Chettinad college of engineering and technology

Department of EEE

EE3018 – Embedded processor

Reading Notes

Prepared By – S.Ragul Ap/EEE

EE3018 EMBEDDED PROCESSORS **L T P C 2 0 2 3**

COURSE OBJECTIVES:

- To introduce the architecture of the ARM processor.
- To train students in ARM programming.
- To discuss memory management, append location development with an ARM processor.
- To involve Discussions/ Practice/Exercise in revising & familiarizing the concepts
- To impart the knowledge on single board embedded processors.

UNIT I ARM ARCHITECTURE **6**

Architecture – Memory Organization – addressing modes -Registers – Pipeline - Interrupts – Coprocessors – Interrupt Structure

UNIT II ARM MICROCONTROLLER PROGRAMMING **6**

ARM general Instruction set – Thumb instruction set –Introduction to DSP on ARM- basic programming.

UNIT III PERIPHERALS OF ARM **6**

ARM: I/O Memory – EEPROM – I/O Ports – SRAM –Timer –UART - Serial Communication with PC – ADC/DAC Interfacing-stepper motor interfacing

UNIT IV ARM COMMUNICATION **6**

ARM With CAN, I^2C, and SPI protocols

UNIT V INTRODUCTION TO SINGLE BOARD EMBEDDED PROCESSOR **6**

Raspberry Pi Architecture - Booting Up RPi- Operating System and Linux Commands -Working with RPi using Python and Sensing Data using Python-programming - GPIO and interfacing peripherals With Raspberry Pi

30 PERIODS

LAB COMPONENTS: **30 PERIODS**

1. Laboratory exercise:
   a) Programming with IDE - ARM microcontroller
   b) Advanced Timer Features, PWM Generator.
   c) RTC interfacing with ARM using Serial communication programming, Stepper motor control.
   d) ARM-Based Wireless Environmental Parameter Monitoring System displayed through Mobile device.
2. Seminar:
   a) ARM and GSM/GPS interfacing
   b) Introduction to ARM Cortex Processor
3. Raspberry Pi based Mini project.

TOTAL: 30+30 = 60 PERIODS

COURSE OUTCOMES:

At the end of this course, the students will have the ability to

CO1: Interpret the basics and functionality of processor functional blocks.
CO2: Observe the specialty of RISC processor Architecture.
CO3: Incorporate the I/O hardware interface of processor with peripherals.
CO4: Emphasis the communication features of the processor.
CO5: Improved Employability and entrepreneurship capacity due to knowledge up gradation on

UNIT – I ARM ARCHITECTURE

Introduction to ARM Architecture

Advanced RISC Machine or Acorn RISC Machine is the architecture with different computing architectures set to be used in different environments. 32-bit and 64-bit can be used here in different computer processors. It was developed by Arm Holdings and the architecture is updated in between. This architecture is specified to be used with CPU, different chips in the system, and in different registers. Reduced Instruction Set Computing helps in creating instructions for the system to be used for several purposes. Smartphones, microcomputers, and embedded devices also use ARM architecture for the instruction set in the registers.

Many Arm-based devices are used all over the world and are one of the popular architectures within the devices. The architecture can be divided into A, R, and M profiles. A profile is mainly for applications, R profile is for real-time and M profile is for Microcontroller. A profile helps to maintain high performance and is designed to run the complex system in Linux or Windows. R profile checks for systems with real-time requirements and is found in networking equipment or embedded control systems. M profile is used in IOT devices and can be synchronized with small and high-power devices.

What is ARM Processor?

ARM Processors derive their name from the acronym "Advanced RISC Machines," signifying their connection to the broader family of CPUs that adhere to the RISC architecture. In essence, Advanced RISC Machines have laid the foundation for the architecture of ARM processors, giving them their distinctive name.

The hallmark of ARM processors lies in their architectural simplicity, resulting in a compact size and reduced complexity. This inherent simplicity allows for optimal performance when integrated into a system, delivering high levels of efficiency.

ARM Processor Architecture

The ARM processor architecture is a family of RISC (Reduced Instruction Set Computing) processors developed by ARM Holdings. It is characterized by its energy efficiency, performance, and scalability, making it a popular choice in a wide range of devices and applications.

The key features of the ARM processor architecture include:

- **Instruction Set Architecture (ISA):** ARM processors use a 32-bit or 64-bit instruction set architecture, depending on the specific version. The ISA defines the supported instructions, addressing modes, and registers.
- **RISC Design:** ARM processors follow the RISC design philosophy, which emphasizes simplicity and efficiency. They have a streamlined instruction set with fixed-length instructions, a large set of general-purpose registers, and a load/store architecture.
- **Thumb Instruction Set:** The Thumb instruction set is an extension of ARM architecture that uses 16-bit instructions instead of 32-bit instructions. It provides improved code density and is commonly used in memory-constrained systems.

- **ARM and Thumb-2 Interworking:** ARM processors support a feature called ARM and Thumb-2 interworking, which allows seamless switching between ARM and Thumb instructions during runtime. This provides flexibility in code optimization and execution.

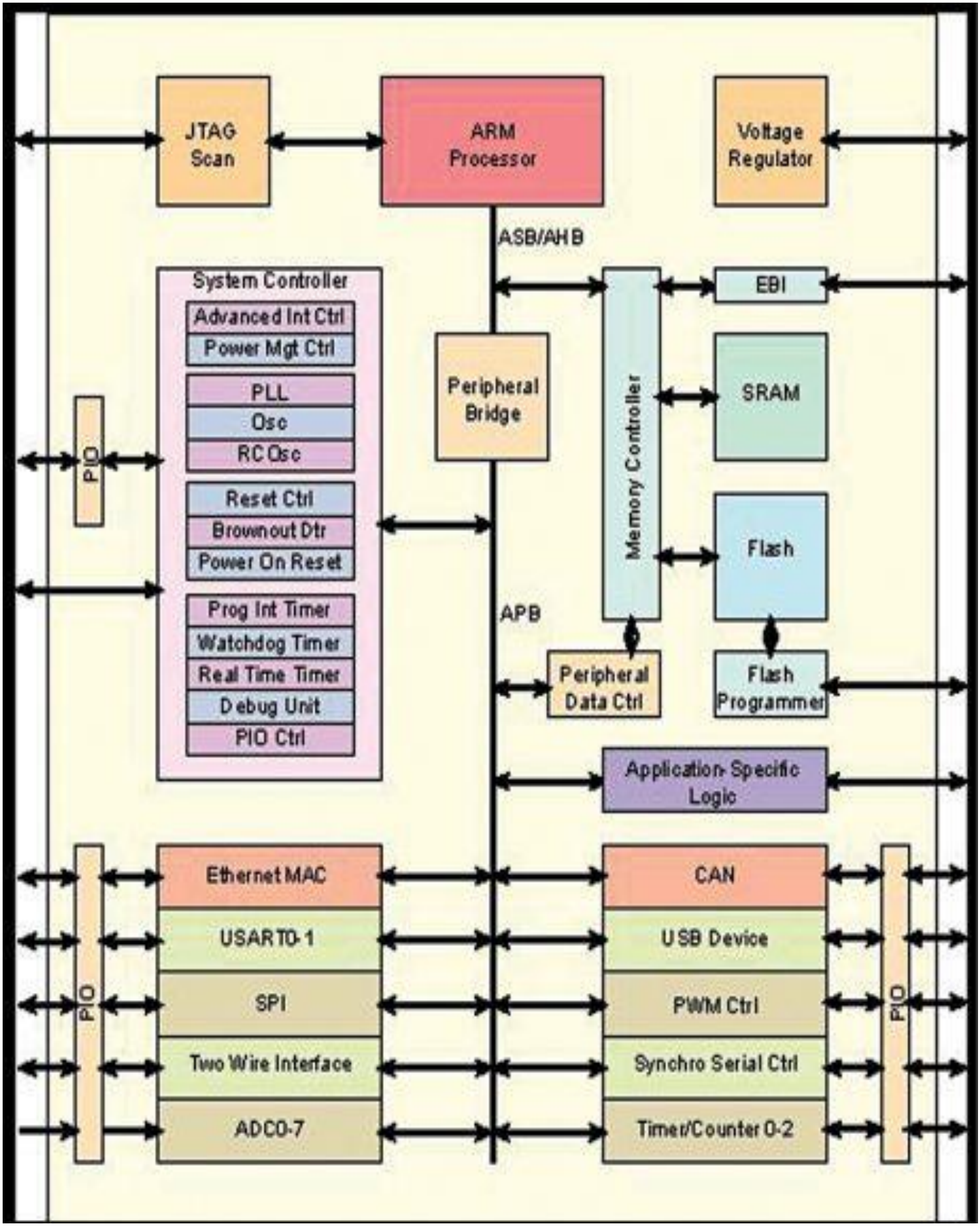
  - **Multiple Execution Modes:** ARM processors have different execution modes, including User mode, Supervisor mode, Interrupt mode, and System mode. Each mode has its own set of privileges and access rights, enabling secure and controlled execution of software.

- **Coprocessors:** ARM processors support coprocessors, which are specialized hardware units that assist in executing specific tasks such as floating-point operations or digital signal processing. These coprocessors can be integrated directly into the processor core or implemented as separate modules.

- **SIMD (Single Instruction, Multiple Data) Extensions:** Some ARM processors feature SIMD extensions, such as the NEON technology, which allows parallel processing of multiple data elements. This is particularly useful for multimedia and signal processing applications.

- **TrustZone:** TrustZone is a security feature available in ARM processors that enables hardware-based isolation between secure and non-secure environments. It allows for the secure execution of sensitive tasks and protects critical data.

- **Power Efficiency:** ARM processors are known for their energy efficiency, consuming lower power compared to other processor architectures. This makes them suitable for battery-powered devices and systems with strict power constraints.



BLOCK DIAGRAM OF ARM ARCHITECTURE

Components of ARM Architecture

Given below are the components mentioned:

- The instruction set in the architecture describes the function of each instruction and explains the representation of the instruction in the memory through encoding. With the help of these instructions, it is easy to manage the architecture and the microprocessors.
- Priority encoders help to load the instruction and to store it in the specified register in order to manage the files. This helps to identify the registers and instructions easily in the architecture.
- Multiplexers are used in the architecture to manage the operation of processor buses. The components are instructed to work in behavioral mode and the components are implemented as an entity. The architecture of the entity is optimized depending on the application of the processor and helps to construct and maintain the design.
- 32-bit inputs are used in Arithmetic and Logic Unit which comes from register files and shifter. The outputs are modified in register flags. There are C flag, V flag, S flag and Z flag. 16 opcodes can be implemented with the help of 4-bit function bus in the microprocessor. V-bit output goes to the V flag and C output to the C flag and so on.
- Booth Multiplier Factor has 32-bit inputs to manage from the register file. The output given is also 32-bit and the multiplication starts when the input is actively working with the processor.
- Also, another multiplication rule is used in the system for 2's complement numbers. The Booth algorithm manages both positive and negative numbers with the same importance and faster multiplication is performed by ignoring 0's and 1's in the process. 16 clock cycle manages the multiplication cycle in the system.
- 32-bit input shifting is represented with the help of Barrel Shifter. The input is either from the register file or from the data given by the user for faster output. Shifter manages these with the help of different controls and also with the functionalities of the system. The logical or arithmetic operation to be performed is indicated in the shift field of the barrel shifter. The quantity is mentioned in the instruction field and it can be as low as 6 bits in the register. It allows up to 32-bit file and responds to left, right, arithmetic right, and rotate right shifts. This works well with multiplexers in the microprocessor.
- The Control unit controls the entire process of the architecture and manages the system operation. It can be made of circuits or can be the combination of functions and circuits in the design. Timing is managed in the control unit and works with a combination of a state machine in the processor. The operation in the processor is managed and controlled with the help of signals from the control unit as these are connected with different components in the system.
- There is a register set in the architecture to help in managing the registers with their corresponding time in the structure. It is important to check for the number of registers and the size of the same for the proper functioning of the system. The function of registers is to manage the files and keep them in the proper place and check for their initial state in the processor.
- The exception model in the architecture helps to know the different levels of access provided in the system and the types of exceptions in the system. When an exception is accepted, the changes undergone in the system are also noticed. Breakpoints are noticed and information is captured with the help of tracepoints.

Features of ARM Processor

As mentioned earlier, ARM processors are built upon the principles of RISC architecture. Consequently, they incorporate essential characteristics associated with RISC architecture, which encompass the following:

1. **Instruction Set:** ARM processors employ 32-bit instructions, enabling the fetching of each instruction in a single cycle, simplifying operations. Fixed-length instructions allow for the fetching of future instructions while

previous ones are being executed. In contrast, CISC architecture lacks this feature, with variable-sized instructions requiring multiple cycles for execution. ARM processors excel in straightforward instruction decoding.

2. **Register Architecture:** RISC machines feature expansive, uniform register files. With 37 registers, each 32 bits in size, only 16 can be utilized at a given time. Unlike CISC processors, where registers have specific dedicated purposes, RISC allows any register to hold either data or an address. This enhances the overall execution process of the system.

3. **Pipelining:** ARM processors adopt a three-stage pipelining approach that maximizes throughput. In this setup, while the first instruction is being executed, the subsequent one is being decoded, and the next-to-next instruction is being fetched. This simultaneous fetching, decoding, and execution enable progress of one step per cycle, saving time. Consequently, ARM processors do not require microcodes for instruction execution, as seen in CISC processors.

4. **Load/Store Model:** ARM architecture follows a load/store model, where all operations occur within registers. Data is loaded from memory into registers using load operations, and operations are performed on that data. Once the operation is completed, the resulting data is stored back into memory. Unlike CISC processors, which support memory-based operations, ARM processors do not allow direct processing on memory.

ARM Instruction Sets

Below are some various ARM Instruction Sets:

Branch Instruction

Branch instructions in ARM processors cause an immediate switch in execution to a specified address location. When a branch (B) instruction is encountered, the processor seamlessly transitions to the designated location and proceeds to execute the operation from there. These instructions facilitate both forward and backward branches, allowing for branching within a range of up to 32 MB.

Data Processing instructions

The various data processing instructions occur within the general-purpose registers. These instructions include:

- **Arithmetic and logic instructions:** These instructions serve the purpose of executing diverse arithmetic and logic operations. They typically involve two source operands, and the resulting output is stored in a designated destination register. Notably, the outcomes of these instructions can be directly written into the Program Counter, which functions as a versatile register.

- **Comparison instructions:** Similar in format to arithmetic and logic instructions, comparison instructions facilitate comparisons between two operands. However, instead of storing the results in registers, these instructions update the contents of flag registers.

- **Multiply instructions:** These instructions are categorized based on the length of the resulting bits after multiplication:

- **32-bit result:** This category represents normal multiplication results, where all 32 bits of the product are stored within a single register.

- **64-bit result:** These instructions produce longer results that span 64 bits. Storing the entire 64-bit value requires the use of two separate registers.

- **Count leading zero instruction:** With this instruction, the device counts the number of consecutive leading zeros in a given sequence, starting from the Most Significant Bit (MSB) to the Least Significant Bit (LSB). The count obtained from this operation is then stored within a register.

Load and Store Instruction

These Instructions are as follows:

Load and Store register: The load register instruction enables the loading of 8-bit, 16-bit, or 32-bit data from memory into a register. Conversely, the store register instruction facilitates the transfer of data from a register to memory.

Load and Store multiple registers: This instruction provides the capability to load or store multiple general-purpose registers as a block from or to memory. It supports various addressing modes, including pre-increment, post-increment, pre-decrement, and post-decrement.

Swap register and memory content: The swap (SWP) instruction functions sequentially in the following manner:

- Loading a value from a specified memory location into a register.
- Storing the current content of the register into the same memory location.
- Simultaneously, the loaded value from memory is stored back into the register.

By maintaining the same register for both steps, the data within the register and memory location are interchanged.

Status Register Transfer Instruction

This instruction enables the transfer of the current program status registers content to or from a general-purpose register. It involves the following steps:

- Setting the value of the condition code flag.
- Configuring the interrupt enables bits.
- Establishing the processor mode.

Applications of ARM Processor

ARM processors have found widespread application across various industries and devices. Some notable applications of ARM processors include:

Mobile Devices: ARM processors have gained significant popularity in smartphones, tablets, and wearable devices due to their power efficiency, compact size, and strong performance capabilities. They are widely used in devices running operating systems like Android and iOS.

Embedded Systems: ARM processors are extensively used in embedded systems, such as automotive electronics, industrial automation, smart appliances, and IoT devices. Their low power consumption, scalability, and real-time processing capabilities make them well-suited for these applications.

Consumer Electronics: ARM processors are found in a range of consumer electronic devices like digital cameras, gaming consoles, set-top boxes, and smart TVs. Their ability to handle multimedia processing, graphics rendering, and connectivity features makes them ideal for delivering rich user experiences.

Automotive Systems: ARM processors play a crucial role in automotive systems, powering functions such as engine control units (ECUs), infotainment systems, driver assistance systems, and telematics. They offer the necessary performance, safety features, and reliability for automotive applications.

Networking and Communication: ARM processors are used in networking equipment such as routers, switches, and wireless access points. Their high-speed data processing capabilities, support for networking protocols, and low power consumption make them suitable for networking and communication infrastructure.

Healthcare Devices: ARM processors find applications in medical devices, such as patient monitoring systems, diagnostic equipment, and portable medical devices. Their low power consumption, real-time processing, and integration with sensor technologies enable advanced healthcare solutions.

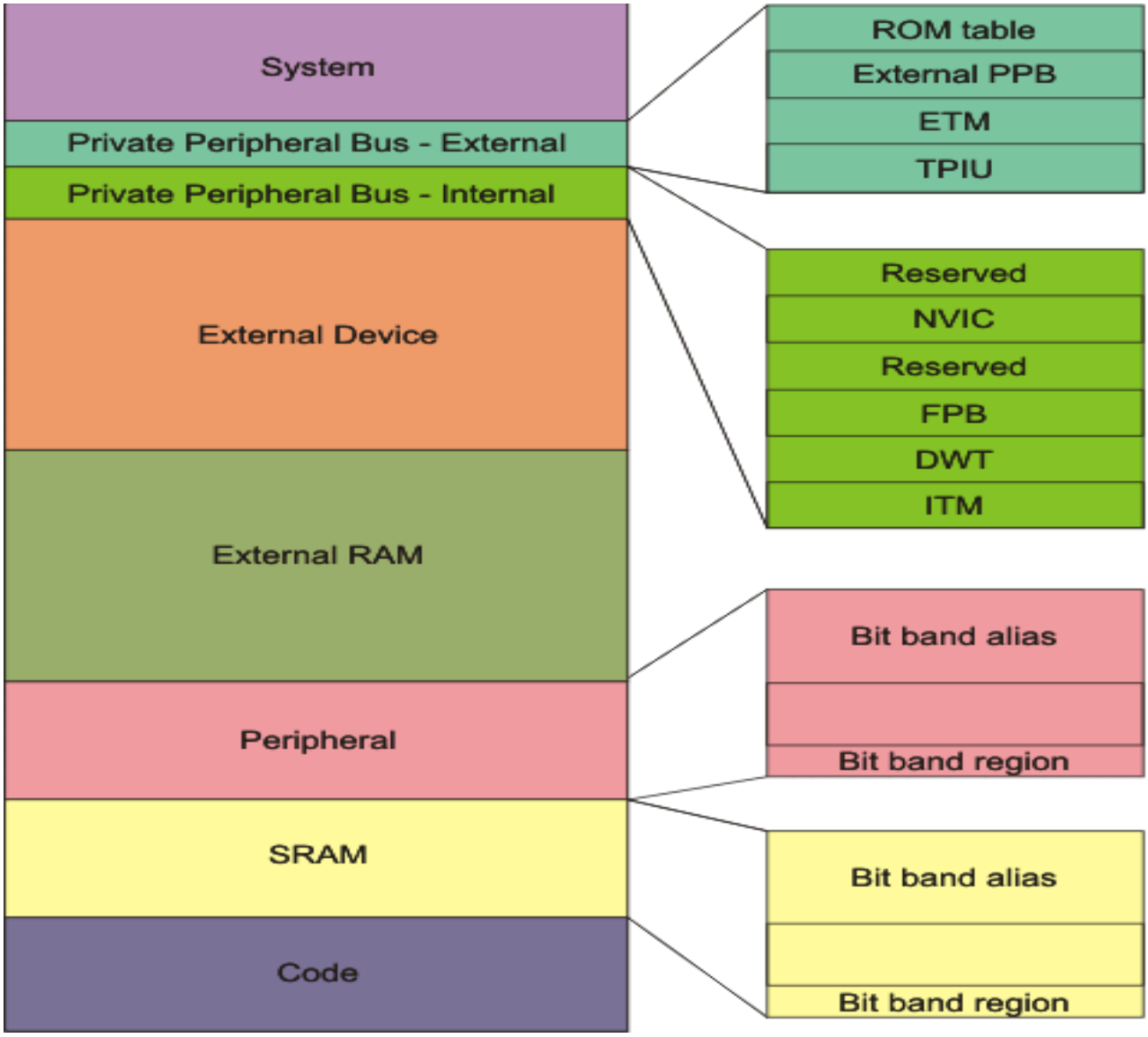
Server and Data Centers: ARM processors are increasingly being adopted in server and data center environments. Their energy efficiency and scalable architecture make them attractive for handling large-scale data processing, cloud computing, and virtualization.

ARM Memory Organization

The Cortex-M3 and Cortex-M4 have a predefined memory map. This allows the built-in peripherals, such as the interrupt controller and the debug components, to be accessed by simple memory access instructions.
Thus, most system features are accessible in program code. The predefined memory map also allows the Cortex-M3 processor to be highly optimized for speed and ease of integration in system-on-a-chip (SoC) designs.

Overall, the 4 GB memory space can be divided into ranges as shown in picture below. The Cortex-M3 design has an internal bus infrastructure optimized for this memory usage.

A graphical representation of the ARM memory is shown in picture below :



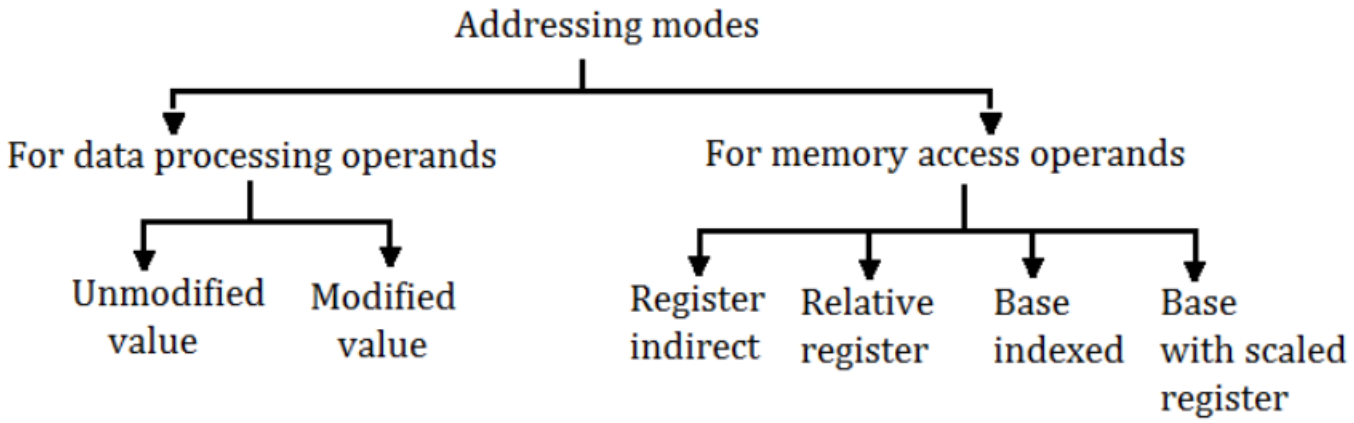
The ARM Cortex-M3 memory is divided into following regions :

- **System** - .
- **Private Peripheral Bus - External** - Provides access to :
  - the Trace Port Interface Unit (TPIU),
  - the Embedded Trace Macrocell (ETM),
  - the ROM table,
  - implementation-specific areas of the PPB memory map.
- **Private Peripheral Bus - External** - Provides access to :
  - the Instrumentation Trace Macrocell (ITM),
  - the Data Watchpoint and Trace (DWT),
  - the Flashpatch and Breakpoint (FPB),
  - the System Control Space (SCS), including the MPU and the Nested Vectored Interrupt Controller (NVIC).
- **External Device** - This region is used for external device memory.
- **External RAM** - This region is used for data.
- **Peripheral** - This region includes bit band and bit band alias areas.

    - Peripheral Bit-band alias - Direct accesses to this memory range behave as peripheral memory accesses, but this region is also bit addressable through bit-band alias.
    - Peripheral bit-band region - Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write.
- **SRAM** - This executable region is for data storage. Code can also be stored here. This region includes bit band and bit band alias areas.
    - SRAM Bit-band alias - Direct accesses to this memory range behave as SRAM memory accesses, but this region is also bit addressable through bit-band alias.
    - SRAM bit-band region - Data accesses to this region are remapped to bit band region. A write operation is performed as read-modify-write.
- **Code** - This executable region is for program code. Data can also be stored here.

Addressing Modes in ARM

The phrase addressing modes in ARM relates to the manner an instruction operand is expressed. Before the operand is actually performed, the addressing mode provides a rule for interpreting or altering the address field of the instruction.



Data Processing Operand (i.e. op1) Addressing Modes

There are two approaches to dealing with these operands.
Unaltered value :The register or a value is delivered unchanged, that is, without any shift or rotation, in this addressing style, for example, I MOV R0, # 1234 H The instantaneous constant value 1234 will be moved into register R0 by this instruction.
Modified value :The provided value or register is shifted or rotated in this addressing mode. As shown below with examples, there are several shift and rotate operations that may be performed.
(1) Leftward logical shift This will move the value of a register by n bits in the direction of the most significant bits.For eg. MOV R0, R1, LSL #2
R0 will become the value of R1 shifted 2 bits once this instruction is executed.

(2) Right logical shift This shifts the value of a register by n bits to the right. MOV R0, R1, LSR R2 are several examples. R0 will have the value of R1 moved right by R2 times once this instruction is executed. R1 and R2 are unaffected.

(3) Right arithmetic shift This is identical to logical shift right, except that for arithmetic shift operations, the MSB is maintained as well as shifted, e.g. MOV R0, R1, ASR #2. R0 will have the value of R1 Arithmetic; moved right by 2 bits after this instruction is executed.

(4) Rotate to the right. This will rotate the value of a register by n bits, for example MOV R0, R1, ROR R2. R0 will have the value of R1 rotated right for R2 times once this instruction is executed.

(5) Extend right rotation This is comparable to Rotate right by one bit with the carry flag put into the MSB,.The value of register R1 will be rotated right through carry by 1 bit once this instruction is executed.

Addressing Modes for Memory Access Operand

To access memory, load and store instructions are utilised, as previously mentioned. The various memory access addressing modes are as follows:

(i) Register indirect addressing mode
(ii) Indirect addressing mode for relative registers
(iIi) Indirect addressing method with a base index
(iv) Scale register addressing mode as a base

As demonstrated in the examples for each addressing method, offset addressing, pre-index addressing, and post-index addressing are all available.

(i) Register indirect addressing mode

A register is utilised in this addressing mechanism to provide the address of the memory region to be accessed. LDR R0, [R1] is an example. This instruction loads the 32-bit word at the memory location contained in register R0 into register R0.

(ii) Indirect addressing mode for relative registers

An instantaneous value applied to a register generates the memory address in this addressing style. This method of addressing allows for both pre- and post-indexing.
For example, (a) LDR R0, [R1, #4]
This instruction loads the register R0 with the word at the memory regions computed by adding the constant address included in the R1 register value 4 to the memory address stored in the R1 register, e.g. (b) LDR R0, [R1, #4]!
This is a type of addressing that comes before the index. This instruction is the same as (a) in that it sets the new address in R1,
i.e. R1 (R1 + 4). e.g. (c)'LDR, [R1], #4
This is referred to as post-index addressing. This instruction loads the word at the memory location supplied in register R1 into register R0. It will then compute the new address by multiplying R1 by 4 and storing it in R1.

(ii) Indirect addressing method with a base index

The memory address is produced by adding the values of two registers in this addressing scheme.. For example, (a) LDR R0, [R1, R2]
Pre indexing and post indexing is also supported.
This instruction will load the word at the memory address determined by adding register R1 and register R2 into register R0.
For example, (b) LDR R0, [R1, R2]!
This is pre-index addressing.
This instruction loads the word at the memory location supplied in register R1 into register R0. The new address will then be calculated by adding the value in register R2 to register R1 and placed in R1.

