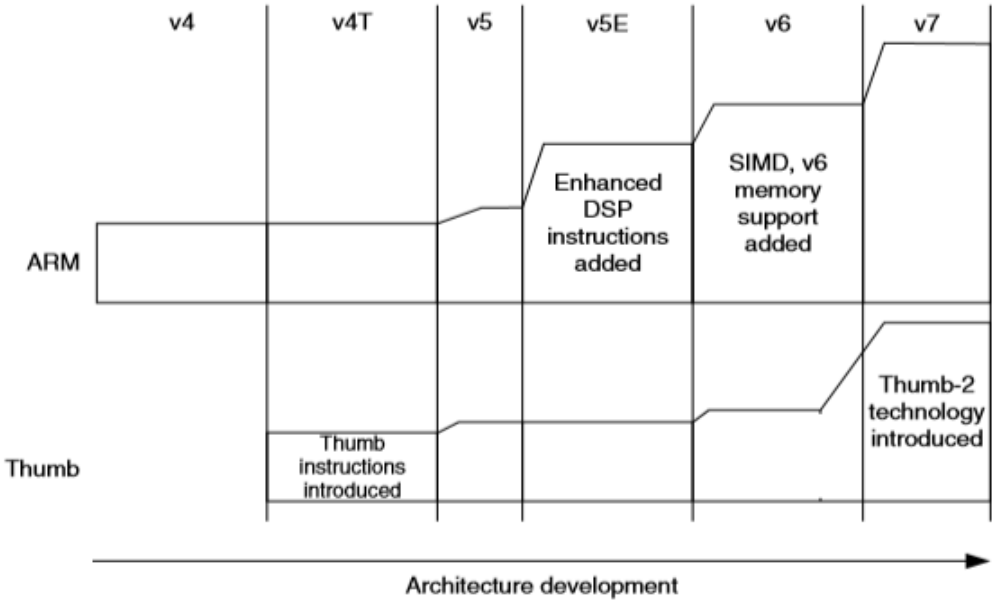


Figure 3.2: Instruction set Enhancement in ARM architectures.

Historically (since ARM7TDMI), two different instruction sets are supported on the ARM processor: the ARM instructions that are 32 bits and Thumb instructions that are 16 bits. During program execution, the processor can be dynamically switched between the ARM state and the Thumb state to use either one of the instruction sets. The Thumb instruction set provides only a subset of the ARM instructions, but it can provide higher code density. It is useful for products with tight memory requirements.

In 2003, ARM announced the Thumb-2 instruction set, which is a new superset

of Thumb instructions that contains both 16-bit and 32-bit instructions. The extended instruction set in Thumb-2 is a superset of the previous 16-bit Thumb instruction set, with additional 16-bit instructions alongside 32-bit instructions. It allows more complex operations to be carried out in the Thumb state, thus allowing higher efficiency by reducing the number of states switching between ARM state and Thumb state. Focused on small memory system devices such as microcontrollers and reducing the size of the processor, the Cortex-M3 supports only the Thumb-2 (and traditional Thumb) instruction set. Instead of using ARM instructions for some operations, as in traditional ARM processors, it uses the Thumb-2 instruction set for all operations. As a result, the Cortex-M3 processor is not backward compatible with traditional ARM processors. That is, you cannot run a binary image for ARM7 processors on the Cortex-M3 processor. Nevertheless, the Cortex-M3 processor can execute almost all the 16-bit Thumb instructions, including all 16-bit Thumb instructions supported on ARM7 family processors, making application porting easy.

With support for both 16-bit and 32-bit instructions in the Thumb-2 instruction set, there is no need to switch the processor between Thumb state (16-bit instructions) and ARM state (32-bit instructions). For example, in ARM7 or ARM9 family processors, you might need to switch to ARM state if you want to carry out complex calculations or a large number of conditional operations and good performance is needed, whereas in the Cortex-M3 processor, you can mix 32-bit instructions with 16-bit instructions without switching state, getting high code density and high performance with no extra complexity. The Thumb-2 instruction set is a very important feature of the ARMv7 architecture. Compared with the instructions supported on ARM7 family processors (ARMv4T architecture), the Cortex-M3 processor instruction set has a large number of new features. For the first time, hardware divide instruction is available on an ARM processor, and a number of multiply instructions are also available on the Cortex-M3 processor to improve data-crunching performance. The Cortex-M3 processor also supports unaligned data accesses, a feature previously available only in high-end processors.

3.3 ARM Architecture

The ARM architecture processor is an advanced reduced instruction set computing [RISC] machine and it's a 32bit reduced instruction set computer (RISC) microcontroller. It was introduced by the Acron computer organization in 1987. This ARM is a family of microcontroller developed by makers like ST Microelectronics,Motorola, and so on. The ARM architecture

comes with totally different versions like ARMv1, ARMv2, etc., and, each one has its own advantage and disadvantages.

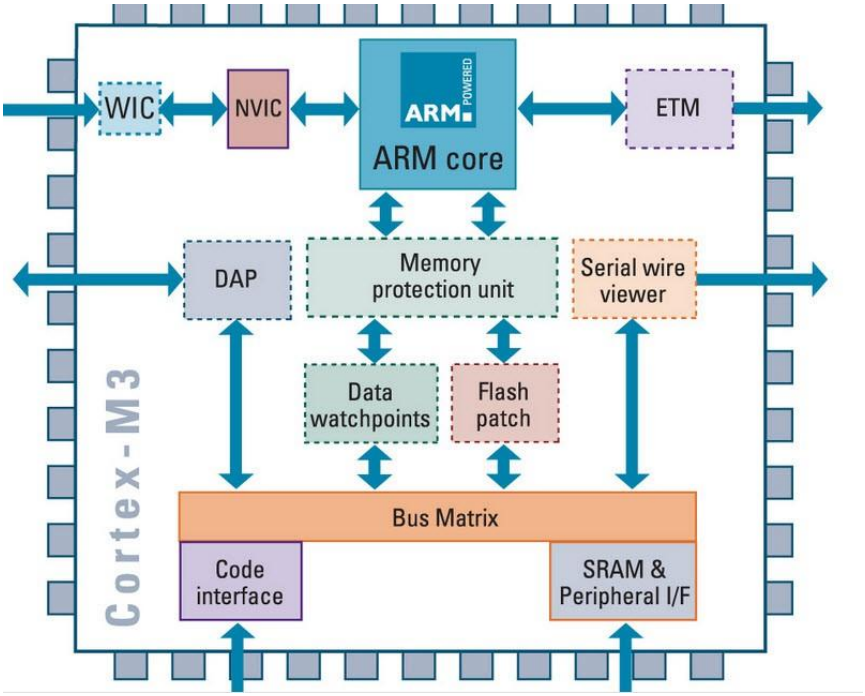


Figure 3.3 ARM Architecture

The ARM Architecture consists of

- Arithmetic Logic Unit
- Booth multiplier
- Barrel shifter
- Control unit
- Register file

The ARM processor conjointly has other components like the Program status register, which contains the processor flags (Z, S, V and C). The modes bits conjointly exist within the program standing register, in addition to the interrupt and quick interrupt disable bits; Some special registers: Some registers are used like the instruction, memory data read and write registers and memory address register.

Priority encoder: The encoder is used in the multiple load and store instruction to point which register within the register file to be loaded or kept .

Multiplexers: Several multiplexers are accustomed to the management operation of the processor buses. Because of the restricted project time, we tend to implement these components in a very behavioral model. Each component is described with an entity. Every entity has its own

architecture, which can be optimized for certain necessities depending on its application. This creates the design easier to construct and maintain.

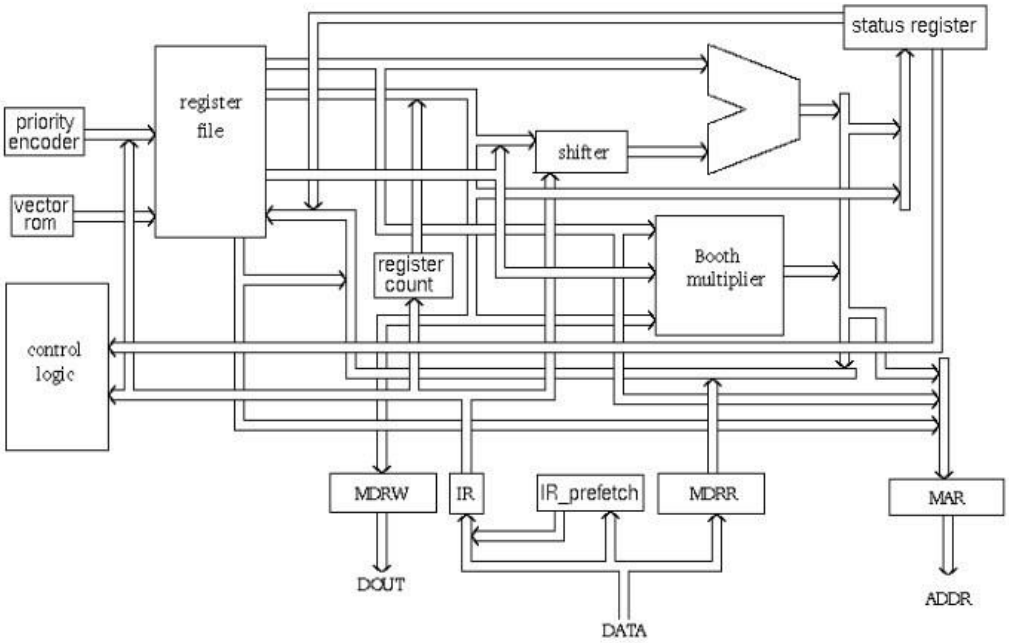


Figure 3.4 ARM Core Block Diagram

Arithmetic Logic Unit (ALU)

The ALU has two 32-bits inputs. The primary comes from the register file, whereas the other comes from the shifter. Status registers flags modified by the ALU outputs. The V-bit output goes to the V flag as well as the Count goes to the C flag. Whereas the foremost significant bit really represents the S flag, the ALU output operation is done by NORed to get the Z flag. The ALU has a 4-bit function bus that permits up to 16 opcode to be implemented.

Booth Multiplier Factor

The multiplier factor has 3 32-bit inputs and the inputs return from the register file. The multiplier output is barely 32-Least Significant Bits of the merchandise. The entity representation of the multiplier factor is shown in the above block diagram. The multiplication starts whenever the beginning 04 input goes active. Fin of the output goes high when finishing.

Booth Algorithm

Booth algorithm is a noteworthy multiplication algorithmic rule for 2's complement numbers. This treats positive and negative numbers uniformly. Moreover, the runs of 0's or 1's

within the multiplier factor are skipped over without any addition or subtraction being performed, thereby creating possible quicker multiplication. The figure shows the simulation results for the multiplier test bench. It's clear that the multiplication finishes only in16 clock cycle.

Barrel Shifter

The barrel shifter features a 32-bit input to be shifted. This input is coming back from the register file or it might be immediate data. The shifter has different control inputs coming back from the instruction register. The Shift field within the instruction controls the operation of the barrel shifter. This field indicates the kind of shift to be performed (logical left or right, arithmetic right or rotate right). The quantity by which the register ought to be shifted is contained in an immediate field within the instruction or it might be the lower 6 bits of a register within the register file.

The shift_val input bus is 6-bits, permitting up to 32 bit shift. The shift type indicates the needed shift sort of 00, 01, 10, 11 are corresponding to shift left, shift right, an arithmetic shift right and rotate right, respectively. The barrel shifter is especially created with multiplexers.

Control Unit

For any microprocessor, control unit is the heart of the whole process and it is responsible for the system operation,so the control unit design is the most important part within the whole design. The control unit is sometimes a pure combinational circuit design. Here, the control unit is implemented by easy state machine. The processor timing is additionally included within the control unit. Signals from the control unit are connected to each component within the processor to supervise its operation.

3.4 ARM Core Register and Modes

An ARM micrcontroller is a load store reducing instruction set computer architecture means the core cannot directly operate with the memory. The data operations must be done by the registers and the information is stored in the memory by an address. The ARM cortex-M3 consists of 37 register sets wherein 31 are general purpose registers and 6 are status registers. The ARM uses seven processing modes to run the user task.

- USER Mode
- FIQ Mode
- IRQ Mode
- SVC Mode

- UNDEFINED Mode
- ABORT Mode
- Monitor Mode

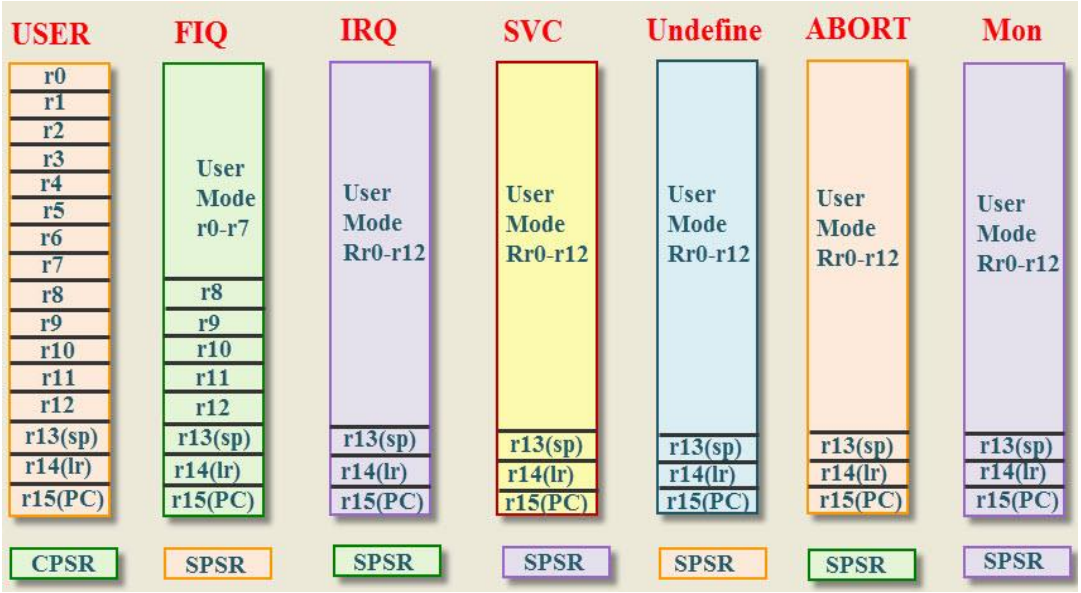


Figure 3.5 ARM Processor Register Modes

- **USER Mode:** The user mode is a normal mode, which has the least number of registers. It doesn't have SPSR and has limited access to the CPSR.
- **FIQ and IRQ:** The FIQ and IRQ are the two interrupt caused modes of the CPU. The FIQ is processing interrupt and IRQ is standard interrupt. The FIQ mode has additional five banked registers to provide more flexibility and high performance when critical interrupts are handled.
- **SVC Mode:** The Supervisor mode is the software interrupt mode of the processor to start up or reset.
- **Undefined Mode:** The Undefined mode traps when illegal instructions are executed. The ARM core consists of 32-bit data bus and faster data flow.
- **THUMB Mode:** In THUMB mode 32-bit data is divided into 16-bits and increases the processing speed.
- **THUMB-2 Mode:** In THUMB-2 mode the instructions can be either 16-bit or 32-bit and it increases the performance of the ARM cortex –M3 microcontroller. The ARM cortex-m3 microcontroller uses only THUMB-2 instructions.

Some of the registers are reserved in each mode for the specific use of the core. The reserved registers are

- Stack Pointer (SP).

- Link Register (LR).
- Program Counter (PC).
- Current Program Status Register (CPSR).
- Saved Program Status Register (SPSR).

The reserved registers are used for specific functions. The SPSR and CPSR contain the status control bits which are used to store the temporary data. The SPSR and CPSR register have some properties that are defined operating modes, Interrupt enable or disable flags and ALU status flag. The ARM core operates in two states 32-bit state or THUMBS state.

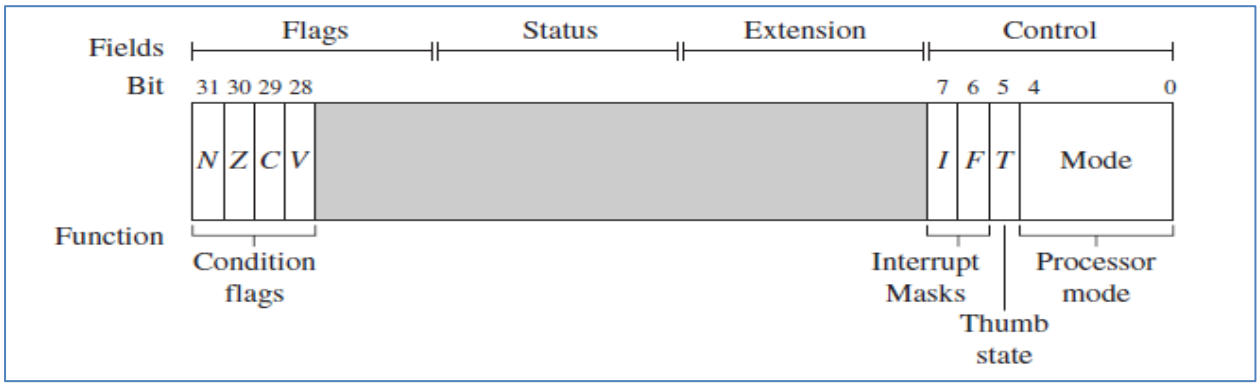


Figure 3.6 : A generic program status register (psr).

The ARM core uses the ***cpsr*** to monitor and control internal operations. The ***cpsr*** is a dedicated 32-bit register and resides in the register file. The ***cpsr*** is divided into **four fields, each 8 bits wide**: **flags**, status, extension, and control. In current designs the extension and status fields are reserved for future use. The control field contains the processor mode, state, and interrupt mask bits. The flags field contains the condition flags.

- First 5 bits is for mode selection
- The processor mode determines which registers are active and the access rights to the ***cpsr*** register itself.
- Each processor mode is either **privileged** or **nonprivileged:** A **privileged** mode allows full read-write access to the **cpsr**. Conversely, a **nonprivileged** mode only allows read access to the control field in the *cpsr* but still allows read-write access to the condition flags.
- There are seven processor modes in total: **six privileged modes** (**abort**, **fast interrupt request**, **interrupt request**, **supervisor**, **system**, and **undefined**) and one **nonprivileged** mode (**user**).

3.5 Instruction Pipelining

The Process of fetching the next instruction while the current instruction is being executed is called as “pipelining”. Pipelining is supported by the processor to increase the speed of program execution. Increases throughput. Several operations take place simultaneously, rather than serially in pipelining. The Pipeline has three stages fetch, decode and execute as shown in figure 3.7.

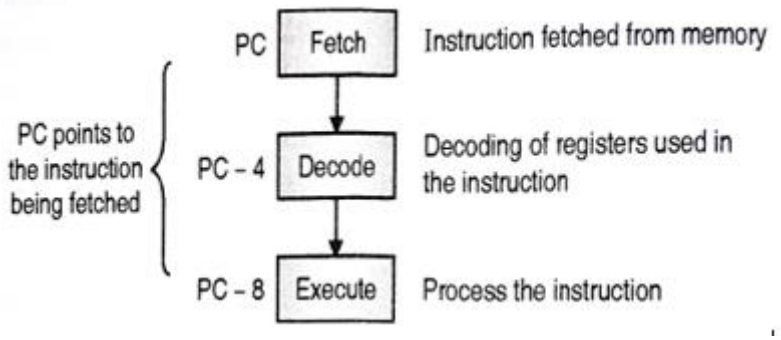


Figure 3.7: 3-stage pipeline

The three stages used in the pipeline are:

(i) Fetch : In this stage the ARM processor fetches the instruction from the memory.

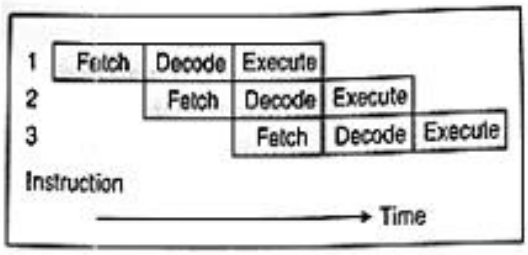
(ii) Decode : In this stage recognizes the instruction that is to be executed.

(iii) Execute 2 In this stage the processor processes the instruction and writes the result back to desired register.

If these three stages of execution are overlapped, we will achieve higher speed of execution. Such pipeline exists in version 7 of ARM processor. Once the pipeline is filled, each instructions require s one cycle to complete execution. Below fig shows three staged pipelined instruction.

In first cycle, the processor fetches instruction 1 from the memory In the second cycle the processor fetches instruction 2 from the memory and decodes instruction 1. In the third cycle the processor fetches instruction 3 from memory, decodes instruction 2 and executes instruction 1. In the fourth cycle the processor fetches instruction 4, decodes instruction 3 and executes instruction 2. The pipeline thus executes an instruction in three cycles i.e. it delivers a throughput equal to one instruction per cycle.

In case of a multi-cycle instruction as shown in Fig. 3.8, instruction 2 (i. e. STR of the store instruction) requires 4 clock cycles and hence the pipeline stalls for one clock pulse. The first instruction completes execution in the third clock pulse, while the second instruction instead of completing execution in fourth clock pulse completes the same in fifth clock pulse. Thereafter every instruction completes execution in one clock pulse as seen in this figure 3.8.



(a) Single cycle instruction execution for a 3-stage pipeline in ARM

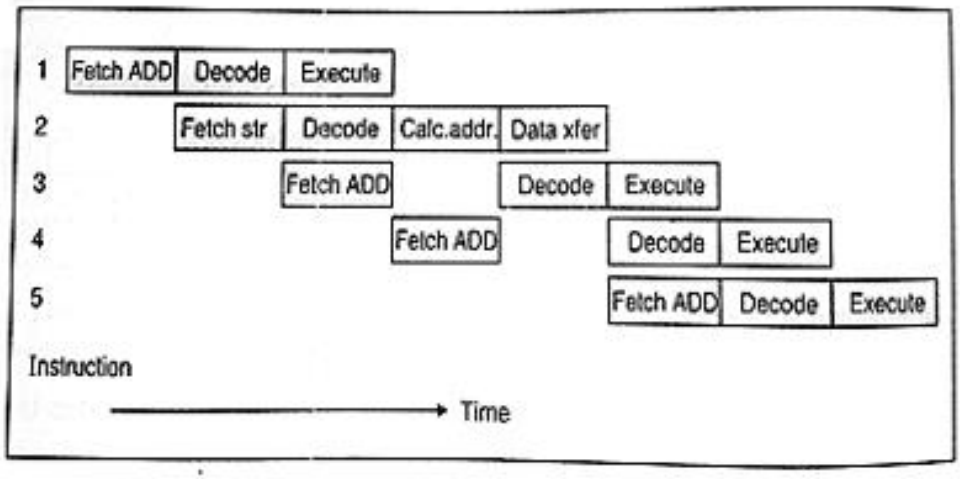


Figure 3.8: Pipelined execution of single cycle and multicycle Instructions

The amount of work done at each stage can be reduced by increasing the number of stages in the pipeline. To improve the performance, the processor then can be operated at higher operating frequency. As more number of cycles are required to fill the pipeline, the system latency also increases. The data dependency between the stages can also be increased as the stages of pipeline increase. So the instructions need to be schedule while writing code to decrease data dependency.

5-Stage Pipeline

A five stage pipelined architecture consists of the following stages.

- **Stage 1 (Instruction Fetch)**
  In this stage the CPU reads instructions from the address in the memory whose value is present in the program counter.
- **Stage 2 (Instruction Decode)**
  In this stage, instruction is decoded and the register file is accessed to get the values from the registers used in the instruction.
- **Stage 3 (Instruction Execute)**
  In this stage, ALU operations are performed.

- **Stage 4 (Memory Access)**
  In this stage, memory operands are read and written from/to the memory that is present in the instruction.
- **Stage 5 (Write Back)**
  In this stage, computed/fetched value is written back to the register present in the instructions.

