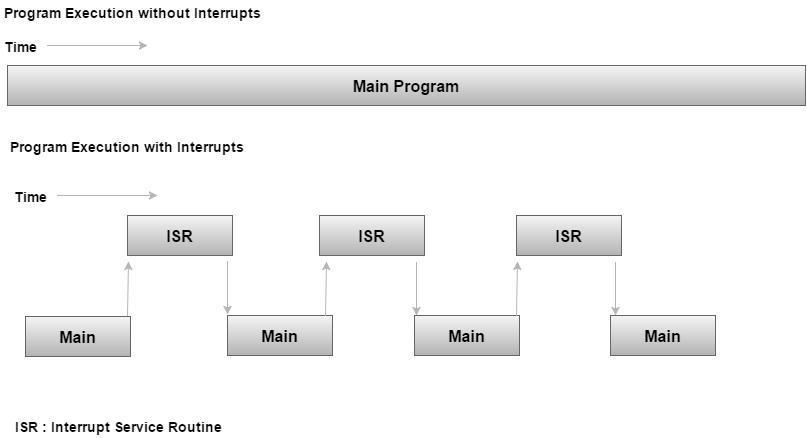
# Interrupt

## What Is An Interrupt? What Is An Interrupt Service Routine?

Imagine that your CPU is busy executing your program instructions. While it’s doing so, there is a peripheral who needs your CPU’s attention to make some work done. For example, a key is pressed. So it sends signal to the CPU which we call INTERRUPT.

Here is the formal definition: An interrupt is a signal to the processor emitted by **hardware** or **software** indicating an event that needs **immediate attention**. Whenever an interrupt occurs, the controller completes the execution of the current instruction and starts the execution of an **Interrupt Service Routine** (ISR) or **Interrupt Handler** which tells the processor or controller what to do when the interrupt occurs.



The interrupts can be either hardware interrupts or software interrupts.

### Hardware Interrupt

A hardware interrupt is an electronic alerting signal sent to the processor from an external device, like a disk controller or an **external peripheral**. For example, when we press a key on the keyboard or move the mouse, they trigger hardware interrupts which cause the processor to read the keystroke or mouse position.

### Software Interrupt

A software interrupt is caused either by an **exceptional condition** or a special instruction in the instruction set which causes an interrupt when it is executed by the processor. For example, if the processor's arithmetic logic unit runs a command to divide a number by zero, to cause a divide-by-zero exception, thus causing the computer to abandon the calculation or display an error message. Software interrupt instructions work similar to subroutine calls.

## Interrupt Service Routine (ISR)

When an interrupt occurs, the microcontroller runs the ISR. For every interrupt, there is a fixed location in memory that holds the address of its ISR. The table of memory locations set aside to hold the addresses of ISRs is called as the **Interrupt Vector Table**.

## Interrupt Vector Table (IVT)

There are six interrupts including RESET in 8051.

|  |  |  |
| --- | --- | --- |
| **Interrupts** | **ROM Location (Hex)** | **Pin** |
| Interrupts | ROM Location (HEX) |  |
| Serial COM (RI and TI) | 0023 |  |
| Timer 1 interrupts(TF1) | 001B |  |
| External HW interrupt 1 (INT1) | 0013 | P3.3 (13) |
| External HW interrupt 0 (INT0) | 0003 | P3.2 (12) |
| Timer 0 (TF0) | 000B |  |
| Reset | 0000 | 9 |

* When the reset pin is activated, the 8051 jumps to the address location 0000. This is power-up reset.
* Two interrupts are set aside for the timers: one for timer 0 and one for timer 1. Memory locations are 000BH and 001BH respectively in the interrupt vector table.
* Two interrupts are set aside for hardware external interrupts. Pin no. 12 and Pin no. 13 in Port 3 are for the external hardware interrupts INT0 and INT1, respectively. Memory locations are 0003H and 0013H respectively in the interrupt vector table.
* Serial communication has a single interrupt that belongs to both receive and transmit. Memory location 0023H belongs to this interrupt.

## Steps to Execute an Interrupt

When an interrupt gets active, the microcontroller goes through the following steps:

* The microcontroller closes the currently executing instruction and saves the address of the next instruction (PC) on the stack.
* It also saves the current status of all the interrupts internally (i.e., not on the stack).
* It jumps to the memory location of the IVT that holds the address of the interrupts service routine.
* The microcontroller gets the address of the ISR from the IVT and jumps to it. It starts to execute the interrupt service subroutine, which is RETI (return from interrupt).
* Upon executing the RETI instruction, the microcontroller returns to the location where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then, it starts executing from that address.