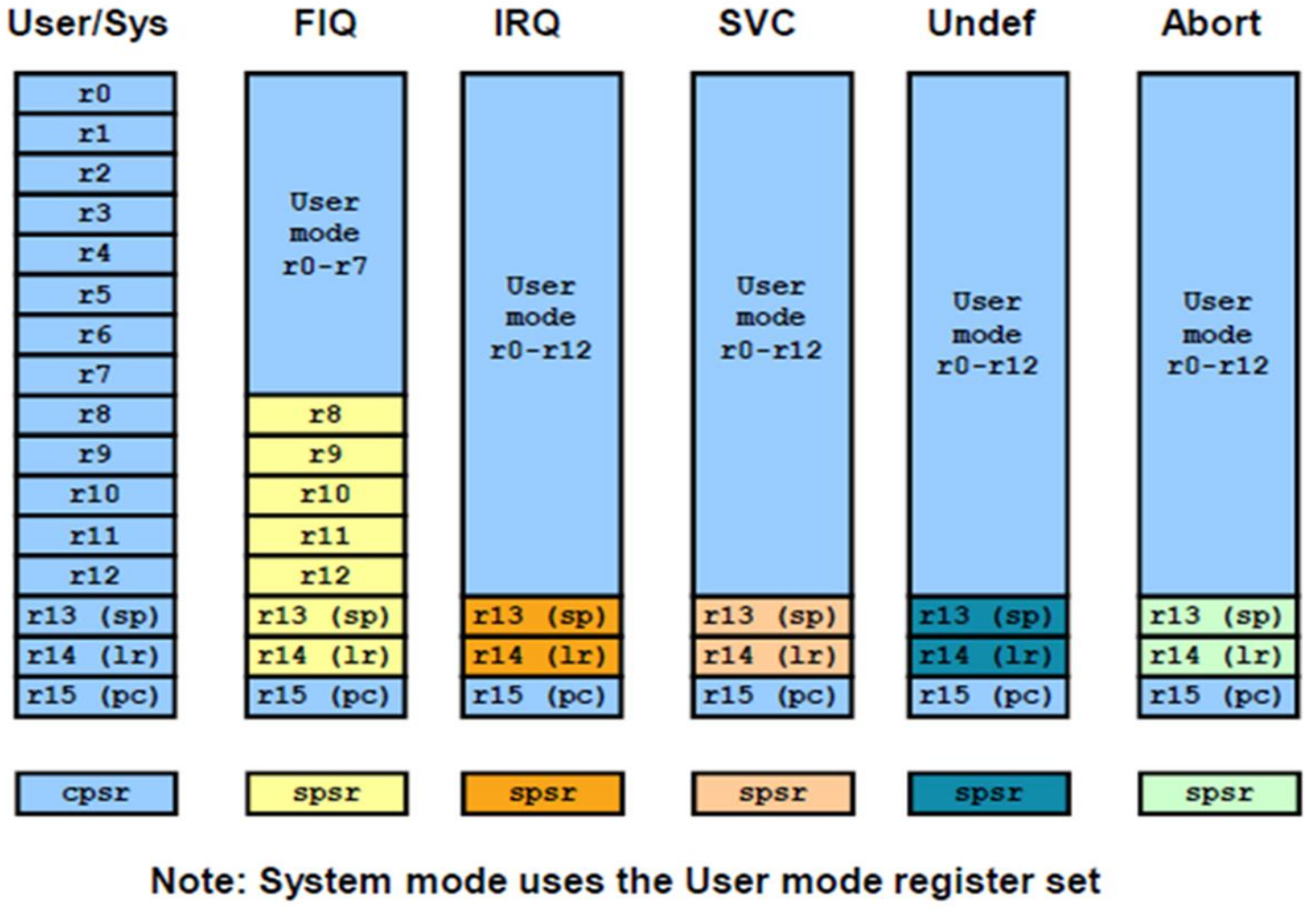
(iii) Scaled register addressing mode as a base

The memory address is produced by adding a register value to another register that is moved left in this addressing scheme. This addressing technique allows for pre-indexing and post-indexing.
For eg., (a) LDR R0, [R1, R2, LSL #2]
The word at the memory location computed by adding register leith register R2 shifted left by 2 bits will be loaded into register R0 using this command. For instance, (b) LDR RO,[R1, R2, LSL #2]!

ARM Registers

Basically there are two types of registers – General purpose registers and Special purpose registers. General-purpose registers hold either data or an address. The letter r is prefixed to the register number to identify them. For example, the label r4 is assigned to register 4. Figure depicts the active registers that are available in user mode, which is a protected state that is often utilized for running programmes. There are seven various modes that the CPU may work in, which we shall go through momentarily. The registers in this example are all 32 bits in size. Up to 18 active registers are available: 16 data registers and 2 processor status registers. The data registers are labeled r0 through r15 by the programmer. The ARM processor contains three registers: r13, r14, and r15, each of which is allocated to a specific duty or unique function. To distinguish them from the other registers, they are typically given separate labels. The coloured registers indicate which special-purpose registers have been allocated.

- Register 13 is traditionally used as the stack pointer (SP) and stores the head of the stack in the current processor mode.
- Register r14 is called the link register (LR) and is where the core puts the return address whenever it calls a subroutine.
- Register 15 is the program counter (PC) and contains the address of the next instruction to be fetched by the processor.



Few of these registers are actually banked and different registers are available for different processor modes.

SP (Stack Pointer)

- The stack pointer, often known as the SP register, is located in register r13.
- Each exception mode has its own version of r13, pointing to a stack specialised to that mode.
- Temporary values are stored in the stack.

LR (The link register)

- The Link register, also known as register r14, is used to store the subroutine's return address.
- When an exception occurs, the version of r14 in the exception mode is set to the address after the instruction is executed.
- SPSR is a copy of the CPSR just before an exception has occurred.

PC (Program Counter)

- The Program Counter, sometimes known as the PC, is located in register r15.
- It's used to determine which instruction will be executed next.
- The PC is referred to as an instruction pointer because it stores the address of the next instruction.

CPSR (Current Processor Status Register)

•Current processor status register (CPSR) contains the current status of the processor.
•This includes various conditional code flags, Interrupt Status Processor mode and other status and control information.
•The exception modes also have a saved processor status register (SPSR), that is used to preserve the value of CPSR when the associated exception occurs.
•Because the User and System modes are not exception modes, there is no SPSR available.

SPSR (Saved Processor Status Register)

In the exception modes there is an additional Saved Processor Status register (SPSR) which holds information on the processor's state before the system changed into this mode i.e. the processor status just before an exception.

ARM Pipelining

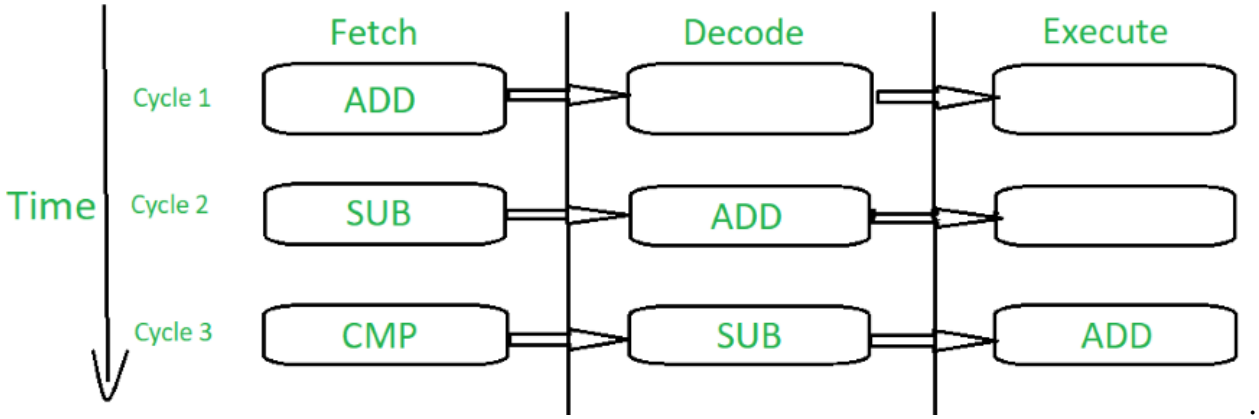
Pipelining is a design method or procedure that improves the efficiency of data processing in computer and microcontroller processors. By keeping the CPU in a continuous fetching, decoding, and execution process known as (the F&E cycle).

- RISC (Reduced instruction set computer) employs a pipelining approach to execute instructions. Pipelining in ARM boosts execution speed.
- by retrieving the instruction and executing it while other instructions are being decoded and executed at the same time.
- As a result, the memory system and CPU can run continually.
- Each ARM family has a distinct pipeline architecture.

3-s pipeline is the mechanism a RISC processor uses to execute instructions. Using a pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed. One way to view the pipeline is to think of it as an automobile assembly line, with each stage carrying out a particular task to manufacture the vehicle.

3-Stage Pipelining Schematic

- Fetch retrieves a memory instruction.
- Decode identifies the to-be-executed instruction.
- The instruction is processed by Execute, and the result is written back to the register.
- The speed of execution is boosted by overlapping the aforementioned steps of execution of various instructions.
- The pipelining technique allows the core to execute an instruction once every cycle, resulting in higher throughput



The figure illustrates the pipeline using a simple example. It shows a sequence of three instructions being fetched, decoded, and executed by the processor. Each instruction takes a single cycle to complete after the pipeline is filled. The three instructions are placed into the pipeline sequentially. In the first cycle the core fetches the ADD instruction from memory. In the second cycle the core fetches the SUB instruction and decodes the ADD instruction. In the third cycle, both the SUB and ADD instructions are moved along the pipeline. The ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched. This procedure is called filling the pipeline. The pipeline allows the core to execute an instruction every cycle. As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency. This in turn increases the performance. The system latency also increases because it takes more cycles to fill the pipeline before the core can execute an instruction. The increased pipeline length also means there can be data dependency between certain stages. You can write code to reduce this dependency by using instruction scheduling.

- It has the ability to finish its procedure in three cycles.
- It utilises the fundamental F&E cycle to achieve maximum throughput.
- This is why, when compared to its other family members, the ARM 7 has the lowest throughput.
- It works with 32-bit data.

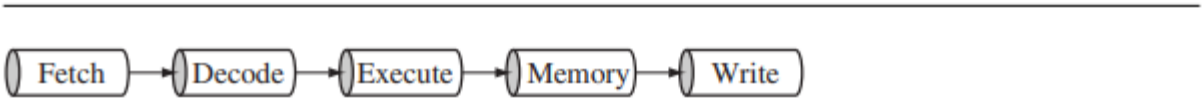
ARM7 Pipeline Characteristics

- An instruction in the ARM pipeline is not processed until it has completed the execution step.
- The PC always refers to the instruction address Plus 8 bytes throughout the execution step.
- PC always refers to the instruction address Plus 4 bytes when the processor is in thumb state.
- The ARM core flushes its pipeline when executing branch instructions or branching via direct change of the PC.
- Even if an interrupt has been raised, an instruction in the execution stage will finish its execution.

5-Stage Pipelining

3 stage pipeline vs 5 stage pipeline

The pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages. The ARM9 adds a memory and writeback stage, which allows the ARM9 to process on average 1.1 Dhrystone MIPS per MHz—an increase in instruction throughput by around 13% compared with an ARM7. The maximum core frequency attainable using an ARM9 is also higher.



ARM9 five-stage pipeline.

5 stage pipelining in ARM

- Pipelining is similar to ARM 7, however there are five phases in ARM 9.
- The procedure takes 5 cycles to finish.
- Pipelining in 5 stages
- It will fetch instructions from memory using the fetch command.
- Decode- It decodes the instructions acquired in the previous cycle.
- ALU – This step performs the instruction that was previously decoded.
- LS1 (Memory) loads/stores data given by load/store instructions.
- LS2(Write) extracts (zero or sign) data loaded by byte or half word load instruction and extends it.
- The throughput is 10 percent to 13 percent greater than ARM 7 due to an improvement in phases and efficiency.
- The ARM 9 core frequency is somewhat higher than the ARM 7 core frequency.

Interrupts on ARM

Most processor families have a mechanism to handle “hardware interrupts”. The ARM Cortex series is no exception – it has surprisingly sophisticated support for them, in fact.

But first: *what is an interrupt?*
You can think of a µC / CPU as performing the following task, millions of times per second:

- fetch the next instruction from memory
- advance the instruction pointer
- perform whatever that instruction says
- rinse and repeat

This is all nice and great, but it completely ignores the outside world. If this was all we had, then we’d have to constantly check all the hardware peripherals whether there is anything we need to take care of: serial input or output, some amount of clock time has passed, a digital or analog pin change, etc. Given the many possible sources of *occasional* work, we’d be wasting our time checking up on all that. With fast I/O, we’d have to check *very often!*
Here’s what an interrupt-aware CPU does instead:

- **if there’s an interrupt event, process it**
- fetch the next instruction from memory
- advance the instruction pointer
- perform whatever that instruction says
- rinse and repeat

So in a way, interrupts are nothing but lots of checking, done in hardware. At the start of every new instruction, the processor does the checking for us. Except that now, it’s virtually instant (and free): all the interrupt signals are OR-ed together to generate a single logic level – when set, there’s an interrupt “pending”, and the processor will divert its attention.

A couple of key points here:

- interrupts happen only *between* instructions
- interrupts can happen between *any* two instructions
- interrupts add overhead, but only when they actually happen

It seems so simple, but as you will see later on, this can cause all sorts of trouble.

An important note in the context of embedded microcontrollers: hardware interrupts are also essential for "waking up" a μC when it's in some low-power / power-down mode.

The stack

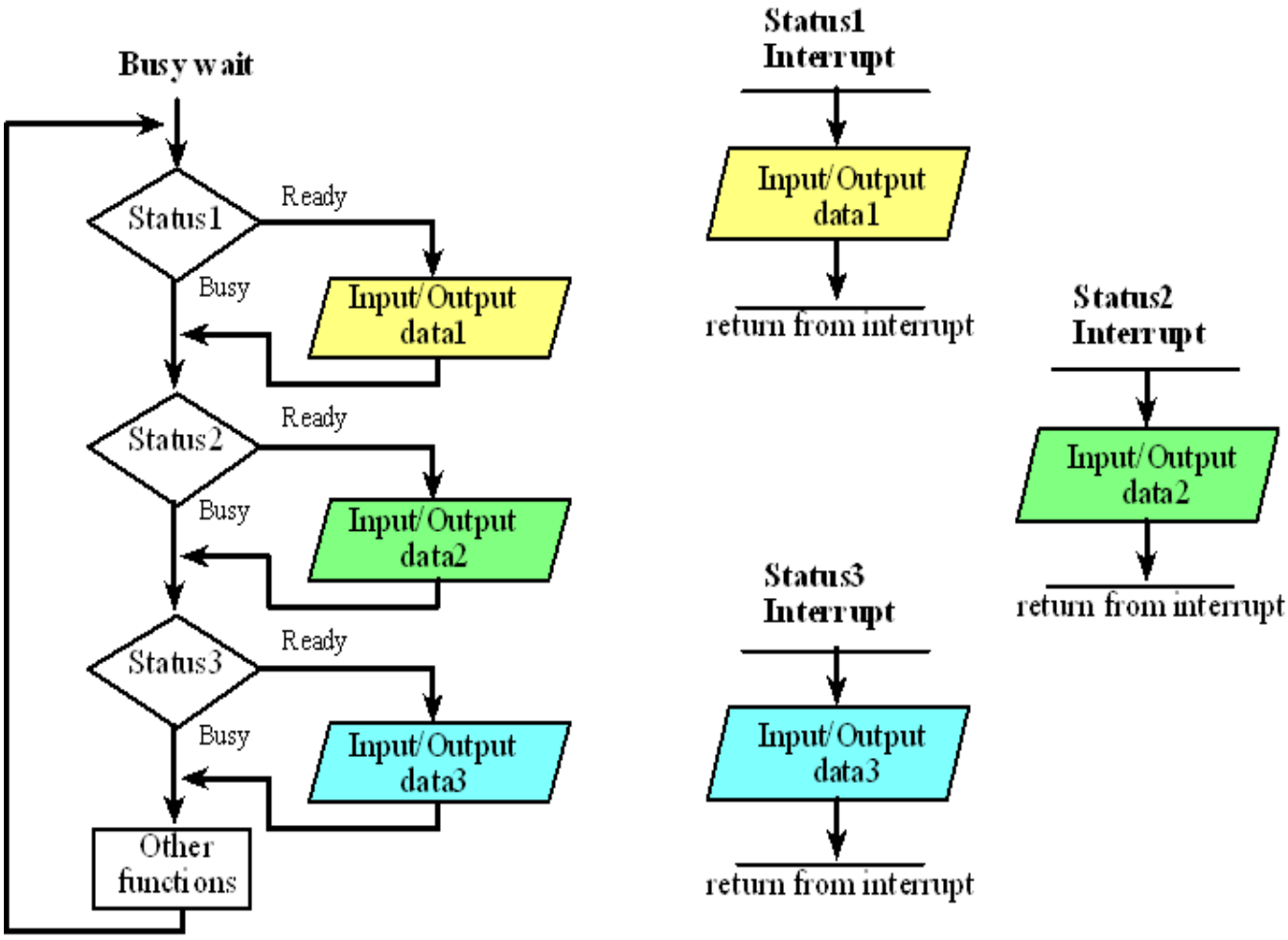
What a processor needs to do to process – or "service" – an interrupt is not trivial: it was doing something, with all sorts of context in its registers. And now it needs to *somehow* suspend that work, take care of the interrupt, and then resume what it was doing before.
To further complicate this: we'd like to be able to write the interrupt code (Interrupt Service Routine, or ISR) in a higher-level language such as C or C++, not just assembly.

This is where the hardware stack plays an essential role: when an interrupt is about to be serviced, all the state of the processor (i.e. its registers, including the instruction pointer) are pushed onto the stack, the instruction pointer is changed to the address of the ISR code, and then… the above loop is simply resumed!

The effect is that all of a sudden, the CPU starts running the ISR code. At the end of that code is a special "return from interrupt" instruction, which pops all the saved registers from the stack, and then again resumes the above loop. We're back where we were before!

There are several clever optimisations for this important mechanism on ARM, such as not saving all the registers and automatically restoring registers on return. This allows ARM chips to efficiently support C/C++ code without any special interrupt entry/exit instructions. But the essential model and mechanism remains unaffected.

Each interrupt routine eats up some stack space when started and gives it back when done.



Interrupt vectors

Hardware interrupts can be used for lots of different purposes. Some interrupts might occur extremely often (many tens of *thousands* of times per second) – in the case of fast peripherals, such as SPI, or a serial port set to a very high baud rate. This in itself is fine, but we need to be careful with overhead. Interrupts "eat away" processor time from the main processing task – if we're not careful, we could end up consuming more time than the processor has (and cause it to totally lock up).

Then again, some interrupts are very infrequent, at least on a μC's MHz performance scale.

The solution to get high performance is called "interrupt vectoring" and causes each *type* of interrupt to jump to a different ISR. The first words in flash memory are reserved precisely for this purpose, and are normally set up to hold the addresses of each of the ISRs.

This means that when any specific ISR is called, it will know exactly what just happened and can do whatever is needed (i.e. copy some data in or out), to then return quickly.

ARM COPROCESSORS

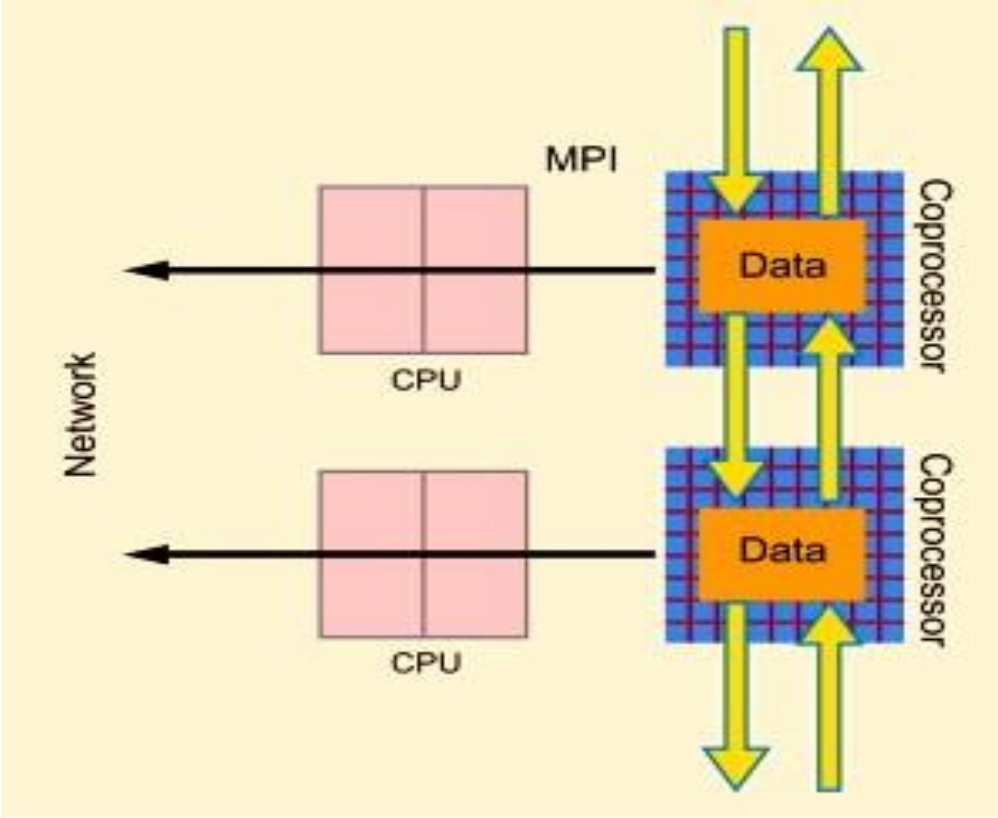
Coprocessors can be attached to the ARM processor. A coprocessor extends the processing features of a core by extending the instruction set or by providing configuration registers. More than one coprocessor can be added to the ARM core via the coprocessor interface.

The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface. Consider, for example, coprocessor 15: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management.

The coprocessor can also extend the instruction set by providing a specialized group of new instructions. For example, there are a set of specialized instructions that can be added to the standard ARM instruction set to process vector floating-point (VFP) operations.
These new instructions are processed in the decode stage of the ARM pipeline. If the decode stage sees a coprocessor instruction, then it offers it to the relevant coprocessor. But if the coprocessor is not present or doesn't recognize the instruction, then the ARM takes an undefined instruction exception, which allows you to emulate the behavior of the coprocessor in software.

Coprocessor-only model

The MPI coprocessor-only or *native* model has the MPI processes launched and residing solely on the coprocessor. MPI libraries, the application, and other needed libraries are uploaded to the coprocessors. Then an application can be launched from the host or from the coprocessor. Once the application is running, MPI network communications between other coprocessors (either on the local node or to other network fabric connected nodes) are managed by the Intel® Coprocessor Communications Link (Intel® CCL) services. Intel CCL provides underlying services to the MPI library to select the optimal transport for MPI messages. One such Intel CCL transport mechanism is the peer-to-peer PCI Express DMA support described in Chapter 8 to directly transfer message data between the coprocessor's memory and a peer InfiniBand (IB) adapter without host memory staging. Figure 9.5 illustrates the MPI Coprocessor-Only model. More details on the structure and components of Intel CCL are described later in this chapter.



Coprocessor Interface

The Armv8 architecture also includes a coprocessor interface designed to allow the integration of hardware accelerators that extend the compute capabilities of the Cortex-M processor. The coprocessors are not peripherals in that they do not provide an interface to the external World like an ADC. Typically, the coprocessor will be an accelerator for specific algorithms such as a DSP or cryptographic accelerator. Each coprocessor is tightly coupled to the Cortex-M processor to provide a low latency interface. Dedicated single cycle instructions are used to move 32- or 64-bit data to and from the Cortex-M processor registers. The Armv8-M architecture provides control and data channels for up to eight separate coprocessors with an addressing scheme shown in Table 7.4

Table 7.4 . Coprocessor Support

Coprocessor	Description
CP0–CP7	Available for vendor implementation
CP8–CP9	Reserved
CP10–CP11	Reserved for FPU
CP12–CP15	Reserved

When an Armv8-M-based microcontroller is fitted with a coprocessor, the silicon vendor will typically provide a supporting library of functions that are used to access the coprocessor features. For example, the NXP lpc55s69 is a Cortex-M33-based microcontroller that includes a cryptographic coprocessor and a DSP coprocessor. Both are supported by vendor libraries so you don't need to do any low-level code development.

ARM Interrupt Structure

A collection of reduced instruction set computer (RISC) instruction set architectures for computer processors that are tailored for different contexts is known as ARM (stylized in lowercase as an arm; originally an abbreviation for Advanced RISC Machines. System-on-a-chip (SoC) and system-on-module (SOM) designs, which combine various components including memory, interfaces, and radios, are examples of devices that other firms design and manufacture using one or more of the architectures that ARM Ltd. develops and licenses. Additionally, it develops cores that use these instruction set architectures and licenses these designs to a large number of businesses, who then use them to create their own products.

The ARM design has gone through multiple iterations. The original ARM1 featured a 32-bit internal structure but only supported 64 MB of main memory due to its 26-bit address space. The ARMv3 series, which has a 32-bit address space, abolished this restriction, while subsequent generations up to ARMv7 maintained this constraint. With its new 32-bit fixed-length instruction set, the 2011-released ARMv8-A architecture gained capability for 64-bit address space and 64-bit arithmetic. Arm Ltd. also released a number of additional instruction sets for various rules. Simultaneous multithreading (SMT) has been added more recently for fault tolerance or improved performance.

For light, portable, battery-powered devices like smartphones, laptops, and tablet computers, as well as other embedded systems, ARM processors are preferred because they are less expensive, consume less power, and generate less heat than their competitors. However, ARM processors are widely employed in servers and desktop computers, notably Fukuku, which will hold the record for the fastest supercomputer from 2020 until 2022. As of 2022, ARM will have created over 230 billion ARM chips, making it the most popular and most abundant family of instruction set architectures (ISA). To include or exclude optional capabilities, there are currently variants of the popular Cortex cores, older "classic" cores, and specialized SecurCore cores available for each of these.

Structure of ARM interrupt:

The following points help us in understanding the structure of ARM interrupt:

- All interrupts are disabled on startup for the ARM CPU until the initialization code turns them on. The Processor Status Registers' bit can be changed to enable or disable the interrupts (PSR or CPSR where C stands for current). The CPSR also determines whether the processor is decoding Thumb instructions and the processor mode (SVC, System, User, etc.). The application can read and write to the CPSR in its entirety when operating in privileged mode, but it can only read the CPSR when operating in non-privileged mode. The processor enters the appropriate interrupt or exception mode in response to an

interrupt or exception, which causes a portion of the main registers to be banked, swapped out, or replaced with set mode registers.

- The interrupt masks' ability to be enabled and disabled is controlled by bits. The two interrupt inputs on the ARM processor can both be regarded as general-purpose interrupts. Interrupt Request (IRQ) and Fast Interrupt Request are the names of the first and second, respectively (FIQ)
- The regular sequential execution of instructions can be stopped by one of seven events on the ARM processor. Since not all events are created equal, the processor must adopt a priority strategy because these events may occur simultaneously. For instance, since it happens when the power to the ARM processor is switched, the Reset has the highest priority. As a result, a reset supersedes all other events when it happens. The only exception to this rule is a Reset event, which takes precedence over all other events when a Data Abort occurs. Since the ARM processor must recognize the event with the highest importance when several events are happening at once, this priority mechanism is crucial.
- The vector table begins at 0x00000000, as was indicated in earlier chapters (ARMx20 processors can optionally locate the vector table address to 0xffff0000). A vector table is a collection of ARM instructions that control the computer (i.e. B, MOV, and LDR). These commands cause the computer to jump to a certain area that can deal with a particular exception or interrupt. Since the FIQ vector is at the end of the table, it can avoid using the B or LDR instruction. The FIQ handler can now begin execution at the FIQ vector point. By preventing the pipe from having to be flushed when the PC is moved, FIQs can conserve processor cycles.
- From an interrupt handler returning The return address from an interrupt or execution handler must be changed because of the processor pipeline. An offset will be present in the address that is kept in the link register. As a result, the offset value must be deducted from the link register.
- The interrupt stacks being set up Depending on the hardware being utilized and the RTOS needs, the interrupt stack may be put in a different location. The target system will crash if the Interrupt Stack extends into the Interrupt vector. Unless a check is made on the stack's extension and a way is provided to deal with that problem when it happens. The IRQ mode stack must first be set up before an interrupt may be enabled. Normally, this is completed in the system's initialization code. Knowing the stack's maximum size is crucial because it allows for the interrupt stack to be allocated that much space. Possible memory configurations with a linear address space are shown below.
- Interrupt handler installation and chaining. The vector table can be fixed for ROM and/or FlashROM-based devices without the need for installation. These systems often copy the entire vector table from ROM to RAM as a block without the need to install individual vectors. Since memory tends to be remapped during initialization, this method is typically employed. Placing a vector entry for the IRQ address (0x00000018) or FIQ address (0x0000001C) so that the entry links to the proper handler is the first step in installing an interrupt handler. Chaining entails inserting a new entry while saving the current vector entry. The original handler may regain control if the newly added handler is unable to handle a certain interrupt source by calling the saved vector entry.
- Another interrupt may occur inside the presently called handler thanks to nested interrupt handlers. This is accomplished by re-enabling the interrupts before the current interrupt has been fully handled by the handler. This feature makes the system more complex for real-time systems. A system failure could result from subtle timing problems introduced by this complexity. These small issues can be very challenging to fix. In order to prevent issues of this nature, the nested interrupt technique needs to be carefully constructed. In order to prevent the next interrupt from filling (overflowing) the stack or corrupting any of the registers, context restoration is protected from interruption.
- If nested interrupts are supported, many common issues can be seen as a result of a rise in complexity. A race condition that results in a cascade of interrupts is one of the key issues. The handler will experience constant interruptions up until the point at which the interrupt stack overflows or the registers become damaged. When designing, efficiency and safety must be balanced. This entails writing code defensively, assuming issues will arise. When possible, the system should examine the stack and take precautions to prevent register corruption.
- Multiple interrupts can be handled using a re-entrant interrupt handler, where interruptions are prioritized. This is significant because interrupts with a higher priority must have a smaller latency. The typically nested interrupt handler is unable to perform this kind of filtering. Re-enabling interrupts early on in the interrupt handler to achieve minimal interrupt latency is the primary distinction between a re-entrant interrupt handler and a nested interrupt handler.

- A prioritized interrupt handler will assign a priority level to a specific interrupt source as opposed to the simple and nested interrupt handlers, which service interruptions on a first-come, first-served basis. The sequence in which the interrupts are handled is determined by a priority level. A desirable property in an embedded system is that a higher-priority interrupt will take precedence over a lower-priority interrupt.

UNIT –II ARM MICROCONTROLLER PROGRAMMING

ARM Instruction Sets

Below are some various ARM Instruction Sets:

Branch Instruction

Branch instructions in ARM processors cause an immediate switch in execution to a specified address location. When a branch (B) instruction is encountered, the processor seamlessly transitions to the designated location and proceeds to execute the operation from there. These instructions facilitate both forward and backward branches, allowing for branching within a range of up to 32 MB.