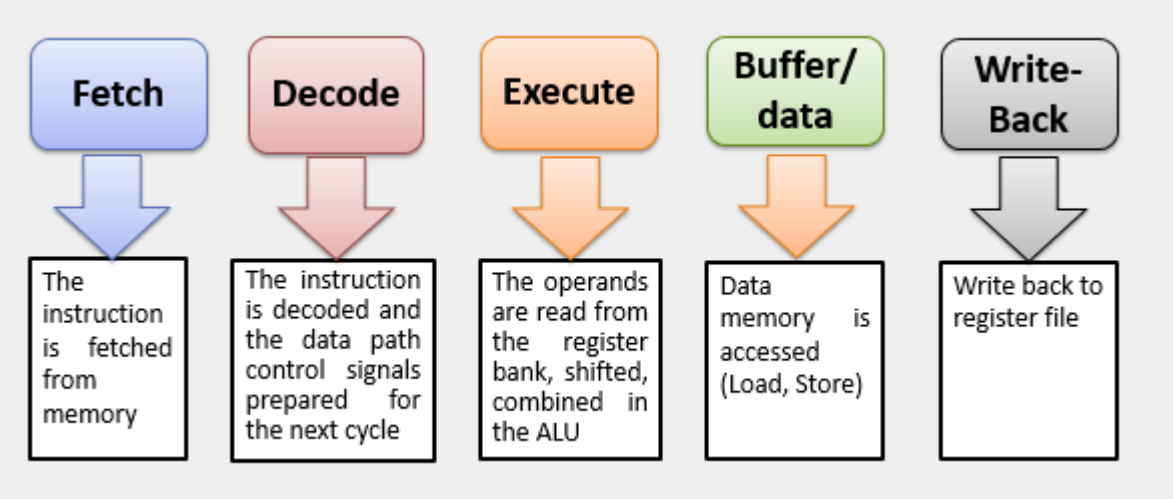


Figure 3.9: Different states in 5-Stages Pipelined architecture

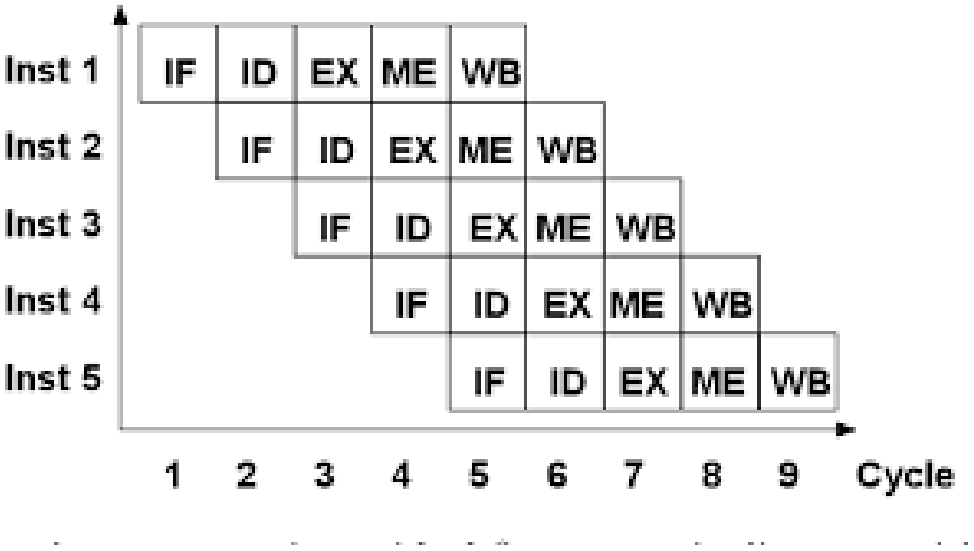


Figure 3.10: Instruction execution in 5-Stages Pipelined architecture

3.5.1 Performance of a pipelined processor

Consider a ‘k’ segment pipeline with clock cycle time as ‘Tp’. Let there be ‘n’ tasks to be completed in the pipelined processor. Now, the first instruction is going to take ‘k’ cycles to come out of the pipeline but the other ‘n – 1’ instructions will take only ‘1’ cycle

each, i.e, a total of ‘n – 1’ cycles. So, time taken to execute ‘n’ instructions in a pipelined processor:

$$ET_{pipeline} = k + n - 1 \text{ cycles}$$
$$= (k + n - 1)\, Tp$$

In the same case, for a non-pipelined processor, execution time of ‘n’ instructions will be:

$$ET_{non\text{-}pipeline} = n * k * Tp$$

So, speedup (S) of the pipelined processor over non-pipelined processor, when ‘n’ tasks are executed on the same processor is:

S = Performance of pipelined processor / Performance of Non-pipelined processor

As the performance of a processor is inversely proportional to the execution time, we have,

$$S = ET_{non\text{-}pipeline} / ET_{pipeline}$$
$$\Rightarrow S = [n * k * Tp] / [(k + n - 1) * Tp]$$
$$S = [n * k] / [k + n - 1]$$

When the number of tasks ‘n’ are significantly larger than k, that is, $n >> k$

$$S = n * k / n$$

$$S = k$$

where ‘k’ are the number of stages in the pipeline.

3.5.2 Pipeline Hazards

Pipeline hazards are situations that prevent the next instruction in the instruction stream from executing during its designated clock cycles. Any condition that causes a stall in the pipeline operations can be called a hazard. There are primarily three types of hazards:

i. Data Hazards

ii. Control Hazards or instruction Hazards

iii. Structural Hazards.

Data Hazards:

A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result of which some operation has to be delayed and the pipeline stalls. Whenever there are two instructions one of which depends on the data obtained from the other.

A=3+A

B=A*4

For the above sequence, the second instruction needs the value of ‘A’ computed in the first instruction. Thus the second instruction is said to depend on the first. If the execution is done in a pipelined processor, it is highly likely that the interleaving of these two instructions can lead to incorrect results due to data dependency between the instructions. Thus the pipeline needs to be stalled as and when necessary to avoid errors.

Structural Hazards

This situation arises mainly when two instructions require a given hardware resource at the same time and hence for one of the instructions the pipeline needs to be stalled. The most common case is when memory is accessed at the same time by two instructions. One instruction may need to access the memory as part of the Execute or Write back phase while other instruction is being fetched. In this case if both the instructions and data reside in the same memory. Both the instructions can’t proceed together and one of them needs to be stalled till the other is done with the memory access part. Thus in general sufficient hardware resources are needed for avoiding structural hazards.

Control hazards

The instruction fetch unit of the CPU is responsible for providing a stream of instructions to the execution unit. The instructions fetched by the fetch unit are in consecutive memory locations and they are executed. However the problem arises when one of the instructions is a branching instruction to some other memory location. Thus all the instruction fetched in the pipeline from consecutive memory locations are invalid now and need to removed(also called flushing of the pipeline).This induces a stall till new instructions are again fetched from the memory address specified in the branch instruction.

Thus the time lost as a result of this is called a branch penalty. Often dedicated hardware is incorporated in the fetch unit to identify branch instructions and compute branch addresses as soon as possible and reducing the resulting delay as a result.

3.6 ARM and Thumb Mode of operation

About ARM and Thumb Mode ARM and Thumb are two different instruction sets supported by ARM cores with a “T” in their name. Limited instruction memory limits the size of the program you can run on your processor, so you want to look for ways to reduce the size of your code. Compile-time optimizations are one obvious way to achieve this, when such optimizations can be found. Increasing the size of the instruction set is another way to do it, but this normally results in an increase in the size of individual instructions across the board, which will lead to a corresponding increase in the amount of storage needed to store the instructions,

which may not be offset by the reduction in the number of instructions needed to write the program. We want to somehow do the same amount of work, yet have the program take up less space. This is where the Thumb extension comes in. Thumb tries to get the best of both worlds by allowing a large (32-bit) instruction set while providing an alternate, small (16-bit) instruction set that can do the bulk of the work while taking up only half the space. They call this concept "code compression", the idea being that the small Thumb instructions are "decompressed" to their equivalent full-size 32-bit ARM instructions before they are run.

For instance, ARM7 TDMI supports Thumb mode. ARM instructions are 32 bits wide, and Thumb instructions are 16 wide. Thumb mode allows for code to be smaller, and can potentially be faster if the target has slow memory. The illustration below shows an example of how the ADD instruction is converted from Thumb to ARM. Notice how the immediate operand, 8 bits in Thumb, is padded with zeroes in its ARM equivalent. Note also that the add instruction takes an additional operand when decompressed.

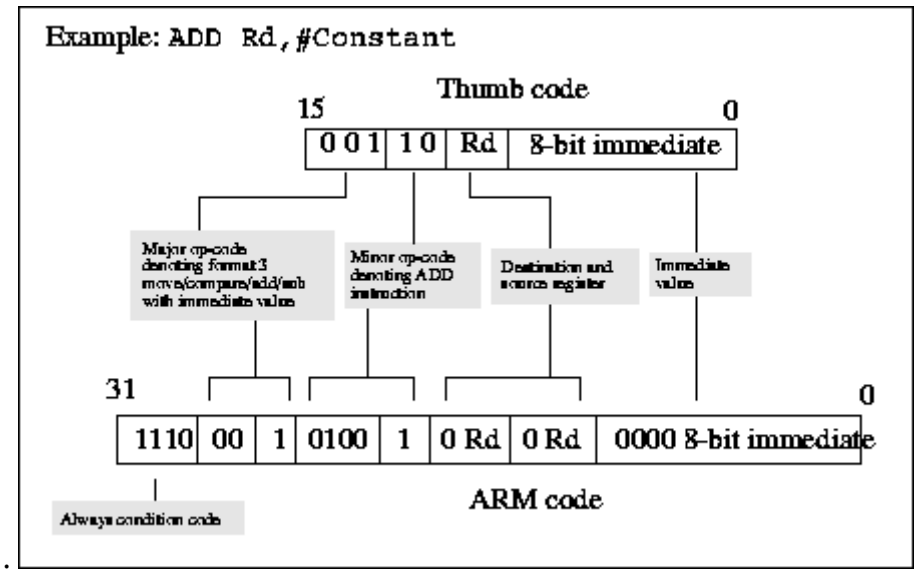


Figure 3.11 Compression with Thumb Instruction sets

A smaller instruction means you must have smaller opcodes, and fewer or smaller (or both) operands. Thumb ensures smaller operands in part by restricting most of its instructions to use 8 general purpose registers in place of the usual 15. A few instructions can access the full register set, such as MOV, to enable workarounds to some of the limitations of a smaller register set. The Thumb instruction set provides most of the functionality required in a typical application. Arithmetic and logical operations, load/store data movements, and conditional and unconditional branches are supported. Based upon the available instruction set, any code written in C could be executed successfully in Thumb state. However, device drivers and exception handlers must often be written at least partly in ARM state.

3.6.1 Register sets in Thumb mode

When operating in the 16-bit Thumb state, the application encounters a slightly different set of registers. Figure 1 compares the programmer's model in that state to the same model in the 32-bit *ARM state*.

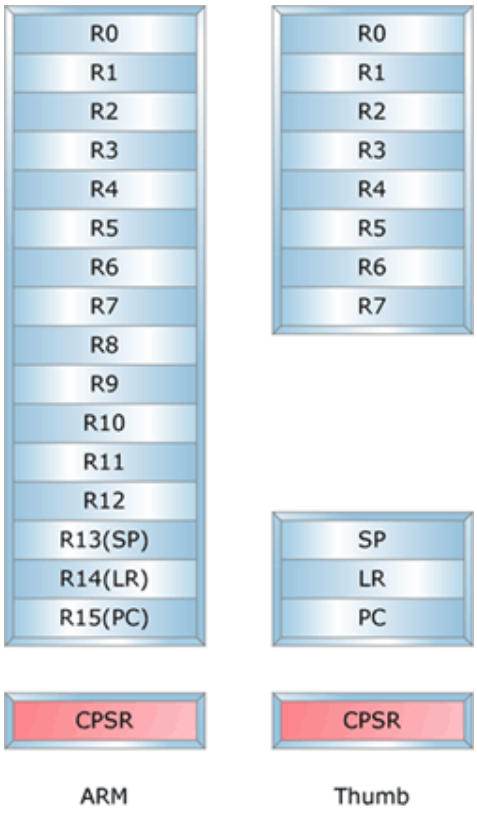


Figure 3.12 ARM vs. Thumb programmer's models

In the ARM state, 17 registers are visible in user mode. One additional register—a saved copy of Current Program Status Register (**CPSR**) that's called **SPSR** (Saved Program Status Register)—is for exception mode only. Notice that the 12 registers accessible in Thumb state are exactly the same physical 32-bit registers accessible in ARM state. Thus data can be passed between software running in the ARM state and software running in the Thumb state via registers R0 through R7. This is done frequently in actual applications.

The biggest register difference involves the **SP** register. The Thumb state has unique stack mnemonics (**PUSH, POP**) that don't exist in the ARM state. These instructions assume the existence of a stack pointer, for which **R13** is used. They translate into load and store instructions in the ARM state.

The **CPSR** register holds the processor mode (user or exception flag), interrupt mask bits, condition codes, and Thumb status bit. The Thumb status bit (**T**) indicates the processor's current state: 0 for ARM state (default) or 1 for Thumb. Although other bits in the CPSR may be

modified in software, it's dangerous to write to **T** directly; the results of an improper state change are unpredictable.

The ARM chip contains a special state bit that tells the CPU whether to expect a compressed Thumb instruction or a standard ARM instruction. This bit is toggled with its own instruction, BX, which must be inserted into the code every time a programmer or compiler wishes to switch between Thumb mode and Standard ARM mode. An obvious result of this is that there is some overhead to switching between modes, thus it is probably not a good idea to switch to Thumb unless it will save you more than two instructions of equivalent ARM code.

3.7 Analog to Digital Converters

Analogue-to-Digital Converters, (ADCs) allow micro-processor controlled circuits, Arduinos, Raspberry Pi, and other such digital logic circuits to communicate with the real world. In the real world, analogue signals have continuously changing values which come from various sources and sensors which can measure sound, light, temperature or movement, and many digital systems interact with their environment by measuring the analogue signals from such transducers.

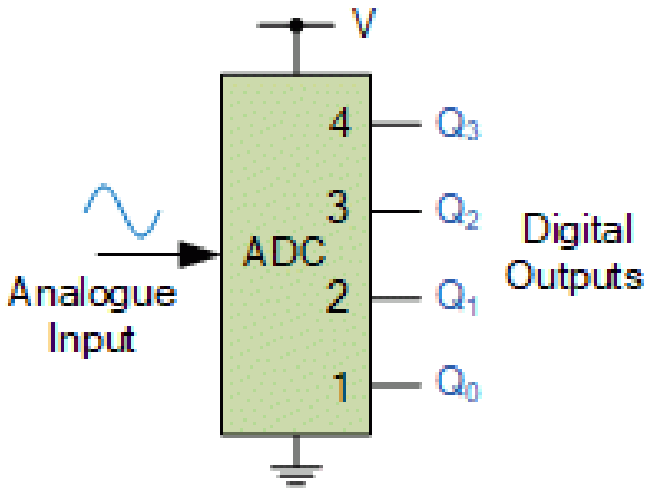


Figure 3.13: Schematic diagram of A/D converter

- The resolution of the ADC is the number of bits it uses to digitize the input samples.
- For an n bit ADC the number of discrete digital levels that can be produced is 2^n.
- Thus, a 12 bit digitizer can resolve 212 or 4096 levels. The least significant bit (lsb) represents the smallest interval that can be detected and in the case of a 12 bit digitizer is 1/4096 or 2.4 x 10-4.
- To convert the lsb into a voltage we take the input range of the digitizer and divide by two raised to the resolution of the digitizer.
- Table 1 shows the lsb for a one Volt (±500 mV) input range for digitizers with resolutions of 8 to 16 bits.

Table 3.1 Resolution in ADC for different bit-size

Resolution	Ideal Dynamic range	Minimum Voltage Increment
8 Bit	256:1	3.92 mV
10 Bit	1024:1	0.98 mV
12 Bit	4096:1	0.244 mV
14 Bit	16384:1	61 µV
16 Bit	65536:1	15 µV

3.8 D/A Converter

A **Digital to Analog Converter (DAC)** converts a digital input signal into an analog output signal. The digital signal is represented with a binary code, which is a combination of bits 0 and 1. This chapter deals with Digital to Analog Converters in detail. The **block diagram** of DAC is shown in the following figure. A Digital to Analog Converter (DAC) consists of a number of binary inputs and a single output. In general, the **number of binary inputs** of a DAC will be a power of two.

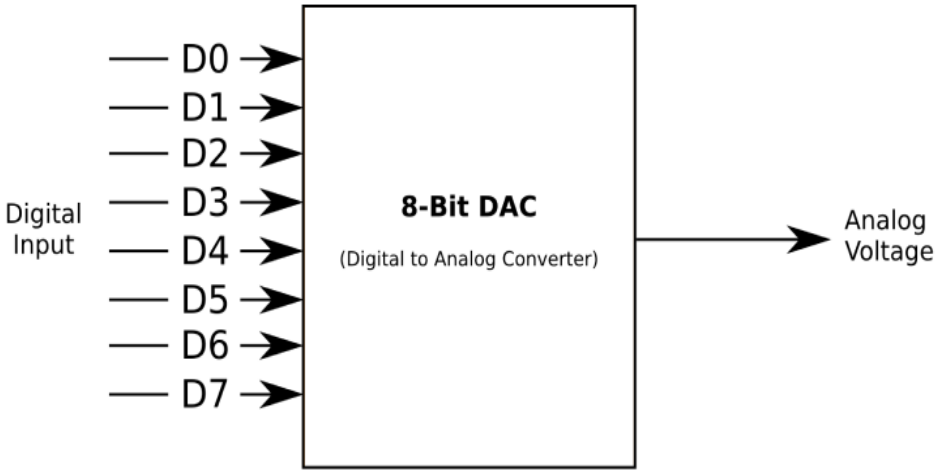


Figure 3.14: Schematic diagram of D/A converter

3.9 Sensors and Actuators

Sensor is a device used for the conversion of physical events or characteristics into the electrical signals. This is a hardware device that takes the input from environment and gives to the system by converting it. For example, a thermometer takes the temperature as physical characteristic and then converts it into electrical signals for the system.

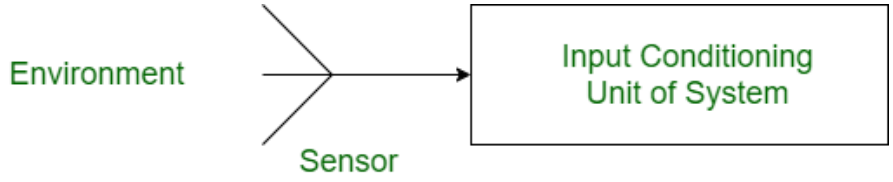


Figure 3.15 Function of a sensor

Actuator is a device that converts the electrical signals into the physical events or characteristics. It takes the input from the system and gives output to the environment. For example, motors and heaters are some of the commonly used actuators.

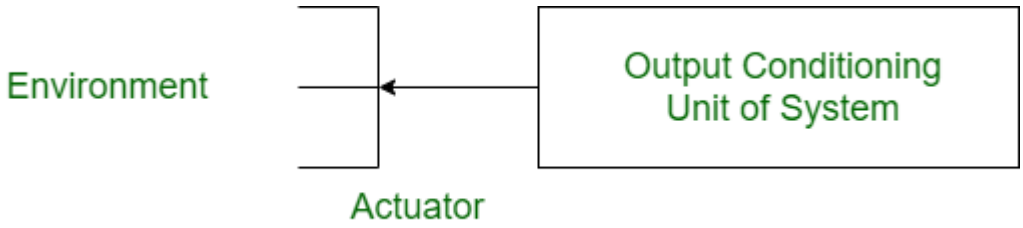


Figure 3.16 Function of a actuator

Table 3.2 Difference between Sensor and Actuator

Sensor	Actuator
It converts physical characteristics into electrical signals.	It converts electrical signals into physical characteristics.
It takes input from environment.	It takes input from output conditioning unit of system.
It gives output to input conditioning unit of system.	It gives output to environment.

Sensor	Actuator
Sensor generated electrical signals.	Actuator generates heat or motion.
It is placed at input port of the system.	It is placed at output port of the system.
It is used to measure the physical quantity.	It is used to measure the continuous and discrete process parameters.
It gives information to the system about environment.	It accepts command to perform a function.
Example: Photo-voltaic cell which converts light energy into electrical energy.	Example: Stepper motor where electrical energy drives the motor.

3.10 Case study- Digital Clock Design

Alarm Clock, Timer and Stopwatch are common time-keeping features. These functions are so frequently used that it is difficult to imagine modern life without a time-keeping application nowadays. Whether it is a scheduled wake up alarm, a stopwatch to track the time one has jogged or a timer and alarm to schedule office tasks, time-keeping is part and parcel of day-to-day life. This is an Arduino project demonstrating a complete time-keeping application. The project is a real-time clock and allows setting alarms, timers and running stopwatch.

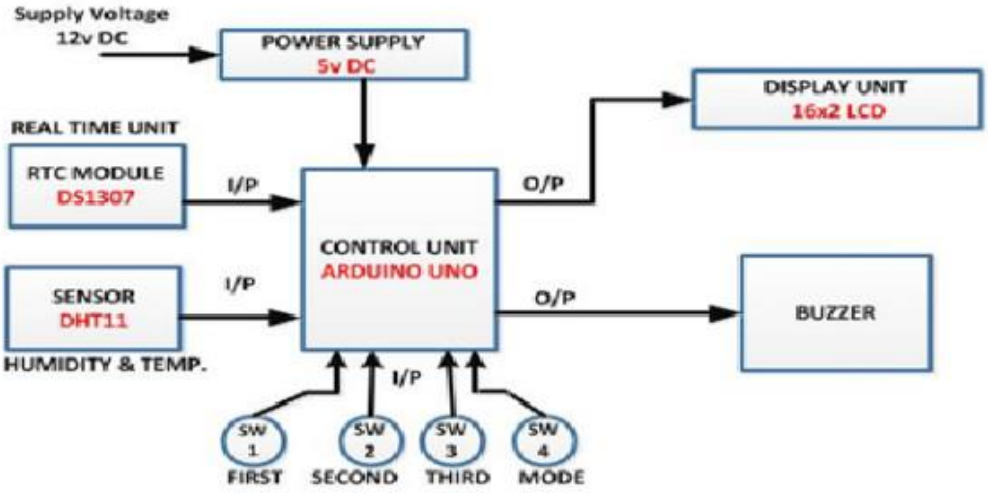


Figure 3.17 Components of Digital clock

It also displays real-time weather conditions with temperature and humidity indications as add-ons. The project has utilized RTC DS1307 for time-keeping and DHT11 sensor for fetching weather information. It is built on Arduino UNO and RTC used is internally powered through a button cell, so the project keeps track of real time and perform user-defined functions irrespective of the continuity of power supply to the circuit. The time and date, temperature and humidity values are displayed on a 16X2 LCD which also provides human interface to set alarm, timer and stopwatch. The users can feed inputs through a 4-switch keypad with switches for the following functions – Mode Selection, ENTER, Increment and SAVE buttons. A buzzer is connected to the Arduino board for realizing alarm and timer alerts.

The project runs under four modes of operations:

1) Default Mode: By default, the project is set to display time, date, temperature and humidity information on the 16X2 LCD screen.

2) Alarm Mode: Here, user can set an alarm. The user enters this mode by pressing Mode selection button once and pressing the ENTER Button thereafter. He can first increase “Hours” value by pressing Increment button and skip to increase “Minutes” value by pressing the ENTER button again. After setting “Hours” and “Minutes” value the user can invoke alarm by pressing the SAVE button. To exit the alarm mode, Increment and mode selection buttons have to be pressed together.

3) Timer Mode: A timer setting mode can be entered by pressing the Mode selection button twice and pressing the ENTER button thereafter. The process for setting and saving time for timer is same as in alarm mode except that “Seconds” value can also be set in this mode. The user can exit the timer mode after setting time by just pressing the mode selection button once again.

4) Stopwatch Mode: To enter stopwatch mode, pressing mode selection button thrice and pressing the ENTER button thereafter works. Here pressing the SAVE button starts the stopwatch, pressing increment button pauses the stopwatch and pressing ENTER button again resets the stop watch. To exit the stopwatch mode, ENTER and Mode Selection buttons have to be pressed together.

The major blocks of the circuit are as follow

1) Power Supply Circuit
2) RTC DS1307 Module
3) DHT11 Temperature and Humidity sensor

4) LCD Display
5) 4-switch keypad
6) Buzzer
7) Microcontroller Board

1) Power Supply – The entire circuit runs on a 5V DC supply. A 12V battery is used to source power to the circuit. The 12V supply is stepped down to 5V by a 7805 voltage regulator. The pin 1 of 7805 receives 12V supply from anode and pin 2 is grounded. The output 5V is generated at pin 3 of the regulator. An LED is also connected in parallel to the output as a visual indicator of power supply.

2) RTC DS1307 Interfacing – The RTC DS1307 has a built in button cell that allows keeping track of real-time irrespective of the power supply. For interfacing with the microcontroller board, SDA and SCL pins of the RTC are connected to the SDA and SCL pins of controller.

3) DHT11 Temperature and Humidity Sensor – This is a digital sensor with inbuilt capacitive humidity sensor and Thermistor. It relays a real-time temperature and humidity reading every 2 seconds as a digital output. The pin 1 and 4 of DHT11 are VCC and Ground respectively.\

4) LCD Display – The 16X2 LCD display is connected to the microcontroller.

5) 4-switch Keypad – The keypad here is a set of four push-to-on switches which are connected to 10, 9, 8 and 7 pins of the Arduino UNO through 1K ohm pull-up resistors. The switches connected at 10, 9, 8 and 7 pins works as SAVE, Increment, Enter and Mode selection buttons respectively. In the circuit diagram, SAVE, Increment, Enter and Mode selection buttons are designated by FIRST, SECOND, THIRD and MODE labels.

6) Buzzer – The buzzer is connected to pin 6 of the Arduino board. A common emitter NPN BC547 transistor circuit is used to relay signal from Arduino pin to the buzzer.

3.11 Internet of Things

IoT (Internet of Things) is an advanced automation and analytics system which exploits networking, sensing, big data, and artificial intelligence technology to deliver complete systems for a product or service. These systems allow greater transparency, control, and performance when applied to any industry or system. IoT systems have applications across industries through their unique flexibility and ability to be suitable in any environment. They enhance data collection, automation, operations, and much more through smart devices and powerful

enabling technology. IoT systems allow users to achieve deeper automation, analysis, and integration within a system. They improve the reach of these areas and their accuracy. IoT utilizes existing and emerging technology for sensing, networking, and robotics.

IoT exploits recent advances in software, falling hardware prices, and modern attitudes towards technology. Its new and advanced elements bring major changes in the delivery of products, goods, and services; and the social, economic, and political impact of those changes.

IoT − Key Features

The most important features of IoT include artificial intelligence, connectivity, sensors, active engagement, and small device use. A brief review of these features is given below −

- **AI** − IoT essentially makes virtually anything "smart", meaning it enhances every aspect of life with the power of data collection, artificial intelligence algorithms, and networks. This can mean something as simple as enhancing your refrigerator and cabinets to detect when milk and your favorite cereal run low, and to then place an order with your preferred grocer.
- **Connectivity** − New enabling technologies for networking, and specifically IoT networking, mean networks are no longer exclusively tied to major providers. Networks can exist on a much smaller and cheaper scale while still being practical. IoT creates these small networks between its system devices.
- **Sensors** − IoT loses its distinction without sensors. They act as defining instruments which transform IoT from a standard passive network of devices into an active system capable of real-world integration.
- **Active Engagement** − Much of today's interaction with connected technology happens through passive engagement. IoT introduces a new paradigm for active content, product, or service engagement.
- **Small Devices** − Devices, as predicted, have become smaller, cheaper, and more powerful over time. IoT exploits purpose-built small devices to deliver its precision, scalability, and versatility.

IoT − Sensors

The most important hardware in IoT might be its sensors. These devices consist of energy modules, power management modules, RF modules, and sensing modules. RF modules manage communications through their signal processing, WiFi, ZigBee, Bluetooth, radio transceiver, duplexer, and BAW.

The sensing module manages sensing through assorted active and passive measurement devices. Here is a list of some of the measurement devices used in IoT.

Table 3.3 Sensing Devices for IoT

1.	accelerometers	7. temperature sensors
2.	magnetometers	8. proximity sensors
3.	gyroscopes	9. image sensors
4.	acoustic sensors	10. light sensors
5.	pressure sensors	11. RFID sensors
6.	humidity sensors	12. micro flow sensors

Wearable Electronics

Wearable electronic devices are small devices worn on the head, neck, arms, torso, and feet. Smartwatches not only help us stay connected, but as a part of an IoT system, they allow access needed for improved productivity.

Current smart wearable devices include −

- **Head** − Helmets, glasses
- **Neck** − Jewelry, collars
- **Arm** − Watches, wristbands, rings
- **Torso** − Clothing, backpacks
- **Feet** − Socks, shoes

Smart glasses help us enjoy more of the media and services we value, and when part of an IoT system, they allow a new approach to productivity.

Standard Devices

The desktop, tablet, and cellphone remain integral parts of IoT as the command center and remotes.

- The **desktop** provides the user with the highest level of control over the system and its settings.
- The **tablet** provides access to the key features of the system in a way resembling the desktop, and also acts as a remote.
- The **cellphone** allows some essential settings modification and also provides remote functionality.

Other key connected devices include standard network devices like **routers** and **switches**.

IoT Software

IoT software addresses its key areas of networking and action through platforms, embedded systems, partner systems, and middleware. These individual and master applications are responsible for data collection, device integration, real-time analytics, and application and process extension within the IoT network. They exploit integration with critical business systems (e.g., ordering systems, robotics, scheduling, and more) in the execution of related tasks.

Data Collection

This software manages sensing, measurements, light data filtering, light data security, and aggregation of data. It uses certain protocols to aid sensors in connecting with real-time, machine-to-machine networks. Then it collects data from multiple devices and distributes it in accordance with settings. It also works in reverse by distributing data over devices. The system eventually transmits all collected data to a central server.

Device Integration

Software supporting integration binds (dependent relationships) all system devices to create the body of the IoT system. It ensures the necessary cooperation and stable networking between devices. These applications are the defining software technology of the IoT network because without them, it is not an IoT system. They manage the various applications, protocols, and limitations of each device to allow communication.

Real-Time Analytics

These applications take data or input from various devices and convert it into viable actions or clear patterns for human analysis. They analyze information based on various settings and designs in order to perform automation-related tasks or provide the data required by industry.

Application and Process Extension

These applications extend the reach of existing systems and software to allow a wider, more effective system. They integrate predefined devices for specific purposes such as allowing certain mobile devices or engineering instruments access. It supports improved productivity and more accurate data collection.

IoT primarily exploits standard protocols and networking technologies. However, the major enabling technologies and protocols of IoT are RFID, NFC, low-energy Bluetooth, low-energy wireless, low-energy radio protocols, LTE-A, and WiFi-Direct. These technologies support the specific networking functionality needed in an IoT system in contrast to a standard uniform network of common systems.

NFC and RFID

RFID (radio-frequency identification) and NFC (near-field communication) provide simple, lowenergy, and versatile options for identity and access tokens, connection bootstrapping, and payments.

- RFID technology employs 2-way radio transmitter-receivers to identify and track tags associated with objects.
- NFC consists of communication protocols for electronic devices, typically a mobile device and a standard device.

Low-Energy Bluetooth

This technology supports the low-power, long-use need of IoT function while exploiting a standard technology with native support across systems.

Low-Energy Wireless

This technology replaces the most power hungry aspect of an IoT system. Though sensors and other elements can power down over long periods, communication links (i.e., wireless) must remain in listening mode. Low-energy wireless not only reduces consumption, but also extends the life of the device through less use.

Radio Protocols

ZigBee, Z-Wave, and Thread are radio protocols for creating low-rate private area networks. These technologies are low-power, but offer high throughput unlike many similar options. This increases the power of small local device networks without the typical costs.

LTE-A

LTE-A, or LTE Advanced, delivers an important upgrade to LTE technology by increasing not only its coverage, but also reducing its latency and raising its throughput. It gives IoT a tremendous power through expanding its range, with its most significant applications being vehicle, UAV, and similar communication.

WiFi-Direct

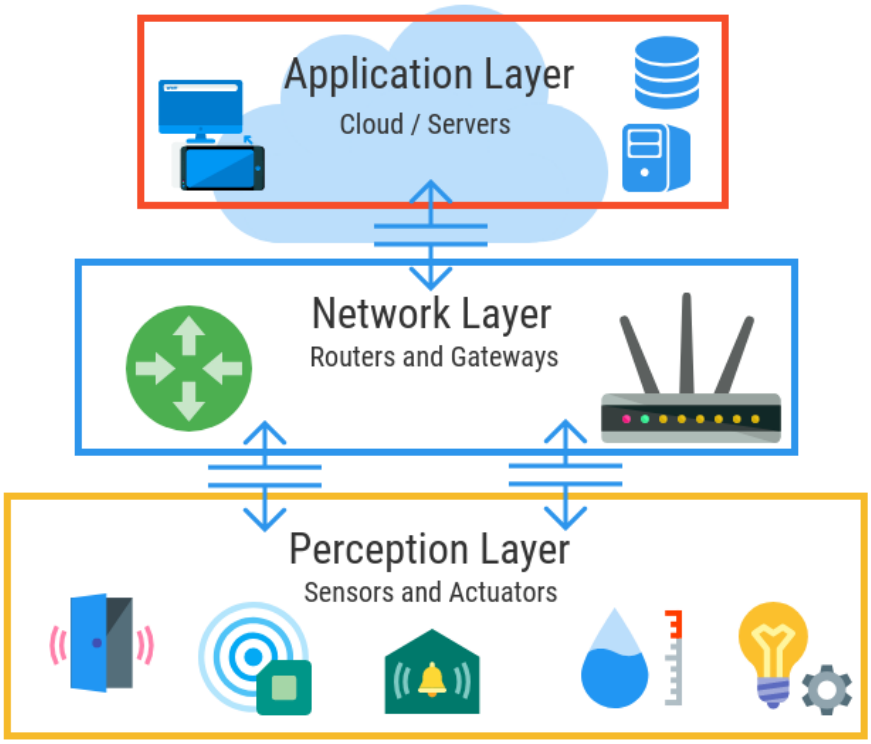
WiFi-Direct eliminates the need for an access point. It allows P2P (peer-to-peer) connections with the speed of WiFi, but with lower latency. WiFi-Direct eliminates an element of a network that often bogs it down, and it does not compromise on speed or throughput.

3.11.1 Three Layer (Tier) IoT Architecture

While there are myriad bits that build a complete end-to-end IoT architecture, this architecture simplifies it down to three fundamental building blocks.

1. Perception layer – Sensors, actuators and edge devices that interact with the environment
2. Network Layer – Discovers, connects and translates devices over a network and in coordination with the application layer

3. Application Layer – Data processing and storage with specialized services and functionality for users



Devices make up a physical or perceptual IoT layer and typically include sensors, actuators and other smart devices. One might call these the "Things" in the Internet of Things. Devices, in turn, interface and communicate to the cloud via wire or localized Radio Frequency (RF) networks. This is typically done through gateways. Oftentimes IoT devices are said to be at the "edge" of the IoT network and are referred to as "edge nodes".

When selecting a device, it is important to consider requirements for specific I/O protocols and potential latency, wired or RF interfaces, power, ruggedness and the device's overall sensitivity. It is critical to determine how much device flexibility your architecture should have.

Many newer devices are IoT ready right out of the box (e.g. are sold with low power bluetooth or are Ethernet enabled). However, most sensors, actuators and legacy devices still interface via conventional "pre-IoT" methods such as analog or serial connections. It is common practice to connect one or more of these conventional devices to microcontrollers, systems on modules (SOMs) or single-board computers (SBCs) with the necessary peripherals (e.g. Arduino, NetBurner, or Raspberry Pi). At a minimum, such collectors provide network

connectivity between the edge nodes and a master gateway. In some instances they may be capable of being configured as a gateway as well.

IoT Gateways are an important middleman element that serves as the messenger and translator between the cloud and clusters of smart devices. They are physical devices or software programs that typically run from the field in close proximity to the edge sensors and other devices. Large IoT systems might use a multitude of gateways to serve high volumes of edge nodes. They can provide a range of functionality, but most importantly they normalize, connect and transfer data between the physical device layer and the cloud. In fact, all data moving between the cloud and the physical device layer goes through a gateway. IoT gateways are sometimes called "intelligent gateways" or "control tiers". [4]

Today, gateways also support additional computing and peripheral functionality such as telemetry, multiple protocol translation, artificial intelligence, pre-processing and filtering massive raw sensor data sets, provisioning and device management. It is becoming common practice to implement data encryption and security monitoring on the intelligent gateway so as to prevent malicious man-in-the-middle attacks against otherwise vulnerable IoT systems. NetBurner devices can be used as robust IoT Gateways, as well as IoT Device Collectors, as mentioned above.

Certain gateways offer an operating system that is specialized for use in embedded and IoT systems along with optimized low-level support for different hardware interfaces, such as NetBurner's SOMs with our custom Real Time Operating System (RTOS) and interface libraries. Managing memory, I/O, timing and interface is not a trivial task. According to Google Cloud, "Generally these abstractions are not easy to use directly, and frequently the OS does not provide abstractions for the wide range of sensor and actuator modules you might encounter in building IoT solutions."[5] Libraries are typically available based on standard protocols. Oftentimes, the most optimized libraries will be part of commercially available development kits and SDKs (as is the case with NetBurner for a multitude of protocols and hardware types).

The **Cloud** is the application layer. It communicates with the gateway, typically over wired or cellular internet. The "Cloud" might be anything from services like AWS or Google Cloud, server farms, or even a company's on-premises remote server. It provides powerful servers and databases that enable robust IoT applications and integrate services such as data storage, big data processing, filtering, analytics, 3rd party APIs, business logic, alerts, monitoring and user interfaces. In a Three Layer IoT Architecture, the "Cloud" is also used to control, configure, and trigger events at the gateway, and ultimately the edge devices.

IoT − Advantages

The advantages of IoT span across every area of lifestyle and business. Here is a list of some of the advantages that IoT has to offer −

- **Improved Customer Engagement** − Current analytics suffer from blind-spots and significant flaws in accuracy; and as noted, engagement remains passive. IoT completely transforms this to achieve richer and more effective engagement with audiences.
- **Technology Optimization** − The same technologies and data which improve the customer experience also improve device use, and aid in more potent improvements to technology. IoT unlocks a world of critical functional and field data.
- **Reduced Waste** − IoT makes areas of improvement clear. Current analytics give us superficial insight, but IoT provides real-world information leading to more effective management of resources.
- **Enhanced Data Collection** − Modern data collection suffers from its limitations and its design for passive use. IoT breaks it out of those spaces, and places it exactly where humans really want to go to analyze our world. It allows an accurate picture of everything.

IoT − Disadvantages

Though IoT delivers an impressive set of benefits, it also presents a significant set of challenges. Here is a list of some its major issues −

- **Security** − IoT creates an ecosystem of constantly connected devices communicating over networks. The system offers little control despite any security measures. This leaves users exposed to various kinds of attackers.
- **Privacy** − The sophistication of IoT provides substantial personal data in extreme detail without the user's active participation.
- **Complexity** − Some find IoT systems complicated in terms of design, deployment, and maintenance given their use of multiple technologies and a large set of new enabling technologies.
- **Flexibility** − Many are concerned about the flexibility of an IoT system to integrate easily with another. They worry about finding themselves with several conflicting or locked systems.

- **Compliance** − IoT, like any other technology in the realm of business, must comply with regulations. Its complexity makes the issue of compliance seem incredibly challenging when many consider standard software compliance a battle

The hardware utilized in IoT systems includes devices for a remote dashboard, devices for control, servers, a routing or bridge device, and sensors. These devices manage key tasks and functions such as system activation, action specifications, security, communication, and detection to support-specific goals and actions.

TEXT / REFERENCE BOOKS

1. Joseph Yiu, “The Definitive Guide to the ARM Cortex-M3”, Newnes, 2nd Edition, 2009.

2. Mark Fisher, “ARM Cortex M4 Cookbook, Packt Publishing, 2016.

3. Lyla B. Das, “Architecture, Programming and Interfacing of Low-power Processors ARM 7, Cortex-M”, Cengage, 1 st Edition, 2017.

4. Steve Furber, "ARM System-on-Chip Architecture” Pearson, 2nd Edition, 2015

Exercise Questions

1. List the features of A, R and M profile-based ARM processors.
2. Identify the any four target applications of ARM cortex-M processor family.
3. Identify the sensors and actuators required for a smart home system.
4. Determine the maximum resolution of A/D converter available in classic ARM-7 processors without pre-scaling.
5. Identify the key communication protocols suitable for Internet of Things.
6. List any four applications of D/A converters.
7. In which mode, do you program the ARM processor, for an application in which cost of memory is much more critical than the execution speed?
8. Illustrate the basic architecture of a classic ARM processer and outline its key features.
9. Consider an instruction pipeline with four stages with the stage delays 5 nsec, 6 nsec, 11 nsec, and 8 nsec respectively. The delay of an inter-stage register stage of the pipeline is 1 nsec. What is the approximate speedup of the pipeline in the steady state under ideal conditions as compared to the corresponding non-pipelined implementation?
10. Explain the Thumb programmer model of ARM processor and its applications.
11. Articulate the 5-layer model of Internet of Things architecture.
12. Develop a system model using ARM processor for seamless real-time vehicle tracking system. Outline the key hardwares required for the system.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – IV SECA3019 – REAL WORLD INTERFACING USING ARM PROCESSOR

[Interfacing the peripherals to LPC2148: GSM and GPS using UART, on-chip ADC using interrupt (VIC), EEPROM using I2C, SD card interface using SPI, on-chip DAC for waveform generation]

4.1 GSM MODULE INTERFACING WITH LPC2148

- GSM (Global System for Mobile Communications) is the technology that underpins most of the world's mobile phone networks.
- GSM is an open, digital cellular technology used for transmitting mobile voice and data services.
- GSM operates in the 900MHz and 1.8GHz bands GSM supports data transfer speeds of up to 9.6 kbps, allowing the transmission of basic data services such as SMS.
- The SIM300 module is a Triband GSM/GPRS solution in a compact plug in module featuring an industry-standard interface

4.1.1 Features of GSM MODEM

- Single supply voltage 3.2v-4.5v
- Typical power consumption in SLEEP Mode: 2.5mA.
- SIM300 tri-band
- MT,MO,CB, text and PDU mode, SMS storage: SIM card
- Supported SIM Card :1.8V,3V

Figure 4.1: GSM modules

- GSM/GPRS module is used to establish communication between a computer and a GSM-GPRS system.
- Global System for Mobile communication (GSM) is an architecture used for mobile communication in most of the countries.
- Global Packet Radio Service (GPRS) is an extension of GSM that enables higher data transmission rate

- SIM7600EI is a complete multi-band LTE/EDGE/GPRS/GSM module solution in LCC type which supports LTE CAT1 up to 10Mbps for downlink and 5Mbps for uplink data transfer.

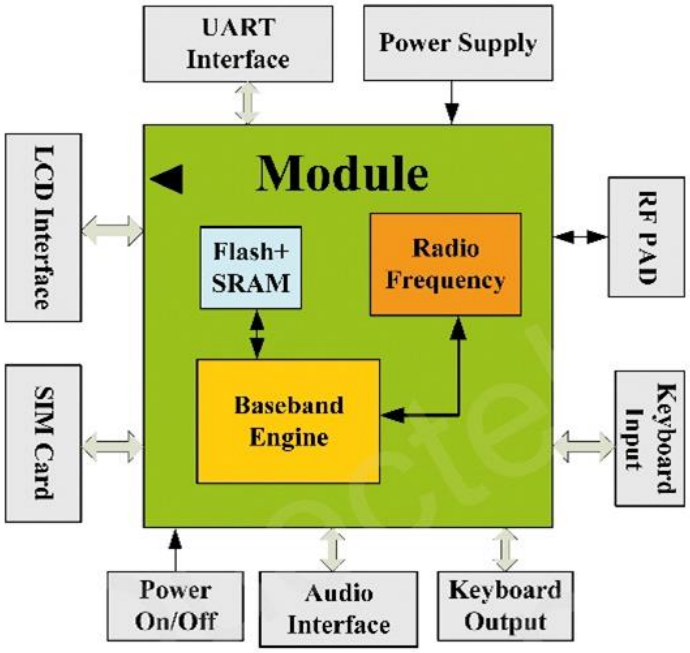


Figure 4.2: GSM Module functional blocks

4.1.2 GSM Mobile Vs GSM Module

- A GSM mobile is a complete system in itself with embedded processors that are dedicated to provide an interface between the user and the mobile network.
- The AT commands are served between the processors of the mobile termination and the terminal equipment.
- The mobile handset can also be equipped with a USB interface to connect with a computer, but it may or may not support AT commands from the computer or an external processor/controller.
- The GSM/GPRS module, on the other hand, always needs a computer or external processor/controller to receive AT commands from.
- GSM/GPRS module itself does not provide any interface between the user and the network, but the computer to which module is connected is the interface between user and network.
- An advantage that GSM/GPRS modules offer is that they support concatenated SMS which may not be supported in some GSM mobile handsets

- Applications of GSM/GPRS module The GSM/GPRS module demonstrates the use of AT commands. They can feature all the functionalities of a mobile phone through computer like making and receiving calls, SMS, MMS etc. These are mainly employed for computer based SMS and MMS services.

4.1.3 AT Commands

AT commands are used to control MODEMs.AT is the abbreviation for Attention.

- These commands come from Hayes commands that were used by the Hayes smart modems.
- The Hayes commands started with AT to indicate the attention from the MODEM.
- The dial up and wireless MODEMs need AT commands to interact with a computer.
- AT commands with a GSM/GPRS MODEM

Table 4.1 GSM AT Commands

GSM AT Commands and their functions	
AT Command	**Function of AT Command**
ATD	Dial
AT+CGMS	Send SMS Message
AT+CMSS	Send SMS Message from storage
AT+CMGL	List SMS Messages
AT+CMGR	Read SMS Messages
AT+CSCA?	Service Centre Address
AT+CPMS	To choose storage from ME or SM
AT+IPR=0	To choose auto from baud rate
AT+CMGF=	To choose PDU Mode or Text Mode

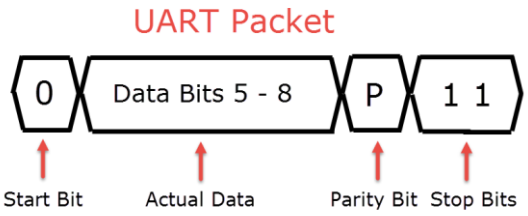


Figure 4.3: UART data format

4.1.4 Interfacing of GSM Module

Figure 4.4 shows interfacing of LPC2148 with GSM modem using UART protocol. MAX232 IC is used for voltage level shifting from 0V/5V to -12V/+12V.

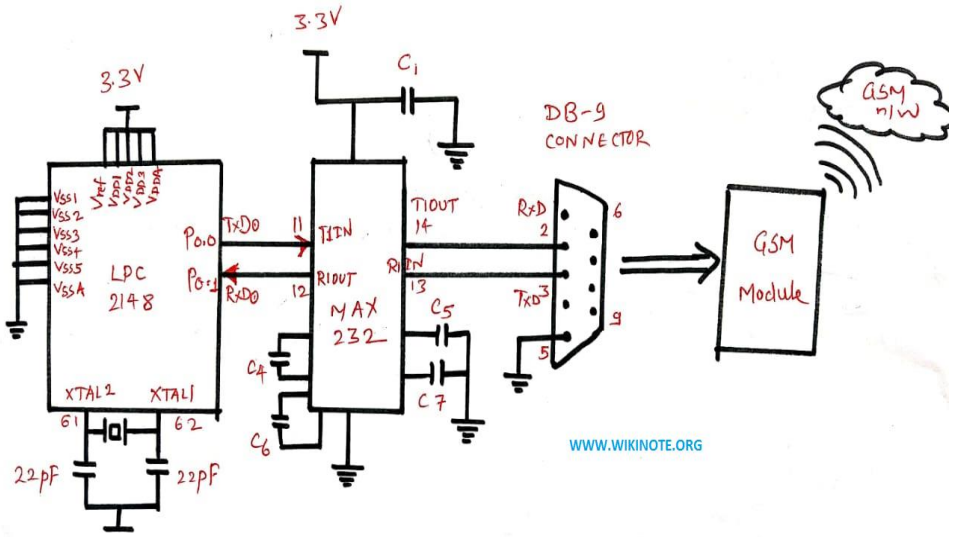


Figure 4.4 GSM modem interfacing with LPC2148

Table 4.2 Pin assignment for GSM interfacing

	UART DB-9 Connector	**LPC2148 Processor Lines**
UART0 (P1) ISP PGM	TXD-0	P0.0
	RXD-0	P0.1
UART1 (P2)	TXD-1	P0.8
	RXD-1	P0.9

Algorithm for GSM module interfacing with LPC2148

1) Start

2) Initialise UART0 or UART1 serial interface using following instruction

PINSEL0=0X0000 0005;//Enable P0.0-TxD0,P0.1-RxD0
U0LCR=0X83; //8-BIT Character length, NO parity,1 stop bit
U0DLL=97; //Baud rate=9600@PCLK=15Mhz – Set the data rate
U0LCR=0X03; **Divisor Latch Access Bit (DLAB)** to Zero

3) Transmit different AT commands through UART module using instruction

while(!(U0LSR&0X20));//Monitor TI flag

4) If transmission buffer is Empty, Transmit AT commands

U0THR=ch; // **U0THR (UART0 Transmit Holding Register)**

5) Provide delay while transmitting each command

6) To transmit a single character use PUTCH function & to transmit a string use PUTS function

7) END

4.1.5 Example Program for GSM Interfacing

```
/*****************************************************************************/
/* Project Name:- GSM Module Interfacing with LPC2148 using UART module      */
/* Device:- LPC2148                                                          */
/* Compiler:- KeilUvision4                                                   */
/* Language:- Embedded C  */
/*****************************************************************************
****************/
#include<lpc21xx.h>   //Includes LPC2148 register definitions
#include "serial.h"
unsigned char GsmSendMsg(unsigned char *msgStr);
void DelayMs(unsigned int count);

int main(void)
{
  Uart0Init();
  Uart0PutS("AT\r\n");
```

```
   DelayMs(500);
   Uart0PutS("ATE0\r\n");  //Turn echo off
  DelayMs(500);
   Uart0PutS("ATD9503XXXXXX;\r\n"); //replace xxxxxxxxxx with number to call
  DelayMs(20000);
   Uart0PutS("ATH0\r\n");      //disconnect call
  DelayMs(3000);
   GsmSendMsg("WIKINOTE FOUNDATION");
  while(1);
}

unsigned char GsmSendMsg(unsigned char *msgStr)
{
    Uart0PutS("AT+CMGF=1\r\n");//Send SMS: Select Text mode
   DelayMs(100);
    Uart0PutS("AT+CMGS=\"9503XXXXXX\"\r\n"); //Send SMS to mobile number
   DelayMs(100);
    Uart0PutS(msgStr);
    DelayMs(100);
    Uart0PutCh(0x1A);          //CNTL + Z
   DelayMs(3000);
   return (1);
}

void DelayMs(unsigned int count)
{
   volatile unsigned int j,k;
   for (j=0;j<count;j++)
      for (k=0;k<6000;k++);
}
```

4.2. GPS MODULE INTERFACING

The SKG13BL is a complete GPS engine module that features super sensitivity, ultra low power and small form factor. The GPS signal is applied to the antenna input of module, and a complete serial data message with position, velocity and time information is presented at the serial interface with NMEA protocol or custom protocol.

It is based on the high performance features of the MediaTek MT3337 single-chip architecture, Its –165dBm tracking sensitivity extends positioning coverage into place like urban canyons and dense foliage environment where the GPS was not possible before. The

small form factor and low power consumption make the module easy to integrate into portable device like PNDs, mobile phones, cameras and vehicle navigation systems.