Data Processing instructions

The various data processing instructions occur within the general-purpose registers. These instructions include:

- **Arithmetic and logic instructions:** These instructions serve the purpose of executing diverse arithmetic and logic operations. They typically involve two source operands, and the resulting output is stored in a designated destination register. Notably, the outcomes of these instructions can be directly written into the Program Counter, which functions as a versatile register.

- **Comparison instructions:** Similar in format to arithmetic and logic instructions, comparison instructions facilitate comparisons between two operands. However, instead of storing the results in registers, these instructions update the contents of flag registers.

- **Multiply instructions:** These instructions are categorized based on the length of the resulting bits after multiplication:

- **32-bit result:** This category represents normal multiplication results, where all 32 bits of the product are stored within a single register.

- **64-bit result:** These instructions produce longer results that span 64 bits. Storing the entire 64-bit value requires the use of two separate registers.

- **Count leading zero instruction:** With this instruction, the device counts the number of consecutive leading zeros in a given sequence, starting from the Most Significant Bit (MSB) to the Least Significant Bit (LSB). The count obtained from this operation is then stored within a register.

Load and Store Instruction

These Instructions are as follows:

Load and Store register: The load register instruction enables the loading of 8-bit, 16-bit, or 32-bit data from memory into a register. Conversely, the store register instruction facilitates the transfer of data from a register to memory.

Load and Store multiple registers: This instruction provides the capability to load or store multiple general-purpose registers as a block from or to memory. It supports various addressing modes, including pre-increment, post-increment, pre-decrement, and post-decrement.

Swap register and memory content: The swap (SWP) instruction functions sequentially in the following manner:

- Loading a value from a specified memory location into a register.
- Storing the current content of the register into the same memory location.
- Simultaneously, the loaded value from memory is stored back into the register.

By maintaining the same register for both steps, the data within the register and memory location are interchanged.

Status Register Transfer Instruction

This instruction enables the transfer of the current program status registers content to or from a general-purpose register. It involves the following steps:

- Setting the value of the condition code flag.
- Configuring the interrupt enables bits.
- Establishing the processor mode.

ARM Thumb Instruction Set

Thumb Instruction Set

The Thumb instruction set consists of 16-bit instructions that act as a compact shorthand for a subset of the 32-bit instructions of the standard ARM. Every Thumb instruction could instead be executed via the equivalent 32-bit ARM instruction. However, not all ARM instructions are available in the Thumb subset; for example, there's no way to access status or coprocessor registers. Also, some functions that can be accomplished in a single ARM instruction can only be simulated with a sequence of Thumb instructions.

What is the difference between Thumb and ARM instructions?

The ARM contains only one instruction set: the 32-bit set. When it's operating in the *Thumb state* , the processor simply expands the smaller shorthand instructions fetched from memory into their 32-bit equivalents.

The difference between two equivalent instructions lies in how the instructions are fetched and interpreted prior to execution, not in how they function. Since the expansion from 16-bit to 32-bit instruction is accomplished via dedicated hardware within the chip, it doesn't slow execution even a bit. But the narrower 16-bit instructions do offer memory advantages.

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.

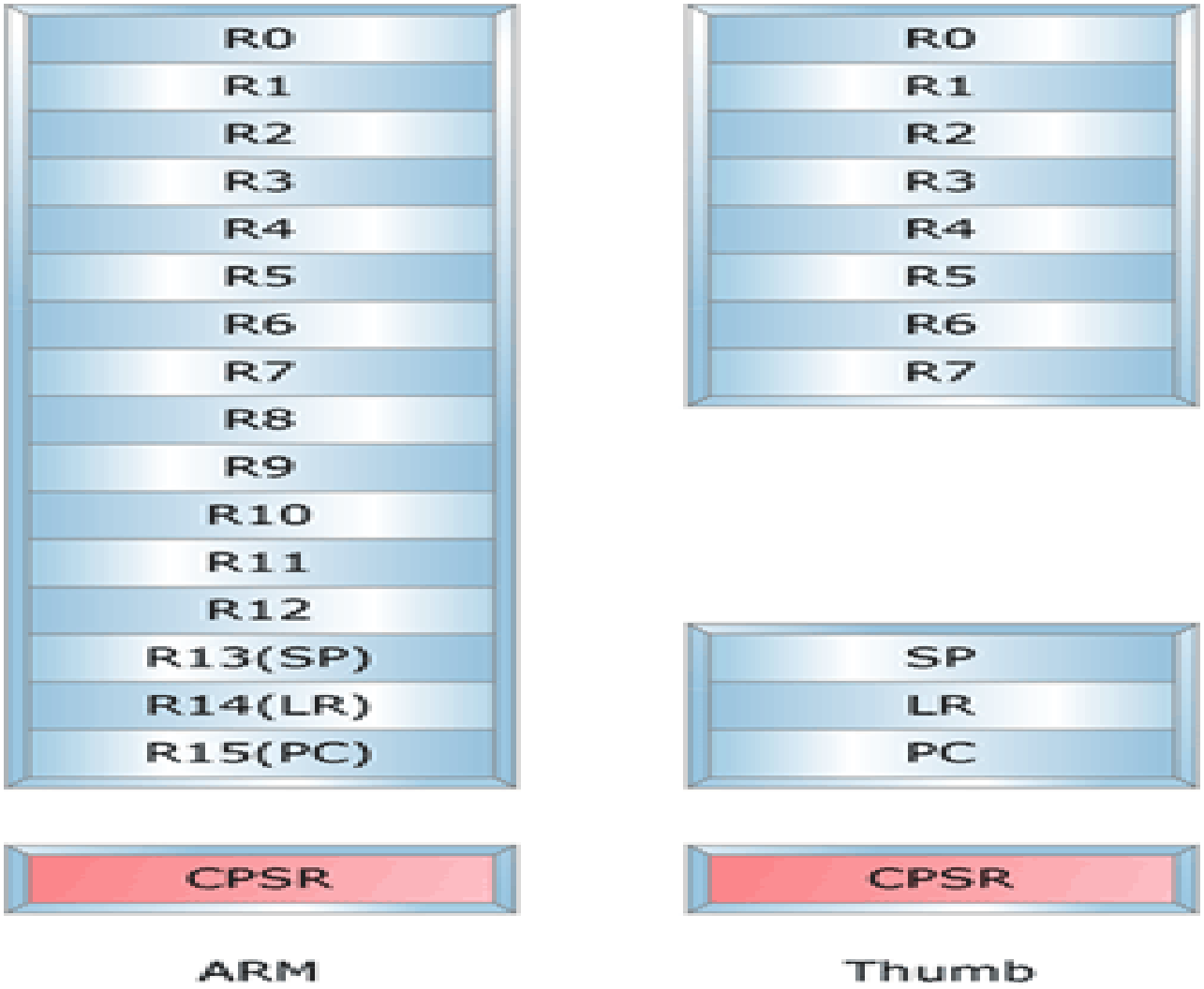
ARM Register Sets

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* .

mov	**R0,#5**	**;Argument to function is in R0**
add	**R1,PC,#1**	**;Load address of SUB_BRANCH, Set for THUMB by adding 1**
BX	**R1**	**;R1 contains address of SUB_BRANCH+1**
;Assembler-specific instruction to switch to Thumb SUB_BRANCH:		
BL	**thumb_sub**	**;Must be in a space of +/- 4 MB**
add	**R1,#7**	**;Point to SUB_RETURN with bit 0 clear**
BX	**R1**	
;**Assembler-specific instruction to switch to ARM SUB_RETURN:**		

Figure 1 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.

Branch and Exchange

There are several ways to enter or leave the Thumb state properly. The usual method is via the Branch and Exchange (**BX**) instruction. See also Branch, Link, and Exchange (**BLX**) if you're using an ARM with version 5 architecture. During the branch, the CPU examines the least significant bit (LSb) of the destination address to determine the new state. Since all ARM instructions will align themselves on either a 32- or 16-bit boundary, the LSB of the address is not used in the branch directly. However, if the LSB is 1 when branching from ARM state, the processor switches to Thumb state before it begins executing from the new address; if 0 when branching from Thumb state, back to ARM state it goes.

Listing 1: How to change into Thumb state, then back

Listing 1 shows one example (not the only one) of using the **BX** instruction to go from ARM to Thumb state and back. This example first switches to Thumb state, then calls a subroutine that was written in Thumb code. Upon return from the subroutine, the system again switches back to ARM state; though this assumes that **R1** is preserved by the subroutine. The **PC** always contains the address of the instruction that is being executed plus 8 (which happens to be **SUB_BRANCH**). The Thumb **BL** instruction actually resolves into two instructions, so 8 bytes are used between **SUB_BRANCH** and **SUB_RETURN** .

When an exception occurs, the processor automatically begins executing in ARM state at the address of the exception vector. So another way to change state is to place your 32-bit code in an exception handler. If the CPU is running in Thumb state when that exception occurs, you can count on it being in ARM state within the handler. If desired, you can have the exception handler put the CPU into Thumb state via a branch.

The final way to change the state is via a return from exception. When returning from the processor's exception mode, the saved value of **T** in the **SPSR** register is used to restore the state. This bit can be used, for example, by an operating system to manually restart a task in the Thumb state—if that's how it was running previously.

Digital Signal Processing made Easy by ARM Architecture

What does ARM have to do with Digital Signal Processing (DSP)?

- ARM seems to be leading the way in this field of processing. The processor has found this as one of its greatest niche markets, mainly because of the steps the company has taken to fit into the embedded market and the architecture it has adopted.
- DSP is prevalent with embedded processor in cell phones, cordless phones, base stations, pagers, modems, Smartphones and PDAs (Personal Digital or Data Assistants).
- Other embedded applications that take advantage of such processors are: disc drive controllers, automotive engine control and management systems, digital auto surround sound, TV-top boxes and internet appliances. Other products are still being modified to take advantage of it: toys, watches, etc. The possible applications are almost endless.

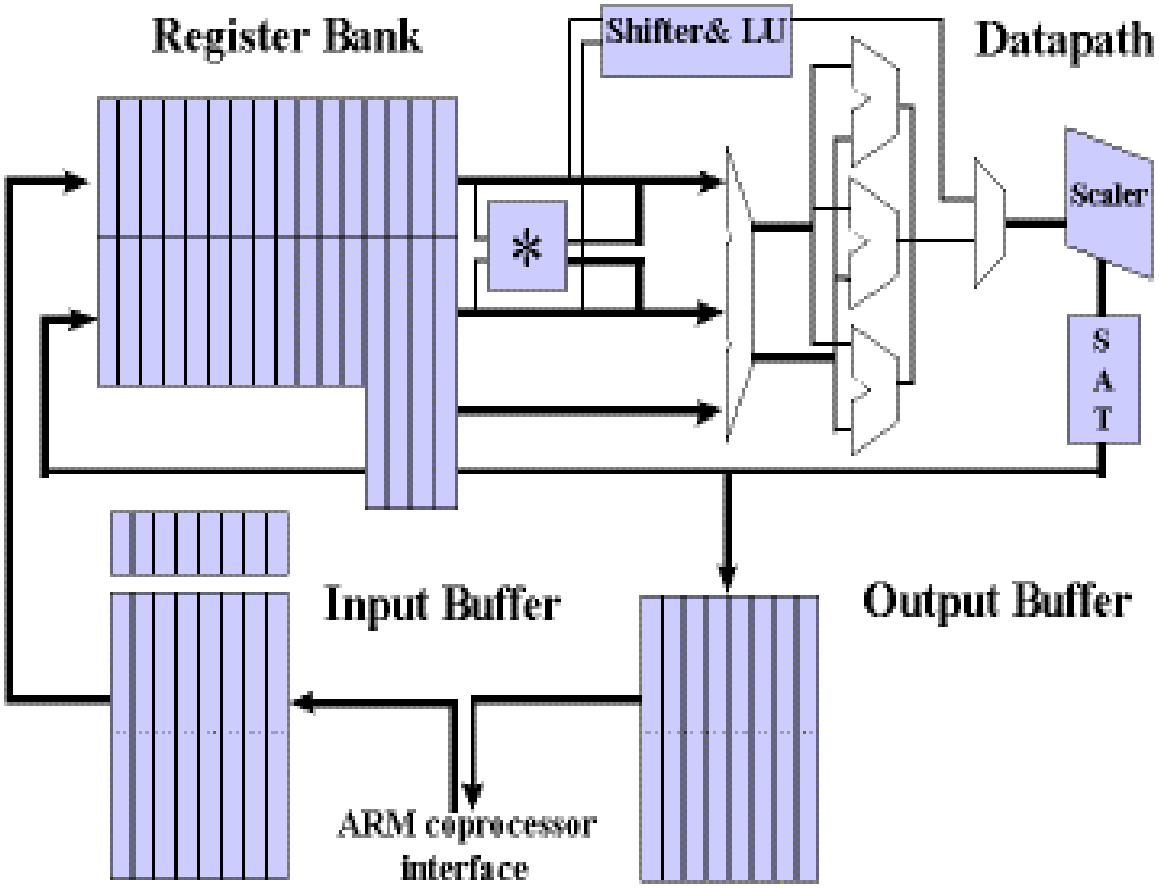
How is this co-processor situation with the Piccolo better for DSP?

- The answer is it helps in many ways. To start with, the integration of the ARM microprocessor with the Piccolo reduces total silicon area by minimizing the amount of on-chip code storage and efficiently using chip memory. This situation is not typically found if two independent processors are being used.
- Improvement to performance through instruction set integration will be achieved through a combination of single-cycle arithmetic operations and the data throughput necessary to sustain that performance.
- Another point that casts light on why the Piccolo solution to DSP has an advantage is that other processors that operate independently are generally based on "Legacy" technology which was not necessarily the best implementation, whereas with the ARM integration there is no dependence on an inadequate standard.

- Other important advantages are the power consumption efficiency that can be obtained which helps lengthen battery life and reduce heat generation, and of course, the cost savings that can be realized with integration. Both are beneficial to support the strong trend toward small, portable, wireless products.

So, all this builds up to something. What does this type of system look like, you ask?

Well here it is -- the Piccolo Architecture



Shown here at top-left is the set of general-purpose registers all of which are programmer accessible and contains thirty-two 16-bit registers or sixteen 32-bit registers to maximize data storage local to the piccolo processor along with four extended precision 48-bit registers. At the bottom are buffers for input and output to minimize memory accesses as well as stalls due to structural hazards encountered by the ARM co-processor interface.

Other notable hardware is a 32-bit barrel shifter for fast scaling of data, 16 * 16 single cycle multiplier, with built in support for extended precision arithmetic, and a split ALU for single-cycle dual 16-bit arithmetic and logical operations in one instruction word or one 32-bit data item arithmetic or logical operation.

Registers have a re-mapping scheme for code optimization and flexibility and there is four nestable zero-overhead hardware loop constructs for executing DSP algorithms.

Looks rather simple, doesn't it. But, a good question you might ask is how does the ARM co-processor interface work? And wouldn't there be a lot of contention between processors sharing data?

Lets start by describing the co-processor architecture support itself. ARM supports a general-purpose extension of its instruction set by adding hardware co-processors.

- This Architecture interface supports up to 16 logical co-processors
- Each co-processor can have up to 16 private registers of not limited to 32-bits.
- Co-processors uses load/store architecture
- For performance most new ARMs restrict the co-processor interface to on-chip use for cache and memory management.

Now lets look at the interface.

- ARM Co-processor Interface is a "Bus watching" system.

- The co-processor is attached through a bus to the ARM processor as the co-processor receives instructions it moves data through the input buffer to its own internal instruction pipeline.
- As the coprocessor instruction begins execution there is a "hand-shake" between the ARM and co-processor that they are both ready to execute the instruction. This protocol includes three signals:

1. Cpi (from ARM to all co-processors).

   A signal for "Co-Processor Instruction," which indicates that ARM has identified a coprocessor instruction and wants it executed.

2. Cpa (from co-processor to ARM).

   A signal for "Co-Processor Absent," which indicates to the ARM that there is no co-processor available to execute the current instruction.

3. Cpb (from co-processor to ARM).

   A signal for "Co-Processor Busy," which tells ARM that the co-processor cannot begin execution of the instruction yet.

What results come from the Hand-shaking? This is the interesting part!

Once the co-processor has received the instruction and it is sitting and waiting for execution there are four possible outcomes based on what hand-shaking occurred.

1. The ARM may not choose to execute this instruction (does not assert cpi), possibly because it fell within a branch shadow or because of some failed condition test (All ARM instructions are conditionally executed.). Result - all co-processors discard instruction.
2. ARM decides to execute (asserts cpi), but no co-processor can take it so cpa stays active, ARM will take the undefined instruction trap and use software to recover.
3. ARM decides to execute and co-processor accepts, but cannot execute yet. Co-processor takes cpa low but leaves cpb high; meanwhile, ARM "busy-waits" until co-processor takes cpb low, stalling instruction stream. However ARM will break off for interrupts.
4. ARM decides to execute and co-processor accepts for immediate execution. Cpi, cpa and cpb are all taken low and both sides commit to complete the instruction.

Special note: Pre-emptive execution.

A co-processor may begin execution of an instruction as soon as receiving in pipeline as long as it can recover state if hand-shaking does not complete.

Basic ARM programming

An **ARM Cortex M** microcontroller is a family of microcontrollers that are based on the ARM architecture. It is a 32-bit processor family that is designed and developed by ARM Holdings. One of the most prominent features of the ARM Cortex M microcontroller is its ability to reduce power consumption while also offering high performance. It is also popular for its affordability, reliability, and scalability.

The ARM Cortex-M microcontrollers support two programming protocols: JTAG (named by the electronics industry association the Joint Test Action Group) and Serial Wire Debug (SWD). There are several ICSP programmers available that support these protocols, including: Keil U-Link 2. Segger J-Link

ARM Cortex M microcontrollers have increasingly been used in various applications and have become the preferred choice for most electronic designers.

In general, microcontrollers are used to control electronic devices and are usually programmed to carry out specific tasks. ARM Cortex M microcontroller follows the same principles and provides additional features such as improved power management and lower power requirements.

Before we dive into the details of programming an *ARM Cortex M* microcontroller, you need to understand the basic concepts and features of the **microcontroller architecture**.

The Architecture and Features of the Microcontroller

An **ARM Cortex M** microcontroller is a suitable choice when designing **microcontrollers** that are used to control electronic devices, IoT devices, and sensors. It has a four-stage pipeline that balances the requirement of a high throughput and low power consumption.

Some popular features of *ARM Cortex M* microcontrollers include the memory protection unit (MPU), the bus interface, and the nested vectored interrupt controller (NVIC).

The Difference between ARM Cortex M and Other Microcontrollers

ARM Cortex M microcontrollers are different from other microcontrollers (such as AVR, PIC, etc.) because they are more powerful, have lower power consumption, fewer instructions and an introduction of a dedicated hardware acceleration mechanism that allows fast operations.

Moreover, the **ARM Cortex M** instruction set is simpler, easier to learn, and more efficient when compared to other instruction sets.

How do I Program my ARM Cortex M3?

Programming is the process of creating instructions that are used to command the hardware to do specific functions. In this section, we'll provide a step-by-step guide on how to program an ***ARM Cortex M3*** microcontroller.

Tools and Software Required for Programming

Before programming an **ARM Cortex M** microcontroller, you need some basic tools and software, including:

- A computer to run the software
- A microcontroller board to program
- A USB to UART converter
- A text editor or an integrated development environment (IDE) such as Keil or Atmel Studio
- CMSIS-DAP compatible debugger or flashing tool

Steps to Program an ARM Cortex M Microcontroller

1. Assemble the hardware: connect the microcontroller board to a personal computer and connect the USB to the UART converter to the board.

2. Download and install the software tools required for programming. This includes the IDE, CMSIS-DAP compatible flashing tool, or debugger.

3. Create a new project in the IDE and select the microcontroller that you want to use for your project.

4. Write your code, making use of libraries and header files available from your microcontroller manufacturer, CMSIS, or open-source libraries.

5. Compile the code to check for errors.

6. Connect the flashing tool or debugger to the board.

7. Flash the code to the board and test successfully.

Start with programming First step is for programming is to arrange GPIO pins. The general purpose I/O pins includes P0.0 to P0.31 and P1.16 to P1.31. Port-0 and Port-1 are 32-bit Input/output ports, and every bit of these ports can be controlled by an individual direction. The operations of port-0 & port-1 depend upon the function of a pin that is selected using the pin connected block. In Port-0, pins like P0.24, P0.26 & P0.27 are not obtainable whereas, in Port-1, the Pins 0 to 15 are not obtainable.

Program: Program for turning LED ON or OFF depending on the status of the pin.
LED is interfaced to P0.0. A switch is interfaced to P0.1 to change the status of the pin.

```
#include <lpc214x.h>
#include <stdint.h>


int main(void)
{
   //PINSEL0 = 0x00000000; /* Configuring P0.0 to P0.15 as GPIO */
   /* No need for this as PINSEL0 reset value is 0x00000000 */
   IO0DIR = 0x00000001;        /* Make P0.0 bit as output bit, P0.1 bit as an input pin  */
   while(1)
   {
      if ( IO0PIN & (1<<1) )  /* If switch is open, pin is HIGH */
      {
```

```
        IO0CLR = 0x00000001;   /* Turn on LED */
      }
      else   /* If switch is closed, pin is LOW */
      {
        IO0SET = 0x00000001;    /* Turn off LED */
      }
   }
}
```

UNIT –III PERIPHERALS OF ARM

ARM processor

Advanced RISC Machine (ARM) Processor is considered to be a family of Central Processing Units that are used in music players, smartphones, wearables, tablets, and other consumer electronic devices.
Advanced RISC Machines create the ARM processor architecture, hence the name ARM. This needs very few instruction sets and transistors. It has very small in size. This is the reason that it is a perfect fit for small-size devices. It has less power consumption along with reduced complexity in its circuits.

They can be applied to various designs such as 32-bit devices and embedded systems. They can even be upgraded according to user needs.

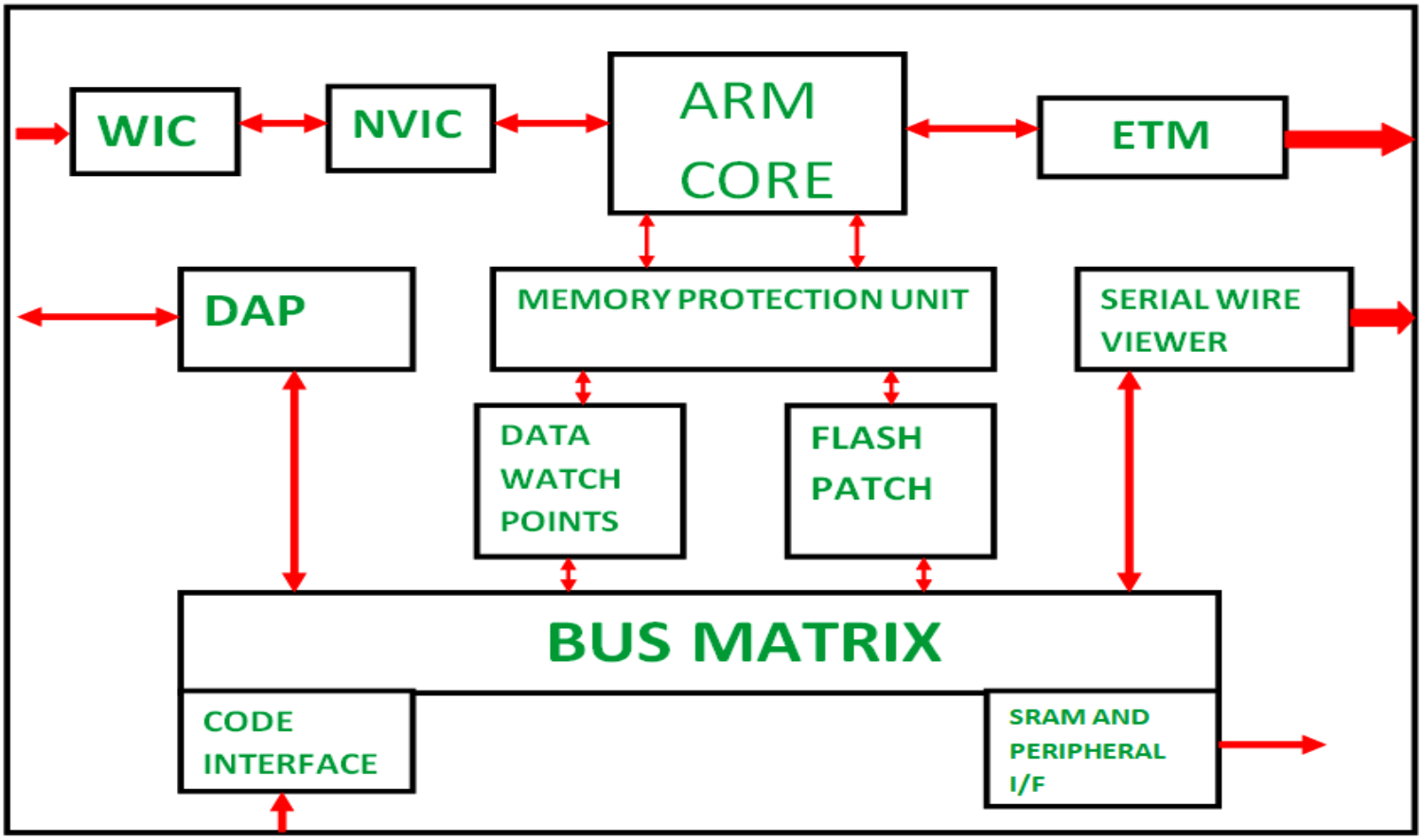
ARM Started With Microcomputing

The applications of the ARM Process start with getting knowledge of the ARM Processor's history. Before ARM, x86 processors were used, which were launched in 1978. Whenever we remove the predefined instructions like complex instructions and hard-to-implement instructions, the remaining instructions take less power and pace and run faster, this is called Reduced Instruction Set Computer (RISC) Architecture. x86 is a Complex Instruction Set Architecture (CISC).

What Makes an ARM Architecture Valuable?

One of the most common electronic architectural designs in the market is Advanced RISC Machine Architecture, even better than x86, which is very common in the server market. ARM Architecture is widely used in smartphones, normal phones, and also in laptops. Though x86 processors have optimized performance ARM Processor gives cost-effective processors with small size, takes less power to run, and also gives better battery life.

ARM Processor is not only limited to mobile phones but is also used in Fugaku, the world's fastest supercomputer. ARM Processor also gives more feasibility to designs of hardware designers and also gives control to designer's supply chains.



Features of ARM Processor

- Multiprocessing System
- Tightly Coupled Memory
- Memory Management
- Thumb-2 Technology
- One-Cycle Execution Time
- Pipelining
- A large number of Registers

1. Multiprocessing Systems: ARM processors are designed to be used in cases of multiprocessing systems where more than one processor is used to process information. The First AMP processor introduced by the name of ARMv6K could support 4 CPUs along with its hardware.

2. Tightly Coupled Memory: The memory of ARM processors is tightly coupled. This has a very fast response time. It has low latency (quick response) that can also be used in cases of cache memory being unpredictable.

3. Memory Management: ARM processor has a management section. This includes Memory Management Unit and Memory Protection Unit. These management systems become very important in managing memory efficiently.

4. Thumb-2 Technology: Thumb-2 Technology was introduced in 2003 and was used to create variable-length instruction sets. It extends the 16-bit instructions of initial Thumb technology to 32-bit instructions. It has better performance than previously used Thumb technology.

5. One-Cycle Execution Time: ARM processor is optimized for each instruction on the CPU. Each instruction is of a fixed length that allows time for fetching future instructions before executing the present instructions. ARM has CPI (Clock Per Instruction) of one cycle.

6. Pipelining: Processing of instructions is done in parallel using pipelines. Instructions are broken down and decoded in one pipeline stage. The channel advances one step at a time to increase throughput (rate of processing).

7. A large number of Registers: A large number of registers are used in ARM processors to prevent large amounts of memory interactions. Records contain data and addresses. These act as a local memory store for all operations.

Difference between ARM and x86

The differences between ARM and x86 are described below.

ARM	x86
ARM uses Reduced Instruction Set Computing Architecture (RISC).	x86 uses Complex Instruction Set Architecture (CISC).
ARM works by executing single instruction per cycle.	x86 works by executing complex instructions at once and it requires more than one cycle.
Performance can be optimized by a Software-based approach.	Performance can be optimized by Hardware based approach.
ARM processors require fewer registers, but they require more memory.	x86 processors require less memory, but more registers.
Execution is faster in ARM Processes.	Execution is slower in an x86 Processor.
ARM Processor work by generating multiple instructions from a complex instruction and they are executed separately.	x86 Processors work by executing complex statements at a single time.
ARM processors use the memory which is already available to them.	x86 processors require some extra memory for calculations.
ARM processors are deployed in mobiles which deal with the consumption of power, speed, and size.	x86 processors are deployed in Servers, Laptops where performance and stability matter.

Advantages of ARM Processor

- ARM processors deal with a single processor at a time, which makes it faster and it also consumes lesser power.
- ARM processors work in the case of a multiprogramming system, where more than one processor is used to process information.
- ARM processors are cheaper than other processors, which makes them usable in mobile phones.
- ARM processors are scalable, and this feature helps it in using a variety of devices.

Disadvantages of ARM Processor

- ARM processors are not stable with x86 processors, and due to this, they cannot be used in Windows Systems.
- ARM processors are not capable of very high performance, which limits them to a variety of applications.
- ARM processor execution is a little hard, which requires skilled programmers to use it.
- ARM processor is inefficient in handling Scheduling instructions.