4.2.1 Features of GPS module

- Ultra high sensitivity: -165dBm
- Built-in 12 multi-tone active interference canceller
- Low power consumption: Typical 22mA@3.3V
- ±10ns high accuracy time pulse (1PPS)
- NMEA Output：GGA,GSA,GSV,RMC
- Advanced Features: AlwaysLocate; AIC
- QZSS,SBAS(WAAS,EGNOS,MSAS,GAGAN)
- UART interface: 4800/9600/38400/115200 bps
- Small form factor: 15x13x2.2mm and RoHS compliant (Lead-free)

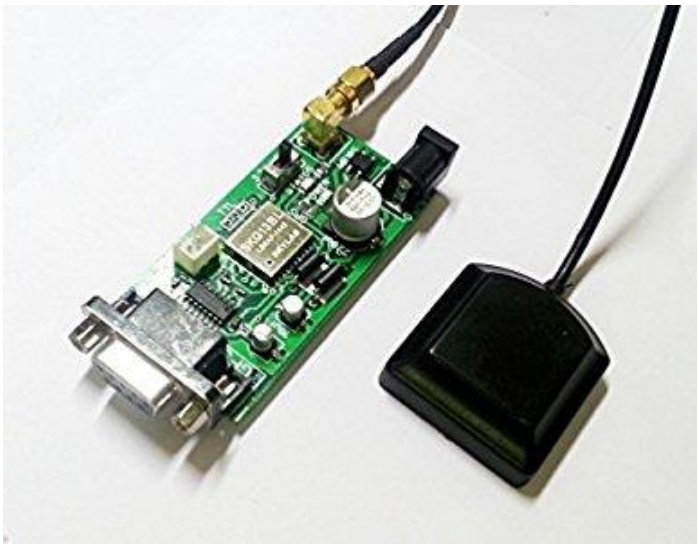


Figure 4.4 GPS module and GPS Antenna

Applications

- LBS (Location Based Service)
- PND (Portable Navigation Device)
- Vehicle navigation system
- Mobile phone
- Extremely fast TTFF at low signal level

4.2.2 Interfacing of GPS Module

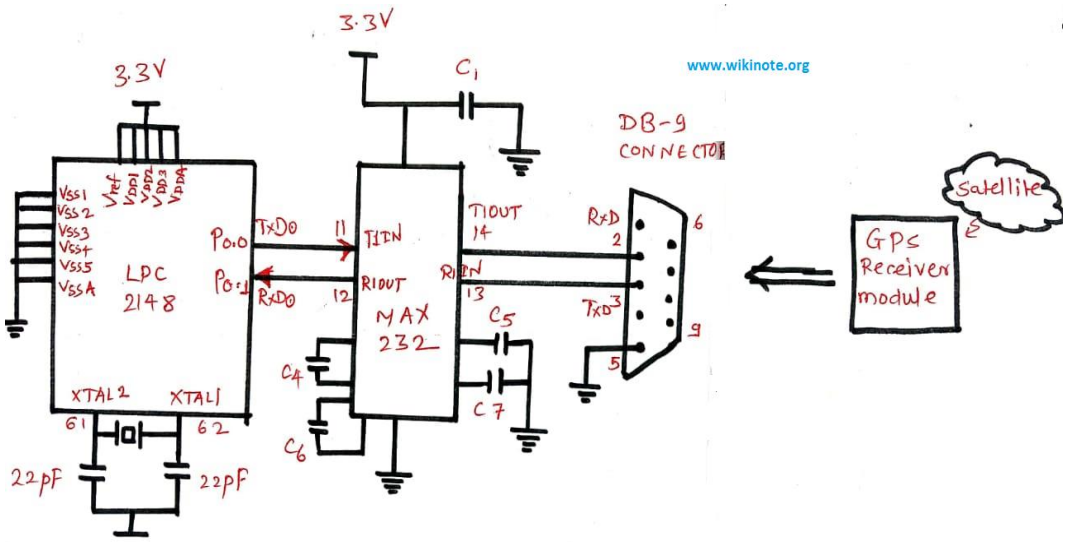


Figure 4.5 Interfacing Circuit for GPS receiver module

Table 4.3 Pin assignment with LPC 2148

	UART DB-9 Connector	LPC2148 Processor Lines
UART0 (P1) ISP PGM	TXD-0	P0.0
	RXD-0	P0.1
UART1 (P2)	TXD-1	P0.8
	RXD-1	P0.9

Algorithm for GPS module interfacing with LPC2148

1) Start

2) Initialise UART0 or UART1 serial interface using following instruction

```
PINSEL0=0X0000 0005;//Enable P0.0-TxD0,P0.1-RxD0
U0LCR=0X83;  //8-BIT Character lenth,NO parity,1 stop bit
U0DLL=97; //Baud rate=9600@PCLK=15Mhz
U0LCR=0X03;//Dlab=0
```

3) Receive GPS Message of location and longitude through UART module using function UARTGetch()

4) Store single character in Variable GPSData

GPSDATA=Uart0Getch();

5) Copy each single received character in array lattitude and longitude

6) Send this array characters to LCD for displaying message

7) END

4.2.3 Example Program for GPS Interfacing

```
#include <lpc214x.h>
#include "serial.h"
#include "lcd.h"
unsigned int j;
unsigned char Gpsdata;            // for incoming serial data
unsigned int finish =0;           // indicate end of message
unsigned int pos_cnt=0;           // position counter
unsigned int lat_cnt=0;           // latitude data counter
unsigned int log_cnt=0;           // longitude data counter
unsigned int flg    =0;           // GPS flag
unsigned int com_cnt=0;           // comma counter
unsigned char lat[20];            // latitude array
unsigned char lg[20];             // longitude array
unsigned int i=0;
unsigned int fg=0;;
void gps(void);
int main(void)
{
 lcd_init();
 Uart0Init();
while(1)
 {
   gps();
   lcdcmd(0x80);
   DisplayLCD1("LT:");
  DisplayLCD1(lat);
  DisplayLCD1("N");
   lcdcmd(0xC0);
   DisplayLCD1("LG:");
  DisplayLCD1(lg);
  DisplayLCD1("E");
 }
}
```

```
void gps()
{
  while(finish==0){

    Gpsdata = Uart0GetCh();
      flg = 1;
    if( Gpsdata=='$' && pos_cnt == 0)   // finding GPRMC header
      pos_cnt=1;
    if( Gpsdata=='G' && pos_cnt == 1)
      pos_cnt=2;
    if( Gpsdata=='P' && pos_cnt == 2)
      pos_cnt=3;
    if( Gpsdata=='R' && pos_cnt == 3)
      pos_cnt=4;
    if( Gpsdata=='M' && pos_cnt == 4)
      pos_cnt=5;
    if( Gpsdata=='C' && pos_cnt==5 )
      pos_cnt=6;
    if(pos_cnt==6 &&  Gpsdata ==','){   // count commas in message
     com_cnt++;
      flg=0;
     }

    if(com_cnt==3 && flg==1){
     lat[lat_cnt++] =  Gpsdata;        // latitude
    flg=0;
     }

    if(com_cnt==5 && flg==1){
      lg[log_cnt++] =  Gpsdata;        // Longitude
     flg=0;
     }

    if( Gpsdata == '*' && com_cnt >= 5 && flg == 1){
      lat[lat_cnt] ='\0';             // end of GPRMC message
     lg[log_cnt]  = '\0';
   com_cnt = 0;                       // end of GPRMC message
     lat_cnt = 0;
      log_cnt = 0;
      flg     = 0;
      finish  = 1;
```

```
    }
   }

  finish = 0;
  pos_cnt = 0;
}
///////$GPRMC,194530.000,A,3051.8007,N,10035.9989,W,1.49,111.67,310714,,,A*74
```

Note:- As we need to send AT Commands using UART , we need to add Program for Serial Communication. Hence Add Serial.c and Serial.h file
Note:- As we want to display Lattitude and Longitude values on LCD we have to add LCD.c and LCD.h files in our keil Project

4.3 LPC2148 INTERFACING WITH ON-CHIP (INTERNAL) ADC

Analog to Digital Converter (ADC) is used to convert analog signal/voltage into its equivalent digital number so that microcontroller can process that numbers and make it human readable. The ADC characterized by resolution. The resolution of ADC indicates the number of digital values. Let's take example: In LPC2148 microcontroller we have in-built 10-bit ADC. So for 10-bit ADC resolution is 10-bit and maximum value will be $\mathbf{2^{10}=1024}$. This means our digital value or discrete level lies between **0 to 1023**. There is one more term important to understand while dealing with ADC and it is step size. **Step size** is the minimum change in input voltage which can be resolved by ADC. The concept of step size is closely associated with the resolution of ADC.

$$\text{Step Size} = \frac{Input\ Range}{Resolution} = \frac{3.3V}{2^{10}}$$

$$= \frac{3.3}{1023} = 0.0032258064\,V$$

$$= 3.23\,\text{mV (Approx.)}$$

So in this case we can measure minimum **2.23 mV (Approx.)** with our microcontroller. This is how step size defines an accuracy of ADC circuit.

4.3.1 Features of ADC

- 2 internal ADC's - ADC0 (6 Channel), ADC1 (8 Channel)
- Type: 10-bit, Successive Approximation type,
- Supports burst mode (repeated conversion at 3-bit to 10-bit resolution)
- Supports simultaneous conversion on both ADC's

- Conversion time: 2.44 micro-seconds
- Start of Conversion by software control / on timer match /transition on a pin
- Range: 0 V – VREF (+3.3 V)
- Max. clock frequency is 4.5 MHz, (by programming ADC Control (ADxCON Register)

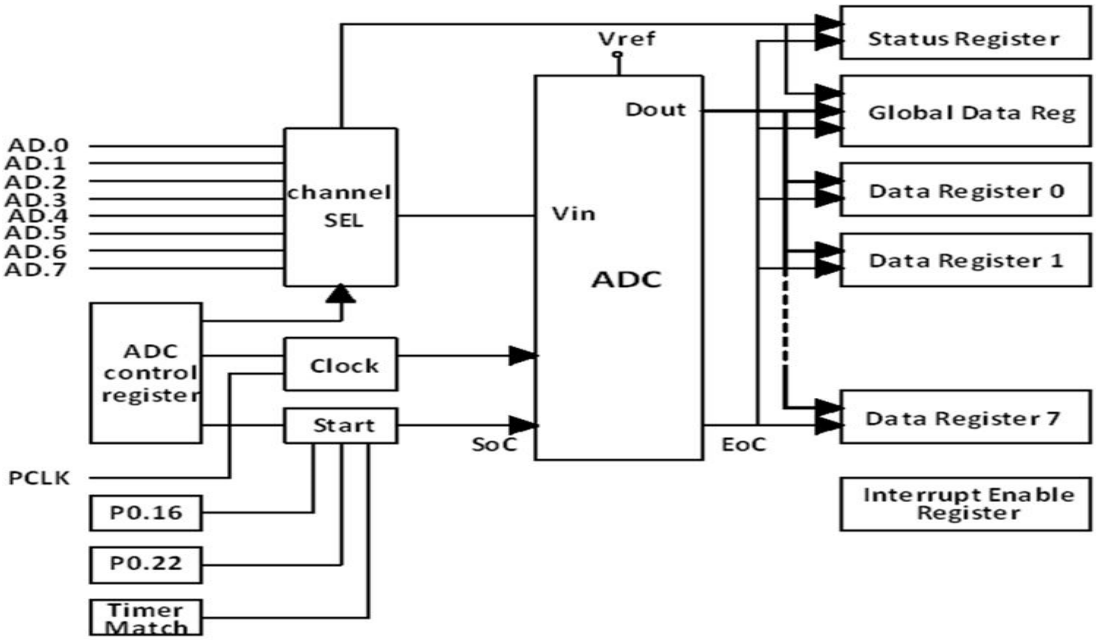


Figure 4.6 On-Chip ADC in LPC2148-Internal Diagram

Table 4.4 Pin Assignment for ADC in LPC2148

Block	**Symbol**	**Description**	**I/O**
ADC0	AD0.1	Channel 1	P0.28
	AD0.2	Channel 2	P0.29
	AD0.3	Channel 3	P0.30
	AD0.4	Channel 4	P0.25
	AD0.6	Channel 6	P0.4
	AD0.7	Channel 7	P0.5
ADC1	AD1.0	Channel 0	P0.6

	AD1.1	Channel 1	P0.8
	AD1.2	Channel 2	P0.10
	AD1.3	Channel 3	P0.12
	AD1.4	Channel 4	P0.13
	AD1.5	Channel 5	P0.15
	AD1.6	Channel 6	P0.21
	AD1.7	Channel 7	P0.22

4.3.2 ADC REGISTERS

1. ADxCON - ADC Control Register-32-bit register

- Useful for Selection of analog input channel, clock frequency to ADC, Resolution, conversion mode, method of issue of SoC, edge for conversion

Table 4.5 ADC Register Configuration

RESERVED	EDGE	START	--	PDN	--	CLKS	BURST	CLKDIV	SEL
31-28	27	26-24	23-22	21	20	19-17	16	15-8	7-0

Bit	Symbol	Description
7-0	SEL (Channel Selection bits)	**Select field:-** Selects which of the AD0.7:0/AD1.7:0 pins is (are) to be sampled and converted. For AD0, bit 0 selects Pin AD0.0, and bit 7 selects pin AD0.7. In software-controlled mode, only one of these bits should be 1. In hardware scan mode, any value containing 1 to 8 can be one
15-8	CLKDIV:	**Clock Division factor Value:-** The APB clock (PCLK) is divided by (this value plus one) to produce the clock for the A/D

<table>
<tr><td></td><td></td><td>converter, which should be less than or equal to 4.5 MHz Typically, software should program the smallest value in this field that yields a clock of 4.5 MHz or slightly less, but in certain cases (such as a high-impedance analog source) a slower clock may be desirable.</td></tr>
<tr><td>16</td><td>BURST</td><td>0; ADC will not perform Repeated A to D Conversion 1; ADC will perform Repeated A to D Conversion The AD converter does repeated conversions at the rate selected by the CLKS field, scanning (if necessary) through the pins selected by 1s in the SEL field. The first conversion after the start corresponds to the least-significant 1 in the SEL field, then higher numbered 1-bits (pins) if applicable. Repeated conversions can be terminated by clearing this bit, but the conversion that’s in progress when this bit is cleared will be completed.
Remark: START bits must be 000 when BURST = 1 or conversions will not start.</td></tr>
<tr><td>19-17</td><td>CLKS</td><td>Clocks:- This field selects the number of clocks used for each conversion in Burst mode, and the number of bits of accuracy of the result in the RESULT bits of ADDR, between 11 clocks (10 bits) and 4 clocks (3 bits).<table><tr><th>CLKS field - 19-18-17</th><th>No. of Clock cycles used per bit conversion</th></tr><tr><td>000</td><td>11 clocks cycles / 10 bit conversion</td></tr><tr><td>001</td><td>10 clocks/ 9 bits</td></tr><tr><td>010</td><td>9 clocks/ 8 bits</td></tr><tr><td>011</td><td>8 clocks/ 7 bits</td></tr></table></td></tr>
</table>

<table>
<tr><td></td><td></td><td><table>
<tr><td>100</td><td>7 clocks/ 6 bits</td></tr>
<tr><td>101</td><td>6 clocks/ 5 bits</td></tr>
<tr><td>110</td><td>5 clocks/ 4 bits</td></tr>
<tr><td>111</td><td>4 clocks/ 3 bits</td></tr>
</table></td></tr>
<tr><td>21</td><td>PDN</td><td>Power Down
PDN=1 The A/D converter is operational.
PDN=0 The A/D converter is in power-down mode.</td></tr>
<tr><td>26-24</td><td>START</td><td><table>
<tr><td>START field - 26-25-24</td><td>Description</td></tr>
<tr><td>000</td><td>No start of Conversion</td></tr>
<tr><td>001</td><td>Start of Conversion Now</td></tr>
</table></td></tr>
<tr><td>27</td><td>Edge</td><td>(In use only when START field contains Values from 010 TO 111)</td></tr>
</table>

- **15-8 CLKDIV**: The APB clock (PCLK) is divided by (this value plus one) to produce the clock for the A/D converter, which should be less than or equal to 4.5 MHz Typically, software should program the smallest value in this field that yields a clock of 4.5 MHz or slightly less, but in certain cases (such as a high-impedance analog source) a slower clock may be desirable.
    - The A/D Converters on the LPC2148 is also called as The conversion speed is selectable by the user.
    - A/D Clock frequency= [Pclk/(CLKDIV+1)] .....<=4.5 MHz

2. A/D Global Start Register (ADxGSR)

- Used to initiate simultaneous conversion on both ADCs

3. A/D Status Register (ADxSTAT)

- Allows simultaneous checking of status of all A/D channels

- Contains done, overrun, interrupt flags

5. A/D Data Registers (ADR0 – ADR7)

- Contains most recent converted data and EoC (Done) status on respected channel

Table 4.6 ADC DATA Register Configuration

DONE	OVERRUN	Reserved	10 bit A/D RESULT	Reserved
31	30	29-16	15-6	5-0

6. Global Data Register

- Contains done bit, most converted data, channel number

Table 4.7 ADC Global Data Register Configuration

DONE	OVERRUN	Reserved	Channel Selection	Reserved	10 bit A/D RESULT	Reserved
31	30	29-28-27	26-25-24	23-16	15-6	5-0

- **DONE (Bit 31)**
  - DONE= 1 ;when an A/D conversion is complete.
  - D0NE=0 ;A/D conversion is in progress

For accurate results, you need to wait until this value is 1 before reading the RESULT bits. (Please note that this value is cleared when you read this register.)

- **OVERRUN (Bit 30)**

While not relevant to the examples used in this tutorial, this value with be 1 if the results of one or more conversions were lost when converting in BURST mode. See the User's Manual for further details. (As with DONE, this bit will be cleared when you read this register.)

- **RESULTS (Bits 15..6)**

If DONE is 1 (meaning the conversion is complete), these 10 bits will contain a binary number representing the results of our analog to digital conversion. It works by measuring the voltage on the analog input pin divided by the voltage on the Vref pin.

Table 4.8 Analog value and its digital equivalent

Analog Input	10-bit Digital output	Digital Output in HEX
0V	0000 0000 00 B	000H
3.3V	1111 1111 11 B	3FFH

Zero means that the voltage on the analog input pin was less than, equal to or close to GND (Vssa), and 0x3FF (or 0011 1111 1111) indicates that the voltage on the analog input pin was close to, equal to or greater than the the voltage on the Vref pin. Anything value between these two extremes will be returned as a 10-bit number (between 0 and 1023).

6. Interrupt Enable Register

- Enables interrupt on EOC channel
- Programming ADC registers – Examples (Construction of control words

4.3.3 ADC Design Example

Select ADC-0, Channel-1, Clock frequency 3.75 MHz (let PCLK is 15 MHz), burst mode repeated conversion) and 10-bit resolution. Power-up ADC and issue start of conversion.

Solution: AD0CR = 0x01210302; // configure SEL, CLKDIV, BURST CLKS & PDN bit fields set START, signal start of conversion

1. Select ADC–1, Channels 0 to 7, clock frequency 4.5 MHz (assume PCLK is 30 MHz), burst mode repeated conversion, 8-bit resolution.

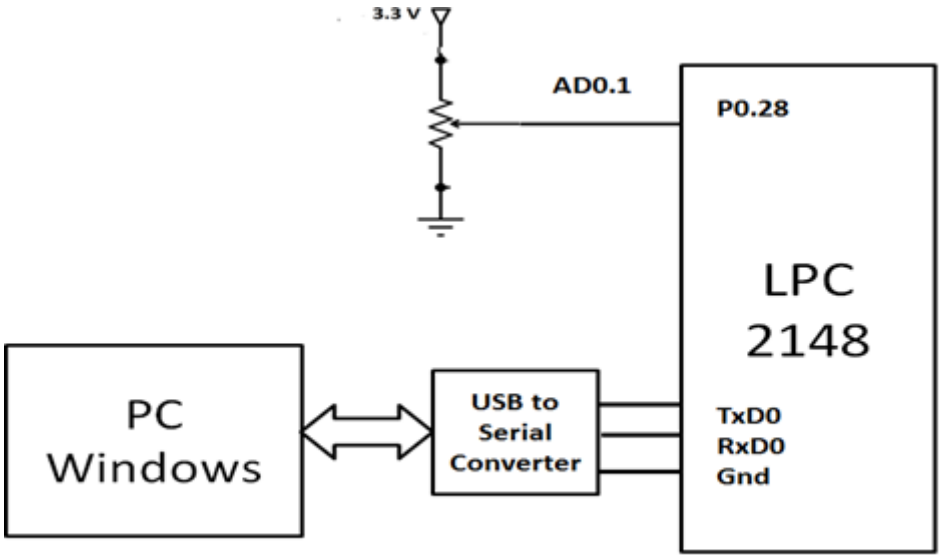


Figure 4.7 On-chip interfacing with Peripherals

a) C Program for on-chip ADC using interrupt

```
#include <lpc214x.h>
#include "serial.h"
#include <stdio.h>
void delay(void);
void ADC_ISR(void) __attribute__ ((interrupt("IRQ")));
int adcdata;
float voltage;
unsigned char volt[3];
int i;
int main(void)
{
 PINSEL0 = 0x00000005;
 PINSEL1 = 0x01000000;
 PINSEL2 = 0x00000000;

 Uart0Init();
   Uart0PutS("\n ADC o/p : ");
```

```
 AD0INTEN = 0x00000002;       ///On completion of AD conversion channel1 will generate
an Interrupt
 VICVectAddr0 = (unsigned int)ADC_ISR;
 VICVectCntl0 = 0x20 | 18;    //// VIRQ and Assign AD0 interrupt Slot0
VICIntEnable = 1 << 18;    ///Enable AD0 interrupt channel of VIC

 AD0CR = 0X01200402; // Channel AD0.1 , Clock 3Mhz, Burst Mode, 11 clocks per 10 bit ,
    //AD conversion is operational, start conversion

while(1){

  }

return 0;
}

void ADC_ISR()
{
if(AD0DR1 & 0x80000000)    ///Monitor EOC bit from AD Data Register of Channel0
{
  adcdata=(AD0DR1 & 0x0000FFC0);
  adcdata=adcdata>>6;         ///Right shift Digital Result by 6 bits
 voltage=((adcdata/1023.0)*3.3);
  sprintf(volt, "%.1f", voltage);      ////Buffer, decimal value. 1 digit fractional value, float
volatage value
 Uart0PutS(volt);    ///print buffer on Hyperterminal
}
 delay();
 AD0INTEN = 0;       ////Disable ADO Interrupr
VICVectAddr=0;   ///End of ISR
}
void delay(void)
{
int i,j;
for(i=0;i<1000;i++)
for(j=0;j<10000;j++);
}
```

b) Embedded C Program for on-chip(Internal ADC) without Interrupt

```
#include<lpc214x.h>
#include<stdio.h>
#include "serial.h"
void delay(void);
```

```
int main()
{
int adcdata;
float voltage;
unsigned char volt[3];
 PINSEL0=0X00000000;
 PINSEL1=0X01000000; //Select P0.28 pin function as Analog i/p
Uart0Init();
 AD0CR=0x00210402; ///CHANNEL1 OF ADC0, ad freq=3MHz,
while(1)
 {
 if(AD0DR1 & 0x80000000) ////EOC bit monitoring
 {
   adcdata=(AD0DR1 & 0x0000FFC0);
   adcdata=adcdata>>6;
   voltage=((adcdata/1023.0)*3.3);
   sprintf(volt, "%.1f", voltage);  ADC o/p=1.2
   Uart0PutS("\n ADC o/p : ");
   Uart0PutS(volt);
   delay();
  }
 }
}
void delay(void)
{
int i,j;
for(i=0;i<1000;i++)
for(j=0;j<10000;j++);
}
```

4.4 Serial Communication Using UART in LPC2148

The characteristics of UART hardware in LPC2148 controller and its associated registers is briefly discussed in this section. The important features of UART hardware in LPC2148 are:

- UART1 is identical to UART0, with the addition of a modem interface.
- 16 byte Receive and Transmit FIFOs.
- Register locations conform to ‘550 industry standard.
- Receiver FIFO trigger points at 1, 4, 8, and 14 bytes.
- Built-in fractional baud rate generator with autobauding capabilities.
- Mechanism that enables software and hardware flow control implementation.
- Standard modem interface signals included with flow control (auto-CTS/RTS) fully

- supported in hardware (LPC2144/6/8 only).

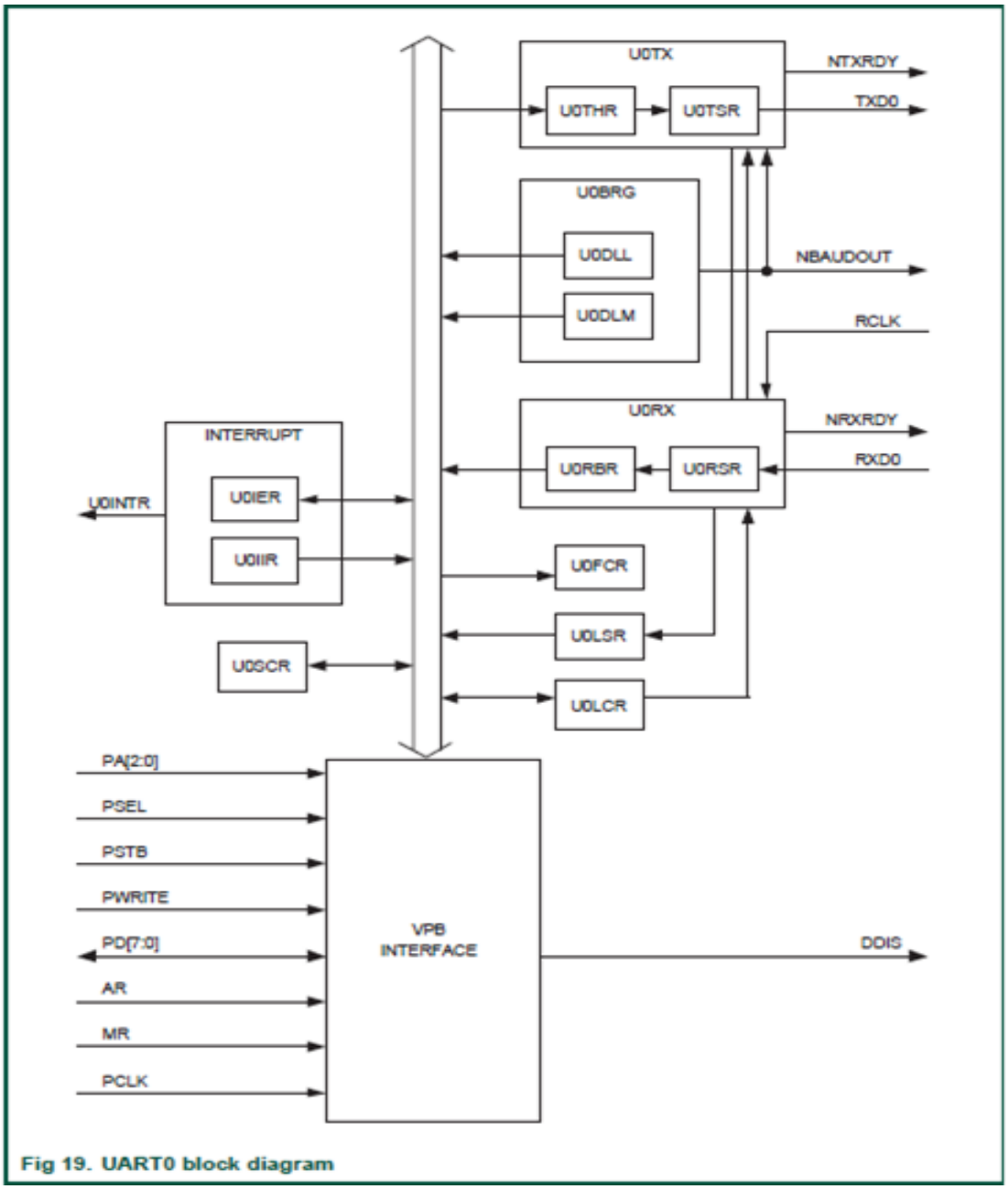


Figure 4.8 UART0 Architecture in LPC2148

U0FCR (FIFO Control Register)

- 8-BIT Byte Addressable register
- This reg is used to enable TX & RX FIFO functionalities
- U0FCR=0x07 is like SCON reg

Table 4.9 U0FCR (FIFO Control Register) bit assignment

U0FCR	**FIFO Control Register**	**-**	**-**	**-**	**-**	**-**	**TX FIFO Reset**	**RX FIFO Reset**	**FIFO Enable**

U0LCR (Line Control Register)

- 8-BIT byte addressable register

Table 4.10 U0LCR (Line Control Register)bit assignment

UART0 Line Control Register (U0LCR - address 0xE000 C00C) bit description				
Bit	Symbol	Value	Description	Reset Value
1:0	Word Length Select	00	5 bit character length	0
		01	6 bit character length	
		10	7 bit character length	
		11	8 bit character length	
2	Stop Bit Select	0	1 stop bit	0
		1	2 stop bits (1.5 if U0LCR[1:0]==00)	

3	Parity Enable	0	Disable parity generation and checking	0
		1	Enable parity generation and checking	
5:4	Parity Select	00	Odd parity. Number of 1s In the transmitted character and the attached parity bit will be odd.	0
		01	Even Parity. Number of is in the transmitted character and the attached parity bit will be even.	
		10	Forced "1" stick parity.	
		11	Forced "0" stick parity.	
6	Break Control	0	Disable break transmission	0
		1	Enable break transmission. Output pin UARTO TXD Is forced to logic 0 when UOLCR[6] Is active high.	
7	Divisor Latch Access Bit (DLAB)	0	Disable access to Divisor Latch	0
		1	Enable access to Divisor Latch	

DLAB (Divisor Latch Buffer)

One high-low pulse across DLAB bit indicates baud rate is successfully loaded.