ARM I/O MEMORY

- In a microprocessor system, there are two methods of interfacing input/output (I/O) devices: memory-mapped I/O and I/O mapped I/O.
- In memory-mapped I/O, input/output devices are mapped to the memory address space of the microprocessor. This means that the I/O devices are treated like memory locations and can be accessed using the same read and write instructions as memory. In other words, the same bus and control signals used for memory access are used for I/O access as well.
- On the other hand, in I/O mapped I/O, input/output devices are mapped to a separate I/O address space that is different from the memory address space. The microprocessor uses special instructions to access the I/O devices using specific I/O address signals, which are separate from the memory address signals.

n the case of the 8085 microprocessor, it uses memory-mapped I/O for accessing input/output devices. The input/output devices are mapped to specific memory locations in the address space, and the microprocessor can access these devices using the same instructions it uses for memory access.

For example, if a device is connected to the memory address 0x8000, the microprocessor can read from or write to the device by using the same instructions it uses to access memory locations, such as:

- MOV A, M ; Load accumulator with data from memory location 0x8000
- MOV M, A ; Write the contents of the accumulator to memory location 0x8000

This method of I/O access is simpler and more efficient in terms of hardware design, as it does not require separate I/O address signals. However, it limits the number of memory locations available for use by the microprocessor.

The microprocessor cannot do anything by itself therefore, It needs to be linked with memory, extra peripherals, or IO devices. This linking is called Interfacing. The interfacing of the I/O devices in 8085 can be done in two ways : 1. Memory-Mapped I/O Interfacing : In this kind of interfacing, we assign a memory address that can be used in the same manner as we use a normal memory location. 2. I/O Mapped I/O Interfacing : A kind of interfacing in which we assign an 8-bit address value to the input/output devices which can be accessed using IN and OUT instruction is called I/O Mapped I/O Interfacing.

Difference between Memory-Mapped I/O Interfacing and I/O Mapped I/O Interfacing :

Features	Memory Mapped IO	IO Mapped IO
Addressing	IO devices are accessed like any other memory location.	They cannot be accessed like any other memory location.
Address Size	They are assigned with 16-bit address values.	They are assigned with 8-bit address values.
Instructions Used	The instruction used are LDA and STA, etc.	The instruction used are IN and OUT.

Cycles	Cycles involved during operation are Memory Read, Memory Write.	Cycles involved during operation are IO read and IO writes in the case of IO Mapped IO.
Registers Communicating	Any register can communicate with the IO device in case of Memory Mapped IO.	Only Accumulator can communicate with IO devices in case of IO Mapped IO.
Space Involved	2^{16} IO ports are possible to be used for interfacing in case of Memory Mapped IO.	Only 256 I/O ports are available for interfacing in case of IO Mapped IO.
IO/M` signal	During writing or read cycles (IO/M` = 0) in case of Memory Mapped IO.	During writing or read cycles (IO/M` = 1) in case of IO Mapped IO.
Control Signal	No separate control signal required since we have unified memory space in the case of Memory Mapped IO.	Special control signals are used in the case of IO Mapped IO.
Arithmetic and Logical operations	Arithmetic and logical operations are performed directly on the data in the case of Memory Mapped IO.	Arithmetic and logical operations cannot be performed directly on the data in the case of IO Mapped IO.
Hardware requirements	Only one set of address and data buses are required for memory and I/O devices	Separate address and data buses are required for memory and I/O devices
Instruction set	Uses the same instructions for accessing both memory and I/O devices	Special instructions are used for accessing I/O devices
Address range	Limited number of memory locations available for use by the microprocessor	Dedicated address space available for I/O devices
Design complexity	Simple to implement and design	More complex to implement and design
Examples of processors	Intel 8085, Motorola 6800	Intel 8255, Zilog Z80

EEPROM

EEPROM stands for Electrically Erasable Programmable Read-Only Memory. EEPROM is a type of non-volatile primary memory and modified version of EPROM (Erasable Programmable Read-Only Memory) which uses electrical signals to erase and program the contents rather than UV signals which was used previously in EPROM. It is used as a chip in computers to store the digital data.
There are two types of EEPROM:

- Serial EEPROM
- Parallel EEPROM

History

EEPROM was developed in 1978 at Intel by *George Perlegos*. Being a non-volatile memory means it retains all the data even power is off and stores a large capacity of data or bits compared to EPROM. It is used as flash memories in its later version and also, to store BIOS of the computer.

Characteristics

- **Less time consuming:** EEPROM takes 5-10 milliseconds to erase the content electronically unlike, EPROM takes minutes to erase the same content using UV signals.
- **Programmable and erasable content:** It can reprogrammed n number of times and that life cycle has to be defined by the manufacture and it can be maximum of 1 million life cycles in modern EEPROMs.
- **No detaching of chip:** To reprogram or erase the content, there is no need to take the chip out of the computer.

Advantages

- Fast erasing of data as it uses electrical signals and can erase all contents or can erase by particular byte.
- Data stored is non-volatile and retains even if the power is off.
- Easy to reprogram without taking it out from computer and does not require any additional equipments for reprogramming.

Disadvantages

- Data retention problem as insulator used is not a perfect insulator and manufacture provides data retention upto 10 years.
- It requires different voltages for reading, writing and erasing the content.

Input Output Ports

•

Ports: The connection point acts as an interface between the computer and external devices like printers, modems, etc.

There are two types of ports :

1. **Internal Port:** It connects the system's motherboard to internal devices like hard disk, CD drive, internal Bluetooth, etc.
2. **External Port:** It connects the system's motherboard to external devices like a mouse, printer, USB, etc.

Some important types of ports are as per follows:

1. **Serial Port :**
   - Used for external modems and older computer mouse
   - Two versions-9pin,25pin
   - Data travels at 115 kilobits per second
2. **Parallel Port :**
   - Used for scanners and printers
   - 25 pin model
3. **Universal Serial Bus (or USB) Port :**
   - It can connect all kinds of external USB devices such as external hard disks, printers, scanners, mouse, keyboards, etc.
   - Data travels at 12 megabits per second.
4. **Firewire Port :**
   - Transfers large amounts of data at a very fast speed.
   - Connects camcorders and video equipment to the computer.
   - Data travels at 400 to 800 megabits per second.
5. **Ethernet Port :**
   - Connects to a network and high-speed Internet.
   - Data travels at 10 megabits to 1000 megabits per second depending upon the network bandwidth.

SRAM

SRAM stands for **Static Random Access Memory**. It is a form of a semiconductor. It is widely used in microprocessors, general computing applications and electronic devices. The SRAM is volatile in nature that means the data stored in it gets all wiped out once the power supply is cut. SRAM comprised of flip flops. It consist of 4-6 transistors, once the flip flop stores the bit it keep it stored until the opposite bit is stored in it.

History

Engineer John Schmidt invented the SRAM in 1964 at Fairchild Semiconductors. The first SRAM is 64-bit and uses p-channel MOS.

Intel released its first 256-bit Intel 1101 SRAM chip in 1969, five years after its invention. But it uses Schottky TTL (Transistor-Transistor-Logic) architecture for its build.

Early SRAMs were manufactured using ceramic plastic. But nowadays, SRAMs are integrated directly to the CPU for faster and better processing.

Characteristics

- **The data is held statically:** The data is stored statically in SRAM and it doesn't need to be refreshed unlike DRAMs.
- **It is a type of Random Access Memory:** The SRAM is a type of Random Access Memory. Random Access Memories are those from which the data can be accessed (read/write) randomly (means any memory location can be accessed), regardless of the memory location that was accessed earlier.
- **It uses flip flop for storing data:** It uses flip flops to store bits. Each flip flop is made up of 4-6 transistors.
- **It is used as a Cache Memory in CPU:** SRAM is used as cache memory for CPUs as they are faster and stores data statically.

Advantages

- It is faster to access and perform operations like read & write.
- The data can be accessed randomly.
- It is used as a cache memory.
- It doesn't need to be refreshed as it stored data statically.
- It has medium power consumption. It requires less power as compared to DRAM.

Disadvantages

- It is expensive.
- It is volatile in nature i.e., data is lost when the memory is not powered.
- It has a low storage capacity.
- It is not possible to refresh the program.
- It has a more complex design and they are bigger in size as well when compared to DRAM.
- It reduces the memory density.

TIMER

- **Timer** is a clock that controls the sequence of an event while counting in fixed intervals of time. A Timer is used for producing precise time delay. Secondly, it can be used to repeat or initiate an action after/at a known period of time. This feature is very commonly used in several applications. An example could be setting up an alarm which triggers at a point of time or after a period of time.

- Timers are used to measure specific time intervals. But in electrical engineering terms, timers are also referred to as counters often. The timer is a component which is extensively used in different embedded systems. They are used to keep a record of time for different events occurring in the embedded systems
- Most of the processors/controllers have inbuilt Timers. Timers in a processor/controller not only generate time delays but they can also be used as counters. They are used to count an action or event. The value of counter increases by one, every time its corresponding action or event occurs.
- Timers in a processor/controller are inbuilt chips that are controlled by special function registers (SFRs) assigned for Timer operations.
- **A counter** is a device that stores (and sometimes displays) the number of times a particular event or process occurred, with respect to a clock signal. It is used to count the events happening outside the microcontroller. In electronics, counters can be implemented quite easily using register-type circuits such as a flip-flop.

Difference between a Timer and a Counter

- The points that differentiate a timer from a counter are as follows −

Timer	Counter
The register incremented for every machine cycle.	The register is incremented considering 1 to 0 transition at its corresponding to an external input pin (T0, T1).
Maximum count rate is 1/12 of the oscillator frequency.	Maximum count rate is 1/24 of the oscillator frequency.
A timer uses the frequency of the internal clock, and generates delay.	A counter uses an external signal to count pulses.

Timers of 8051 and their Associated Registers

- The 8051 has two timers, Timer 0 and Timer 1. They can be used as timers or as event counters. Both Timer 0 and Timer 1 are 16-bit wide. Since the 8051 follows an 8-bit architecture, each 16 bit is accessed as two separate registers of low-byte and high-byte.
- Timer 0 Register
- The 16-bit register of Timer 0 is accessed as low- and high-byte. The low-byte register is called TL0 (Timer 0 low byte) and the high-byte register is called TH0 (Timer 0 high byte). These registers can be

accessed like any other register. For example, the instruction **MOV TL0, #4H** moves the value into the low-byte of Timer #0.

- Timer 1 Register
- The 16-bit register of Timer 1 is accessed as low- and high-byte. The low-byte register is called TL1 (Timer 1 low byte) and the high-byte register is called TH1 (Timer 1 high byte). These registers can be accessed like any other register. For example, the instruction **MOV TL1, #4H** moves the value into the low-byte of Timer 1.

- TMOD (Timer Mode) Register
- Both Timer 0 and Timer 1 use the same register to set the various timer operation modes. It is an 8-bit register in which the lower 4 bits are set aside for Timer 0 and the upper four bits for Timers. In each case, the lower 2 bits are used to set the timer mode in advance and the upper 2 bits are used to specify the location.

-

- **Gate** − When set, the timer only runs while INT(0,1) is high.
- **C/T** − Counter/Timer select bit.
- **M1** − Mode bit 1.
- **M0** − Mode bit 0.

- GATE
- Every timer has a means of starting and stopping. Some timers do this by software, some by hardware, and some have both software and hardware controls. 8051 timers have both software and hardware controls. The start and stop of a timer is controlled by software using the instruction **SETB TR1** and **CLR TR1** for timer 1, and **SETB TR0** and **CLR TR0** for timer 0.
- The SETB instruction is used to start it and it is stopped by the CLR instruction. These instructions start and stop the timers as long as GATE = 0 in the TMOD register. Timers can be started and stopped by an external source by making GATE = 1 in the TMOD register.

- C/T (CLOCK / TIMER)
- This bit in the TMOD register is used to decide whether a timer is used as a **delay generator** or an **event manager**. If C/T = 0, it is used as a timer for timer delay generation. The clock source to create the time delay is the crystal frequency of the 8051. If C/T = 0, the crystal frequency attached to the 8051 also decides the speed at which the 8051 timer ticks at a regular interval.
- Timer frequency is always 1/12th of the frequency of the crystal attached to the 8051. Although various 8051 based systems have an XTAL frequency of 10 MHz to 40 MHz, we normally work with the XTAL frequency of 11.0592 MHz. It is because the baud rate for serial communication of the 8051.XTAL = 11.0592 allows the 8051 system to communicate with the PC with no errors.

M1	**M2**	**Mode**
0	0	13-bit timer mode.
0	1	16-bit timer mode.
1	0	8-bit auto reload mode.

Different Modes of Timers

- Mode 0 (13-Bit Timer Mode)
- Both Timer 1 and Timer 0 in Mode 0 operate as 8-bit counters (with a divide-by-32 prescaler). Timer register is configured as a 13-bit register consisting of all the 8 bits of TH1 and the lower 5 bits of TL1. The upper 3 bits of TL1 are indeterminate and should be ignored. Setting the run flag (TR1) does not clear the register. The timer interrupt flag TF1 is set when the count rolls over from all 1s to all 0s. Mode 0 operation is the same for Timer 0 as it is for Timer 1.

- Mode 1 (16-Bit Timer Mode)
- Timer mode "1" is a 16-bit timer and is a commonly used mode. It functions in the same way as 13-bit mode except that all 16 bits are used. TLx is incremented starting from 0 to a maximum 255. Once the value 255 is reached, TLx resets to 0 and then THx is incremented by 1. As being a full 16-bit timer, the timer may contain up to 65536 distinct values and it will overflow back to 0 after 65,536 machine cycles.

- Mode 2 (8 Bit Auto Reload)
- Both the timer registers are configured as 8-bit counters (TL1 and TL0) with automatic reload. Overflow from TL1 (TL0) sets TF1 (TF0) and also reloads TL1 (TL0) with the contents of Th1 (TH0), which is preset by software. The reload leaves TH1 (TH0) unchanged.
- The benefit of auto-reload mode is that you can have the timer to always contain a value from 200 to 255. If you use mode 0 or 1, you would have to check in the code to see the overflow and, in that case, reset the timer to 200. In this case, precious instructions check the value and/or get reloaded. In mode 2, the microcontroller takes care of this. Once you have configured a timer in mode 2, you don't have to worry about checking to see if the timer has overflowed, nor do you have to worry about resetting the value because the microcontroller hardware will do it all for you. The auto-reload mode is used for establishing a common baud rate.

- Mode 3 (Split Timer Mode)
- Timer mode "3" is known as **split-timer mode**. When Timer 0 is placed in mode 3, it becomes two separate 8-bit timers. Timer 0 is TL0 and Timer 1 is TH0. Both the timers count from 0 to 255 and in case of overflow, reset back to 0. All the bits that are of Timer 1 will now be tied to TH0.
- When Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be set in modes 0, 1 or 2, but it cannot be started/stopped as the bits that do that are now linked to TH0. The real timer 1 will be incremented with every machine cycle.

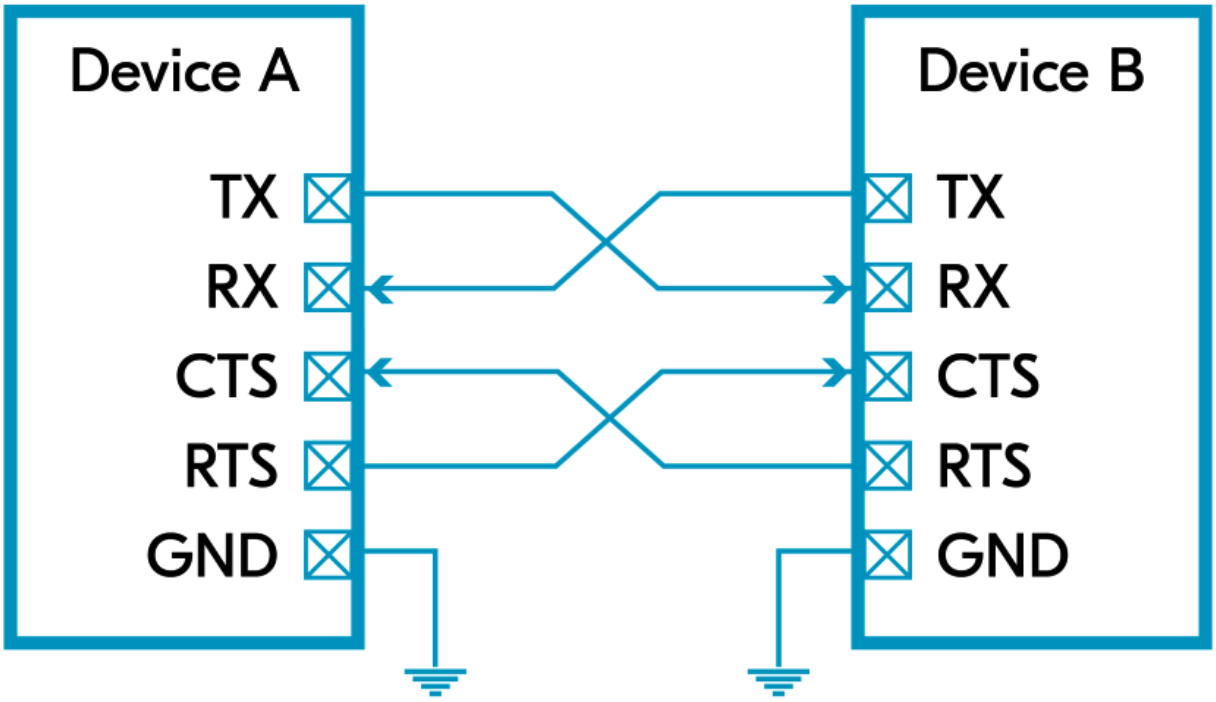
Universal Asynchronous Receiver Transmitter (UART) Protocol

UART means Universal Asynchronous Receiver Transmitter Protocol. UART is used for serial communication from the name itself we can understand the functions of UART, where **U** stands for Universal which means this protocol can be applied to any transmitter and receiver, and **A** is for Asynchronous which means one cannot use clock signal for communication of data and **R** and **T** refers to Receiver and Transmitter hence UART refers to a protocol in which serial data communication will happen without clock signal.
UART is established for serial communication. In this article, we will discuss how parallel communication is established with respect to serial communication using UART as well as how to configure UART and what is the data format in UART. Later on, we will discuss the Pros and Cons of the UART.

UART Basics

UART is a Universal Asynchronous Receiver Transmitter protocol that is used for serial communication. Two wires are established here in which only one wire is used for transmission whereas the second wire is used for

reception. Data format and transmission speeds can be configured here. So, before starting with the communication define the data format and transmission speed. Data format and transmission speed for communication will be defined here and we do not have a clock over here that's why it is referred to as asynchronous communication with UART protocol. Here we will see how this protocol is designed physically.



Here, DEVICE A that is having transmitter PIN and a receiver pin; DEVICE B is having a receiver and transmission pin. The Transmitter of DEVICE A is one that should be connected with the receiver pin of DEVICE B and the transmitter pin of DEVICE B should be connected with the receiver pin of DEVICE A we just need to connect two wires for communication.

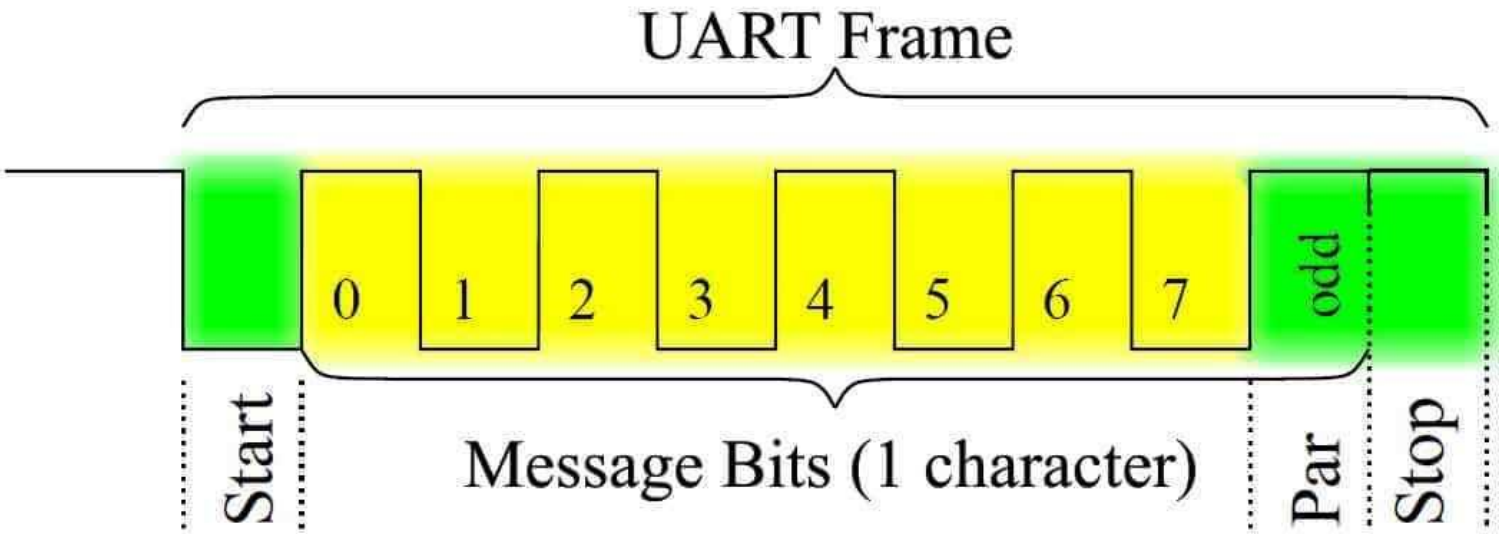
If DEVICE A wants to send data, then it will be sending data on the transmitter's pin and here receiver of this DEVICE B will receive it over and if DEVICE A wants to receive the data, then that is possible on the RX line that will be forwarded by TX of DEVICE B. On comparing this serial communication of UART with parallel then it can be observed that in parallel multiple buses are required. Based on the number of lines bus complexity of UART is better but parallel communication is good in terms of speed.

So, when speed is required at that time we should select parallel communication and for a low-speed application, UART must be used and the bus complexity will be less.

The configuration of UART is done before transmission, both of these devices are connected with protocol and should know the speed of data transmission. First, define the speed of both devices. Now, configure the speed of DEVICE A and B for data transmission which is referred to as Baud Rate so here Baud Rate will be the same for DEVICE A and B otherwise both of these devices cannot understand at what speed and at what rate data is coming. After that, configure the data length so here DEVICE A and DEVICE B both are configured at fixed data length if DEVICE A is transmitting data, then it is configured with fixed data. Like if DEVICE A is configured with the eight-bit size of data then DEVICE B should also be configured at the same size of data which is eight bits. After this, check data transmission or receiving time, forward start bits, and stop bits.

Now we will see the data format and when the communication is according to UART protocol. We are using NRZ encoding for data communication.

UART Data Format



Suppose DEVICE A is sending data to DEVICE B, and the transmitter of DEVICE A will send data to the receiver of Device B then it will be logic high. Now, send the start bit that will be logic 0 and once we have the start bit, DEVICE B will know that somebody is communicating. Now there is the same speed configuration with both devices. So, after the start bit, DEVICE A can forward data.

Consider 8 bits of data length, so we will be forwarding 8 bits and those 8 bits will be received by DEVICE B a parity bit can also be used which is optional, but this is quite effective. By using the parity bit, it can be identified whether the received data is correct or not. Suppose we are sending 1 1 1 0 0 0 1 0. Now, we have 4 ones; an even number of ones are there hence the parity is even and for that, logic 0 will be assigned. Suppose we are receiving data with some error, say zero is converted into one; Now incorrect data that is 1 1 1 1 0 0 1 0 for this incorrect data parity will be 0 as there are 5 ones, here is a mismatch in the parity bit and hence it is confirmed that the received data has some error.

Pros of UART Protocol

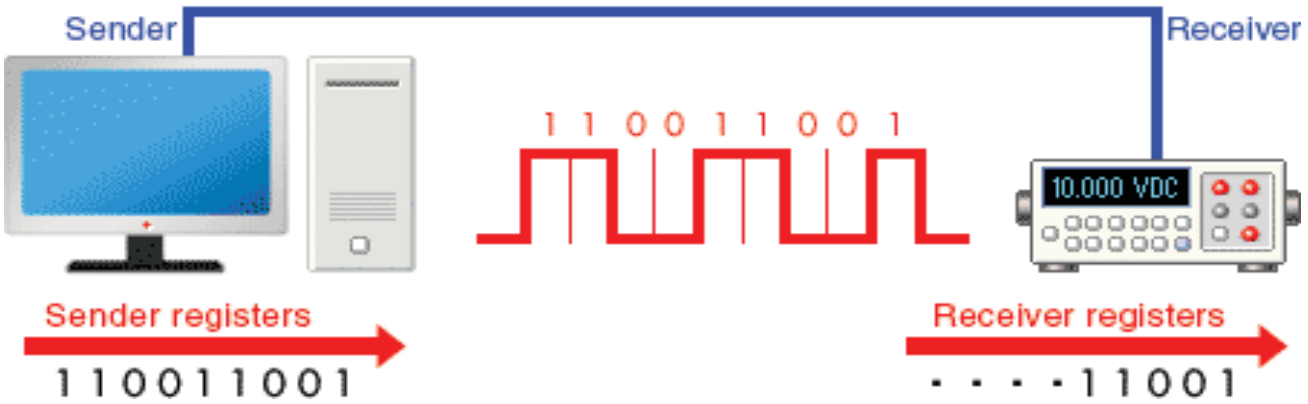
- It is having less physical interfacing based on only two lines.
- Simple to configure data and data size. Speed is also configurable. In the majority of cases, this baud rate is 9600 for the UART protocol. Full duplex mode configuration is possible by using two wires so that is the major advantage of UART.
- Error detention is possible

Cons of UART Protocol

- UART is having serial communication, therefore, it has less speed.
- Start bit, stop bit, and the parity bit is other overhead.
- Since this is asynchronous communication so here there are many things that we need to do in configuration, for instance, we should configure both devices at the same speed because the clock signal is absent.

SERIAL COMMUNICATION WITH PC

Serial communication is a communication method that uses one or two transmission lines to send and receive data, and that data is continuously sent and received one bit at a time.Since it allows for connections with few signal wires, one of its merits is its ability to hold down on wiring material and relaying equipment costs.



Serial communication standards

RS-232C/RS-422A/RS-485 are EIA (Electronic Industries Association) communication standards.Of these communication standards, RS-232C has been widely adopted in a variety of applications, and it is even standard equipment on computers and is often used to connect modems and mice.Sensors and actuators also contain these interfaces, many of which can be controlled via serial communication.

RS-232C

This serial communication standard is widely used and is often equipped on computers as standard.It is also called "EIA-232".The purpose and timing of the signal lines and the connectors have been defined (D-sub 25-pin or D-sub 9-pin).Currently the standard has been revised with the addition of signal lines and is formally called "ANSI/EIA-232-E".However, even now it is generally referred to as "RS-232C

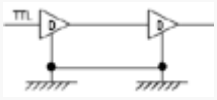
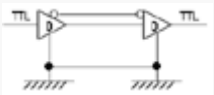
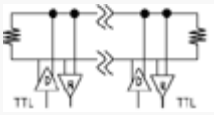
RS-422A

This standard fixes problems in RS-232C such as a short transmission distance and a slow transmission speed.It is also called "EIA-422A".The purpose and timing of the signal lines are defined, but the connectors are not.Many compatible products primarily adopt D-sub 25-pin and D-sub 9-pin connectors.

RS-485

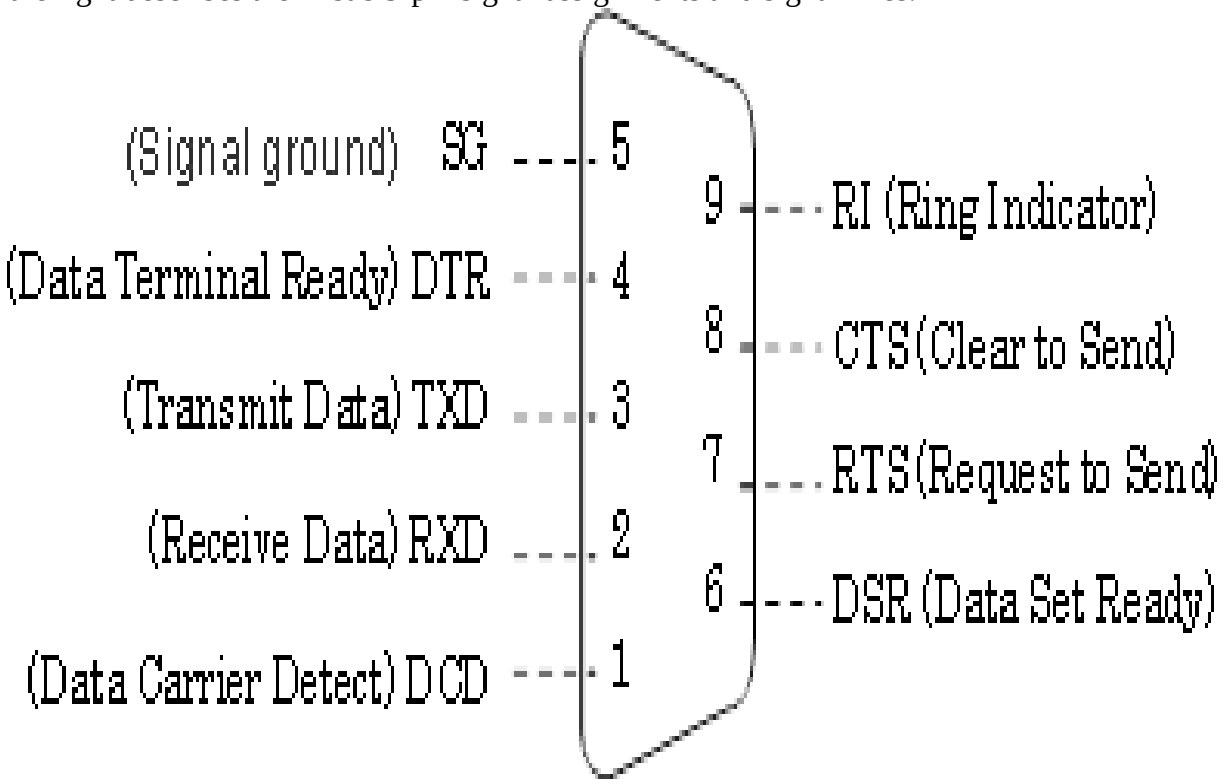
This standard fixes the problem of few connected devices in RS-422A.It is also called "EIA-485".RS-485 is forward compatible standard with RS-422A.The purpose and timing of the signal lines are defined, but the connectors are not.Many compatible products primarily adopt D-sub 25-pin and D-sub 9-pin connectors.