- **DLAB=1** baud rate is loading
- **DLAB=0** After loading baud rate DLAB must be zero.

U0LSR (Line Status Register)

- 8-bit byte addressable register
- Consists of different flag bits, TI interrupt & RI interrupt flag bit

Table 4.11 U0LSR (Line Status Register) bit assignment

UART0 Line Status Register				
Bit	Symbol	Value	Description	Reset value
0	Receiver Data Ready (RDR)		U0LSR0 is set when the U0RBR holds an unread character and is cleared when the UART0 RBR FIFO is empty.	0
		0	U0RBR is empty.	
		1	U0RBR contains valid data.	
1	Overrun Error (OE)		The overrun error condition is set as soon as it occurs. An U0LSR read clears U0LSR1. U0LSR1 is set when UART0 RSR has a new character assembled and the UART0 RBR FIFO is full. In this case, the UART0 RBR FIFO will not be overwritten and the character in the UART0 RSR will be lost.	0
		0	Overrun error status is inactive.	
		1	Overrun error status is active.	
			When the parity bit of a received character is in the wrong state, a parity error occurs. An U0LSR read clears U0LSR[2]. Time of	

2	Parity Error		parity error detection is dependent on U0FCR(0). **Note**: A parity error is associated with the character at the top of the UART0 RBR FIFO.	0
		0	Parity error status is Inactive.	
3	Framing Error (FE)		When the stop bit of a received character is a logic 0. a framing error occurs. 0 An U0LSR read dears U0LSR[3]. The time of the framing error detection is dependent on U0FCR0. Upon detection of a framing error, the Rx will attempt to resynchronize to the data and assume that the bad stop bit is actually an early start bit. However, it cannot be assumed that the next received byte will be correct even if there is no Framing Error. **Note**: A framing error is associated with the character at the top of the UART0 RBR FIFO.	0
		0	Framing error status is inactive.	
		1	Framing error status is active.	
4	Break Interrupt (BI)		When RXD0 is held in the spacing state (all 0's) for one full character transmission (start, data, parity, stop), a break interrupt occurs. Once the break condition has been detected, the receiver goes idle until RXD0 goes to marking state (all 1s). An U0LSR read clears this status bit. The time of break detection is dependent on U0FCR(0). **Note**: The break interrupt is associated with the character at the top of the UART0 RBR FIFO.	0
		0	Break interrupt status is inactive.	

		1	Break interrupt status is active.	
5	Transmitter Holding Register Empty (THRE)		THRE is set immediately upon detection of an empty UART0 THR and is cleared on a U0THR write.	1
		0	U0THR contains valid data.	
		1	U0THR is empty.	
6	Transmitter Empty (TEMT)		TEMT is set when both U0THR and U0TSR are empty; TEMT is cleared when either the U0TSR or the U0THR contain valid data.	1
		0	U0THR and/or the U0TSR contains valid data.	
		1	U0THR and the U0TSR are empty.	
7	Error in RX FIFO (RXFE)		UOLSR(7) is set when a character with a Rx error such as framing error, parity error or break interrupt, is loaded into the U0RBR. This bit is cleared when the U0LSR register is read and there are no subsequent errors in the UART0 FIFO.	0
		0	U0RBR contains no UART0 RX errors or U0FCR[0]=0.	
		1	UART0 RBR contains at least one UART0 RX error.	

DLR (Divisor Latch Register)

- DLR is 16-bit register
- Used to load baud rate
- As the baud rate is 8-bit value, divide DLR into two parts DLM & DLL (8-bit each)

For 9600 baud rate
U0DLL=0x63; //(Pclk=12Mhz)
U0DLM=0x00

U0DLL:U0DLM=[Pclk/16*Desired Baud rate]

U0THR (Transmit Hold Register)

- 8-bit byte addressable reg.
- Data can be loading to U0THR, whenever transmitting data

U0THR='A' //THR buffer register is used only for transmitting

U0RBR (UART0 Receive Buffer Register)

- 8-bit byte addressable reg.
- Data can be loading into U0RBR, whenever receiving data.
- a = U0RBR //RBR buffer register is used only for transmitting

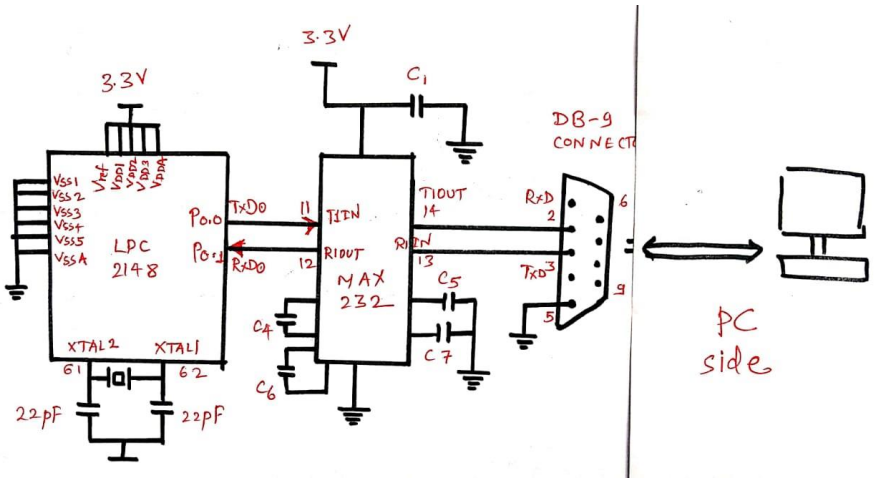


Figure 4.9 Circuit for serial communication with LPC2148 and PC

4.4.2 Algorithm for UART serial communication

1) Start
2) Initialise UART0 serial interface using following instruction

PINSEL0=0X0000 0005;//Enable P0.0-TxD0,P0.1-RxD0

U0LCR=0X83; //8-BIT Character lenth,NO parity,1 stop bit, DLAB=1
U0DLL=97; //Baud rate=9600@PCLK=15Mhz
U0LCR=0X03;//DLAB=0
3) LPC2148 will receive characters transmitted by PC

4) LPC2148 will transmit the characters received back to PC

3) Transmit different AT commands through UART module using instruction
while(!(U0LSR&0X20));//Monitor TI flag

4) If transmission buffer is Empty,Transmit single character at a time
U0THR=ch;

5) Provide delay while transmitting each command
6) To transmit a single character use PUTCH function & to transmit a string use PUTS function
7) END

4.4.3 Embedded C program for Serial Transmission and Reception

```
#include<lpc21xx.h>   //Includes LPC2148 register definitions

void Uart0Init (void)      // Initialize Serial Interface
{
    PINSEL0 = 0x00000005;          //Enable RxD0 and TxD0
   U0LCR = 0x83;                 // 8 bits, no Parity, 1 Stop bit
   U0DLL = 97;                   // 9600 Baud Rate @ 15MHz PCLK
   U0LCR = 0x03;       // DLAB = 0
}

void Uart0PutCh (unsigned char ch)  // Write character to Serial Port
{
    U0THR = ch;
  while (!(U0LSR & 0x20));
}
```

```
void  Uart0PutS(unsigned char *str)  //A function to send a string on UART0
{
while(*str)
{
    Uart0PutCh(*str++);
}
}
unsigned char Uart0GetCh (void)  // Read character from Serial Port
{
 while (!(U0LSR & 0x01));
 return (U0RBR);
}
int main()
{
unsigned char a;
Uart0Init();
while(1)
{
a=Uart0GetCh();
Uart0PutCh(a);
}
}
```

4.5 LPC2148 INTERFACING WITH EEPROM USING I2C

I2C is a two-wire synchronous serial communication protocol. SDA line is used for transferring data and SCK is used for transferring clock information. Every device connected to an I2C bus has a unique address.

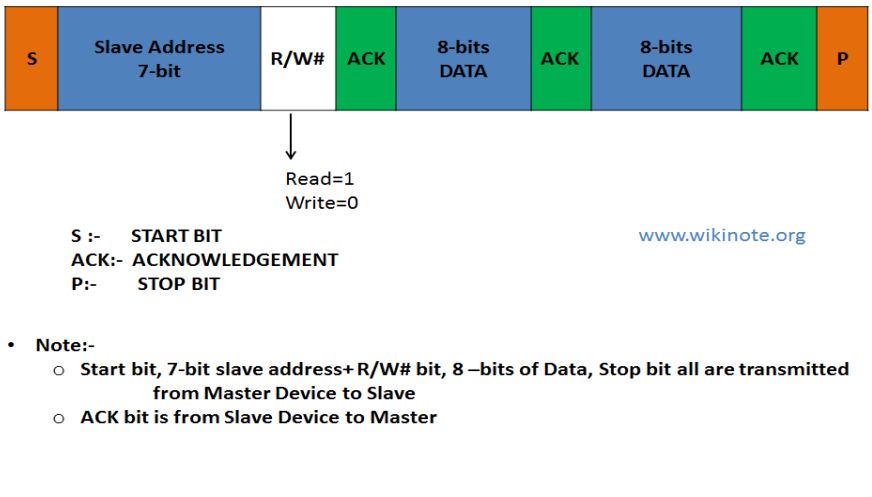


Figure 4.10: I2C frame format

I2C communication protocol involves communication between a slave and a master. The device which initiates the communication and which provides the clock is referred to as a master device. The devices which receive the clock signal and receive/transmit data according to the clock signal is termed as a slave device. Each device on the bus is accessed using its slave address.

START condition

- **STEP-1)** First the MCU will issue a START condition. The devices connected to the bus will listen to the START condition and will stay ready to begin the communication process.
- **STEP-2)** Then MCU will send the address of the device with which it needs to communicate. Master indicates the action to be performed with the device whether to read or write along with the address.
- **STEP-3)** All devices connected to the bus will receive the address and will compare it with its own address. If the addresses match with each other, the device will send back an ACKNOWLEDGEMENT signal to the master device. If they don't match they will simply wait for the bus to be released with a STOP condition.
- **STEP-4)** Once the MCU sends the address and corresponding device acknowledges, the MCU can start transmitting or receiving data.
- **STEP-5)** When the data transmission or reception is complete, the MCU will stop communicating by sending a STOP condition.

STOP condition

- **STEP-6)** STOP condition indicates that the bus is released and it can be used by any other master (if any) connected to the I2C bus.

- After a master generate a start condition I2C bus will solely belong to it. The bus will be freed only if the master generate a STOP condition. Any other master connected to the bus can access the bus after a STOP is identified on the bus.
- If the master device which uses the bus needs to communicate with a different slave it should generate a RESTART. Instead if it tries to stop current communication and then start again it may lose access to the bus. RESTART is nothing but a start signal without a stop in the bus.

4.5.1 Features of I2C module in LPC2148

- Two fast I2C buses (I2C0, I2C1)
- Standard I2C compliant bus interfaces that may be configured as Master, Slave, or Master/Slave.
- Arbitration between simultaneously transmitting masters without corruption of serial data on the bus.
- Programmable clock to allow adjustment of multiple I2C data transfer rates.
    - Standard- 100 kbps
    - Fast- 400 kbps
    - High Speed- 3.4 Mbps
- Bidirectional data transfer between masters and slaves.
- Serial clock synchronization allows devices with different bit rates to communicate via one serial bus.
- Serial clock synchronization can be used as a handshake mechanism to suspend and resume serial transfer.
- The I2C bus may be used for test and diagnostic purposes.

Applications

Interfaces to external I2C standard parts

- Serial RAMs, ROMs
- LCDs
- Tone generators

Table 4.12 Pin Description for I2C communication

Pin	Type	Description	LPC2148 Pins
SDA0/1	Input/Output	I2C Serial Data	P0.3 and P0.14
SCL0/1	Input/Output	I2C Serial Clock	P0.2 and P0.11

Table 4.13 I2C Registers

Generic Name	Description	Access	Reset value	I2Cn Register name & Address
I2CONSET	**I2C Control Set Register.** When a one is written to a bit of this register. the corresponding bit in the I2C control register is set. Writing a zero has no effect on the corresponding bit in the I2C control register.	R/W	0x00	I2C0CONSET - 0xE001 C000 I2C1CONSET - 0xE005 C000
I2STAT	**I2C Status Register.** During I2C operation, this register provides detailed status codes that allow software to determine the next action needed.	RO	0xF8	I2C0STAT - 0xE001 C0004 I2C1STAT - 0xE005 C004
I2DAT	**I2C Data Register.** During master or slave transmit mode. data to be transmitted is written to this register. During master or slave receive mode, data that has been received may be read from this register.	R/W	0x00	I2C0DAT - 0xE001 C008 I2C1DAT - 0xE005 C008
I2ADR	**I2C Slave Address Register.** Contains the 7 bit slave address for operation of the I2C interface in slave mode. and is not used in master mode. The least significant bit determines	R/W	0x00	I2C0ADR - 0xE001 C00C I2C1ADR - 0xE005 C00C

	whether a slave responds to the general call address.			
I2CSCLH	**SCH Duty Cycle Register High Half Word.** Determines the high time of the RC clock.	R/W	0x04	I2C0SCLH - 0xE001 C010 I2C1SCLH - 0xE005 C010
I2CSCLL	**SCL Duty Cycle Register Low Half Word.** Determines the low time of the 12C clock. I2nSCLL and I2nSCLH together determine the clock frequency generated by an I2C master and certain times used in slave mode.	R/W	0x04	I2C0SCLL - 0xE001 C014 I2C1SCLL - 0xE005 C014
I2CONCLR	**I2C Control Clear Register.** When a one is written to a bit of this register. the corresponding bit in the I2C control register is cleared. Writing a zero has no effect on the corresponding bit in the PC control register.	WO	NA	I2C0CONCLR - 0xE001 C018 I2C1CONCLR - 0xE005 C018

Table 4.14 I2CxCONSET Register

Bit	Symbol	Description
0-1	--	Reserved

2	AA	Assert Acknowledge AA=1; request an acknowledge
3	SI	I2C Serial Interrupt SI=1; indicate state change
4	STO	STOP STO=1; sends stop condition
5	STA	START STA=1; sends START condition
6	I2CEN	I2CEN=1; I2C interface enable
7	-	Reserved

4.5.2 Features of EEPROM IC (AT24C512)

- The AT24C512 provides 524,288 bits of serial electrically erasable and programmable read only memory (EEPROM) organized as 65,536 words of 8 bits each.
- The device's cascadable feature allows up to four devices to share a common two-wire bus.
- The device is optimized for use in many industrial and commercial applications where low power and low-voltage operation are essential.
- The devices are available in space saving8-pin PDIP, 8-lead EIAJ SOIC, 8-lead JEDEC SOIC, 8-lead TSSOP, 8-lead Leadless Array (LAP), and 8-lead SAP packages. In addition, the entire family is available in 2.7V (2.7V to 5.5V) and 1.8V (1.8V to 3.6V) versions.

Pin Configurations

Pin Name	Function
A0–A1	Address Inputs
SDA	Serial Data
SCL	Serial Clock Input
WP	Write Protect
NC	No Connect

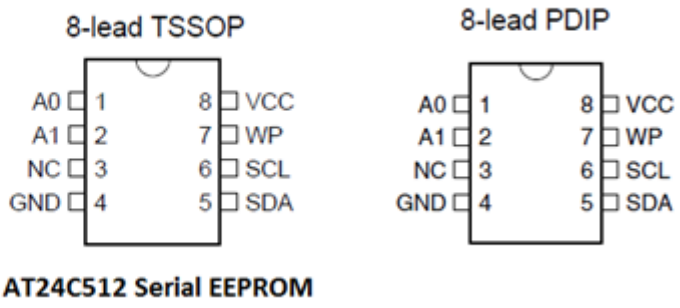


Figure 4.11 Pin details of EEPROM IC

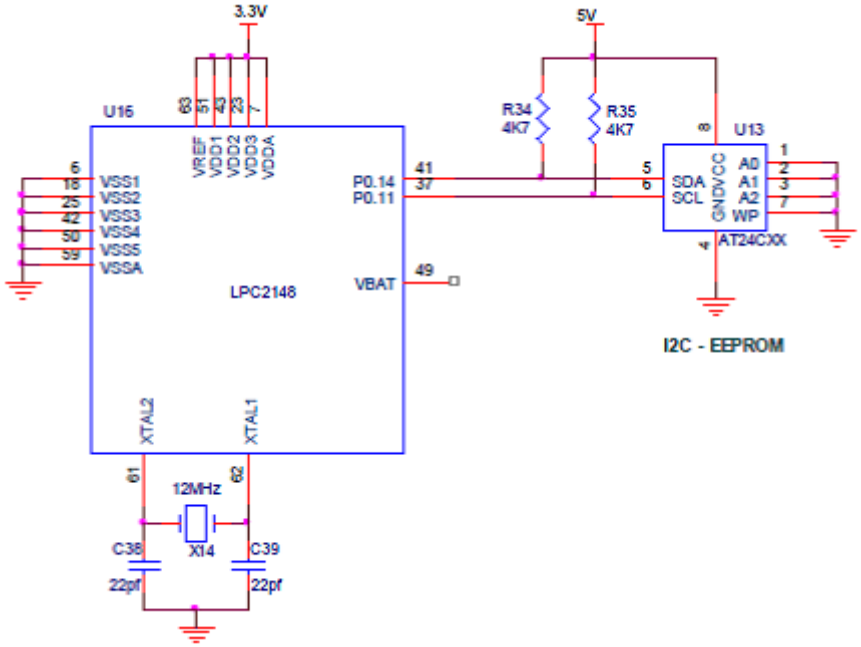


Figure 4.12 Interfacing EEPROM IC with LPC2148

4.5.3 Algorithm for the Interfacing EEPROM

1) Start

2) Initialize I2C bus interface

PINSEL0=0X10400050; //Configure P0.11-SCL1 & P0.14-SD1
I2CSCLH=150;
I2CSCLL=150; //SET I2C frequency=[Pclk/(I2CSCLL+I2CSCH)]

3) Transmit the slave address(Page address,Page offset,No. of bytes)

4) Enable I2C bus interface

I2CCONSET=0X40;////I2CEN=1

5) Master (LPC2148) will transmit START signal

I2CCONSET=0X20;//STA=1

6) Transmit slave address(7-bit address,R/W=0; write operation)

7) Wait for acknowledgement

8) Tansmit Page address and page offset at which data is to be written

9) Wait for acknowledment

10) Transmit data using I2CDAT register

11) Wait for acknowledge

12) After successful transmission of data , master wil transmit STOP condition

I2CCONSET=0X10;//STO=1

13) Disable I2C interface

I2CCONCLR=0X40; //I2CENC=1

14) END

Example Program

```
#include <LPC214x.h>
#include <stdio.h>
#include "serial.h"

#define EEPROM_Addr 0xA0     //device address
#define I2Cwrite 0x00   //LSB bit 0 (write)
#define I2Cread  0x01   //LSB bit 1 (read)

#define I2C_ENABLE  1 << 6     //I2C Enable bit
#define I2C_START 1 << 5     //Start Bit
#define I2C_STOP  1 << 4     //Stop Bit
#define I2C_SI  1 << 3     //I2C interrupt flag
#define I2C_AACK   1 << 2     //assert ACK flag
unsigned char write_array[10] = {11,12,13,14,15,16,17,18,19,20};
unsigned char read_array[10];
unsigned char val[4];
void I2CInit(void)
{
```

```
 PINSEL0 |= 0x00000050;       //P0.2 -> SCL0  P0.3 -> SDA0 I2C0CONCLR  =
I2C_ENABLE | I2C_START | I2C_STOP | I2C_SI | I2C_AACK; //clear all the bits in
CONTROL register
//set I2C clock to work at 100Khz
I2C0SCLH = 0x4B ;       //set the high time of i2c clock; (15mhz / 100khz / 2)
I2C0SCLL = 0x4B ;       //set the low time of i2c clock;

 I2C0CONSET = I2C_ENABLE ;     //enable the I2C Interface
}
void I2CStart(void)          //Function to initiate a start condition on the I2C bus
{
unsigned int status;
I2C0CONCLR = (I2C_START | I2C_STOP | I2C_SI | I2C_AACK);  // clear all the bits in
CONCLR register
I2C0CONSET = (I2C_ENABLE );           //Enable the I2C interface
I2C0CONSET = (I2C_START);          //set the STA bit
while(!((status=I2C0CONSET)& I2C_SI));      //wait till interrupt flag becomes set
}
void I2CStop(void)
{
unsigned int status;
I2C0CONCLR = I2C_START | I2C_SI | I2C_AACK;    //clear all bits
I2C0CONSET = I2C_STOP;         //set STOP bit
}
void I2Csend(unsigned char data)
{
unsigned int status;
I2C0DAT = data;
I2C0CONCLR = I2C_START | I2C_STOP ;      // clear start bit for next operation
I2C0CONCLR = I2C_SI;        // clear interrupt flag
while(!((status=I2C0CONSET)& I2C_SI));        //wait till interrupt flag becomes set
}

unsigned char I2Cget(void)
{
unsigned char data;
unsigned int status;

I2C0CONCLR = I2C_START | I2C_STOP;
I2C0CONCLR = I2C_SI;        // clear interrupt flag
I2C0CONSET = I2C_AACK;           // send ack to continue further data transfer
while(!((status=I2C0CONSET)& I2C_SI));     //wait till interrupt flag becomes set
data = I2C0DAT;
return data;
```

```
}


int main()
{
unsigned int i,j;
Uart0Init();       //initialize UART with 9600 baudrate
Uart0PutS("\r\nI2C EEPROM\r\n");
I2CInit();        //initialize I2C




/* Write Sequence */
Uart0PutS("\r\n Writing Data.....\r\n");
I2CStart();       //Assert START
I2Csend(EEPROM_Addr | I2Cwrite);  //Device address with LSB bit 0
I2Csend(0x13);         //Address higher byte
I2Csend(0x49);      //Address lower byte
for(i=0;i<10;i++)
 I2Csend(write_array[i]);   //write the array to EEPROM
I2CStop();
    //Assert STOP
for(i=0;i<10;i++)
{
 sprintf(val,"%d",write_array[i]);    //display read data
Uart0PutS(val);
 Uart0PutS("\r\n");
}
/* Read Sequence */
Uart0PutS("\r\n Reading.....\r\n");
I2CStart();        //Assert START
I2Csend(EEPROM_Addr | I2Cwrite);  //Device address with LSB bit 0 (Dummy Write)
I2Csend(0x13);      //Address higher byte
I2Csend(0x49);       //Address lower byte
I2CStart();       //Assert Restart
I2Csend(EEPROM_Addr | I2Cread);  //Device address with LSB bit 1
for(i=0;i<10;i++)
 read_array[i] = I2Cget();   //Read EEPROM
I2CStop();       //Assert STOP


/*Display Write and Read Data*/
```

```
for(i=0;i<10;i++)
{
 sprintf(val,"%d",read_array[i]);    //display read data
Uart0PutS(val);
 Uart0PutS("\r\n");
}
while(1);      //stop here forever
return 0;
}
```

4.6 SD CARD INTERFACING WITH LPC2148

4.6.1 Features of SPI Module in LPC2148

- Single complete and independent SPI controller.
- Compliant with Serial Peripheral Interface (SPI) specification.
- Synchronous, Serial, Full Duplex Communication.
- Combined SPI master and slave.
- Maximum data bit rate of one eighth of the input clock rate.
- 8 to 16 bits per transfer

Table 4.15 SPI Pin Description in LPC2148

Pin Name	Type	Pin Description	LPC2148 Pins
SCK0	Input / Output	Serial Clock	P0.4
SSEL0	Input	Slave Select	P0.7
MISO0	Input / Output	Master In Slave Out	P0.5
MOSI0	Input / Output	Master Out Slave In	P0.6

Table 4.16 SPI Registers

Name	Description	Access
S0SPCR	SPI Control Register. This register controls the operation of the SPI.	R/W
S0SPSR	SPI Status Register. This register shows the status of the SPI.	Read Only
S0SPDR	SPI Data Register. This bi-directional register provides the transmit and receive data for the SPI. Transmit data is provided to the SPI0 by writing to this register. Data received by the SPI0 can be read from this register.	R/W
S0SPCCR	SPI Clock Counter Register. This register controls the frequency of a master's SCK0.	R/W
S0SPINT	SPI Interrupt Flag. This register contains the interrupt flag for the SPI interface.	R/W

SPI Control Register (S0SPCR)

The S0SPCR register controls the operation of the SPI0 as per the configuration bits setting.

Table 4.17 SOSPCR Register description

Bits	15-12	11-8	7	6	5	4	3	2	1	0
Symbol	Reserved	BITS	SPIE	LSBF	MSTR	CPOL	CPHA	Bit Enable	-	-

Bits	Symbol	Description
0-1	Reserved	-
2	BIT FIELD ENABLE	0 ;The SPI controller sends and receives 8 bits of data per transfer. 1; The SPI controller sends and receives the number of bits selected by bits field (11:8)
3	CPHA	Clock Phase Control

		0; The data is sampled on first clock edge 1; The data is sampled on second clock edge
4	CPOL	Clock Polarity 0; Serial Clock (SCK) is active High 1; Serial Clock (SCK) is active High
5	MSTR	Master mode select. 0; The SPI operates in Slave mode.0 1 ;The SPI operates in Master mode.
6	LSBF	LSB First controls which direction each byte is shifted when transferred. 0; SPI data is transferred MSB (bit 7) first. 1 ;SPI data is transferred LSB (bit 0) first.
7	SPIE	Serial peripheral interrupt enable. 0; SPI interrupts are inhibited.0 1; A hardware interrupt is generated each time the SPIF or WCOL bits are activated
11-8	BITS FIELD	When bit 2 of this register is 1, this field controls the number of bits per transfer: • 1000 - 8 bits per transfer • 1001- 9 bits per transfer • 1010- 10 bits per transfer • 1011 -11 bits per transfer • 1100 -12 bits per transfer • 1101 -13 bits per transfer • 1110 -14 bits per transfer • 1111 -15 bits per transfer • 0000 -16 bits per transfer
15-12	RESERVED	Reserved

SPI STATUS REGISTER(S0SPSR)

The S0SPSR register controls the operation of the SPI0 as per the configuration bits setting.

Table 4.18 SOSPSR Register description

Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Symbol	SPIF	WCOL	ROVR	MODF	ABRT	-	-	-

Bits	**Symbol**	**Description**
0-2	Reserved	-
3	ABRT	**Slave abort.** When 1, this bit indicates that a slave abort has occurred. This bit is cleared by reading this register.
4	MODF	**Mode fault.** when 1, this bit indicates that a Mode fault error has occurred. This bit is cleared by reading this register, then writing the SPI control register.
5	ROVR	**Read overrun**. When 1, this bit indicates that a read overrun has occurred. This bit is cleared by reading this register.
6	WCOL	**Write collision.** When 1, this bit indicates that a write collision has occurred. This bit is cleared by reading this register, then accessing the SPI data register.
7	SPIF	**SPI transfer complete flag.** When 1, this bit indicates when a SPI data transfer is complete. When a master, this bit is set at the end of the last cycle of the

		transfer. When a slave, this bit is set on the last data sampling edge of the SCK. This bit is cleared by first reading this register, then accessing the SPI data register.

SPI Data Register (S0SPDR)

This bi-directional data register provides the transmit and receive data for the SPI.Transmit data is provided to the SPI by writing to this register. Data received by the SPI can be read from this register. When a master, a write to this register will start a SPI data transfer. Writes to this register will be blocked from when a data transfer starts to when the SPIF status bit is set, and the status register has not been read.

Table 4.19 SPI Data Register (S0SPDR)

Bits	Symbol	Description
7-0	Data Low	
15-8	Data HIGH	If bit 2 of the SPCR is 1 and bits 11:8 are other than 1000, some or all of these bits contain the additional transmit and receive bits. When less than 16 bits are selected, the more significant among these bits read as zeroes.

SPI Clock Counter Register (S0SPCCR)

- This register controls the frequency of a master's SCK. The register indicates the number of PCLK cycles that make up an SPI clock.
- The value of this register must always be an even number. As a result, bit 0 must always be 0.
- The value of the register must also always be greater than or equal to 8.

- SPI (SCLK) Frequency = PCLK / SPCCR Value Max. Freq=1.875 Mhz

Note:-Violations of this can result in unpredictable behaviour

4.6.2 SD Cards

Digital (SD) cards are removable flash-based storage device SD means ‘secure digital’ and MMC means ‘multimedia card.’ You can insert these cards in your media player, PDA or digital camera. Their small size, relative simplicity, low power consumption and low cost make them an ideal solution for many applications.

Figure 4.13 SD Cards from different manufacturers

- SD/MMC cards have their own architecture and signals.
- These are universal low-cost, high-speed data storage cards.
- MMCs work at 20 MHz, while SD cards work at up to 25 MHz's,
- The two memories work in two different modes: SD mode and serial peripheral interface (SPI).

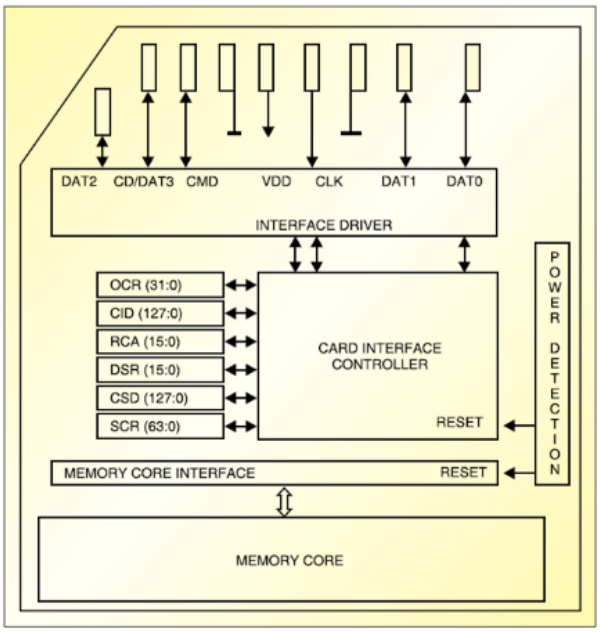


Figure 4.13 SD card internals

Table 4.20 SD card pin details

Pin	Signal	Function (SD mode)	Function (SPI mode)
1	DAT3/CS	Data line 3	Chip select/slave select (SS)
2	CMD/DI	Command line	Master-out slave-in (MOSI)
3	VSS1	Ground	Ground
4	VDD	Supply voltage	Supply voltage
5	CLK	Clock	Clock (SCLK)
6	VSS2	Ground	Ground
7	DAT0/DO	Data line 0	Master-in slave-out (MISO)
8	DAT1/IRQ	Data line 1	Unused or IRQ
9	DAT2/NC	Data line 2	Unused

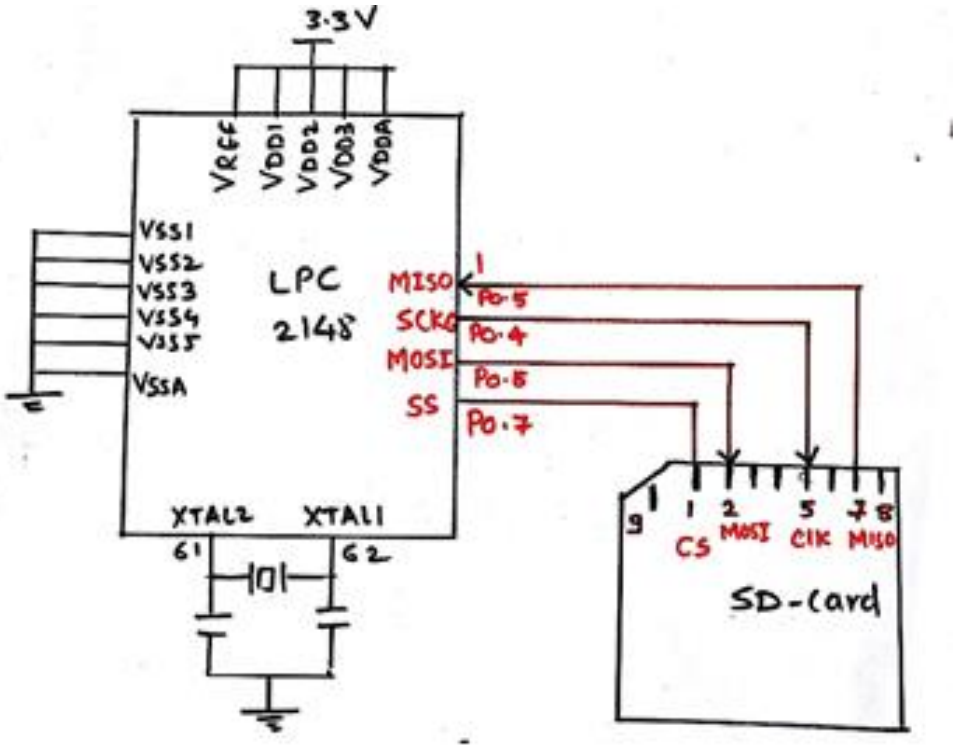


Figure 4.14 SD card Interfacing diagram with LPC2148

SD Memory interfaces to the host point-to-point (in Fig. an ARM microcontroller is the host). This type of interfacing is very popular in the industry. In serial peripheral interface (SPI) mode, you can use following signals of the host:

1. CS: Host to card chip-select signal

2. CLK: Host to card clock signal

3. MOSI (master -out slave-in): Host to card single bit data signal

4. MISO (master - in slave - out) : Card to host single-bit data signal

Now many companies are manufacturing suitable hosts for the SD bus interface.

For example, Philips is manufacturing LPC2148 microcontroller with MOSI and MISO

- Master-slave mode of communication is used for multiple slave devices in the SD architecture.
- MOSI is a unidirectional signal used to transfer serial data from the master to the slave. When the host is master, data can move from the host to the SD card. That's why MOSI is connected to data input (DI) of the SD/MMC card.
- The MISO signal transfers serial data from the slave to the master. When the SD is a slave, serial data is output on MISO signal. When the SD is a master, it clocks in serial data from this signal.
- SD memory cards use 1- or 4-bit bus width and star topology to connect multiple cards, while MMC cards use 1-bit bus width and bus topology for reading multiple cards.

Steps to switch from SD-Bus mode to SPI Bus mode of Operation

- All communications between the host and the card are controlled by the host.
- Messages in the spi bus protocol consist of commands, responses and tokens.
- The card returns a response to every command received and also a data response token for every write command
- The sd card wakes up in sd card mode, and it will enter the spi mode if its **cs (chip select or slave select) line is held low.** When a reset command is sent to the card
- **The card can only be returned to the sd mode after a power down and power up sequence** then the spi mode is entered.
- The card is in the non protected mode where CRC checking is not used CRC checking can be turned on and off by sending command CRC_on_off command name cmd59 to the card.

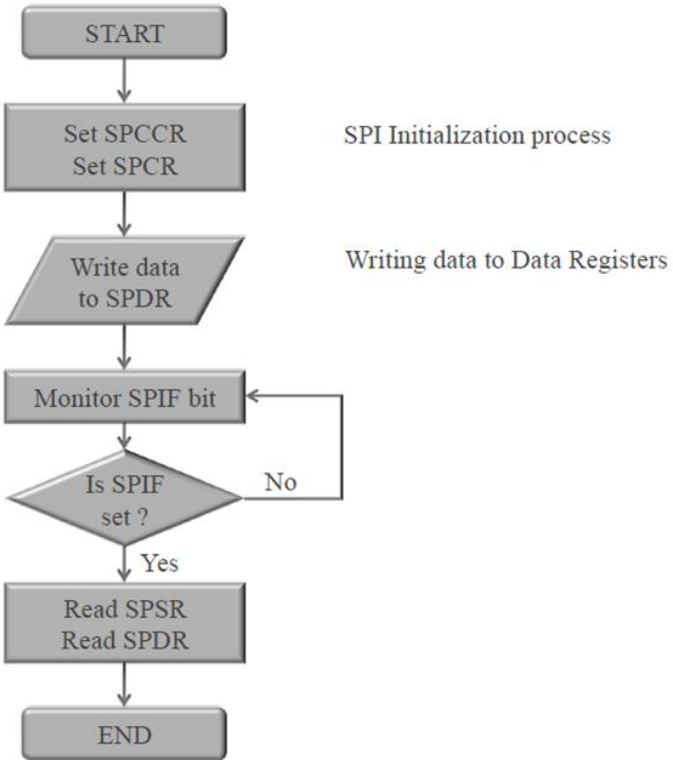


Figure 4.15 Algorithm for R/W operation with SD card

4.6.3 Example Program for SD card Interfacing

```
/**********************************/
        Function for initializing SPI
/**********************************/
void spi_init()
{
 PINSEL0=0X00001505;// Select MOSI=P0.6, MISO=P0.5, SCK=P0.4,SSEL=P0.7
S0SPCCR=0X08; // clock is divided by 8 ( SPI Clock freq = PCLK / S0SPCCR Value)
S0SPCR=0X0020; // select as master mode
}
/**********************************/
        Function for sending a char
/**********************************/
void spi_master(char a)
{
S0SPDR=a;   //write character “a” to be transmitted in S0SPDR
while(!(S0SPSR & 0X80)); // wait till SPIF=1 i.e., complete transfer of data
}
/**********************************/
        Function for receiving a char
/**********************************/
```

```
char spi_slave(void)
{
while(!(S0SPSR & 0X80)); // wait till SPIF=1 i.e., complete reception of data

return S0SPDR; //pick-up received character which is arrived in S0SPDR

}
```

4.7 DIGITAL TO ANALOG CONVERTER (DAC) IN LPC2148

- LPC2148 has one 10-bit DAC
- Settling time software selectable
- DAC output can drive max of 700 micro-Ampere or 350 micro-Ampere
- DAC peripheral has only one register, DACR

Table 4.21 DAC Register Pin Description

Pin	Type	Description
AOUT	Output	**Analog Output**. After the selected settling time after the DACR is written with a new value, the voltage on this pin (with respect to V_{SSA}) is VALUE/1024 * $V_{REF.}$
V_{REF}	Reference	**Voltage Reference**. This pin provides a voltage reference level for the D/A converter.
V_{DDA}, V_{SSA}	Power	**Analog Power and Ground**. These should be nominally the same voltages as V_3 and V_{SSD}, but should be isolated to minimize noise and error.

Table 4.22 Digital to Analog Control Register (DACR) Description

31-17	16	15-6	5-0
Reserved	BIAS	10-bit Digital Value	Reserved

Bit	Symbol	Value	Description	Reset value

5:0	-		Reserved, user software should not write ones to reserved NA bits. The value read from a reserved bit is not defined.	NA
15:6	VALUE		After the selected settling time after this field is written with a 0 new VALUE, the voltage on the A_{OUT} pin (with respect to VssA) is VALUE/1024 * V_{REF}.	0
16	BIAS	0	The settling time of the DAC is 1 µs max, and the maximum current is 700 µA.	0
		1	The settling time of the DAC is 2.5 µs and the maximum current is 350 µA.	
31:17	-		Reserved, user software should not write ones to reserved NA bits. The value read from a reserved bit is not defined.	NA

4.7.1 DAC Design Example

Configure DAC register for generating with 3.3V V_{REF} & Select 350 micro AMPERE settling time.

1. 0V,
2. 1.65V,
3. 3.3V

Note: $\mathbf{A_{OUT}}$= V_{REF} * (10 bit Digital Value/Resolution)

Solution:

1. DACR = 0x00010000; //A_{OUT} = 0V
2. DACR = 0x00018000; //A_{OUT} = 1.65V
3. DACR = 0x0001FFC0; //A_{OUT} = 3.3 V

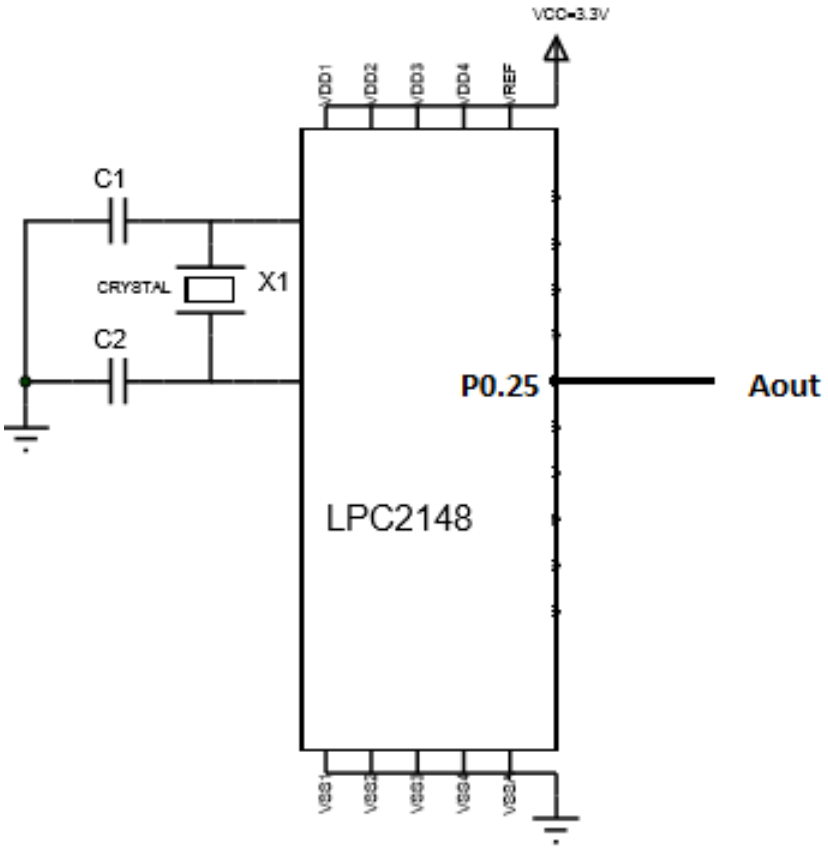


Figure 4.16 Configuring internal DAC of LPC2148

4.7.2 C-Program for Sine Waveform Generation using DAC in LPC2148

```
#include <lpc214x.h>
#include <stdint.h>
void delay_ms(uint16_t j)
{
   uint16_t x,i;
for(i=0;i<j;i++)
 {
   for(x=0; x<6000; x++);    /* loop to generate 1 milisecond delay with Cclk = 60MHz */
 }
}

int main (void)
{
 uint16_t value;
```

```
uint8_t i;
 i = 0;
 PINSEL1 = 0x00080000; /* P0.25 as DAC output */
uint16_t sin_wave[42] = {
512,591,665,742,808,873,926,968,998,1017,1023,1017,998,968,926,873,808,742,665,591,51
2,
            436,359,282,216,211,151,97,55,25,6,0,6,25,55,97,151,211,216,282,359,436 };
while(1)
 {
   while(i !=42)
    {
     value = sin_wave[i];
     DACR = ( (1<<16) | (value<<6) );///Bias bit=1, Digital Value left shifted by 6 bits
    delay_ms(1);
     i++;
    }
    i = 0;
  }
}
```

4.7.3 C-code for Triangular Waveform Generation

```
#include <lpc214x.h>
#include <stdint.h>
void delay_ms(uint16_t j)
{
   uint16_t x,i;
for(i=0;i<j;i++)
 {
   for(x=0; x<6000; x++);    /* loop to generate 1 milisecond delay with Cclk = 60MHz */
 }
}

int main (void)
{
 uint16_t value;
uint8_t i;
 i = 0;
 PINSEL1 = 0x00080000; /* P0.25 as DAC output */
while(1)
 {
    value = 0;
   while ( value != 1023 )
    {
```

```
    DACR = ( (1<<16) | (value<<6) );
    value++;
   }
  while ( value != 0 )
   {
    DACR = ( (1<<16) | (value<<6) );
    value--;
   }
 }

}
```

4.7.4 C-Program for Square Waveform Generation

```
#include <lpc214x.h>
#include <stdint.h>
void delay_ms(uint16_t j)
{
  uint16_t x,i;
for(i=0;i<j;i++)
 {
  for(x=0; x<6000; x++);    /* loop to generate 1 milisecond delay with Cclk = 60MHz */
 }
}

int main (void)
{
 uint16_t value;
uint8_t i;
 i = 0;
 PINSEL1 = 0x00080000; /* P0.25 as DAC output */
while(1)
 {
   value = 1023;
   DACR = ( (1<<16) | (value<<6) );
   delay_ms(100);
   value = 0;
   DACR = ( (1<<16) | (value<<6) );
   delay_ms(100);
 }
}
```

TEXT / REFERENCE BOOKS

1. Tularam M Bansod " Microcontroller Programming (8051, PIC, ARM7 ARM Cortex)", 1st Edition, Shroff Publishers,2017
2. A Getting Started Guide for MDK Version 5- Reference Manual, Keil, 2015
3. Steve Furber, "ARM System-on-Chip Architecture" Pearson, 2nd Edition, 2015

Exercise Questions

1. Mention the key specifications of LPC2148 ARM processor.
2. How many I/O devices can be connected to LPC2148 by using UART interface?
3. Mention the four logic signals used in SPI protocols
4. Outline the logic signals used in I2C protocol.
5. Contrast SPI and I2C protocol.
6. Outline the role of shift register in master slave devices of SPI bus.
7. Give the specifications of on-chip DAC in LPC2148 processor.
8. Contrast GSM and GPS modems.
9. Design a circuit with LPC2148 ARM processor and develop a C code to measure the intensity of ambient light and temperature and automatically control a AC lamp.
10. Design a circuit with LPC2148 ARM processor and develop a C code to read an analog voltage and convert it to equivalent digital value.
11. Design a circuit with LPC2148 ARM processor and develop a C code to generate a triangular waveform with a frequency of 1KHz using the DAC module.
12. Design a circuit with LPC2148 ARM processor and a GSM modem to control an agriculture water pump-set, by sending SMS from a mobile phone.
13. Design a circuit with LPC2148 ARM processor and develop a C code to read data from EEPROM using I2C protocol.
14. Develop a system model for a real-time data acquisition and logging system with SD card storage unit.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – V SECA3019 - ARM CORTEX PROCESSORS

ARM CORTEX PROCESSORS

[Introduction to ARM CORTEX series, improvement over classical series and advantages for embedded system design. CORTEX-A, CORTEX-M, CORTEX-R processors series, versions, features and applications, need of operating system in developing complex applications in embedded system, Firmware development for ARM Cortex, Survey of CORTEX-M3 based controllers, its features and comparison]

5.1 ARM Architecture classification

The ARM architecture processor is an advanced reduced instruction set computing [RISC] machine and it's a 32bit reduced instruction set computer (RISC). It was introduced by the Acron computer organization in 1987. Several Chip manufacturers started making microcontrollers using the ARM architecture for the CPU core and adding their own peripheral devices to it. They are called as ARM microcontrollers. This ARM family of microcontroller are developed by makers like ST Microelectronics, Motorola, NXP and so on. The relative simplicity of ARM processors makes them suitable for low power applications. As a result, they have become dominant in the mobile and embedded electronics market, as relatively low-cost, small microprocessors and microcontrollers. ARM processors account for approximately 90% of all embedded 32-bit RISC processors and are used extensively in consumer electronics, including personal digital assistants (PDAs), tablets, mobile phones, digital media and music players, hand-held game consoles, calculators and computer peripherals such as hard drives and routers.

The ARM architecture comes with totally different versions like ARMv1, ARMv2, etc., (shown in figure 5.1) and each one has its own advantage and disadvantages. Some years ago, ARM has launched a new generation of its core identified by the name: CORTEX.

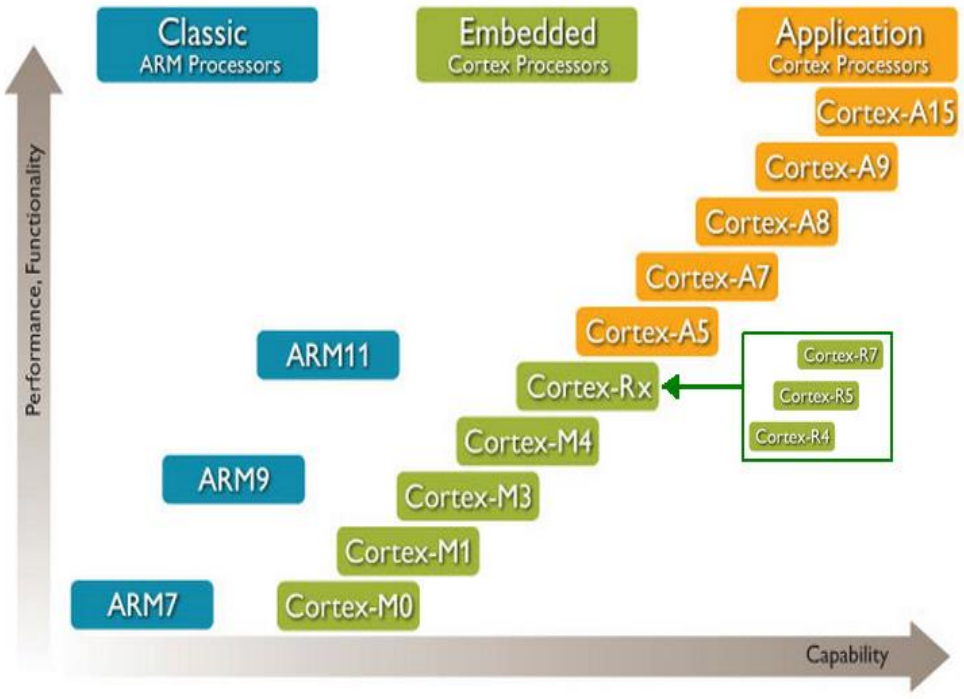


Figure 5.1 ARM Processor families

5.2 ARM Cortex Series Processors

The ARM® Cortex® series of cores encompasses a very wide range of scalable performance options offering designers a great deal of choice and the opportunity to use the best-fit core for their application without being forced into a one-size-fits-all solution. The Cortex portfolio is split broadly into three main categories:

- Cortex-A -- application processor cores for a performance-intensive systems
- Cortex-R – high-performance cores for real-time applications
- Cortex-M – microcontroller cores for a wide range of embedded applications demanding low lost with optimum performance

5.2.1 Cortex-A Series

Cortex-A processors provide a range of solutions for devices that make use of a rich operating system such as Linux or Android and are used in a wide range of applications from low-cost handsets to smartphones, tablet computers, set-top boxes and also enterprise networking equipment. The first range of Cortex-A processors (A5, A7, A8, A9, A12, A15 and A17) is based on the ARMv7-A architecture. Each core shares a common feature set including items such as the NEON media processing engine, Trustzone for security extensions, and single- and double-precision floating point support along with support for several instruction sets (ARM, Thumb-2, Thumb, Jazelle and DSP). Together this group of processors offers design flexibility by providing the required peak performance points while delivering the desired power efficiency. While the Cortex-A5 core is the smallest and lowest power member of the Cortex A series, it offers the possibility of multicore performance and is compatible with the larger members of the series (A9 and A15). The A5 is a natural choice for designers who have previously worked with the ARM926EJ-S or ARM1176JZ-S processors as it enables higher performance and lower silicon cost.

The Cortex-A7 is similar in power consumption and area to the Cortex-A5 but brings a performance increase in the range of 20 percent as well as full architectural compatibility with the Cortex-A15 and Cortex-A17. The Cortex-A7 is an ideal choice for cost-sensitive smartphone and tablet implementations, and it can also be combined with a Cortex-A15 or Cortex-A17 in what ARM refers to as a “big.LITTLE” processing configuration.

The big.LITTLE configuration is essentially a power optimization technology; a high-performance CPU (e.g., Cortex-A17) and an ultra-efficient CPU (e.g., Cortex-A7) are combined to provide higher sustained performance and also to enable significant overall power savings by relying on the more efficient core in cases of low to moderate performance

requirements from the application, saving potentially 75 percent of CPU energy and as such extending battery life. This configuration offers a significant advantage to the developer as the performance demands of smartphones and tablets is advancing much faster than the capacity of batteries can keep pace.