Parameter	RS-232C	RS-422A	RS-485
Transmission mode	Simplex	Multi-point simplex	Multi-point multiplex
Max. connected devices	1 driver 1 receiver	1 driver 10 receivers	32 drivers 32 receivers

Parameter	RS-232C	RS-422A	RS-485
Max. transmission rate	20Kbps	10Mbps	10Mbps
Max. cable length	15m	1200m	1200m
Operation mode	Single-ended (unbalanced type)	Differential (balanced type)	Differential (balanced type)
Connection image			
Features	Short distance Full-duplex 1:1 connection	Long distance Full-duplex, half-duplex 1:N connection	Long distance Full-duplex, half-duplex N:N connection

Signal assignments and connectors

In RS-232C, the connectors to use and the signal assignments have been defined and are standardized.The figure to the right describes the D-sub 9-pin signal assignments and signal lines.



Pin No.	Signal name	Description	
1	DCD	Data Carrier Detect	Carrier detect
2	RxD	Received Data	Received data
3	TxD	Transmitted Data	Transmitted data
4	DTR	Data Terminal Ready	Data terminal ready
5	SG	Signal Ground	Signal ground or common return
6	DSR	Data Set Ready	Data set ready
7	RTS	Request To Send	Request to send
8	CTS	Clear To Send	Clear to send
9	RI	Ring Indicator	Ring indicator
CASE	FG	Frame Ground	Maintenance ground or earth

Connection method

In RS-232C, the connectors and signal assignments have been standardized, so many standard-compliant cables are available commercially.However, equipment comes in the following types, and depending on the equipment that will be connected, a straight cable or a crossover cable is required.

Equipment type

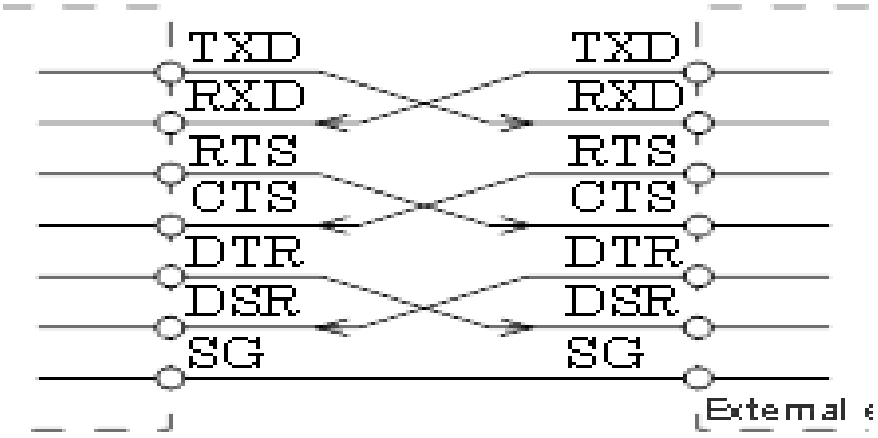
DCE

Data communication equipment.This term indicates equipment that passively operates such as modems, printers, and plotters.

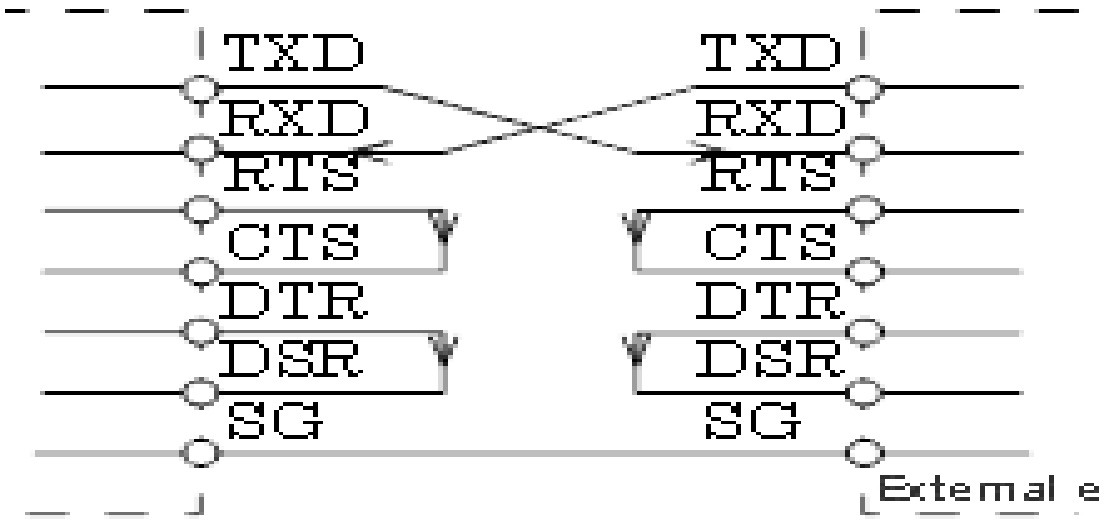
DTE

Data terminal equipment.This term indicates equipment that actively operates such as computers.

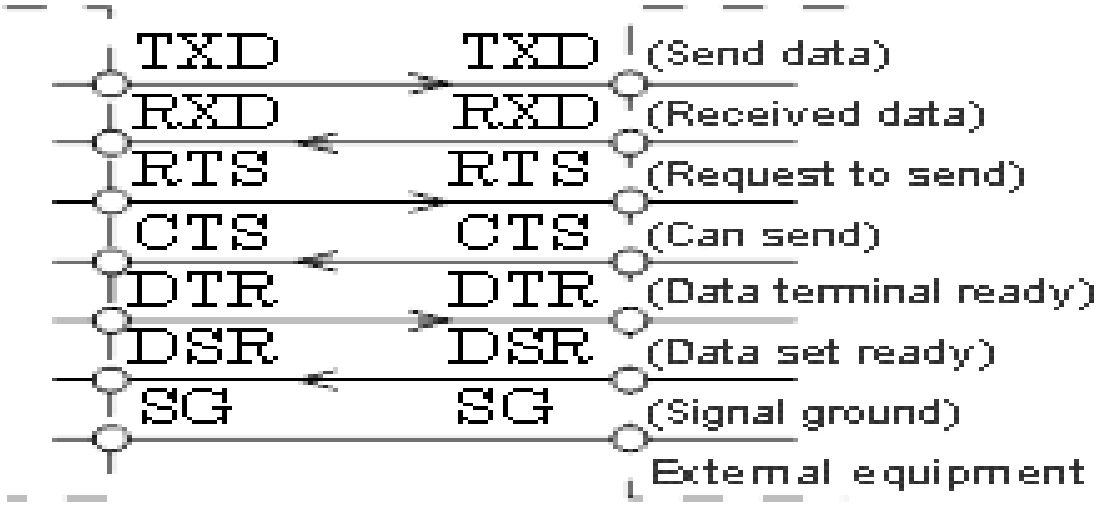
Crossover cable connection (1)



Crossover cable connection (2)



Straight cable connection



Half-duplex communication and full-duplex communication

Full-duplex communication

A method where send and receive both have their own transmission line so data can be simultaneously sent and received.

Half-duplex communication

A method where communication is performed using one transmission line while switching between send and receive.For this reason, simultaneous communication cannot be performed.

Asynchronous communication and synchronous communication

In serial communication, data is sent one bit at a time using one signal line, so in order for the receiving side to accurately receive the data, the sending side must know at what speed it is sending each bit.In RS-232C, synchronous communication and asynchronous communication standards have been defined.For peripheral equipment used for measurements or control, the previously mentioned full-duplex communication and asynchronous communication are typically used.

Synchronous communication

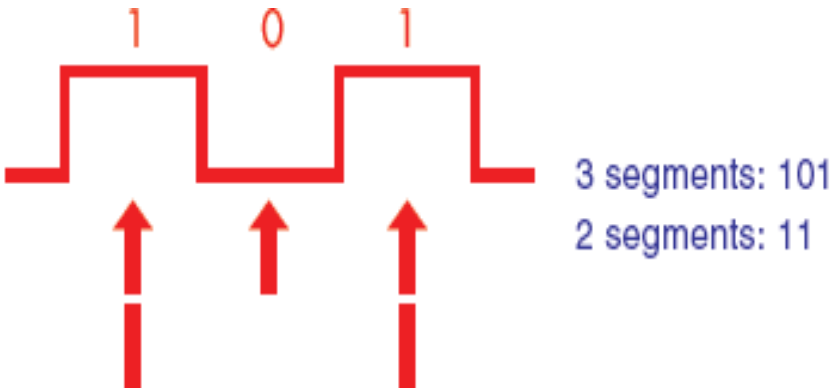
This method sends and receives data synchronized to a clock generated by the other equipment or by a self-generated clock.Communication is performed is based on a synchronization signal added to each bit from the sending side.This has good data transmission efficiency but there is a demerit in that the transmission procedure becomes complicated.

Asynchronous communication

This method sends and receives data synchronized to each side's own self-generated clock.Normal communication is not possible if the transmission rate settings do not match.In other words, both the sending side and the receiving side initially agree on how many bits to transfer each second, and then each creates a synchronization signal of a frequency that matches that transmission rate. In asynchronous communication, data is sent and received one bit at a time on one data line, so if each side's communication condition settings do not initially match, normal communication is not possible.Matching the computer (controller) side settings to the peripheral equipment side settings is the normal setup method.

Transmission rate

Specifies the number of bits to send each second.The unit is bps (bits per second) and is selected from 300, 600, 1200, 2400, 4800, 9600, 19200, and so on.By matching the settings and timing, the data delimiters correspond, and data can be normally sent and received.For this reason, a start bit is added to each item of data (1 byte) to acquire the correct timing.



Stop bit length

This sets the length of the bit that indicates the end of the data.This is normally selected as 1 bit, 1.5 bits, or 2 bits.The start bit length is fixed as 1 bit so this setting is not necessary.

Data bit length

This specifies how many bits each item of data is composed from.This depends on the device being used, but normally specify 7 bits for alphanumeric characters and symbols, and specify 8 bits for 1 byte binary data.

Parity check setting

This is a function to find errors in the data and is selected from "even parity check (EVEN)", "odd parity check (ODD)", or "no parity check (NONE)".

Parity check details

On the sending side, a parity bit of "1" or "0" is added to the data so as to make the number of "1" data bits even for EVEN and odd for ODD.On the receiving side, the number of "1" data bits is counted and the data is judged as being correct if the number is even when EVEN and odd when ODD.

Handshake (flow control)

When sending and receiving data between devices, data may be lost when data is sent when the receiving side is not in the receiving state, so it is important in communication to check the other side's state.The handshake (flow control) is a function that maintains the reliability of communications. A signal is sent from the sending side to the receiving side that states, "data is being sent", and the receiving side receives that signal and reads the data from the signal line.Then it sends a reply to the sending side that states, "the data was received".In other words, data can be transferred while each side checks the sending and receiving of data.

Software handshake (XON/XOFF flow control)

This is a control method where the "XOFF code" is sent to the sending side to request that sending be temporarily interrupted when the remaining free space in the receive buffer gets low on the receiving side.When there is a sufficient amount of free space, the "XON code" is sent to request that the sending side restart sending.

Hardware handshake

The control lines (RTS or DTR) are automatically turned on or off as an alternative to sending the XON/XOFF codes in software flow control.The RTS signal and the CTS signal or the DTR signal and the DSR signal must be connected to each other.

ADC INTERFACING WITH ARM PROCESSOR

In electronics world there are many varieties of analog sensors in the market that are used to measure temperature, speed, displacement, pressure etc. Analog sensors are used to produce output that are continuously changing over

the time. These signals from analog sensors tend to be very small in value from a few micro-volts (uV) to several milli-volts (mV), so some form of amplification is required. For using these analog signals in microcontroller we need to **convert analog signal into digital signal** as the microcontroller understands and process only digital signals. So most of the microcontroller has an inbuilt important feature called **ADC (Analog to Digital convertor).** Our microcontroller **ARM7-LPC2148** also has an ADC feature.

In this tutorial we will see **how to use ADC in ARM7-LPC2148** by supplying a varying voltage to an Analog pin and display it on the 16x2 LCD screen after analog to digital conversion. So let's start by a short introduction about ADC.

What is ADC?

As said earlier ADC stands for Analog to digital conversion and it is used to convert analog values from real world into digital values like 1's and 0's. So what are these analog values? These are the ones that we see in our day to day life like temperature, speed, brightness etc. These parameters are measured as analog voltages by respective sensors and then these Analog values are converted into Digital values for microcontrollers.

Let us assume that our ADC range is from 0V to 3.3V and we have a 10-bit ADC this means our input voltage 0-3.3 Volts will be split into 1024 levels of discrete analog values(2^{10} = 1024). Meaning 1024 is the resolution for a 10-bit ADC, similarly for a 8-bit ADC resolution will be 512 (28) and for a 16-bit ADC resolution will be 65,536 (216). **LPC2148 has the 10 bit resolution ADC**.

With this if the actual input voltage is 0V then the MCU's ADC will read it as 0 and if it is 3.3V the MCU will read 1024 and if it somewhere in between like 1.65v then the MCU will read 512. We can use the below formulae to calculate the digital value that will be read by the MCU based on the Resolution of the ADC and Operating voltage.

(ADC Resolution / Operating Voltage) = (ADC Digital Value / Actual Voltage Value)

ADC in ARM7-LPC2148

- The LPC2148 contain **two** analog to digital converters.
- These converters are single 10-bit **successive approximation** analog to digital converters.
- While ADC0 has **six** channels, ADC1 has **eight** channels.
- Therefore, total number of available **ADC inputs for LPC2148 is 14**.
- It converts input voltage in range of (0 to 3.3V) only. It must not exceed 3.3V the voltage reference. As it will damage the IC and also provide uncertain values.

Some important feature of ADC in LPC2148

- Each converter capable of performing more than 400000 10-bit samples per second.
- Every analog input has a dedicated result register to reduce interrupt overhead.
- Burst conversion mode for single or multiple inputs.
- Optional conversion on transition on input pin or timer match signal.
- Global Start command for both converters.

Also check how to use ADC in other Microcontrollers:

- How to Use ADC in Arduino Uno?
- Interfacing ADC0808 with 8051 Microcontroller
- Using ADC Module of PIC Microcontroller
- Raspberry Pi ADC Tutorial
- How to use ADC in MSP430G2 - Measuring Analog Voltage

- How to use ADC in STM32F103C8

Digital-to-analog (DAC) converter

The digital-to-analog converter (DAC) is a device widely used to convert digital pulses to analog signals. In this section we discuss the basics of interfacing a DAC to the ARM Microcontroller (LPC2148).

Recall from your digital electronics book the two methods of creating a DAC: binary weighted and R/2R ladder. The vast majority of integrated circuit DACs, including the MC1408 (DAC0808) used in this section, use the R/2R method since it can achieve a much higher degree of precision. The first criterion for judging a DAC is its resolution, which is a function of the number of binary inputs. The common ones are 8, 10, and 12 bits. The number of data bit inputs decides the resolution of the DAC since the number of analog output levels is equal to 2″, where *n* is the number of data bit inputs. Therefore, an 8-input DAC

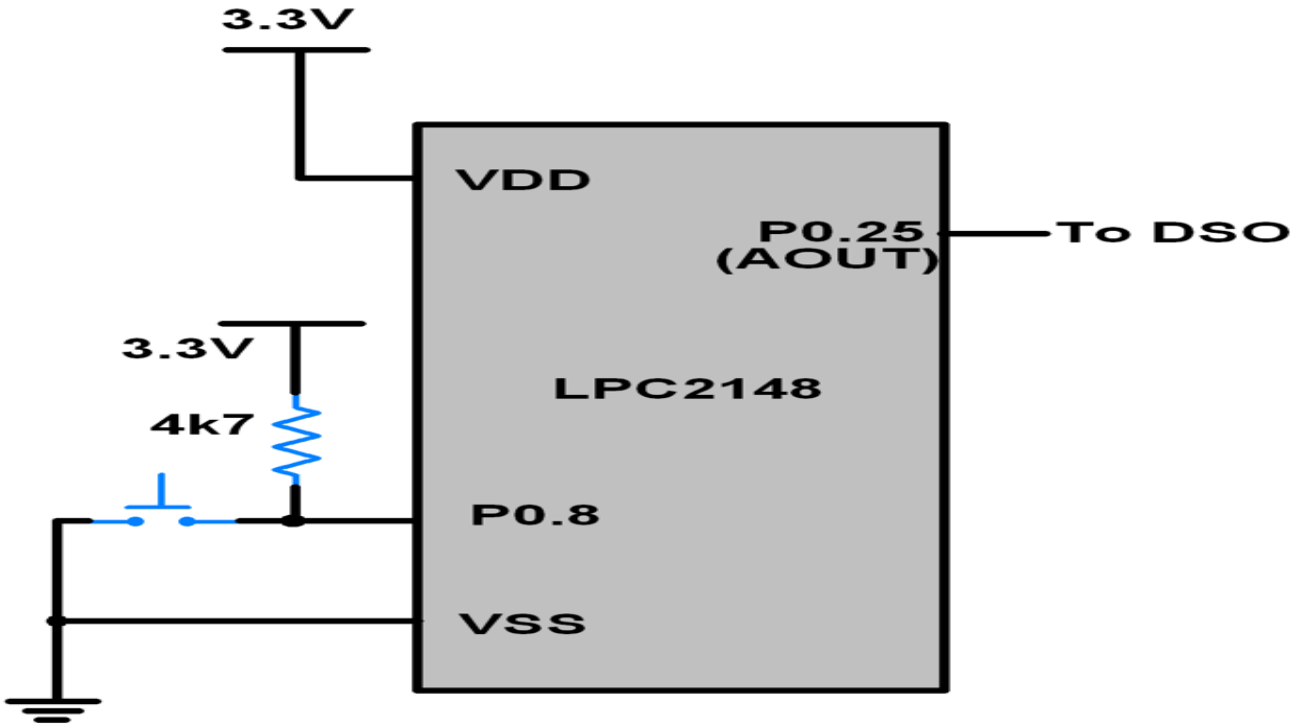
such as the DAC0808 provides 256 discrete voltage (or current) levels of output.Similarly, the 12-bit DAC provides 4096 discrete voltage levels. There are also16-bit DACs, but they are more expensive.

LPC2148 DAC (Digital to Analog Converter)Introduction to DAC

Digital to Analog Converter (DAC) are mostly used to generate analog signals (e.g. sine wave, triangular wave etc.) from digital values.

- LPC2148 has 10-bit DAC with resistor string architecture. It also works in Power down mode.
- LPC2148 has Analog output pin (AOUT) on chip, where we can get digital value in the form of Analog output voltage.
- The Analog voltage on AOUT pin is calculated as ((**VALUE**/1024) * VREF). Hence, we can change voltage by changing **VALUE**(10-bit digital value) field in **DACR** (DAC Register).
- e.g. if we set **VALUE** = 512,
  then, we can get analog voltage on AOUT pin as ((512/1024) * VREF) = VREF/2.

Interfacing Diagram

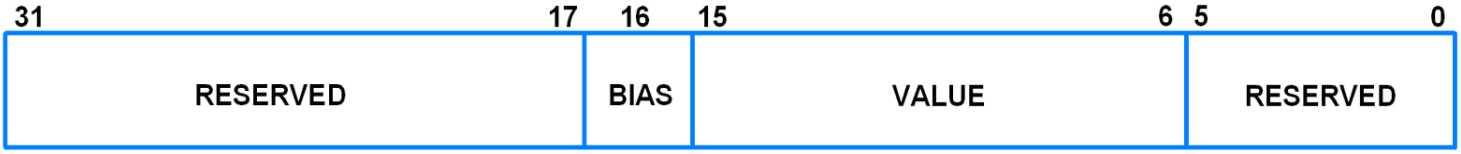


DAC Register Configuration

Let's see the Register used for DAC

DACR (DAC Register)

- DACR is a 32-bit register.
- It is a read-write register.



DACR (DAC Register)

- **Bit 5:0 –** RESERVED

- **Bits 15:6 –** VALUE

  This field contains the 10-bit digital value that is to be converted in to Analog voltage. We can get Analog output voltage on AOUT pin and it is calculated with the formula (VALUE/1024) * VREF.
- **Bit 16 –** BIAS

  0 = Maximum settling time of 1µsec and maximum current is 700µA

  1 = Settling time of 2.5µsec and maximum current is 350µA

  Note that, the settling times are valid for a capacitance load on the AOUT pin not exceeding 100 pF. A load impedance value greater than that value will cause settling time longer than the specified time.
- **Bit 31:17 –** RESERVED

Interfacing Stepper Motor with LPC2148

Stepper motor is a brushless DC motor that divides the full rotation angle of 360° into number of equal steps.

The motor is rotated by applying a certain sequence of control signals. The speed of rotation can be changed by changing the rate at which the control signals are applied.

Various stepper motors with different step angles and torque ratings are available in the market.

Microcontroller can be used to apply different control signals to the motor to make it rotate according to the need of the application.

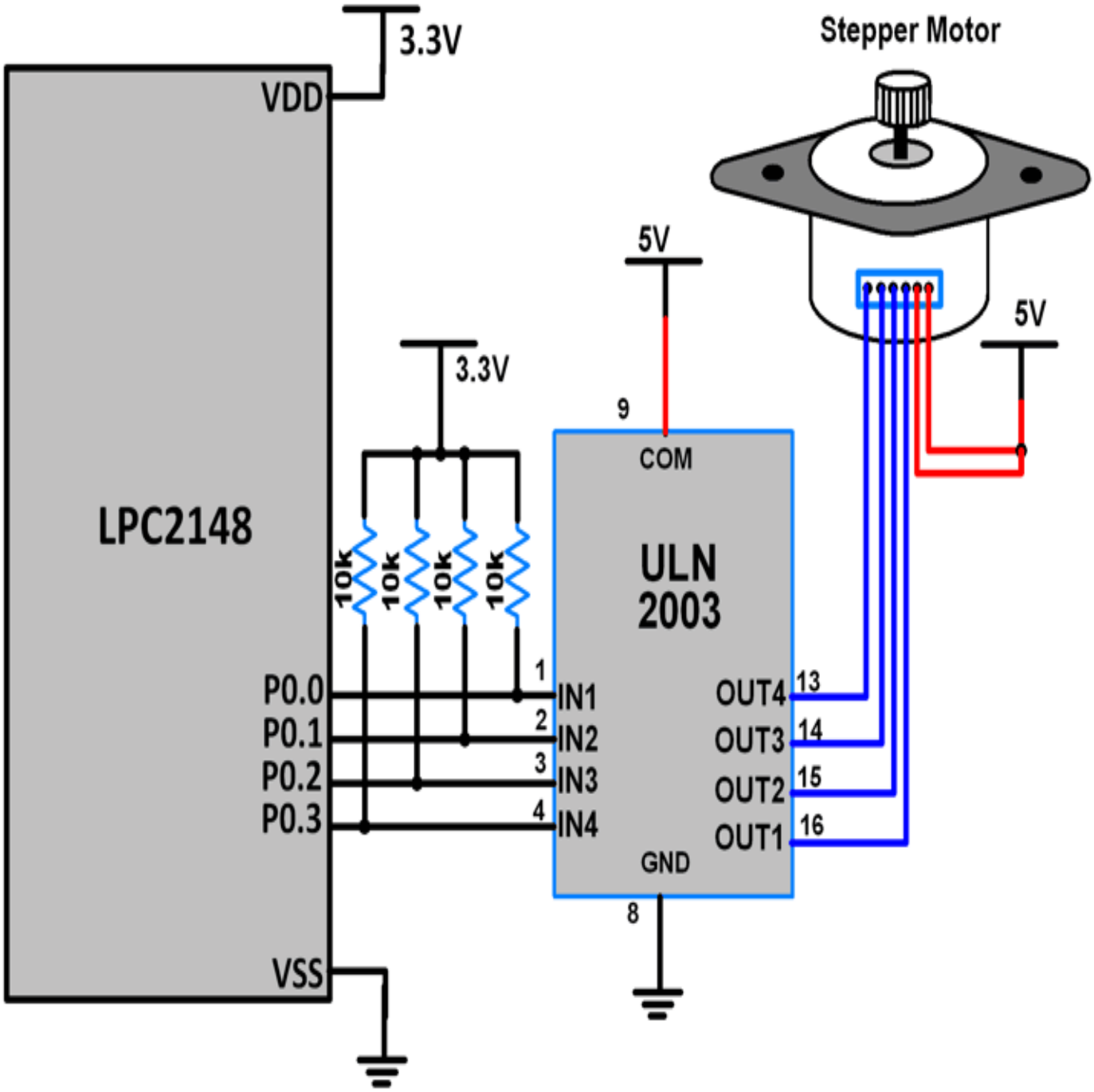
For more information about Stepper Motor and how to use it, refer the topic Stepper Motor in the sensors and modules section.

Controlling a stepper motor using **LPC2148 Development Board**. It works by turning ON & OFF a four I/O port lines generating at a particular frequency.
The ARM7 **LPC2148 Development Board** has four numbers of I/O port lines, connected with I/O Port lines (P1.16 – P1.19) to rotate the stepper motor. ULN2803 is used as a driver for port I/O lines, drivers output connected to stepper motor, connector provided for external power supply if needed.

Source Code

The Interfacing stepper motor control with LPC2148 program is very simple and straight forward, that control the stepper motor in clockwise, counter clockwise and also a particular angular based clockwise by using switches. The I/O port lines are used to generate pulses for stepper motor rotations. C programs are written in Keil software.

Interfacing Diagram



Testing the Stepper Motor with LPC2148

Give +3.3V power supply to **LPC2148 Development Board**; the Stepper Motor is connected with **LPC2148 Development Board**. When the program is downloading into LPC2148 in Primer Board, the LED output is working that the LED is ON some time period and the LED is OFF some other time period for a particular frequency. Now, the stepper motor is rotating.

☞If you are pressed switch sw1, then the stepper motor is rotating in clockwise direction.

☞If you are pressed switch sw2, then the stepper motor rotating in anti-clockwise direction.

☞If you are pressed switch sw3, then the motor is rotating the required angle which angle is set in code.

If you are not reading any output from LED, then you just check the jumper connections & check the LED is working.

If the stepper motor is not rotating then check the motor connections. Otherwise you just check the code with debugging mode in Keil.

UNIT –IV – ARM COMMUNICATION

Controller Area Network (CAN) Prototyping With the ARM Cortex-M3 Processor

---

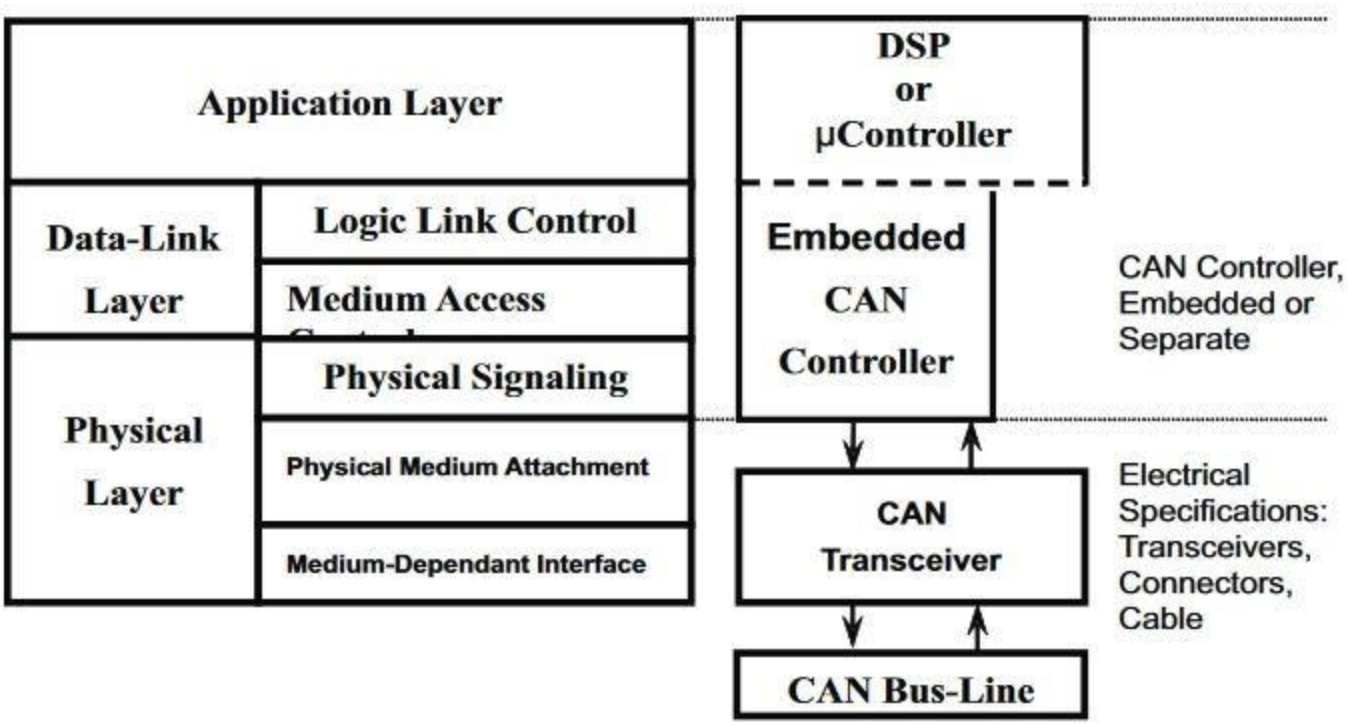
Communication protocols like UART (Serial), I2C and SPI are very popular because several peripherals can be interfaced with Arduino using these protocols. CAN (Controller Area Network) is another such protocol, which isn't very widely popular in general, but find several applications in the automotive domain

- CAN is a message-based protocol (i.e., the message and content are more important than the sender). A message transmitted by one device is received by all devices,including the transmitting device itself.
- If multiple devices are transmitting at the same time, the device with the highest priority continues transmission, while others back off. Note that since CAN is a message based protocol, IDs are assigned to messages and not the devices.
- It uses two lines for data transmission CAN_H and CAN_L. The differential voltage between these lines determines the signal. A positive difference above a threshold indicates a 1, while a negative voltage indicates a 0
- The devices in the network are called nodes. CAN is very flexible in the sense that newer nodes can be added to the network, and nodes can be removed as well. All the nodes in the network only share two lines.
- Data transmission happens in frames. Each data frame contains an 11 (base frame format) or 29 (extended frame format) identifier bits and 0 to 8 data bytes.

CAN Bus is used extensively in:

- Transportation systems (rail vehicle, aircraft, marine, etc.)
- Industrial machine control systems
- Home and building automation (e.g. HVAC, elevators)
- Mobile machines (construction and agriculture equipment)
- Medical devices and laboratory automation, as well as in many other embedded and deeply embedded applications.

System Components and Network Structure



CAN Bus Types

CAN High Speed (CAN 2.0B)

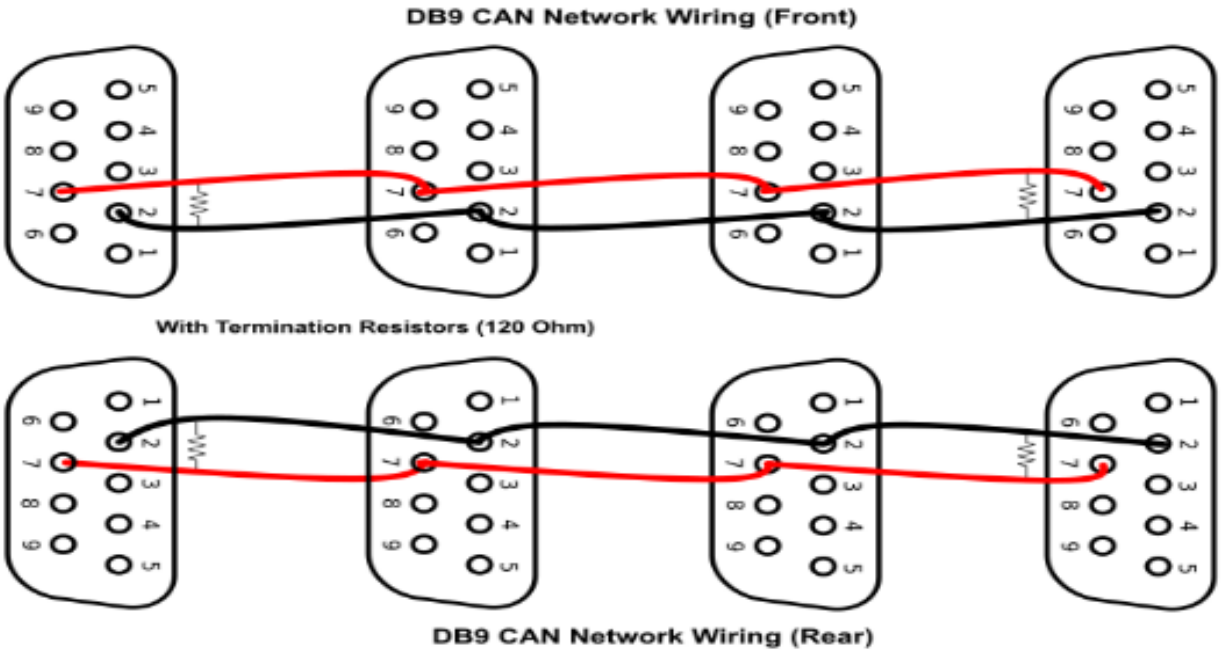
- Speed: Up to 1Mbps
- Range: 40m
- 29bit Message Identifier
- Termination with 120 Î© Resistor

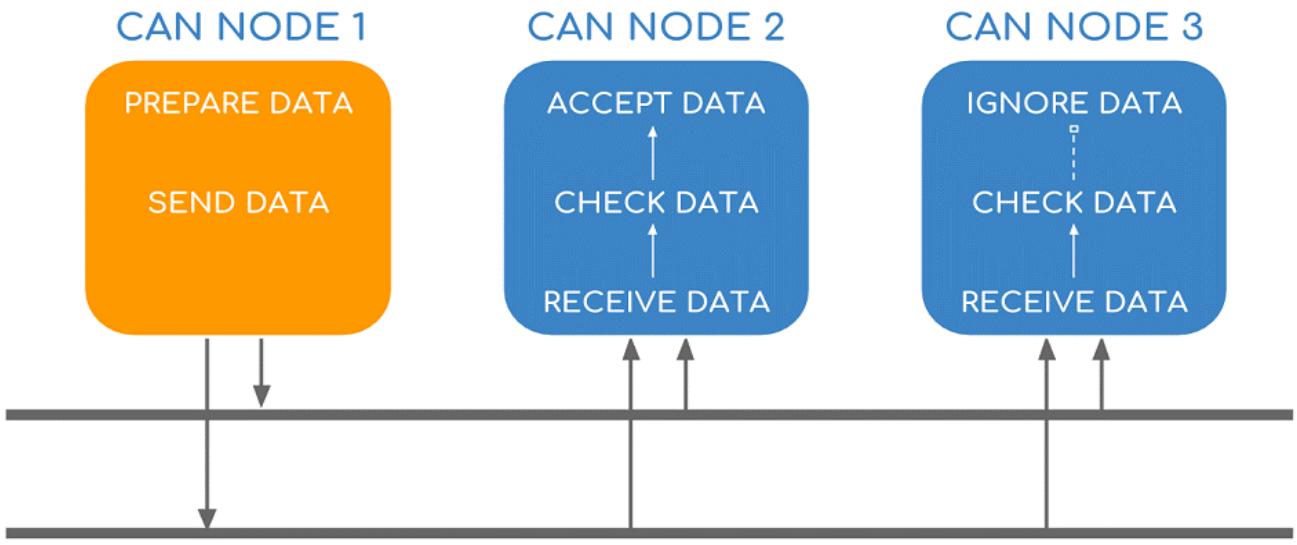
CAN Low Speed (CAN 2.0A)

- Speed: Up to 125Kbps
- Range: 500m
- 11bit Message Identifier
- overall termination resistance should be about 100 Î©

CAN FD (Flexible Data Rate)

- Speed: Up to 15Mbps
- Range 10m





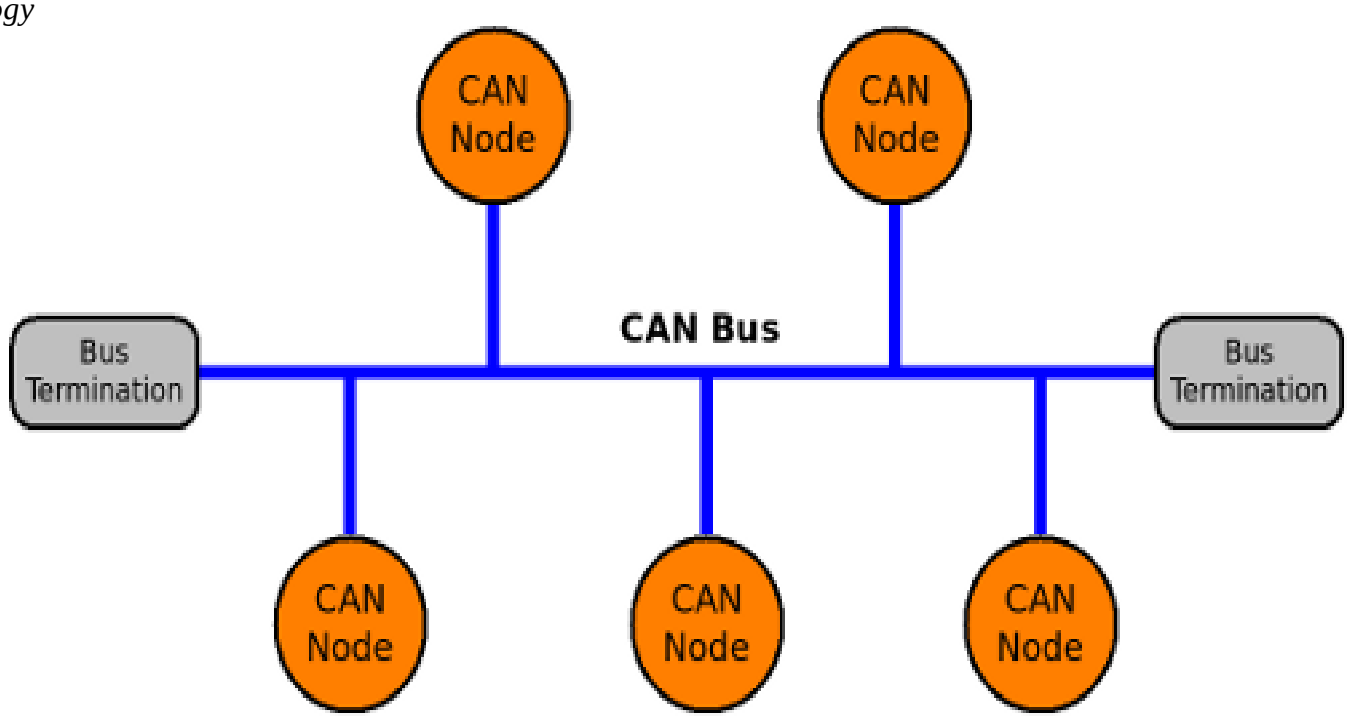
CAN BUS Notwork Wiring Diagram

Is CAN Synchronous or Asynchronous?

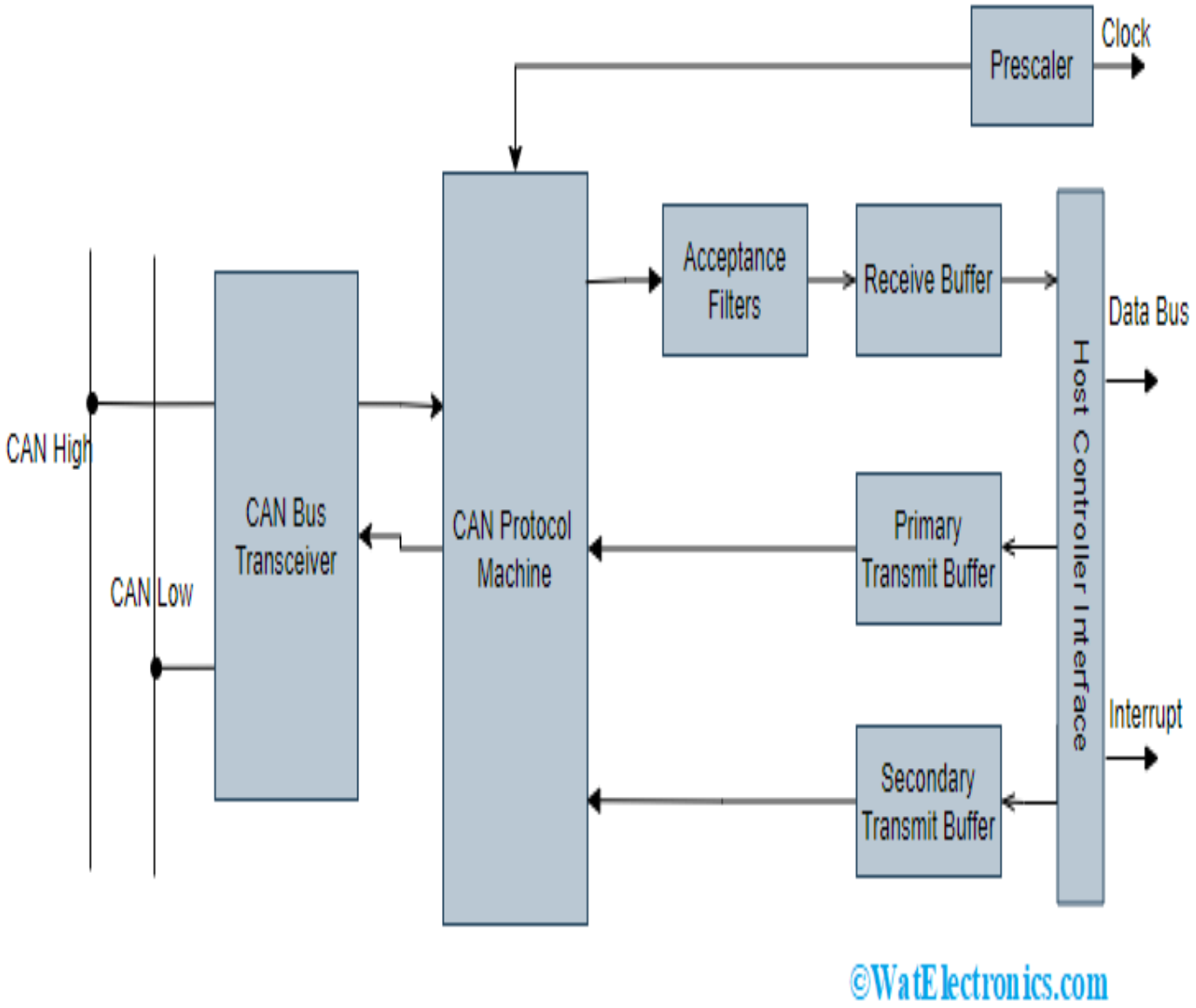
CAN data transmission uses a lossless bitwise arbitration method of contention resolution. This arbitration method requires all nodes on the CAN network to be synchronized to sample every bit on the CAN network at the same time. Therefore, some call CAN synchronous. (Unfortunately, the term synchronous is imprecise since the data is transmitted without a clock signal in an asynchronous format) All nodes are connected to each other through a two-wire bus. The wires are a twisted pair with a 120 Î© (nominal) characteristic impedance.

NOTE: The CAN bus must be terminated. The termination resistors are needed to suppress reflections as well as return the bus to its recessive or idle state.

Topology



CAN allows multiple devices (referred to as "nodes"), two or more nodes are required on the CAN network to communicate. The complexity of the node can range from a simple I/O device or embedded computer or a gateway. Each node requires a: Central processing unit: microprocessor, or host processor CAN controller: often an integral part of the microcontroller CAN Transceiver: Defined by ISO 11898-2/3 Medium Access Unit [MAU] standards.

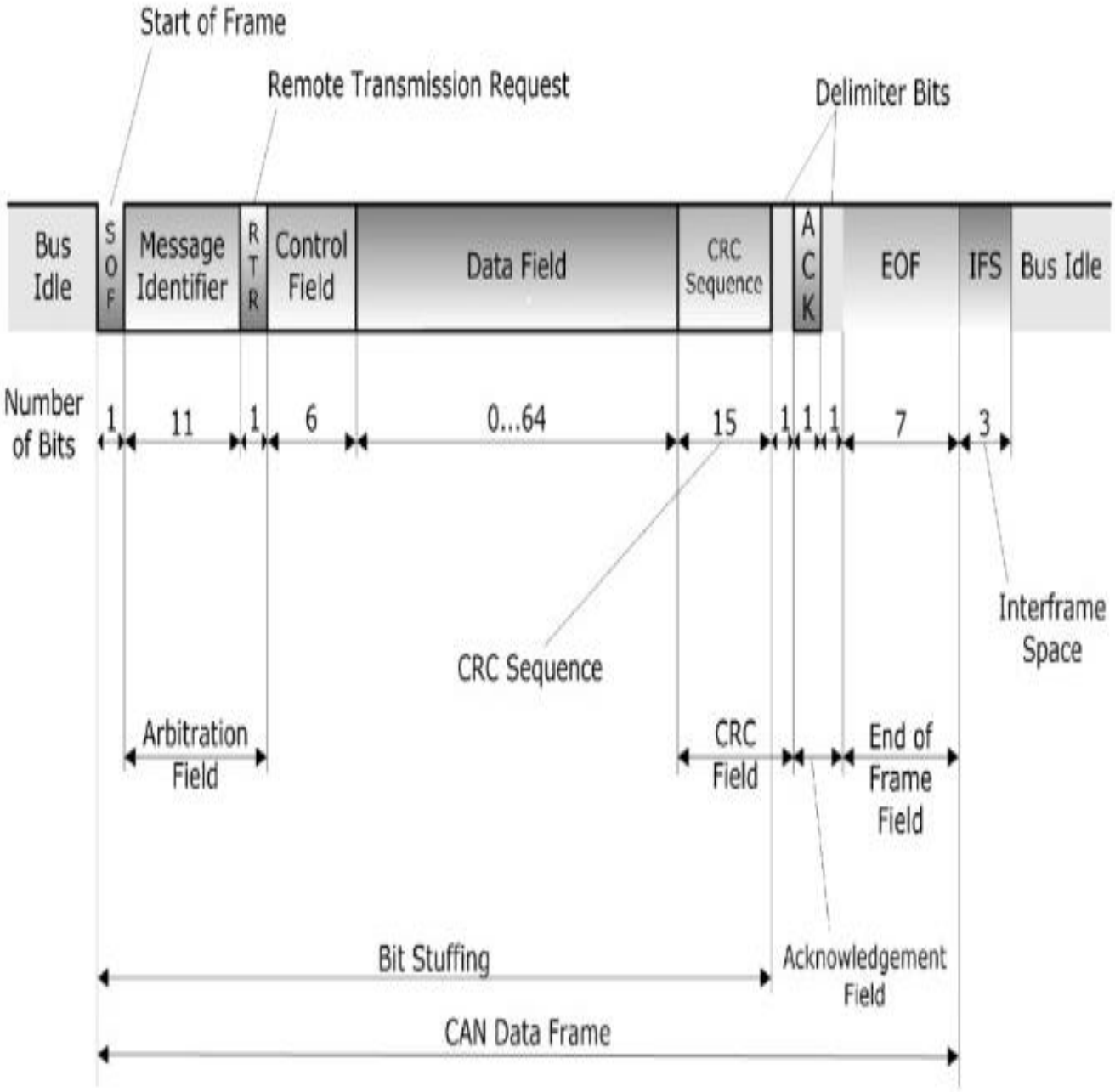


Data And Remote Frames

The following will address data and remote frames and how they, consequently, are distinguished from each other. Both, Data Frame and Remote Frame, are very similar. Basically, the Remote Frame is a Data Frame without the Data Field.

Per definition a CAN data or remote frame has the following components:

- **SOF** (Start of Frame) - Marks the beginning of data and remote Frames
- **Arbitration Field** – Includes the message ID and RTR (Remote Transmission Request) bit, which distinguishes data and remote frames
- **Control Field** – Used to determine data size and message ID length
- **Data Field** – The actual data (Applies only to a data frame, not a remote frame)
- **CRC Field** - Checksum
- **EOF** (End of Frame) – Marks the end of data and remote frames



Applications

1. Passenger vehicles, trucks, buses (combustion vehicles and electric vehicles)
2. Agricultural equipment.
3. Electronic equipment for aviation and navigation.
4. Industrial automation and mechanical control.
5. Elevators, escalators.
6. Building automation.

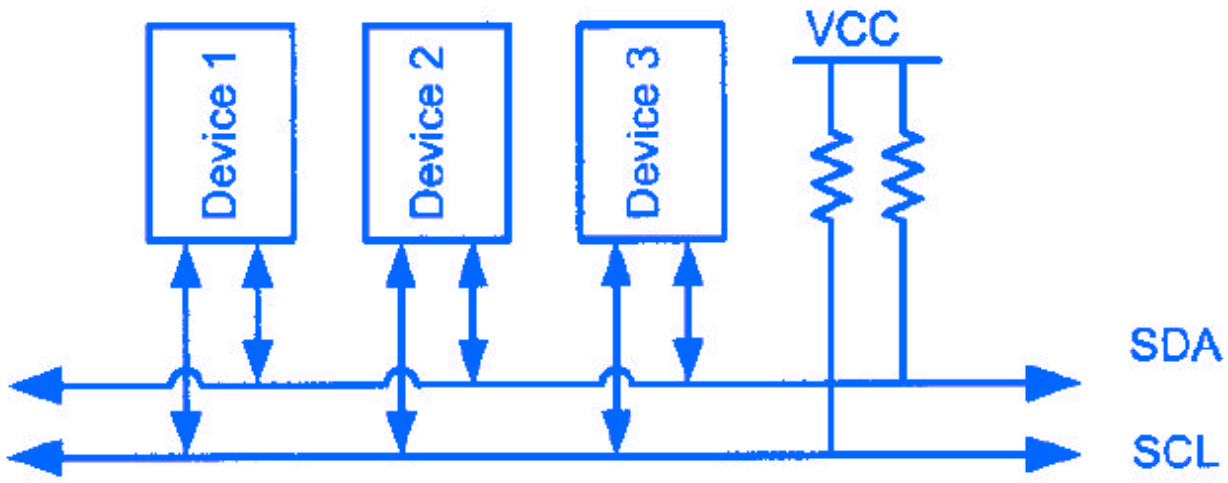
7. Medical instruments and equipment.

- **I2C PROTOCOL**

Introduction to I2C

I2C (Inter Integrated Circuit) is a serial bus interface connection protocol first invented by Philips Semiconductor (Now NXP Semiconductors). It is also called as TWI (two wire interface) since it uses only two wires for communication. I2C uses a handshaking mechanism for communication. Hence, it is also called as acknowledgment based communication protocol.

I2C PROTOCOL:



This protocol uses 2 bidirectional open drain pins SDA and SCK for data communication. SCL(Serial Clock) is used to synchronize the data transfer between these two chips and SDA to transfer the data to the devices. Therefore this protocol will allow us to reduce communication pins, package size and power consumption drastically. Each devices connected to the I2C line is known as nodes and the communication lines should be activated by means of a pull up resistor.Each data bit is transferred on the SDA line is synchronized by a high to low pulse clock on the SCL line. And the data line cannot change when the clock line is low.

START AND STOP CONDITIONS:

I2C communications are initiated and terminated by means of a START and STOP Conditions. START condition is generated by a high to low change in SDA line when SCL is high wheras STOP condition is generated by low to high change in SDA line when SCL is low.

I2C works in two modes namely,

- **Master Mode**

  Master is responsible for generating clock and initiating communication

- **Slave Mode**

  Slave receives the clock and responds when addressed by the Master

Two wires of the I2C interface are SDA (serial data) and SCL (serial clock).

- SDA is Serial Data wire used for data transfer in between master and slave
- SCL is Serial Clock wire used for clock synchronization. Clock is provided by the master

There are two types of data transfer possible on I2C interface depending on Read/Write operation i.e.

Data transfer from Master Transmitter to Slave Receiver and

Data transfer from Slave Transmitter to Master Receiver

LPC2148 I2C

- LPC2148 has two I2C interfaces. i.e. I2C0 & I2C1
- It has Programmable clock to support adjustable I2C transfer rates
- LPC2148 I2C is byte oriented and has four operating modes:

  Master Transmitter mode

  Master Receiver mode

  Slave Transmitter mode

  Slave Receiver mode

ARM LPC2148 I2C Pins

SCL0: - Serial Clock pin of I2C0.

SDA0: - Serial Data pin of I2C0.

SCL1: - Serial Clock pin of I2C1.

SDA1: - Serial Data pin of I2C1.

Let's see the registers that are used to initialize and control the operation of LPC2148 I2C.

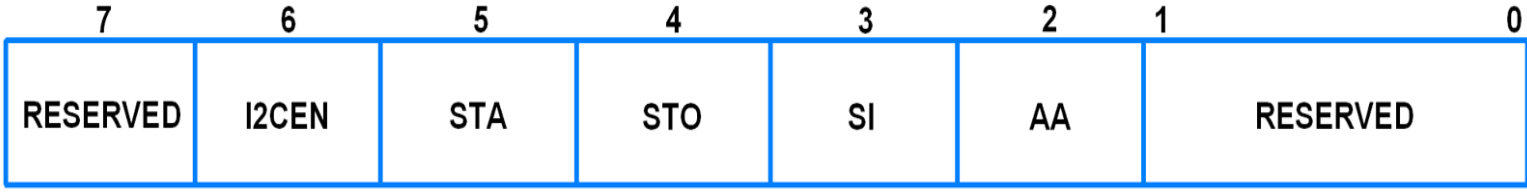
I2C0 and I2C1 have similar register formats and register names. The only difference in their register names is the I2C0 or I2C1 acronym at the beginning of the register name.

Here we will be showing I2C0 registers. Same can be used for I2C1 by changing their name (Replace the I2C0 part by I2C1).

I2C Registers

1. **I2C0CONSET (I2C0 Configuration Set Register)**

- It is an 8-bit read-write register.
- It is used to control the operation of the I2C0 interface.



I2C0CONSET (I2C0 Configuration Set Register)

- Writing a 1 to a bit in this register causes corresponding bit in the I2C control register to set.
- Writing a 0 has no effect.
- **Bit 2 – AA (Assert Acknowledge Flag)**
  When set to 1, Acknowledge (SDA LOW) is returned during acknowledge clock pulse on SCL.
  Otherwise, Not acknowledge (SDA HIGH) is returned.
- **Bit 3 – SI (I2C Interrupt Flag)**
  This bit is set when the I2C state changes. Else it remains reset.
- **Bit 4 – STO (Stop Flag)**
  In Master Mode, setting this bit causes the I2C interface to transmit a STOP condition. In Slave Mode, setting this bit causes recover from an error condition if any.
  This bit is cleared by hardware automatically.

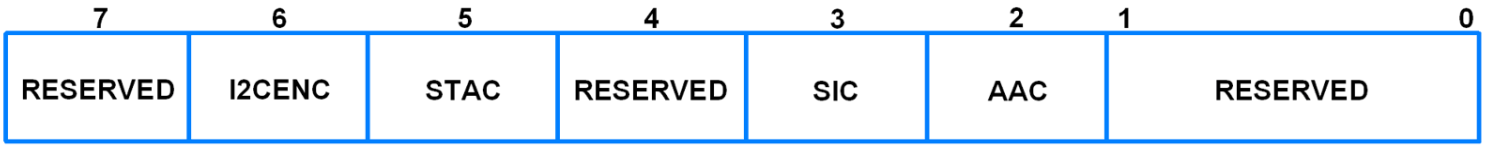
- **Bit 5 – STA (Start Flag)**

  Setting this bit causes the I2C interface to enter in master mode and transmit a START condition or transmit a repeated START condition if it is already in master mode.
- **Bit 6 – I2CEN (I2C Interface Enable)**

  Set this bit to enable I2C interface.

2. **I2C0CONCLR (I2C0 Configuration Clear Register)**

- It is an 8-bit write only register.



I2C0CONCLR (I2C0 Configuration Clear Register)

- Writing a 1 to a bit in this register causes corresponding bit in the I2C control register to clear.
- Writing a 0 has no effect.
- **Bit 2 – AAC (Assert Acknowledge Flag Clear)**

  Setting this bit clears the AA bit in I2C0CONSET register.
- **Bit 3 – SIC (I2C Interrupt Flag Clear)**

  Setting this bit clears the SI bit in I2C0CONSET register.
- **Bit 5 – STAC (Start Flag Clear)**

  Setting this bit clears the STA bit in I2C0CONSET register.
- **Bit 6 – I2CENC (I2C Interface Disable)**

  Setting this bit clears the I2CEN bit in I2C0CONSET register.

3. **I2C0STAT (I2C0 Status Register)**

- It is an 8-bit read only register.
- It reflects the present status of the I2C interface.

I2C0STAT (I2C0 Status Register)

- **Bit 2:0 – Unused bits**

  These are unused bits and are always 0.

- **Bit 7:3 – Status**

  These are status bits which provide the status of the current event on I2C bus.

  There are 26 possible status codes.

  Example, I2C0STAT = 0x08. This code indicates that a start condition has been transmitted.

  For other status codes, refer Table 232 (Page 228) to table 235 of the datasheet given in the attachments section below.

4. **I2C0DAT (I2C0 Data Register)**

- It is an 8-bit read-write register.

I2C0DAT (I2C0 Data Register)

- It contains the data to be transmitted or received.
- This register can be read or written to only when SI = 1.

5. **I2C0SCLL (I2C0 SCL Low Duty Cycle Register)**

- It is a 16-bit register.

I2C0SCLL (I2C0 SCL Low Duty Cycle Register)

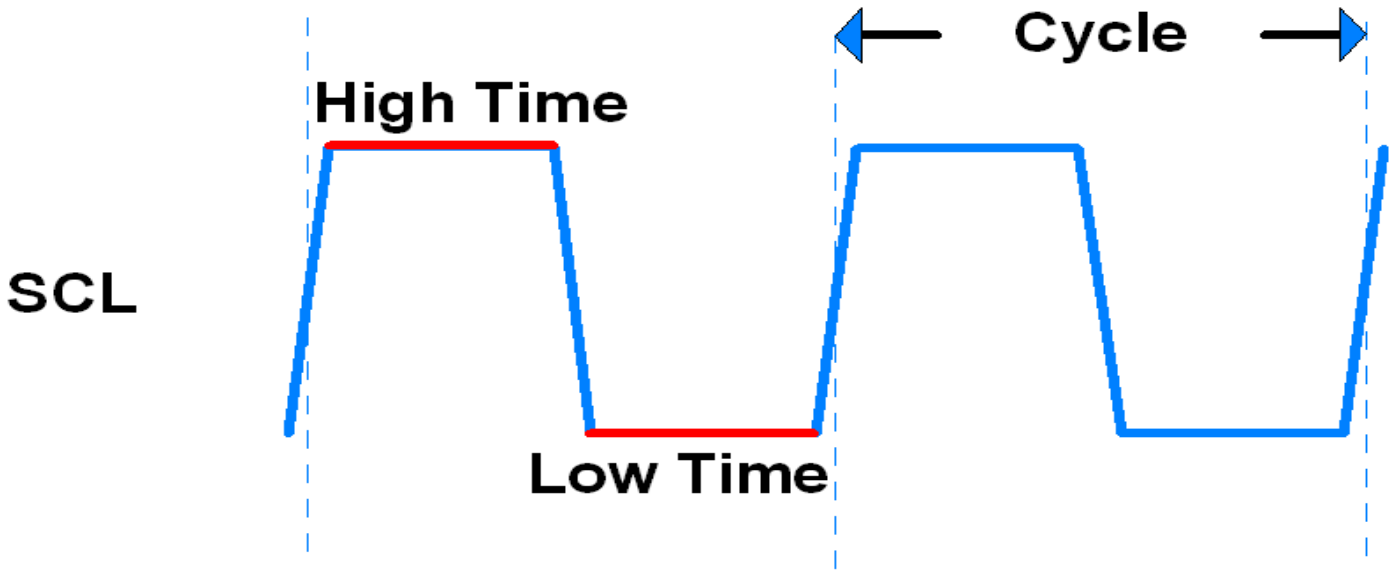
- This register contains the value for SCL Low time of the cycle.