Design methodologies such as big.LITTLE, as part of an overall system design strategy, can significantly help reduce this battery technology gap. Moving to the other end of the Cortex-A scale, let's consider the Cortex-A15 and Cortex-A17 cores. These are both very high-performance processors and again are available in a variety of configurations. The Cortex-A17 is the most efficient "mid-range" processor, and it squarely targets premium smartphones and tablets. The Cortex-A9 has been widely deployed in that market, but the Cortex-A17 offers an increase of more than 60percent (cycle for cycle) compared to the Cortex-A9 and achieves this performance while also improving overall power efficiency. The Cortex-A17 can be configured with up to four cores, each of which contains a fully out-of-order pipeline. As mentioned previously, the Cortex-A17 can be combined with the Cortex-A7 for an effective big.LITTLE configuration, and it can also be combined with high-end mobile graphics processors (such as the MALI from ARM), resulting in a very efficient design overall. The Cortex-A15 is the highest performance member of this series, providing (in a mobile configuration) twice the performance you would get from a Cortex-A9. While being perfectly adequate in applications such as high-end smartphones or tablets, a multi-core Cortex-A15 processor running at 2.5 GHz opens up the possibility of using a Cortex-A processor in applications such as low-power servers or wireless infrastructure.

The Cortex-A15 is the first processor from ARM to incorporate hardware support for data management and arbitration of virtualized software environments. Applications in those software environments are able to simultaneously access the system capabilities, making it possible to implement devices with virtual environments that are robust and isolated from each other. The latest additions – the Cortex-A50 series – extend the reach of the Cortex-A series into low-power servers. These processors are built on the ARMv8 architecture and bring with them support for AArch64 – an energy-efficient 64-bit execution state that can operate alongside the existing 32-bit execution state. An obvious reason for the move to 64-bit is the support of more than 4GB of physical memory, which is already achieved on Cortex-A15 and Cortex-A7. In this case, the move to 64-bit is really about providing better support for server applications where a growing number of operating system and application implementations are using 64-bit, and the Cortex-A50 series delivers a power optimized solution for this scenario. The same is largely true for the desktop market, and support for 64-bit will enable the CortexA50 series to be more broadly adopted into this segment and will

provide some level of future-proofing for the eventual migration of 64-bit operating systems into mobile applications.

5.2.2 Cortex-R Series

The Cortex-R processors target high-performance real-time applications such as hard disk controllers (or solid state drive controllers), networking equipment and printers in the enterprise segment, consumer devices such as Blu-ray players and media players, and also automotive applications such as airbags, braking systems and engine management. The Cortex-R series is similar in some respects to a high-end microcontroller (MCU) but targets larger systems than you would typically use a standard MCU. The Cortex-R4, for example, is well suited for automotive applications. It can be clocked up to 600 MHz (delivering 2.45 DMIPS/MHz), has an 8-stage pipeline with dual-issue, pre-fetch and branch prediction and a low latency interrupt system that can interrupt multi-cycle operations to quickly serve the incoming interrupt. It can also be implemented in a dual-core configuration with the second Cortex-R4 being in a redundant lock-step configuration with logic for fault detection making it ideal for safety critical systems.

Networking and data storage applications are well served by the Cortex-R5, which extends the feature set offered by the Cortex-R4 to offer increased efficiency and reliability and enhance error management in dependable real-time systems. One such system-level feature is the low latency peripheral port (LLPP) to enable fast peripheral reads and writes (instead of having to perform a read-modify-write on the entire port). The Cortex-R5 can also be implemented as a “lock-step” dual-core system with the processors running independently, each executing its own programs with its own bus interfaces, and interrupts. This dual-core implementation makes it possible to build very powerful, flexible systems with real-time responses. The Cortex-R7 significantly extends the performance reach of the series, with clock speeds in excess of 1 GHz and a performance of 3.77 DMIPS/MHz.

The 11-stage pipeline on the Cortex-R7 now adds out-oforder execution along with improved branch prediction. There are several options for multi-core implementations as well: lock-step, symmetric multi-processing and asymmetric multi-processing. The Cortex-R7 also has a fully integrated generic interrupt controller (GIC) supporting complex priority-based interrupt handling. It is worth noting, however, that despite its high-performance levels, the Cortex-R7 is it not suitable for running rich operating systems (such as Linux and Android), which remains the domain of the Cortex-A series.

5.2.3 Cortex-M Series

The Cortex-M series is designed specifically to target the already very crowded microcontroller unit (MCU) market. The Cortex-M series is built on the ARMv7-M architecture (used for Cortex-M3 and Cortex-M4), and the smaller Cortex-M0+ is built on the ARMv6-M architecture. The first Cortex-M processor was released in 2004, and it quickly gained popularity when a few mainstream MCU vendors picked up the core and started producing MCU devices. It is safe to say that the Cortex-M has become for the 32-bit world what the 8051 is for the 8-bit – an industry-standard core supplied by many vendors, each of which dip the core in their own special sauce to provide differentiation in the market. The Cortex-M series can be implemented as a soft core in an FPGA, for example, but it is much more common to find them implemented as MCU with integrated memories, clocks and peripherals. Some are optimized for energy efficiency, some for high performance and some are tailored to a specific market segment such as smart metering. The Cortex-M3 and Cortex-M4 are very similar cores. Each offers a performance of 1.25 DMIPS/MHz with a 3-stage pipeline, multiple 32-bit busses, clock speeds up to 200 MHz and very efficient debug options. The significant difference is the Cortex-M4 core's capability for DSP. The Cortex-M3 and Cortex-M4 share the same architecture and instruction set (Thumb-2). However, the Cortex-M4 adds a range of saturating and SIMD instructions specifically optimized to handle DSP algorithms.

For example, consider the case of a 512 point FFT running every 0.5 second on equivalent off-the-shelf Cortex-M3 and Cortex-M4 MCUs. For comparison, the Cortex-M3 would consume around three times the power that a Cortex-M4 would need for the same job. There is also the option to get a single precision floating point unit (FPU) on a Cortex-M4. If your application requires floating point math, you will get this done considerably faster on a Cortex-M4 than you will on a Cortex-M3. That said, for an application that is not using the DSP or FPU capabilities of the Cortex-M4, you will see the same level of performance and power consumption on a Cortex-M3. In other words, if you need DSP functionality, go with a Cortex-M4. Otherwise, the Cortex-M3 will do the job. For applications that are particularly cost sensitive or are migrating from 8-bit to 32-bit, the smallest member of the Cortex-M series might be the best choice. The Cortex-M0+ performance sits a little below that of the Cortex-M3 and Cortex-M4 at 0.95 DMIPS/MHz but is still compatible with its bigger brothers. The Cortex-M0+ uses a subset of the Thumb-2 instruction set, and those instructions are predominantly 16- bit operands (although all data operations are 32-bit), which lend themselves nicely to the 2-stage pipeline that the Cortex-M0+ offers. This brings some overall power saving to the system through reduced branch shadow, and the pipeline will in most cases hold the next four instructions. The Cortex-M0+ also has a dedicated bus

for single-cycle GPIO, meaning you can implement certain interfaces with bit-bashed GPIO like you would on an 8-bit MCU but with the performance of a 32-bit core to process the data.

Another key difference on the Cortex-M0+ is the addition of the micro trace buffer (MTB). This peripherals allows you to dedicate some of the on-chip RAM to store program branches while in debug.– These branches can then be passed back up to the integrated development environment (IDE), and the program flow can be reconstructed. This capability provides a rudimentary form of instruction trace and compensates for not having the extended trace macrocell (ETM) found on the Cortex-M3 and Cortex-M4. The level of debug information you can extract from a Cortex-M0+ is significantly higher than that which you can get from an 8-bit MCU, meaning those hard to solve bugs just got easier to fix.

Table 5.1 Comparison of Cortex-M and Cortex-R series

Features	Cortex M3	Cortex M4	Cortex R4
	32-bit Microcontroller	32-bit Microcontroller	32-bit embedded real-time CPU
Architecture	v7M	v7M	v7R
ISA	Thumb/Thumb2	Thumb/Thumb2	Thumb2 and ARM
Pipeline	3-stage single-issue	3-stage single-issue with branch prediction	8-stage dual-issue in-order with branch prediction
TCM	No	No	Yes
Cache	No	No	Yes
Memory Management	Memory Protection Unit (MPU)	Memory Protection Unit (MPU)	Memory Protection Unit (MPU)
Multi-core	No	No	Yes (redundant dual-core capability)
Floating-point Unit (FPU)	No	Single-precision	Single and double precision
SIMD/DSP support	No	8 and 16-bit SIMD and DSP instructions	Both SIMD and DSP instructions
Reliability Features	None	None	ECC/parity RAMs & Redundant core interface
Interrupt Controller	On-chip (up to 240 interrupts)	On-chip (up to 240 interrupts)	External Interrupt Controller Interface
Interrupt Latency	12 cycles	12 cycles	20 cycles
HW Divide	Yes	Yes	Yes
Software Compatibility	Thumb/Thumb2	Thumb/Thumb2	Binary compatible with M3 and M4

Table 5.2 Comparison of Cortex-M and Cortex-R series

	Cortex-A	Cortex-R	Cortex-M
Architecture profile	ARMv7-A ARMv8-A	ARMv7-R ARMv8-R	ARMv7-M ARMv8-M
Instruction set	32-bit/64-bit	32-bit	32-bit
Interrupts	Software managed	Deterministic software managed	Hardware managed
Bus interface	AMBA® AXI	AMBA AXI	AMBA AHB/AXI
Operating system support	Rich OS/RTOS	RTOS	RTOS
Example processors	Cortex-A7 Cortex-A35	Cortex-R8 Cortex-R52	Cortex-M7 Cortex-M33

Applications of ARM Cortex Series Processors

- The **cortex-A** stands for Application which will help in performance-intensive applications such as Android, Linux and many other applications related to handsets, tablets, desktops and laptops.
- The **Cortex-R** stands for the real-time application which is used in the safety-critical applications and where we need real-time responses of the system such as Automotive, medical, defence, avionics and server-side technologies where data related operations are executed.
- The **Cortex-M** stands for the Microcontroller which is used in most of our daily life applications also starting from the automation to DSP applications, sensors, smart displays, IoT applications and many more. The cortex-M series is an ocean of possibilities with a large number of probabilities and configurations.

5.2 ARM Cortex-M3 processor

The idea behind the Cortex-M3 architecture was to design a processor for cost-sensitive applications while providing high-performance computing and control1. These applications include automotive body systems, industrial control systems and wireless networking/sensor products. The M3 series introduced several important features to the 32-bit ARM processor architecture including:

• Non-maskable interrupts

• Highly-deterministic, nested, vectored interrupts

• Atomic bit manipulation

• Optional memory protection (MPU) In addition to excellent computational performance, the Cortex-M3 processor's advanced interrupt structure ensures prompt system response to real-world events while still offering low dynamic and static power consumption.

The Cortex-M3 and M4 processors share many common elements including advanced on-chip debug features, 3-stage pipeline and the ability to execute the full ARM instruction set or the subset used in THUMB2 processors. The Cortex-M4 processor's instruction set is enhanced by a rich library of efficient DSP features including extended single-cycle cycle 16/32-bit multiply-accumulate (MAC), dual 16-bit MAC instructions, optimized 8/16-bit SIMD arithmetic and saturating arithmetic instructions. Overall, the most noticeable difference between M3 and M4 is the optional single-precision (IEEE-754) Floating Point Unit (FPU) available with the M4. So Cortex-M4 processor core is best for digital signal processing applications.

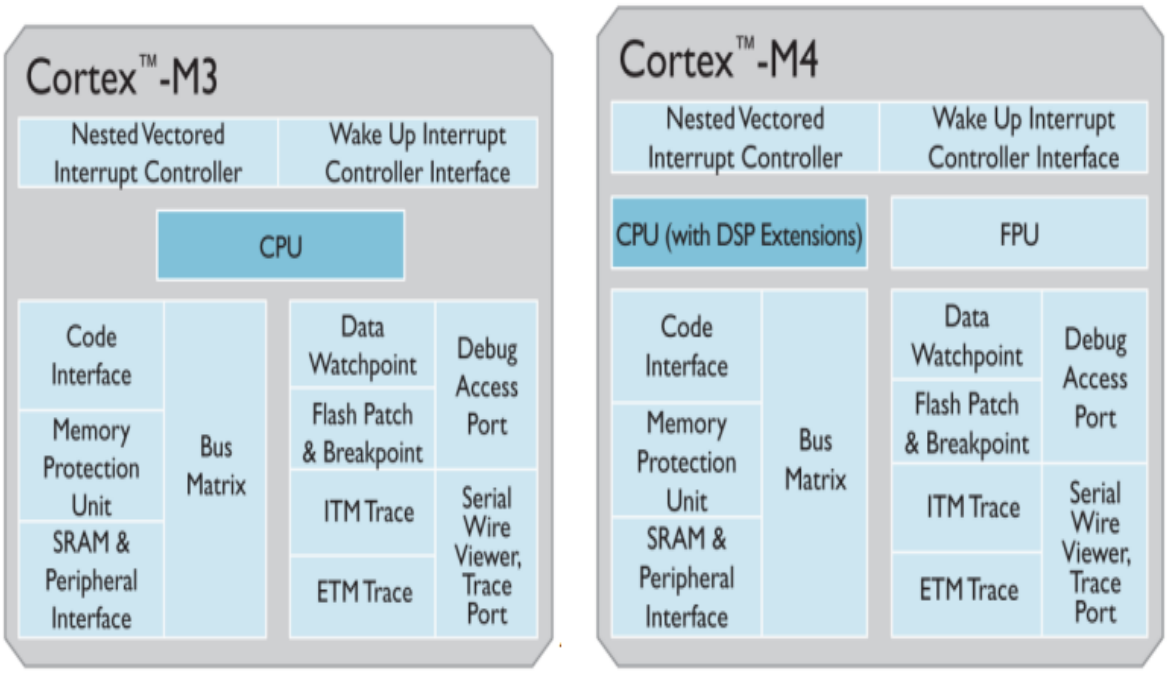


Figure 5.2 Cortex-M3 and Cortex-M4 Comparison

Features of ARM Cortex-M3 core

- Armv7-M architecture
- Bus interface 3x AMBA AHB-lite interface (Harvard bus architecture) AMBA ATB interface for CoreSight debug components
- Thumb/Thumb-2 subset instruction support
- 3-stage pipeline
- Nested Vectored Interrupt Controller (NVIC)
- Optional 8 MPU regions with sub-regions and background region
- Integrated Bit-field Processing Instructions and Bus Level Bit Banding
- Non-maskable interrupt + 1 to 240 physical interrupts with 8 to 256 priority levels
- Wake-up interrupt controller
- Hardware single-cycle (32x32) multiply, Hardware Divide (2-12 cycles), Saturated Adjustment support
- Integrated WFI and WFE Instructions and Sleep On Exit capability. Sleep and Deep Sleep Signal, Optional Retention Mode with Arm Power Management Kit
- Optional JTAG and Serial Wire Debug ports. Up to 8 breakpoints and 4 watchpoints
- Optional Instruction (ETM), Data Trace (DWT), and Instrumentation Trace (ITM)

5.3 Product Development with STM32 Cortex-M3 Microcontrollers

Figure 5.3 below shows the steps involved in converting ideas to product using the STM32 series microcontrollers. The first step is to identify a suitable STM32 microcontroller or STM32 development board that has inbuilt peripherals needed by the application

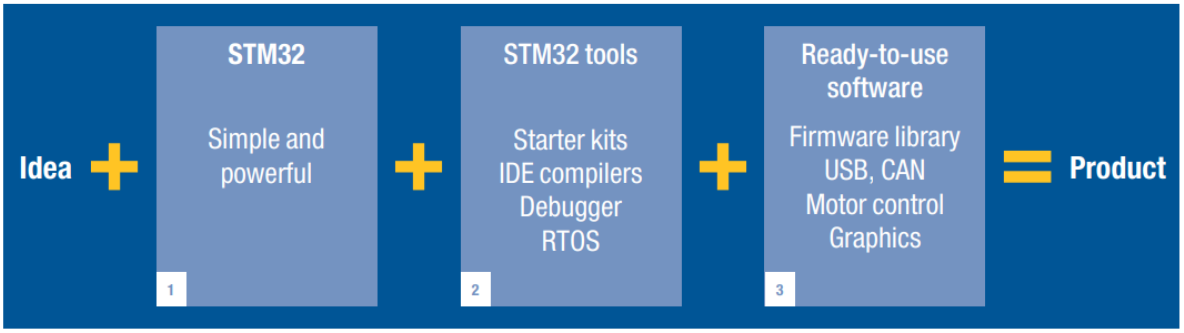


Figure 5.3 Product Development steps with STM32 microcontroller series

5.3.1 Choosing STM32 Microcontrollers/Boards

Select from the broad range of development boards available in the market for developing your applications.

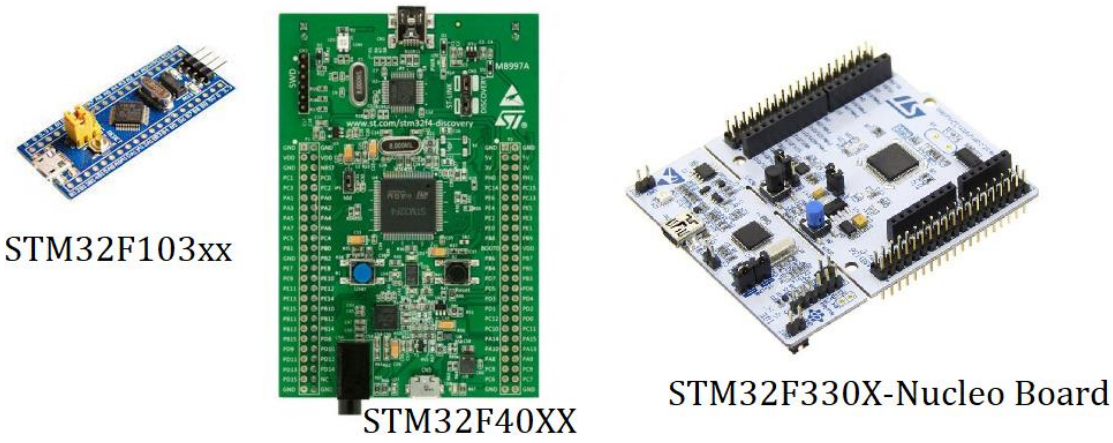


Figure 5.4 STM32 Development Boards

5.3.2 STM32 software Development Tools

STMicroelectronics' STM32 family of 32-bit ARM Cortex-M core-based microcontrollers is supported by a wide range of software integrated development environments (IDEs) with C, C++, Pascal and JAVA support and debuggers from STMicroelectronics and major 3rd-parties (free versions are available) that are complemented by tools from ST allowing to configure and initialize the MCU or monitor its behavior in run time. The popular IDE for STM32 processors are STM32Cube, mBED, Keil, Arduino IDE, Eclipse in Linux and emIDE. The code/memory optimization level and code portability achieved with different STM32 software tools is illustrated in figure 5.5.

5.3.3 STM32 Firmware Library

STM32 firmware library The STM32 firmware library provides easy access to all features of the standard device peripherals of the STM32. This free software package provides drivers for all standard device features and peripherals, from GPIO and timers to CAN, I2 C, FSMC, I2 S, SDIO, DAC, SPI, UART, ADC and more. The fully documented and tested C source code requires only basic knowledge of C programming, is compatible with any C compiler for ARM-core-based microcontrollers.

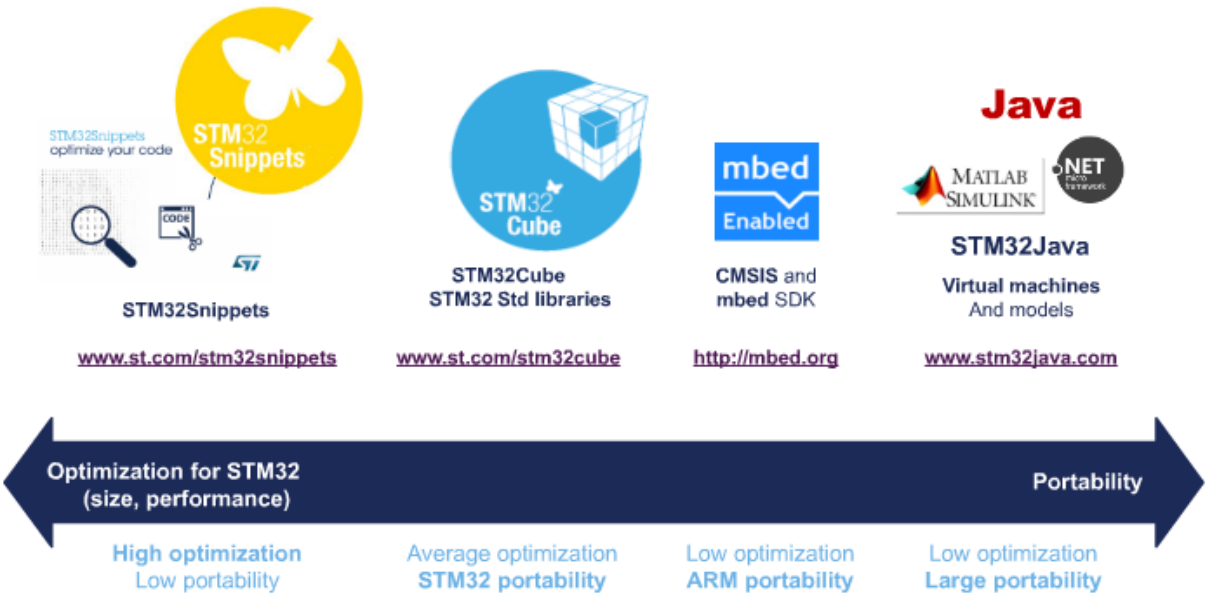


Figure 5.5 Performance of different STM32 Software Development Tools

5.4 Need for Operating System in Embedded Applications

For each embedded product, software developers need to consider whether they need an operating system; and if so, what type of an OS. Operating systems vary considerably, from real-time operating systems with a very small memory footprint to general-purpose OSes such as Linux with a rich set of features.

Choosing a proper type of operating system for your product – and consequently working out the required features of the embedded processor – depends significantly on whether you face a hard real-time requirement. Safety-critical and industrial systems such as an anti-lock braking system or motor control will have hard maximum response times. At the other end of the spectrum, consumer systems such as audio or gaming devices may be able to tolerate buffering, as long as the average performance is adequate. Such systems are said to have soft real-time requirements.

Bare metal

A hard real-time requirement can be achieved by writing so called bare-metal software that directly controls the underlying hardware. Bare-metal programming is typically utilized when the processor resources are very limited, the software is simple enough, and/or the real-time requirements are so tight that introduction of a further abstraction layer would complicate meeting these hard real-time requirements. The disadvantage to this approach is

that such bare-metal software needs to be written as a single task (plus interrupt routines), making it difficult for programmers to maintain the software as its complexity grows.

Real-time operating systems

When dealing with more complex embedded software, it is often advantageous to employ a Real-Time Operating System (RTOS). It allows the programmer to split the embedded software into multiple threads whose execution is managed by the small, low-overhead "kernel" of the RTOS. The use of the multi-threaded paradigm enables developers to create and maintain more complex software while still allowing for a sufficient reactivity. RTOSes typically operate with a concept of "priority" assigned to individual threads. The RTOS can then "pre-empt" (temporarily halt) lower-priority threads in favor of those with higher priority, so that the required real-time constraints can be met. The use of an RTOS often becomes necessary when adopting complex libraries or protocol stacks (such as TCP/IP or Bluetooth) as this third-party software normally consists of multiple threads already. Today there is a wide choice of open source and commercially licensed RTOSes.

The embedded processor requirements of a simple RTOS, such as FreeRTOS or Zephyr, are truly modest. It is sufficient to have a RISC-V processor with just machine mode (M) and a timer peripheral. These RTOSes can therefore run on any of the Codasip RISC-V cores or Western Digital SweRV Cores. However, rigorous software development is needed as machine mode offers unconstrained access to all memory and peripherals with associated risks. Extra protection is possible through a specialized RTOS such as those developed for functional safety, like SAFERTOS, or for security.

If a processor core supports both machine (M) and user (U) privilege modes and has physical memory protection (PMP), it is possible to establish separation between trusted code (with unconstrained access) and other application code. With PMP, the trusted code sets up rules for each portion of the application code, saying which parts of memory (or peripherals) it is allowed to access. PMP can for instance be used to prevent third-party code from interfering with the data of the rest of the application, or to detect stack overflows. Employing PMP therefore increases the safety and security of a system, but at the cost of additional hardware required for its support.

Rich operating systems

For applications requiring a more advanced user interface, sophisticated I/O and networking, such as in set-top boxes or entertainment systems, an RTOS is likely to be too simplistic. The same applies if there are complex computations, requirements for a full process isolation and multitasking, filesystem & storage support, or a full separation of

application code from hardware via device drivers. Systems like these generally have soft real-time requirements and can be best served by a general-purpose rich operating system such as Linux. As noted in an earlier post, Linux requires multiple RISC-V privilege modes – machine, supervisor, and user modes (M, S, U) – as well as a memory management unit (MMU) for virtual-to-physical address translation. Also, the memory footprint of such system is significantly larger compared to a simple RTOS.

Finally, for embedded systems that require both hard real-time responses and features of a rich operating system like Linux, it is common to design them with two communicating processor subsystems, one supporting an RTOS and the other running Linux.

5.5 Survey of ARM Cortex-M3 based Microcontrollers

Chip Manufactures like ST Microelectronics, NXP, Motorola, Stellaris, Texas Instruments are manufacturing several microcontrollers using the ARM Cortex-M3 core and adding their own choice peripherals. Features of few of them is discussed in this section.

5.5.1 NXP LPC1345 32bit ARM Microcontroller

The LPC1345FHN33 is an Arm Cortex-M3 based microcontroller for embedded applications featuring a high level of integration and low power consumption. The Arm Cortex-M3 is a next generation core that offers system enhancements such as enhanced debug features and a higher level of support block integration. The LPC1345FHN33 operates at CPU frequencies of up to 72 MHz. The Arm Cortex-M3 CPU incorporates a 3-stage pipeline and uses a Harvard architecture with separate local instruction and data buses as well as a third bus for peripherals. The Arm Cortex-M3 CPU also includes an internal prefetch unit that supports speculative branching.