6. **I2C0SCLH (I2C0 SCL High Duty Cycle Register)**

- It is a 16-bit register.

I2C0SCLH (I2C0 SCL High Duty Cycle Register)

- This register contains the value for SCL High time of the cycle.



The frequency and duty cycle of SCL is decided using I2C0SCLL and I2C0SCLH. I2C0SCLH contains the TON (High) time and I2C0SCLL contains the TOFF (Low) time.

The frequency is calculated as follows:

$$I2CBitFrequency = \frac{Fpclk}{I2C0SCLL + I2C0SCLH}$$

Example: Assume FPCLK = 30MHz, I2CBitFrequency = 300KHz, Duty Cycle = 50%.

As duty cycle is 50%, I2C0SCLL = I2C0SCLH.

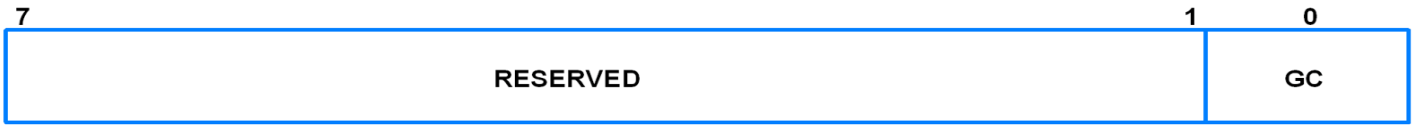
Hence,

$$300000 = \frac{30000000}{2xI2C0SCLH}$$

Hence, I2C0SCLL = I2C0SCLH = 50

7. **I2C0ADR (I2C0 Slave Address Register)**

- It is an 8-bit read-write register.



I2C0ADR (I2C0 Slave Address Register)

- It is only used when device is in Slave Mode.
- **Bit 0 – GC (General Call)**

  When this bit is set, the general call address (0x00) is recognized.
- **Bit 7:1 – Address**

  It contains the I2C device address for slave mode.
-

<u>STEPS TO PROGRAM I2C</u>:

MASTER MODE:

1. Load the values in the I2SCLH and I2SCLL register to set the required bit frequency.
2. Enable the I2EN bit in the I2CONSET register.
3. Set the Acknowledgement bit and start bit in the I2CONSET register.
4. Check for the status 0x08 in the I2STAT register which was defined in the datasheet.
5. Transmit the slave address, followed by the databytes.
6. Send the stop signal by activating the bit STO in the I2CONSET register.
7. Disable the I2C by writing 1 to the bit I2EN in the I2CONCLR register,

SLAVE MODE:

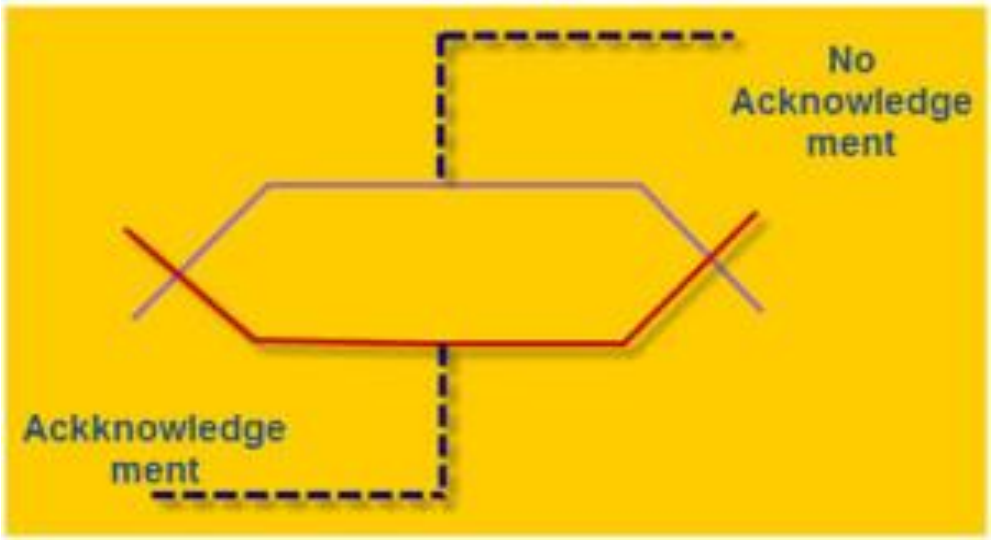
1. Slave address register should be loaded with the slave address.
2. The I2EN bit must be set to enable the I2C function and AA bit must be set to acknowledge general address or call address.
3. Receive or transmit data based on the direction bit received along with the slave address.

Kindly refer the datasheet for all modes of program, since the topic is vast and cannot be covered within this article. Also datasheet did an exceptional job in explaining the registers and modes of I2C operation. The link to download the datasheet is available below.

DATASHEET DOWNLOAD

Acknowledgement (ACK) and No Acknowledgement (NCK) Condition

Each byte transmitted over the I2C bus is followed by an acknowledge condition from the receiver, which means, after the master pulls SCL low to complete the transmission of 8-bit, the SDA will be pulled low by the receiver to the master. If, after the transmission of the receiver does not pull, the SDA line LOW is considered to be a NCK condition.



Data Transfer Format

Start: Primarily, the data transfer sequence initiated by the master generating the start condition.
7-bit Address: After that the master sends the slave address in two 8-bit formats instead of a single 16-bit address.
R/W: If the read and write bit is high, then the write operation is performed.

ACK: If the write operation is performed in the slave device, then the receiver sends the 1-bit ACK to the microcontroller.
Stop: After completion of the write operation in the slave device, the microcontroller sends the stop condition to the slave device.

Master to Slave Read Operation

The data is read back from the slave device in the form of bit or bytes – read the most significant bit first and read the least significant bit last.

The Data Read Format

Start: Primarily, the data transfer sequence is initiated by the master generating the start condition.
7-bit Address: After that the master sends the slave address in two 8-bit formats instead of a single 16-bit address.
R/W: If the read and write bit is low, then the read operation is performed.
ACK: If the write operation is performed in the slave device, then the receiver sends the 1-bit ACK to the microcontroller.
Stop: After completion of the write operation in the slave device, the microcontroller sends the stop condition to the slave device.

Applications of I2C

It is the best choice for those applications that require less costly and easy implementation rather than high-speed.

1. Reading certain memory ICs
2. Accessing DACs and ADCs
3. Transmitting and controlling user-directed actions
4. Reading hardware sensors
5. Communicating with multiple micro-controller

Advantages of I2C

There are the following advantages:

1. It provides flexible data transmission rates.
2. It provides long-distance communication than SPI.
3. Each device on the bus is controlled independently.

4. It increases the complexity of firmware or low-level hardware.
5. This protocol imposes overhead that also reduces throughput.
6. This protocol requires only two cables.
7. It can accommodate several master interactions through arbitration and collision detection.

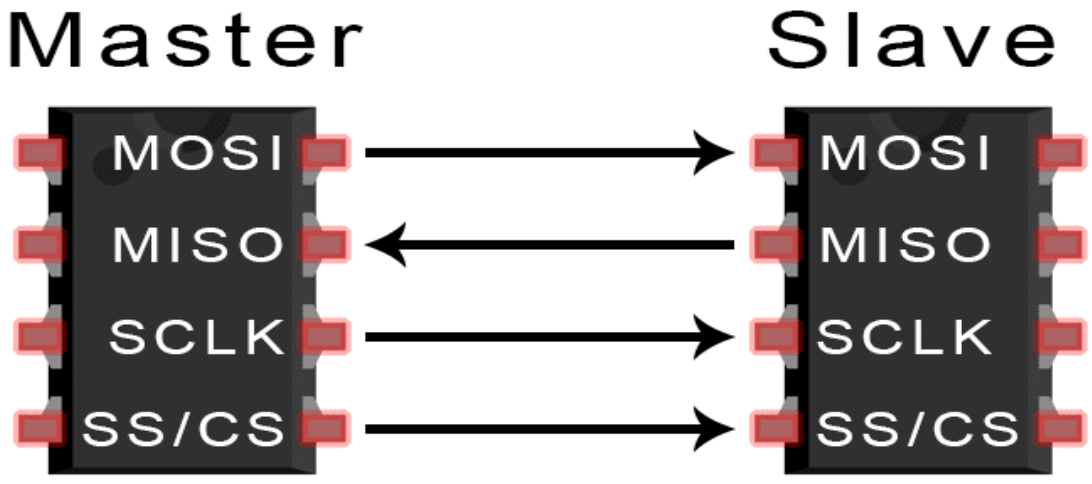
Disadvantages of I2C

1. The complexity of hardware increases when no. of master/slave devices are high in the circuit.
2. It provides a half-duplex mode for communication.
3. It is managed by the stack.
4. Many devices have multiple addresses stored, which can cause conflicts.

SPI COMMUNICATION PROTOCOL

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

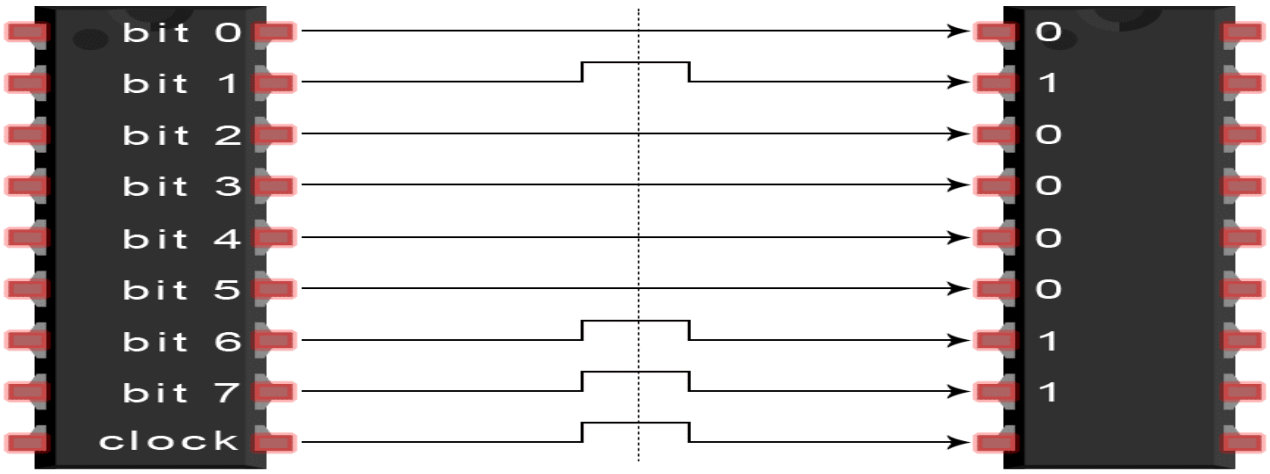
One unique benefit of SPI is the fact that data can be transferred without interruption. Any number of bits can be sent or received in a continuous stream. With I2C and UART, data is sent in packets, limited to a specific number of bits. Start and stop conditions define the beginning and end of each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



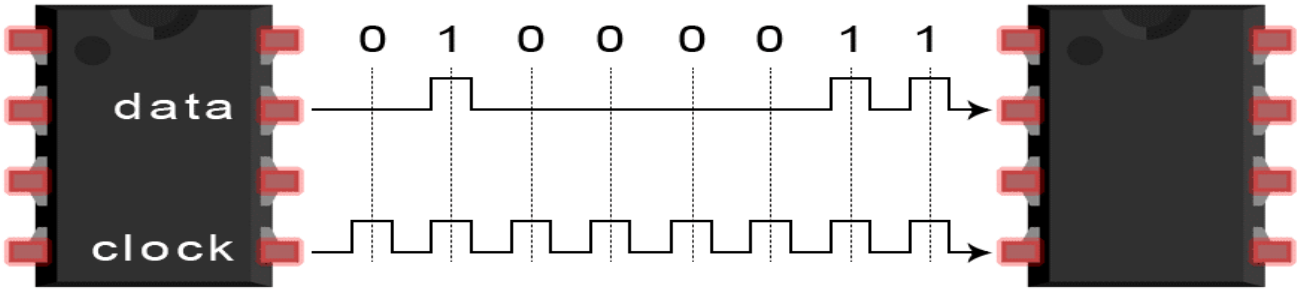
SERIAL VS. PARALLEL COMMUNICATION

Electronic devices talk to each other by sending *bits* of data through wires physically connected between devices. A bit is like a letter in a word, except instead of the 26 letters (in the English alphabet), a bit is binary and can only be a 1 or 0. Bits are transferred from one device to another by quick changes in voltage. In a system operating at 5 V, a 0 bit is communicated as a short pulse of 0 V, and a 1 bit is communicated by a short pulse of 5 V.

The bits of data can be transmitted either in parallel or serial form. In parallel communication, the bits of data are sent all at the same time, each through a separate wire. The following diagram shows the parallel transmission of the letter "C" in binary (01000011):



In serial communication, the bits are sent one by one through a single wire. The following diagram shows the serial transmission of the letter "C" in binary (01000011):



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

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

SCLK (Clock) – Line for the clock signal.

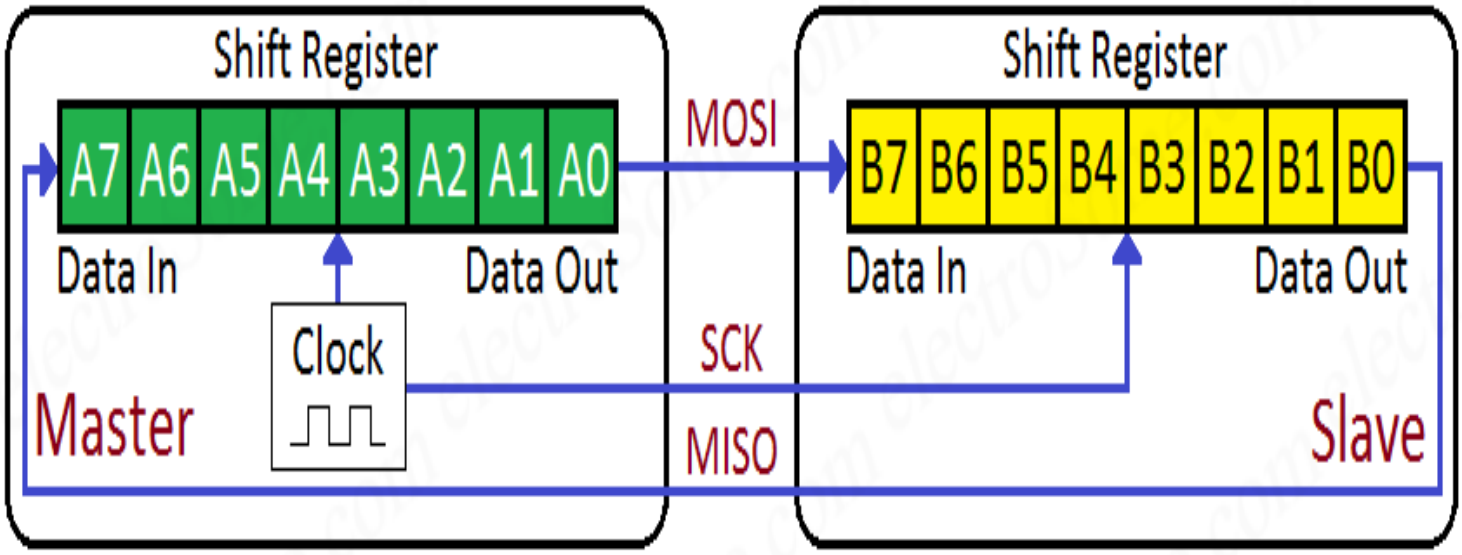
SS/CS (Slave Select/Chip Select) – Line for the master to select which slave to send data

HOW SPI WORKS

THE CLOCK

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

Any communication protocol where devices share a clock signal is known as *synchronous.* SPI is a synchronous communication protocol. There are also *asynchronous* methods that don't use a clock signal. For example, in UART communication, both sides are set to a pre-configured baud rate that dictates the speed and timing of data transmission.
The clock signal in SPI can be modified using the properties of *clock polarity* and *clock phase*. These two properties work together to define when the bits are output and when they are sampled. Clock polarity can be set by the master to allow for bits to be output and sampled on either the rising or falling edge of the clock cycle. Clock phase can be set for output and sampling to occur on either the first edge or second edge of the clock cycle, regardless of whether it is rising or falling.



SLAVE SELECT

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

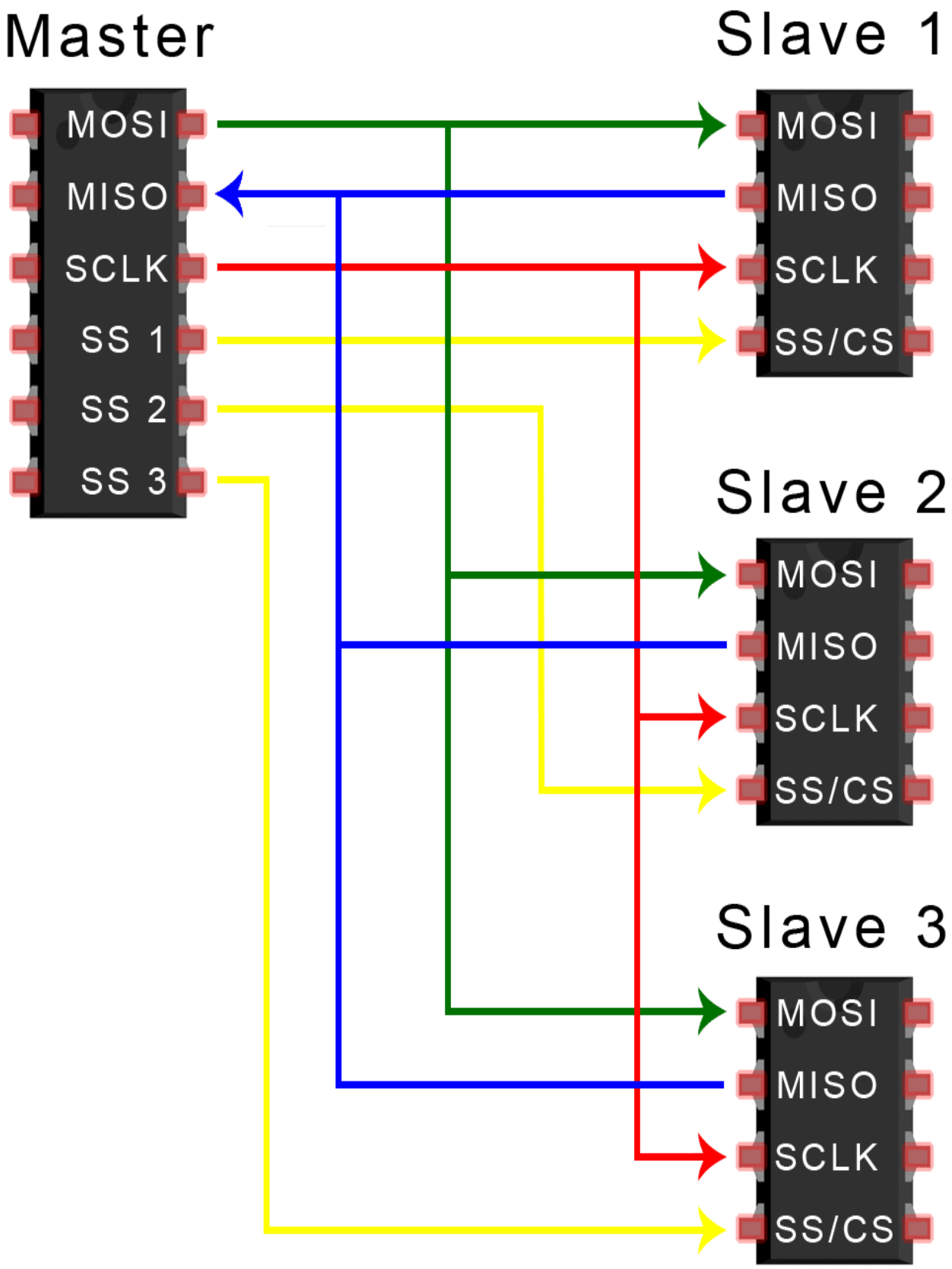
MULTIPLE SLAVES

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

MOSI AND MISO

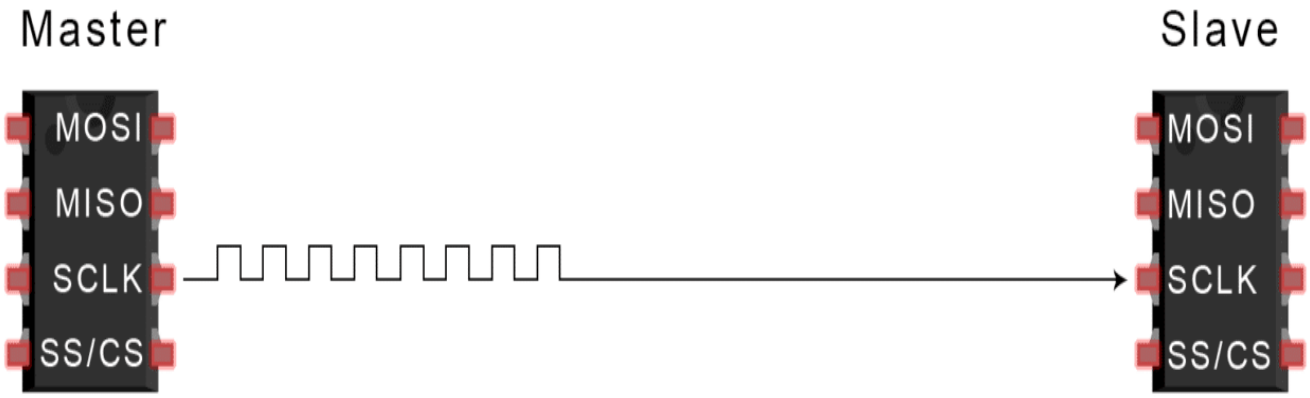
The master sends data to the slave bit by bit, in serial through the MOSI line. The slave receives the data sent from the master at the MOSI pin. Data sent from the master to the slave is usually sent with the most significant bit first.

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


Master
MOSI
MISO
SCLK
SS 1
SS 2
SS 3
Slave 1
MOSI
MISO
SCLK
SS/CS
Slave 2
MOSI
MISO
SCLK
SS/CS
Slave 3
MOSI
MISO
SCLK
SS/CS

STEPS OF SPI DATA TRANSMISSION

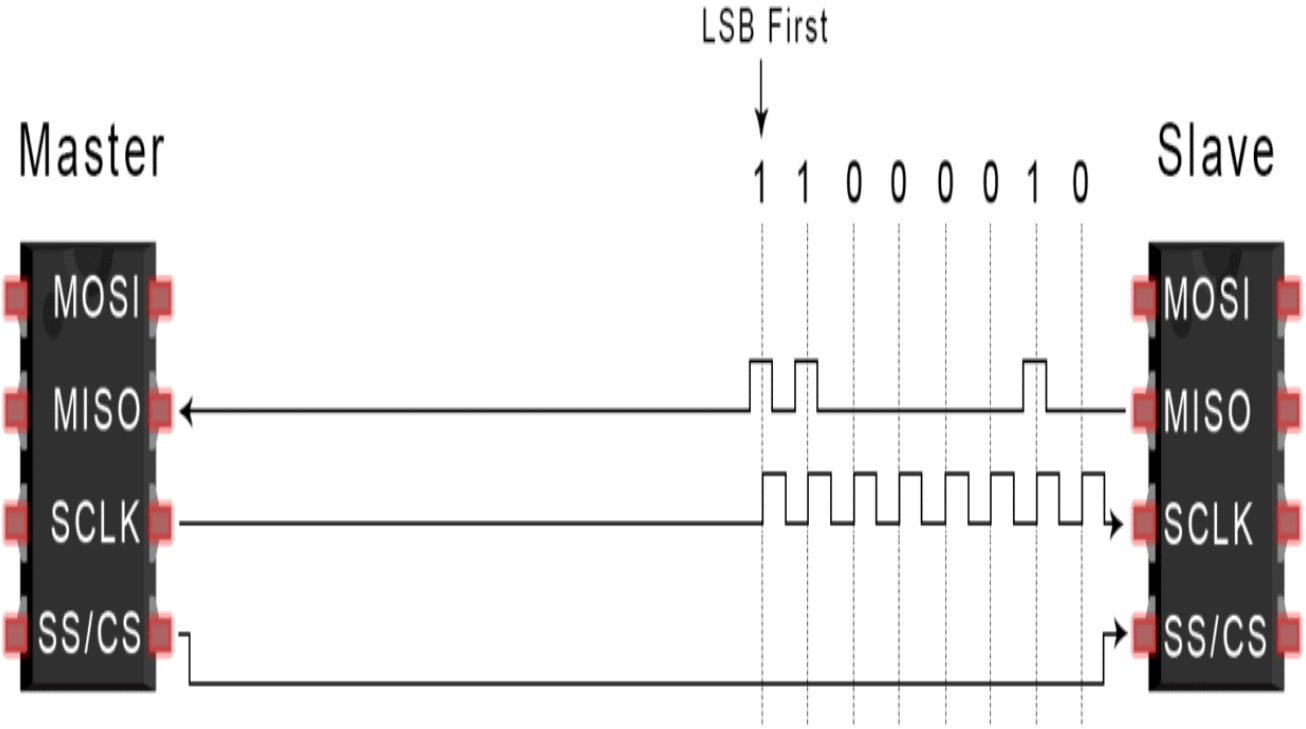
1. The master outputs the clock signal:



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

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

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



ADVANTAGES

- No start and stop bits, so the data can be streamed continuously without interruption
- No complicated slave addressing system like I2C
- Higher data transfer rate than I2C (almost twice as fast)
- Separate MISO and MOSI lines, so data can be sent and received at the same time

DISADVANTAGES

- Uses four wires (I2C and UARTs use two)
- No acknowledgement that the data has been successfully received (I2C has this)
- No form of error checking like the parity bit in UART
- Only allows for a single master

Applications of SPI

- Memory: SD Card, MMC, EEPROM, and Flash.
- Sensors: Temperature and Pressure.
- Control Devices: ADC, DAC, digital POTS, and Audio Codec.
- Others: Camera Lens Mount, Touchscreen, LCD, RTC, video game controller, etc.

UNIT –V

INTRODUCTION TO SINGLE BOARD EMBEDDED SYSTEMS

Introduction to Raspberry Pi

For a precise definition, Raspberry Pi is a single-board computer, designed for education, experimentation, and DIY projects. Raspberry Pi was first introduced by the Raspberry Pi Foundation in the United Kingdom and is now widely used by hobbyists and professionals, worldwide.

More highlights on Raspberry Pi:

- The Raspberry Pi is affordable, and requires low power with a compact design, making it ideal for various applications.
- Primarily, it is used as media centers, web servers, gaming consoles, and robotics.
- The Raspberry Pi features a Broadcom System-on-Chip (SoC) with an ARM processor, RAM, storage, GPIO (general-purpose input/output) pins, and various interfaces, such as USB, Ethernet, HDMI, and Wi-Fi.

- It can run multiple operating systems, including Raspbian, Ubuntu, and others, and supports various programming languages like Python, C++, and Scratch.
- The Raspberry Pi components include the board, a power supply, a microSD card for storage and operating system, a display (optional), and various accessories, such as a case, a keyboard, a mouse, and more.
- The Raspberry Pi can be connected to multiple sensors, actuators, and modules to create interactive projects and experiments.
- Raspberry Pi offers an array of applications in different industry sectors like education, research, entertainment, and many others.

What is Raspberry Pi Foundation?

The Raspberry Pi Foundation is a UK-based esteemed charitable group devoted to advancing computer programming education and digital making. Primarily, the organization is responsible for developing and innovating Raspberry Pi devices.

Furthermore:

- From establishing the Raspberry Pi website, publishing the MagPi magazine, and generating exciting projects and tutorials. The group is committed to inspiring and engaging learners from diverse backgrounds of all ages.
- The Raspberry Pi community supports its budding computer engineers by organizing forums, blogs, and events that feature the limitless potential and imagination of projects and applications.
- All revenue generated from selling Raspberry Pi products and services is dedicated to strengthening the foundation's mission – to broaden technology education across the globe.
- This organization has had far-reaching implications in democratizing access to computing technology and has allowed people to learn, create and make the most of technology.

- he inception of Raspberry Pi has revolutionized the tech industry. Raspberry Pi Pico is the smallest of all, yet it delivers the same results as a general computer system.

  Below are a few features of Raspberry Pi that make it worth the hype.

- **General Purpose Input/Output (GPIO) Pins**

- Raspberry Pi has a set of GPIO pins that allow users to connect external devices and components, such as sensors, motors, and LED lights, and control them programmatically. These pins can create many projects, from simple sensors and actuators to complex robotics and automation systems.

- **Low Power Consumption**

- Raspberry Pi is designed to be energy-efficient, with a power consumption of only a few watts. It suits it for running on batteries or solar panels, making it ideal for remote or off-grid applications.

- **Operating System Support**

- Raspberry Pi supports a range of operating systems, including Raspbian, Ubuntu, and Windows 10 IoT Core, which allows users to choose the best OS for their specific application or project.

- **Compact Size**

- Raspberry Pi is compact and portable, with a credit-card size form design. It makes carrying around and integrating into various projects and applications accessible.

- **Video and Graphics Support**

- Raspberry Pi's powerful graphics processing unit (GPU) supports full HD video playback and 3D graphics rendering. It makes it ideal for media centers and gaming applications.

- **Wireless Connectivity**

- Raspberry Pi has built-in Wi-Fi and Bluetooth connectivity, allowing it to connect to the internet and communicate with other devices wirelessly. It makes it suitable for IoT applications and remote monitoring and control systems.

- **Expandability**

- Raspberry Pi has various expansion options, such as adding additional memory and storage using microSD cards, USB drives, and external hard drives. It also has a range of accessory boards, known as HATs (Hardware Attached on Top), that provide additional functionality, such as GPS tracking, audio processing, and environmental sensing.

- Examples of Raspberry Pi projects that utilize these features include a smart home automation system that uses GPIO pins to control lights and appliances, a media center that uses the GPU for video playback and HDMI output, and a remote monitoring system that requires wireless connectivity and environmental sensors to monitor temperature and humidity levels.

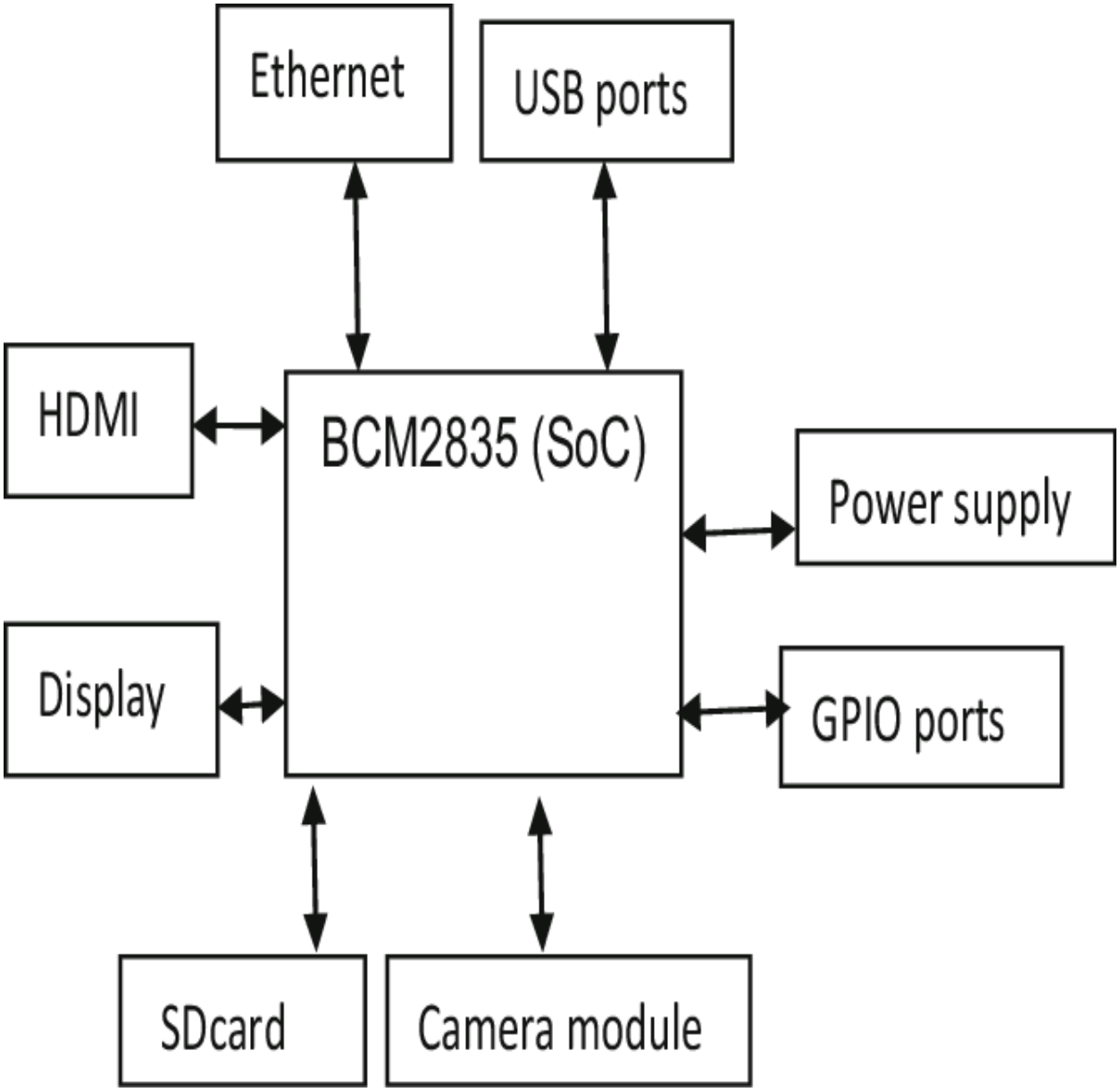
Raspberry Pi Architecture

- The diagram highlights various components and ports, such as the ARM architecture-based SoC, USB ports, Ethernet port, HDMI port, GPIO pins, and more. Additionally, the board is available in multiple models like the 1, 2, 3, 4, Zero, and Compute Module, each with distinctive specifications and capacitie

Components of Raspberry Pi

The Raspberry Pi packs a powerful punch in a small package with a 32-bit ARM processor, commonly used in mobile devices.

- The Pi boasts a RISC (Reduced Instruction Set Computing) architecture, offering streamlined instructions, reduced power consumption, and faster processing.
- This powerful processor runs at a clock speed of 1.2 GHz and is supported by a Broadcom BCM2835 System-on-Chip (SoC).
- Meanwhile, fans of graphics-intensive applications, such as video playback and gaming, include a powerful VideoCore IV GPU that can decode 1080p video at 30 frames per second.



The Raspberry Pi architecture incorporates a memory subsystem comprising 1 GB of RAM.

- The RAM is shared between the ARM processor and the GPU, and software configurations vary the split.
- Moreover, a microSD card slot stores the operating system and all user data.
- In addition to various input/output (I/O) interfaces, the Pi offers 40 GPIO (General Purpose Input/Output) pins, enabling connections with sensors, actuators, and other devices.

Raspberry Pi Applications

The Raspberry Pi is a versatile device for various applications due to its low cost and ease of use. The possibilities are endless, from educational programs and home automation systems to media centers and robotics.

Here are some of the most common benefits of the Raspberry Pi:

- **Education:** As it is accessible and low-cost, it is widely deployed in educational settings, robotics, and electronics.
  Examples: UK's National Curriculum for Computing, allowing students to practice programming in the Python language.
- **Home Automation:** Integrating sensors and other electronics, the Raspberry Pi can create a fully automated and intelligent home system.
  Examples: Devices such as lights and security cameras can be programmed to turn on or off, and the ambient temperature can be regulated from anywhere.
- **Media Centers:** Raspberry Pi can assemble media centers that stream movies and music from remote locations and local storage.
  Examples: Kodi, a hugely popular software, is installable on the Raspberry Pi, enabling users to access media content with a few simple clicks.
- **Internet of Things (IoT):** The Raspberry Pi is a valuable tool for creating IoT devices that collect and transmit data from connected devices and sensors.
  Examples: Developing a weather station: with Raspberry Pi, data such as temperature and humidity can be transferred to a web server, reporting readings in real-time.
- **Gaming:** The Raspberry Pie can be utilized to build gaming consoles, granting users access to classic titles from the 80s and 90s.
  Examples: RetroPie, a software that emulates classic gaming consoles, can be installed on the Raspberry Pi to provide a retro gaming experience.
- **Robotics:** The Raspberry Pi is an excellent resource for developing robots or drones. Using it, one can build a drone remotely operated from a smartphone or tablet.
  Examples: Raspberry Pi has a wide range of applications due to its versatility, low cost, and ease of use.

Booting Your Raspberry Pi for the First Time

After you're done writing the Raspberry Pi OS to a microSD card, it's time for the moment of truth.

1**. Insert the microSD card** into the Raspberry Pi.

2. **Connect the Raspberry Pi** to a monitor, keyboard and mouse.
3. **Connect an Ethernet cable** if you plan to use wired Internet.
4. **Plug the Pi in** to power it on.

If you had used the Raspberry Pi Imager settings to create a username and password, you'll be able to go straight into the desktop environment, but if not, you will get a setup wizard.

Using the Raspberry Pi First-Time Setup WIzard

If you chose a username and password in Raspberry Pi Imager settings, before writing the microSD card, you will get the desktop on first boot. But, if you did not, you'll be prompted to create a username and password and enter all the network credentials by a setup wizard on first boot. If that happens, follow these steps to finish setting up your Raspberry Pi.

1. **Click Next** on the dialog box.
2. **Set your country and and language** and click Next. The default choices may already be the correct ones.
3. **Enter a username and password** you wish to use for your primary login. **Click Next**.
4. **Toggle Reduce the size of the desktop" to on** if the borders of the desktop are cut off. Otherwise, just **click Next**.
5. **Select the appropriate Wi-Fi network** on the screen after, provided that you are connecting via Wi-Fi. If you don't have Wi-Fi or are using Ethernet, you can skip this.
6. **Enter your Wi-Fi password** (unless you were using Ethernet and skipped).
7. **Click Next** when prompted to Update Software. This will only work when you are connected to the Internet, and it can take several minutes. If you are not connected to the Internet, click Skip.
8. **Click Restart.**

Working with Rpi using with python

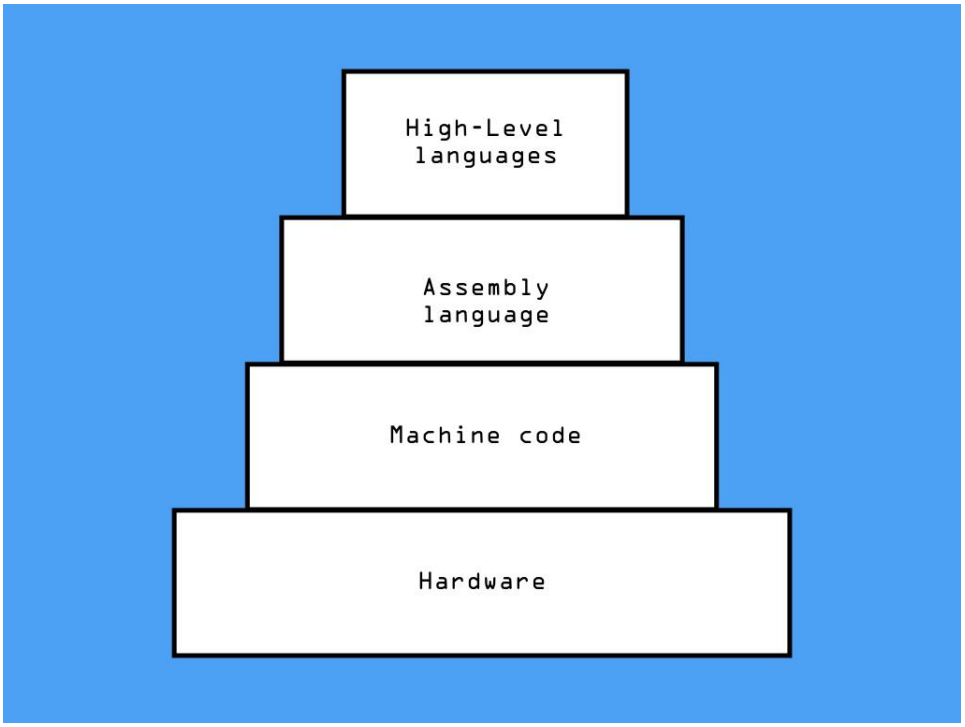
The Raspberry Pi made physical computing and programming accessible to many -- it is relatively inexpensive, and almost anyone could simply connect a monitor, keyboard, and mouse to get started. **Python** is an easy to start with high-level language, and it is an integral part of the Raspberry Pi's operating system.

In this guide, we'll show you how to get started with the Thonny IDE, learn about basic data types and control flow statements that are readily used when working with sensors and actuators on the Raspberry Pi.
Complete this guide to get started with Python programming on the Raspberry Pi.

Step 1 Overview

- The Raspberry Pi was created so that computing could be more accessible for everyone -- it is relatively inexpensive, and anyone could simply connect a monitor, keyboard, and mouse to get started. As sensors, actuators and other devices can be easily attached to it via its handy General purpose Input Output (GPIO) connector, it also serves as a gentle introduction into physical computing. The other impetus was to make programming more accessible, especially for beginners and kids. That's why **Python** is an integral part of the Raspberry Pi's operating system!
- In this guide, we'll show you how to get started with programming in Python on the Pi. We'll also briefly talk about the commonly used data types.

Step 2 What is a programming language?



- To communicate with a computer, we require the use of programming languages. A **programming language** is a language engineered to communicate instructions to a machine which understands machine code.

  Machine language or **machine code** is a lower-level form that is read and executed by the computer. This language is comprised of binary, 0's and 1's. In this machine code, all instructions, memory locations, numbers, and characters are represented in 0's and 1's.

  Assembly language implements a symbolic representation of machine code, that is to say, it replaces replaces the 0's and 1's with alphanumeric symbols in order to make it easier to remember and work with.

  While they can run and execute fast, the main disadvantage is that it is tedious for humans to use, maintain, or debug. Enter **high-level languages**, these are languages that seem a lot more like our human language, enabling us to focus on problem-solving.

Still, programming languages such as Java or C++ require the use of a compiler, which is a program that converts human-readable instructions or code, into machine code, and is then executed and run.

On the other hand, a programming language such as Python is executed directly using an **interpreter**, rather than needing to go through a compiler. The code is read one at a time, where each statement is translated into machine language and then executed.

- While a compiler scans the entire program and translates it as a whole into machine code, an interpreter translates the program one statement at a time.

Step 3 Starting Python for the First Time

One of the challenges when beginning to use Python is the need to install Python and other related software. Thankfully, all this is already set up for you in the latest version of Raspbian with PIXEL desktop, specifically Raspbian Buster at this time of writing.

The easiest way to get started is to use Thonny, a new IDE (integrated development environment) that can be easily accessed by navigating to **Raspberry Pi icon (menu) > Programming > Thonny Python IDE**

- You will see a script editor and a shell, you enter a program in the script editor and run it in the shell. Let's write our first program!

Step 4 Save and execute code

- Enter the following code into the IDE.
- Click on **Save**
- Name this file and click **OK**
- Click on **Run**
- Down the bottom in the IDE, you will see the output here.

Step 5 Run Python via Terminal

- Another way to write and run python programs is through the terminal. To access the terminal, click on the terminal icon up the top left-hand corner.
- Type the following command: nano helloworld.py
- Enter the following code
- Press CTRL+O to save
- Press CTRL+X to exit

- Now you can run the saved script by using the command: python helloworld.py

  You should see the program output in the terminal window.
- You can also write Python programs remotely via SSH. If you haven't yet, check out our guide on how to enable SSH on your Raspberry Pi.

Step 6 Data Types: Numbers

- One useful feature of a programming language, is the ability to manipulate variables. A **variable** can be thought of as a container, or a name, that holds a value.

  So far, in our test-thonny program, we've seen the **integer** data type (falls under the Numbers category) being used i.e. a = 17. The equal symbol here is an **assignment operator**, this assigns a value to a variable; We declared a variable, a, with an integer value of 17.

  Integers can be positive or negative whole numbers.
- Another commonly used Numbers data type is the **floating point number**. These are values with a decimal point or scientific notation such as 32.0 or 4.5. This divides whole numbers into fractional parts.

  For example, 32 is an integer, and 32.0 is a floating point number.
- Then there's the **long** or long integers, which are integers of infinite size. They look like integers but are followed by the letter 'L'

  For example: 150L is a long integer.
- Occasionally you may also come across **complex numbers**. They have applications related to mathematics and are represented by the formula **x + yi**

Step 7 Data Types: String

- Aside from integer numbers, there are other data types that we can use in our program. The first we'll take a quick look at is the **string** data type.

  A string is a sequence of characters, meaning that it is an ordered sequence of other values.

  Try entering this into the editor:

```
snack = 'cookie'
letter = snack[1]
print(letter)
```

- The first statement is a sequence of characters that make up the word cookie. The second statement selects character number 1 from snack and assigns it to letter.

  What are those brackets? The expression in these brackets is called an **index**, this indicates which character in the sequence you are trying to access.

  The index begins at 0, hence snack[1] will output 'o' instead of 'c'.

Step 8 Data Types: Boolean

- Python also allows the use of **boolean** data types. These are values that can either be True or False.

  The example here uses the **== operator** which compares two **operands**, and produces True if they are equal. Otherwise, it produces False.
- The == operator is one of the **relational operators**. Others include:

```
a != b    #a is not equal to b
a > b     #a is greater than b
a < b     #a is less than b
a >= b   # a is greater than or equal to b
a <= b   # a is less than or equal to b
```

Step 9 Data Types: List

- Sometimes, you may want a variable to hold a collection of values rather than just a single value. For that, there's the **list** data type, which can hold a collection of other data types. To create a list, simply use the [and] to contain its contents, for example:

```
snack = ['cookie', 'candy', 'chips', 'nuggets']
```

- Try printing the value at index 3:

  print(snack[3])

Step 10 Data Types: Dictionary

- When you want to access a collection of values, and you know where just it is placed, then you'd want to use a **dictionary**.

Dictionaries are an alternative to lists for storing collections of data. The difference is that a dictionary stores a **key and value** pair. As such, you can use the key to retrieve the value without needing to search through the entire collection.

To create a dictionary, use the { } notation.

For example:

inventory = {'LEDs' : '9000', 'Breadboards' : '5000', 'Transistors' = '6700', Pushbuttons = '6755'}

- Commonly used methods with the dictionary data type include the **keys()** method which will state all the keys in the dictionary.
- The **values()** method retrieves all the values in the dictionary.
- Please note that **dictionary.has_key('keyname')** has been deprecated. Instead, you should use '**keyname' in dictionary**

This will return true if the keyname is found in the dictionary, otherwise it will return false.

Step 11 Data Types: Tuple

- **Tuples** are like lists, in which it is a collection of values, but they are immutable which means its order cannot be changed. So they are especially useful in cases where the value cannot be modified.

They are separated by a comma, written in parentheses instead of square brackets, can be created as follows:

colours = ('blue', 'green', 'red', 'turquoise', 'purple', ['pastel pink', 'pastel blue', 'pastel yellow'])

In this example, colours contains five strings and one list.

- Since lists are mutable, if the tuple contains a list, then it can be modified:

colours[5][1] = 'pastel green'

Here, 'pastel blue' will be replaced with 'pastel green’

Step 12 Indentation

```
a = 15
if a > 20:
   print("a is larger than 20")
```

- Notice the leading whitespaces, or indentation? Python uses indentation to figure out what statements belong together. Consider the following example:

```
a = 15
if a > 20:
   print("a is larger than 20")
```

The indented print statement will let Python know if it should be executed if the statement returns True.

- You may use 4 consecutive spaces for a level of indentation.

Step 13 Control Flow: Conditional statements

```
import RPi.GPIO as GPIO
import Adafruit_DHT

sensor = Adafruit_DHT.DHT11
LED=40

GPIO.setmode(GPIO.BOARD)

GPIO.setup(LED,GPIO.OUT) #Red LED
GPIO.output(LED,GPIO.LOW)

humidity, temperature = Adafruit_DHT.read_retry(sensor, gpio)
```

```
if humidity is not None and temperature is not None:
    print('Temp={0:0.1f}*C  Humidity={1:0.1f}%'.format(temperature, humidity))
else:
    print('Failed to get reading. Try again!')

if temperature > 30:
    GPIO.output(LED, GPIO.HIGH)
else:
    GPIO.output(LED, GPIO.LOW)

GPIO.cleanup()
```

- Usually, a program is executed from top to bottom, line by line. This would be all fine if the program's purpose was to say, make a simple one-off calculation as we did in test-thonny.py

  What if you wanted an LED to light up when the temperature goes beyond a threshold? For that kind of program, you may need to include control flow statements. These include **conditional statements** which are used to determine whether or not a specific condition has been met, this is done through testing whether a condition is true or false.
- Imagine we had a DHT11 temperature and humidity sensor, as well as a 3mm Red LED connected up to the Raspberry Pi's GPIO pins. The pseudo code can be as follows:

```
if temperature > 30:
     turn LED to ON
else:
    turn LED to OFF
```

- The actual code may look like the one on the left. As you can see, it follows the logic of:

```
if condition_is_true:
    do something()

else:
    do something else()
```

- As mentioned in the previous step, it is important to note the **indentation** determines whether a block of code needs to be executed when a specific condition is met. Try modifying the indentation in your code, and note what happens to the program execution

Step 14 Control Flow: for statements

```
for x in range (0, 5):
    print "hello"
```

- The **for** statement can be used when there is a block of code which you want to repeat a number of times. For example, say you wanted to print 'hello' five times

Step 15 Control Flow: while statements

```
import RPi.GPIO as GPIO
import time
LED = 18

GPIO.setmode(GPIO.BCM)
GPIO.setup(LED, GPIO.OUT)

while (True):
    GPIO.output(LED, True)
    time.sleep(1)
    GPIO.output(LED, False)
    time.sleep(1)
```

- The **while statement** can be used to repeat blocks of code over and over again. They are controlled by a conditional expression. For example, say you wanted an LED to blink on and off repeatedly, you could use the following code.

Step 16 Breaking out of the loop

```
import RPi.GPIO as GPIO
import time
LED = 18

GPIO.setmode(GPIO.BCM)
GPIO.setup(LED, GPIO.OUT)

i = 0
while (True):
    print("Value of i is now ", i)
    GPIO.output(LED, True)
    time.sleep(1)
```

```
GPIO.output(LED, False)
time.sleep(1)
i += 1
if i > 10:
    break
```

- To break out of a loop, you can use an if statement with **break** as such.
- After i is larger than 10, the LED will stop flashing altogether.

Step 17 Conclusion

sensing data in R or Python

About Remote Sensing Data

Remote sensing is the science of studying things without touching them. You can use remote sensing systems, to study how Earth systems change over time. For example, scientists use, high powered cameras, not unlike the camera in your smartphone, mounted on airplanes and satellites to capture images of the earth as it changes over time. Other sensors such as lidar (light detection and ranging) are used to collect height data which can be used to measure how trees and forests and even development changes over time.

Active vs Passive Remote Sensing

There are two types of remote sensing sensors: active and passive sensors. Passive sensors measure existing energy, often from the sun. The camera in your smartphone or iPad is an example of a passive remote sensing sensor. To capture a picture, this camera records sunlight, reflected off objects. In contract, an active remote sensing sensor creates its own energy source. Lidar (also sometimes referred to as *active laser scanning*) is an example of an active remote sensing sensor. Lidar systems have a laser on board that emits light that then reflects off of objects, like trees, on the Earth's surface.

Raspberry Pi Interfaces

Raspberry Pi is most popular SBC(Single Board Computer). We can used Raspberry Pi as an IoT device and IoT Gateway. In this article we discuss Raspberry Pi Interfaces. Interfaces used for connecting Sensors and actuators.

What is Raspberry pi ?

The Raspberry Pi is a low cost, **credit-card sized computer** that plugs into a computer monitor or TV, and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python. It's capable of doing everything you'd

expect a desktop computer to do, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games."

If you know about Raspberry Pi more, Visit this : ***Raspberry Pi Tutorials***

If you have a Raspberry Pi and you want to setup for use in Headless mode, Visit This :

Raspberry Pi Headless Mode Setup

Raspberry pi has ***Serial, SPI and I2C*** interfaces for data transfer.

Serial : The Serial interface on Raspberry Pi has receive (Rx) and transmit (Tx) pins for communication with serial peripherals.

SPI : Serial Peripheral Interface (SPI) is a synchronous serial data protocol used for communicating with one or more peripheral devices. in an SPI connection, there are five pins on Raspberry Pi for SPI interface :

- **MISO (Master in slave out)** – Master line for sending data to the peripherals.
- **MOSI (Master out slave in)** – Slave line for sending data to the master.
- **SCK (Serial Clock)** – Clock generated by master to synchronize data transmission
- **CE0 (Chip Enable 0)** – To enable or disable devices
- **CE0 (Chip Enable 1)** – To enable or disable devices

I2C :

The I2C interface pins on Raspberry Pi allow you to connect hardware modules. I2C interface allows synchronous data transfer with just two pins – **SDA (data line)** an **SCL (Clock Line)**.

Operating system and linux commands using Rpi

Sometimes it's hard to keep track of all the Raspberry Pi commands you use, so I created a list of some of the most useful and important ones that will make using Linux on the Raspberry Pi a lot easier. But first a quick note about user privileges…

There are two user "modes" you can work with in Linux. One is a user mode with basic access privileges, and the other is a mode with administrator access privileges (AKA super user, or root). Some tasks can't be performed with basic privileges, so you'll need to enter them with super user privileges to perform them. You'll frequently see the prefix `sudo` before commands, which means you're telling the computer to run the command with super user privileges.

An alternative to entering `sudo` before each command is to access the *root command prompt*, which runs every command with super user privileges. You can access root mode by entering `sudo su` at the command prompt. After entering `sudo su`, you'll see the `root@raspberrypi:/home/pi#` command prompt, and all subsequent commands will have super user privileges.

Most of the commands below have a lot of other useful options that I don't mention. To see a list of all the other available options for a command, enter the command, followed by - -help.

GENERAL COMMANDS

- apt-get update: Synchronizes the list of packages on your system to the list in the repositories. Use it before installing new packages to make sure you are installing the latest version.
- apt-get upgrade: Upgrades all of the software packages you have installed.
- clear: Clears previously run commands and text from the terminal screen.
- date: Prints the current date.
- find / -name example.txt: Searches the whole system for the file example.txt and outputs a list of all directories that contain the file.
- nano example.txt: Opens the file example.txt in the Linux text editor Nano.
- poweroff: To shutdown immediately.
- raspi-config: Opens the configuration settings menu.
- reboot: To reboot immediately.
- shutdown -h now: To shutdown immediately.
- shutdown -h 01:22: To shutdown at 1:22 AM.
- startx: Opens the GUI (Graphical User Interface)

FILE AND DIRECTORY COMMANDS

- cat example.txt: Displays the contents of the file example.txt.
- cd /abc/xyz: Changes the current directory to the /abc/xyz directory.
- cp XXX: Copies the file or directory XXX and pastes it to a specified location; i.e. cp examplefile.txt /home/pi/office/ copies examplefile.txt in the current directory and pastes it into the /home/pi/ directory. If the file is not in the current directory, add the path of the file's location (i.e. cp /home/pi/documents/examplefile.txt /home/pi/office/ copies the file from the documents directory to the office directory).
- ls -l: Lists files in the current directory, along with file size, date modified, and permissions.
- mkdir example_directory: Creates a new directory named example_directory inside the current directory.
- mv XXX: Moves the file or directory named XXX to a specified location. For example, mv examplefile.txt /home/pi/office/ moves examplefile.txt in the current directory to the /home/pi/office directory. If the file is not in the current directory, add the path of the file's location (i.e. cp /home/pi/documents/examplefile.txt /home/pi/office/ moves the file from the documents directory to the office directory). This command can also be used to rename files (but only within the same directory). For example, mv examplefile.txt newfile.txt renames examplefile.txt to newfile.txt, and keeps it in the same directory.
- rm example.txt: Deletes the file example.txt.
- rmdir example_directory: Deletes the directory example_directory (only if it is empty).

- scp user@10.0.0.32:/some/path/file.txt: Copies a file over SSH. Can be used to download a file from a PC to the Raspberry Pi. user@10.0.0.32 is the username and local IP address of the PC, and /some/path/file.txt is the path and file name of the file on the PC.
- touch example.txt: Creates a new, empty file named example.txt in the current directory.

NETWORKING AND INTERNET COMMANDS

- ifconfig: To check the status of the wireless connection you are using (to see if wlan0 has acquired an IP address).
- iwconfig: To check which network the wireless adapter is using.
- iwlist wlan0 scan: Prints a list of the currently available wireless networks.
- iwlist wlan0 scan | grep ESSID: Use grep along with the name of a field to list only the fields you need (for example to just list the ESSIDs).
- nmap: Scans your network and lists connected devices, port number, protocol, state (open or closed) operating system, MAC addresses, and other information.
- ping: Tests connectivity between two devices connected on a network. For example, ping 10.0.0.32 will send a packet to the device at IP 10.0.0.32 and wait for a response. It also works with website addresses.
- wget http://www.website.com/example.txt: Downloads the file example.txt from the weband saves it to the current directory

  .

SYSTEM INFORMATION COMMANDS

- cat /proc/meminfo: Shows details about your memory.
- cat /proc/partitions: Shows the size and number of partitions on your SD card or hard drive.
- cat /proc/version: Shows you which version of the Raspberry Pi you are using.
- df -h: Shows information about the available disk space.
- df /: Shows how much free disk space is available.
- dpkg - -get-selections | grep XXX: Shows all of the installed packages that are related to XXX.
- dpkg - -get-selections: Shows all of your installed packages.
- free: Shows how much free memory is available.
- hostname -I: Shows the IP address of your Raspberry Pi.
- lsusb: Lists USB hardware connected to your Raspberry Pi.
- UP key: Pressing the UP key will print the last command entered into the command prompt. This is a quick way to repeat previous commands or make corrections to commands.
- vcgencmd measure_temp: Shows the temperature of the CPU.
- vcgencmd get_mem arm && vcgencmd get_mem gpu: Shows the memory split between the CPU and GPU.