Features of NXP LPC1345 features

- Arm Cortex-M3 processor, running at frequencies of up to 72 MHz
- 32 kB on-chip flash program memory with a 256 byte page erase function
- In-System Programming (ISP) and In-Application Programming (IAP)
- 2 kB on-chip EEPROM data memory with on-chip API support
- 10 kB SRAM data memory
- 16 kB boot ROM with API support
- 26 General Purpose I/O (GPIO) pins
- Four general purpose counter/timers
- Programmable Windowed WatchDog Timer (WWDT)

- 2-bit ADC with eight input channels and sampling rates of up to 500 kSamples/s
- USB 2.0 full-speed device controller and I2C bus
- USART with fractional baud rate generation

5.5.2 ST STM32F102/103 32bit ARM Microcontroller

The STM32 family of 32-bit Flash microcontrollers is based on the breakthrough ARM Cortex-M3 core – a core specifically developed for embedded applications that require a combination of high-performance, realtime, low-power and low-cost operation. The STM32 family benefi ts from the Cortex-M3 architectural enhancements (including the Thumb-2® instruction set) that deliver improved performance combined with better code density, and a tightly coupled nested vectored interrupt controller that significantly speeds response to interrupts, all combined with industry-leading power consumption. STMicroelectronics was a lead partner in developing the Cortex-M3 core and is now the first leading MCU supplier to introduce a product family based on the core.

The STM32 also embeds a real-time clock (RTC) running either from a 32 kHz quartz oscillator or an internal RC oscillator. The RTC has a separate power domain, with an embedded switchover to run either from a dedicated coin cell battery or from the main supply. Its typical current consumption is 1.4 µA at 3.3 V. It embeds up to 84 bytes for data backup. Start-up time from low-power modes is lower than 6 µs typical from stop mode, and 50 µs typical from standby mode and reset.

Hardware features of STM32F102 and F103 series

- 2x USB OTG (one with HS support)
- Audio: dedicated audio PLL and 2 half duplex I²S
- Up to 15 communication interfaces (including 6 USARTs running at up to 7.5 Mbit/s, 3x SPI running at up to 30 Mbit/s, 3x I²C, 2x CAN, SDIO)
- Analog: two 12-bit DACs, three 12-bit ADCs reaching 2 MSPS or 6 MSPS in interleaved mode
- Up to 17 timers: 16- and 32-bit timers
- The STM32F205/215 devices cover from 128 Kbytes to 1 MByte of Flash, up to 128 Kbytes of SRAM

Performance Limits of STM32F10x series microcontrollers

- Low voltage 2.0 V to 3.6 V operation
- Clock Frequency - 72 MHz

- Startup time from stop < 6 µs
- Startup time from standby 50 µs
- USB - 12 Mbit/s
- USART up to 4.5 Mbit/s
- SPI - 18 MHz master and slave
- I2C - 400 kHz
- GPIO 18 MHz maximum toggle
- PWM timer 72 MHz clock input
- SDIO Up to 48 MHz
- I2S From 8 kHz to 48 kHz sampling frequencies
- ADC 12-bit, 1 µs conversion time
- DAC 2-channel, 12-bit

5.6 Firmware Development for STM32Fxxx ARM Cortex Microcontroller

An alternative to Arduino is the STM32F103C8T6 microcontroller-based development board, which is often called as the Blue Pill (Matrix reference). This microcontroller is based on ARM Cortex-M3 Architecture manufactured by STMicroelectronics. STM32F103C8T6 is a very powerful Microcontroller and with its 32-bit CPU, it can easily beat Arduino UNO in performance. As an added bonus, you can easily program this board using your Arduino IDE (although with some tweaks and additional programmer i.e. USB to USART converter). Coming to the Blue Pill board itself, you get the board and two male header strips for you to solder on to the board.

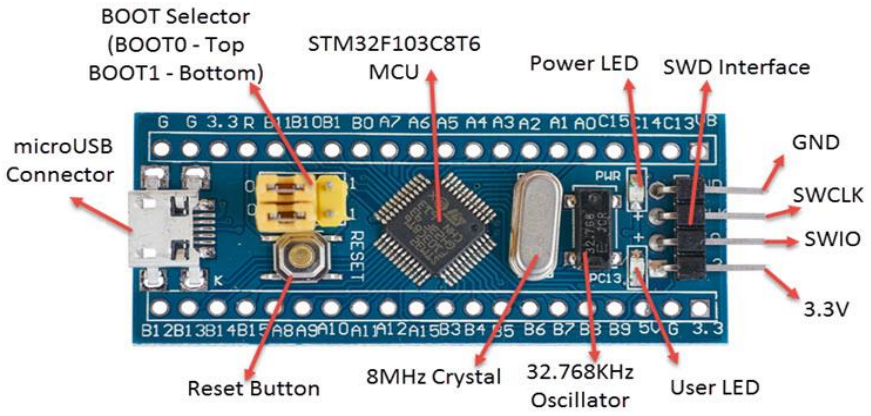


Figure 5.6 STM Blue Pill Board

The other features of the board are as follows:

- It contains the main MCU – the STM32F103C8T6 in a Quad Flat Package.

- A Reset Switch – to reset the Microcontroller.
- microUSB port – for serial communication and power.
- BOOT Selector Jumpers – BOOT0 and BOOT1 jumpers for selecting the booting memory.
- 8 MHz Crystal – Main Clock for MCU and 32.768KHz Oscillator – RTC Clock.
- 3.3V regulator (on the bottom) – converts 5V to 3.3V for powering the MCU.

On either long edge of the board, there are pins for connecting various Analog and Digital IO and Power related stuff. The following image shows the pin configuration of the board along with different functions supported by each pin.

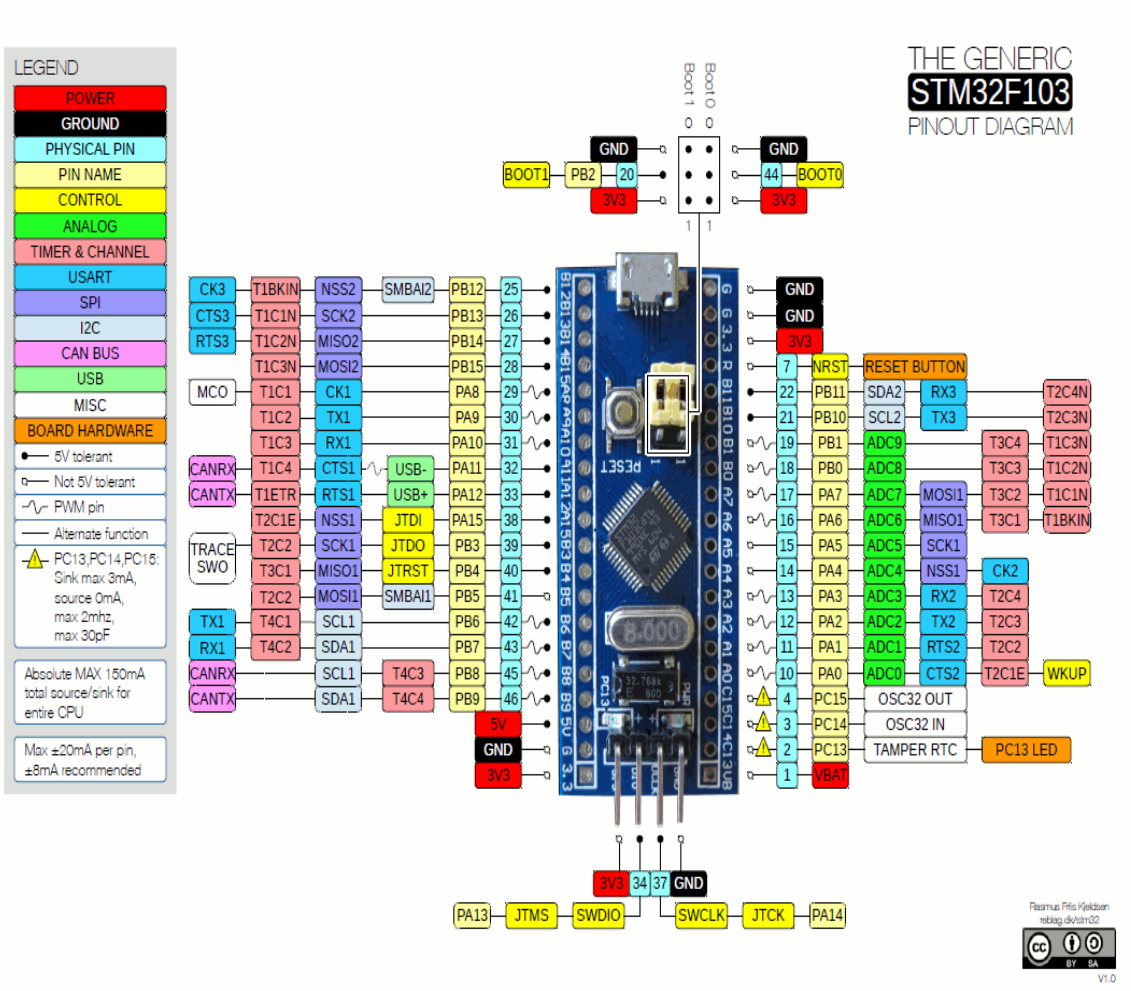


Figure 5.7 Pin layout of STM32 Bluepill Board

5.6.1 Programming STM32F103C8T6 Blue Pill Board using Arduino IDE

- Install Arduino IDE. After that open your Arduino IDE and select File -> Preferences. You will find a tab called “Additional Boards Manager URLs”. Copy the following link and paste it there as shown in figure.

"*https://github.com/stm32duino/BoardManagerFiles/raw/master/STM32/package_stm_index.json*"

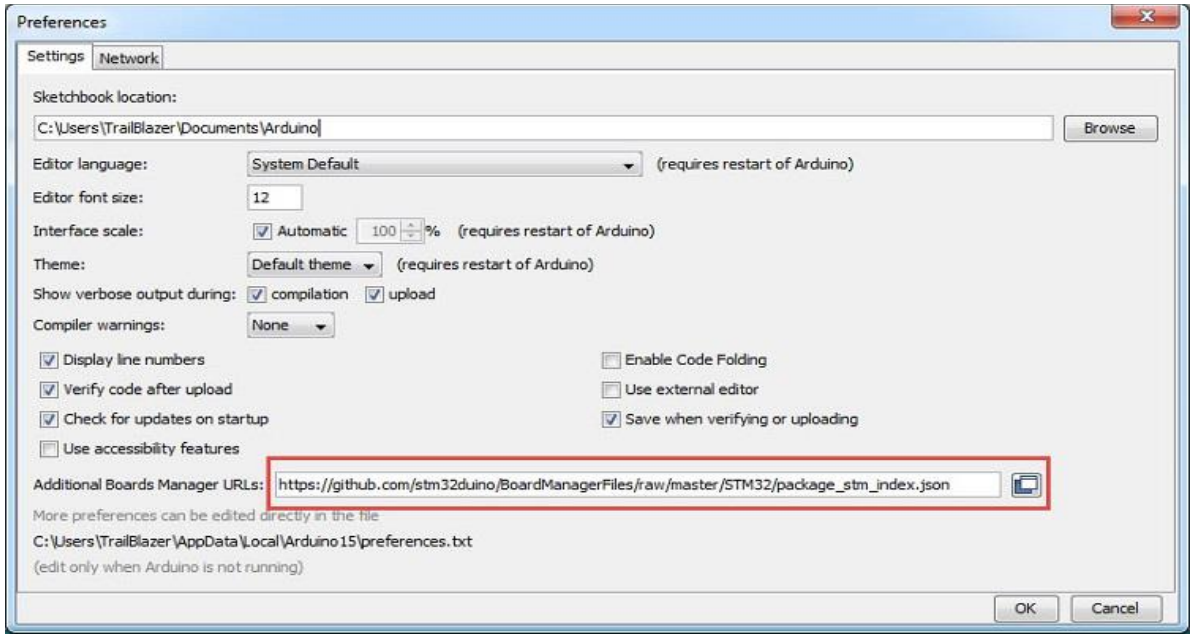


Figure 5.8 Entering Board URL in Arduino IDE

Now, go to Tools -> Board -> Board Manager… option and search for "stm32". You will get a result like "STM32 Cores by STMicroelectronics". This will take some time as it will download and install some of the necessary files and tools.

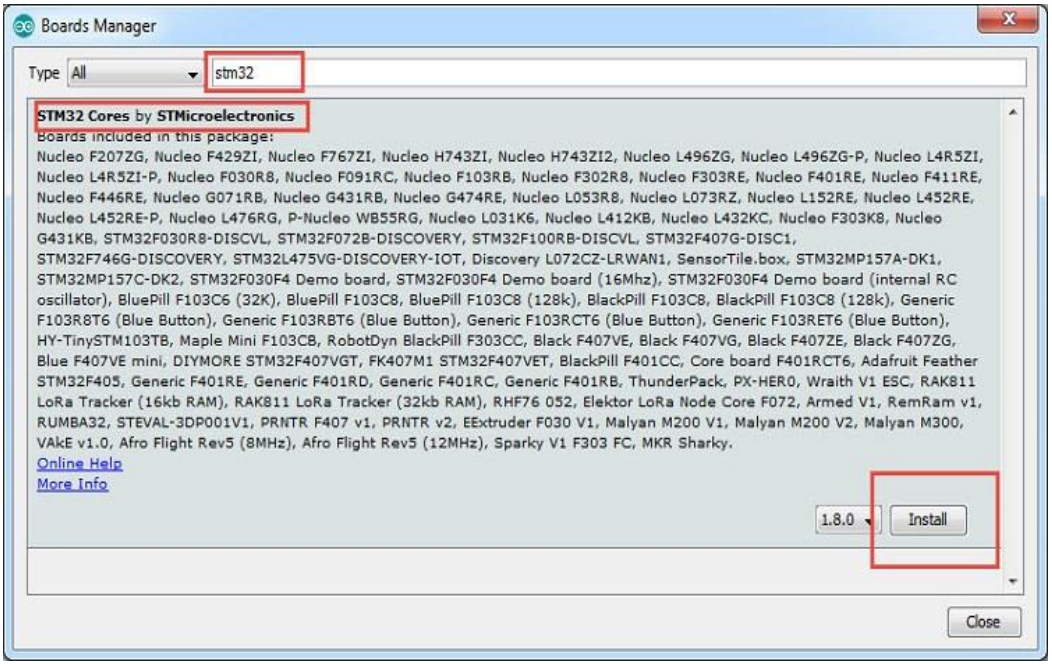


Figure 5.8 Installing STM32 Library in Arduino IDE

Now you can select the board from Tools -> Board -> Generic STM32F1 series. Once you select this board, a bunch of options will appear below for customizing your board type. The first important option is “Board part number”. Make sure that “BluePill F103C8” is selected.

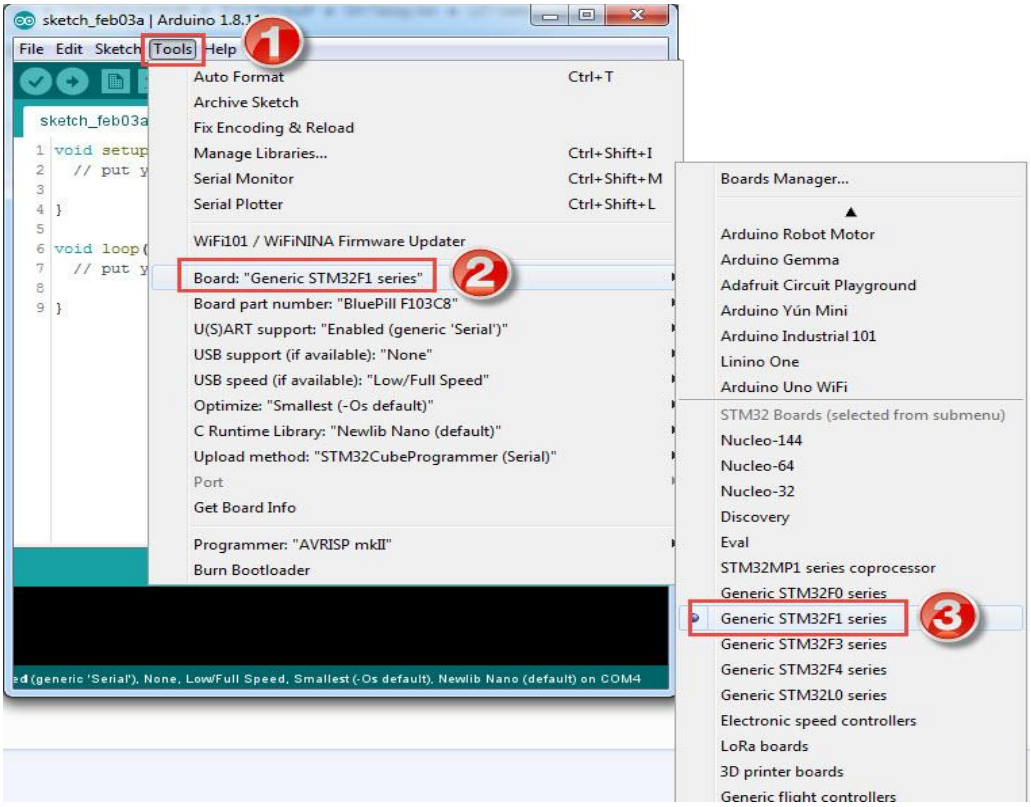


Figure 5.9 Selecting STM32 board for Programming

The other important options are “U(S)ART support”, make it as “Enabled (generic ‘Serial’)” and “Upload method”, make its as “STM32CubeProgrammer (Serial)”. You can leave the remaining options as their default values.

Write the Blinky program as follows. It is similar to the Arduino Blinky sketch but instead if LED_BUILTIN, I have used PC13 as the LED is connected to that pin of the MCU.

After this, you can click on Upload and the IDE will start compiling the code. It will take some time for compiling. Once the compilation is successful, it will automatically invoke the STM32CubeProgrammer tool. If everything goes well, the IDE will successfully program the STM32 Board.

Code

```
void setup() {
   pinMode(PC13, OUTPUT);
}
void loop() {
  digitalWrite(PC13, HIGH);
  delay(1000);
  digitalWrite(PC13, LOW);
  delay(1000);
}
```

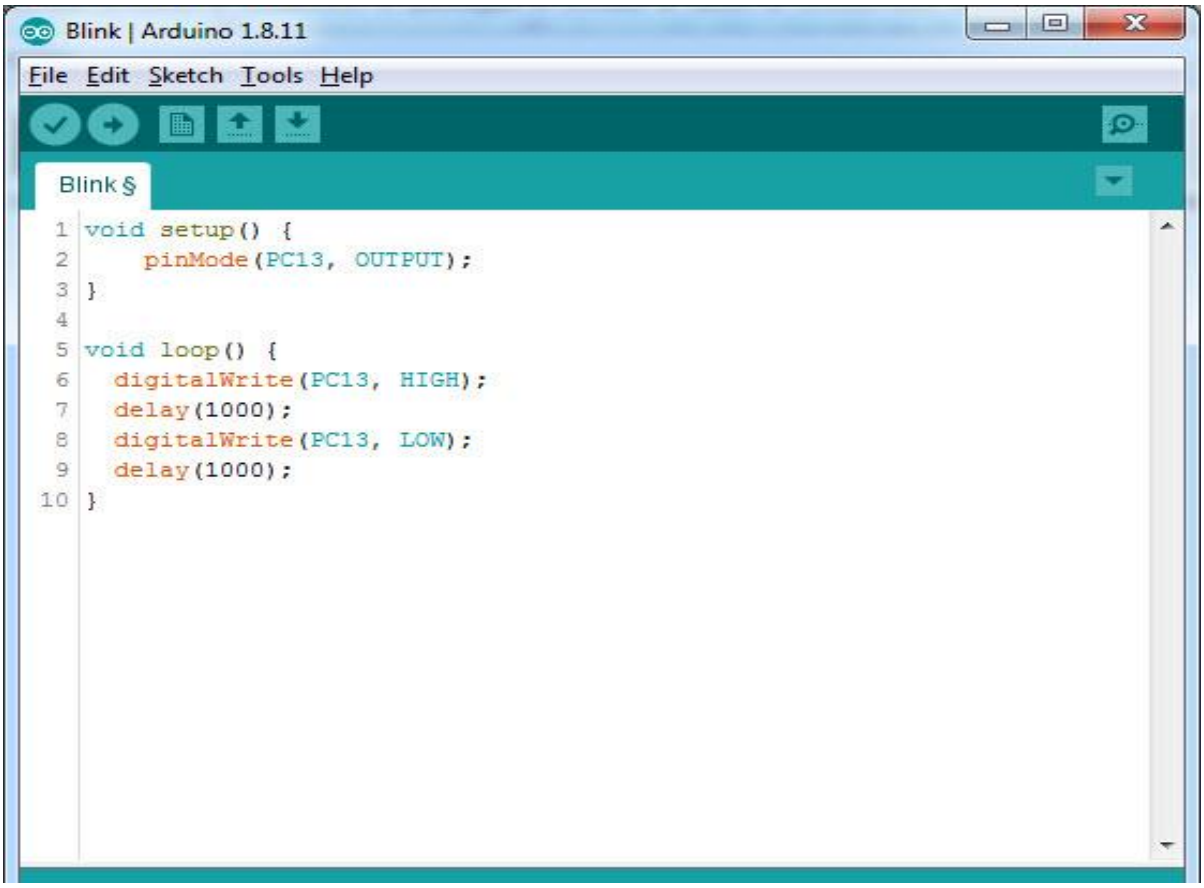
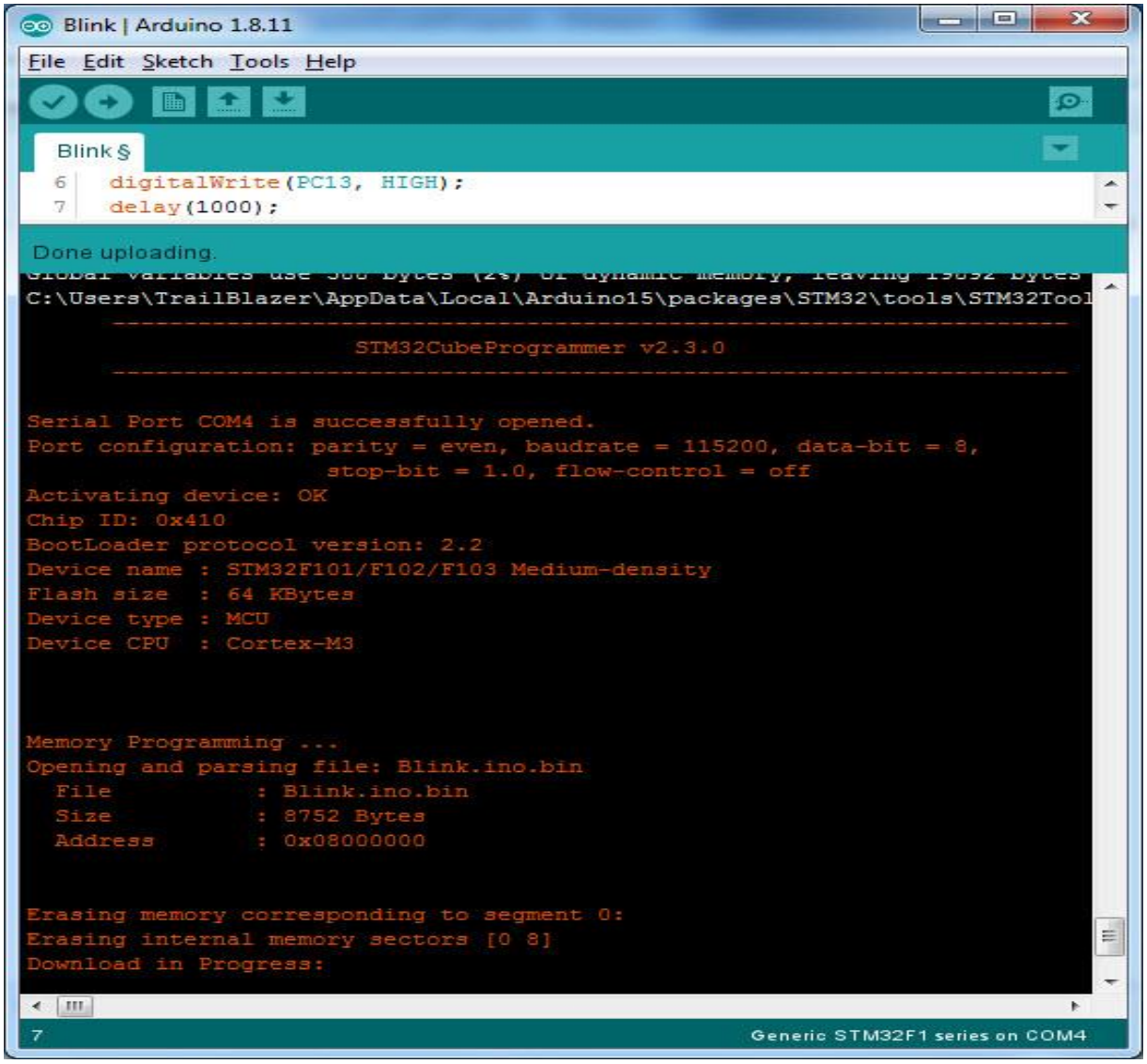


Figure 5.10 Writing the Code for STM32 in Arduino IDE



It will automatically reset the MCU and you can notice the LED blinking. Don't forget to move the BOOT0 pins back to LOW position so that the next time you power-on the board, it will start running the previously uploaded program.

TEXT / REFERENCE BOOKS

[1] Joseph Yiu, "The Definitive Guide to the ARM Cortex-M3", Newnes, 2nd Edition, 2009.

[2] Muhammad Ali Mazidi, "The STM32F103 Arm Microcontroller and Embedded Systems: Using Assembly and C",1st Edition, Naimi and Mazidi books,2019

Exercise Questions

1. Identify four applications for which ARM cortex-A processor is best suited.
2. Identify four applications for which ARM cortex-R processor is best suited.
3. Outline the key features of Cortex-M ARM processor family.
4. Mention two microcontrollers that are based on Cortex-M architecture.
5. List the any four operating systems that can be used with ARM processors.
6. Outline the key benefits of developing OS based embedded systems compared to bare metal embedded systems.
7. Distinguish firmware and software.
8. Contrast BIOS firmware and EFI firmware.
9. Discuss on the evolution of ARM Processor Architectures.
10. Illustrate the functional architecture of ARM Cortex-M3 32-bit processor and outline the functions of each modules.
11. Compare and contrast CORTEX-A, CORTEX-M, CORTEX-R processors.
12. Discuss the importance and applications of on-chip embedded In-circuit emulators (ICE) in ARM cortex processors
13. Exempifly the different stages of embedded system development process with a flow chart.