## Edge Triggering vs. Level Triggering

Interrupt modules are of two types – level triggered or edge triggered.

|  |  |
| --- | --- |
| **Level Triggered** | **Edge Triggered** |
| A level-triggered interrupt module always generates an interrupt whenever the level of the interrupt source is asserted. | An edge-triggered interrupt module generates an interrupt only when it detects an asserting edge of the interrupt source. The edge gets detected when the interrupt source level actually changes. It can also be detected by periodic sampling and detecting an asserted level when the previous sample was de-asserted. |
| If the interrupt source is still asserted when the firmware interrupt handler handles the interrupt, the interrupt module will regenerate the interrupt, causing the interrupt handler to be invoked again. | Edge-triggered interrupt modules can be acted immediately, no matter how the interrupt source behaves. |
| Level-triggered interrupts are cumbersome for firmware. | Edge-triggered interrupts keep the firmware's code complexity low, reduce the number of conditions for firmware, and provide more flexibility when interrupts are handled. |

## Enabling and Disabling an Interrupt

Upon Reset, all the interrupts are disabled even if they are activated. The interrupts must be enabled using software in order for the microcontroller to respond to those interrupts.

IE (interrupt enable) register is responsible for enabling and disabling the interrupt. IE is a bit-addressable register.

Interrupt Enable Register

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| EA | - | ET2 | ES | ET1 | EX1 | ET0 | EX0 |

Where:

* **EA** − Global enable/disable.
* **-** − Undefined.
* **ET2** − Enable Timer 2 interrupt.
* **ES** − Enable Serial port interrupt.
* **ET1** − Enable Timer 1 interrupt.
* **EX1** − Enable External 1 interrupt.
* **ET0** − Enable Timer 0 interrupt.
* **EX0** − Enable External 0 interrupt.

To enable an interrupt, we take the following steps:

* Bit D7 of the IE register (EA) must be high to allow the rest of register to take effect.
* If EA = 1, interrupts will be enabled and will be responded to, if their corresponding bits in IE are high. If EA = 0, no interrupts will respond, even if their associated pins in the IE register are high.

## Interrupt Priority in 8051

We can alter the interrupt priority by assigning the higher priority to any one of the interrupts. This is accomplished by programming a register called **IP** (interrupt priority).

The following figure shows the bits of IP register. Upon reset, the IP register contains all 0's. To give a higher priority to any of the interrupts, we make the corresponding bit in the IP register high.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| - | - | - | - | PT1 | PX1 | PT0 | PX0 |

|  |  |  |
| --- | --- | --- |
| - | IP.7 | Not Implemented |
| - | IP.6 | Not Implemented |
| - | IP.5 | Not Implemented |
| - | IP.4 | Not Implemented |
| PT1 | IP.3 | Defines the Timer 1 interrupt priority level |
| PX1 | IP.2 | Defines the External Interrupt 1 priority level |
| PT0 | IP.1 | Defines the Timer 0 interrupt priority level |
| PX0 | IP.0 | Defines the External Interrupt 0 priority level |

## Interrupt inside Interrupt

What happens if the 8051 is executing an ISR that belongs to an interrupt and another one gets active? In such cases, a high-priority interrupt can interrupt a low-priority interrupt. This is known as **interrupt inside interrupt**. In 8051, a low-priority interrupt can be interrupted by a high-priority interrupt, but not by any another low-priority interrupt.

## Triggering an Interrupt by Software

There are times when we need to test an ISR by way of simulation. This can be done with the simple instructions to set the interrupt high and thereby cause the 8051 to jump to the interrupt vector table. For example, set the IE bit as 1 for timer 1. An instruction **SETB TF1** will interrupt the 8051 in whatever it is doing and force it to jump to the interrupt vector table.

## General Thumb Rules When Designing an ISR

* Create a list of events that your system needs to handle.
* Create a map of ISRs that will handle these events.
* Think about the complexity of certain ISRs.
* Think about the speed with which certain devices would expect response or service.
* Keep the ISRs short in terms of speed.
* Avoid loops.
* Avoid complex instructions.
* Use NMIs (Non-Maskable Interrupts) only to handle catastrophic events.

## ISR vs Function Call

**Main point:**

A function (or procedure or sub-routine more generally) must be **explicitly called** and is part of the same context and thread of execution as its caller.

A hardware ISR is not explicitly called but rather **invoked by some external event** (external to the processor core that is - on-chip peripherals may generate interrupts). When an interrupt is called, the context of the current thread is automatically preserved before switching to the ISR. On return, the reverse context switch occurs restoring the state of the processor before the interrupt so that execution continues from the point of interruption.

The mechanism can be complicated by the presence of a multi-threaded operating system or scheduler whereby the ISR itself may cause a thread-context switch so that on return from an ISR a different thread of execution or context is switched in. Such mechanisms are managed by the operating system in this case.

There is another kind of ISR supported on some processors - that of a *software interrupt*. A software interrupt is used like a function call in the sense that it is explicitly invoked by an instruction rather than a single event, but offers an indirection mechanism whereby the caller does not need to know the address of the ISR and indeed that address may change. In that sense it is little different than calling a function through a pointer, but because it is an ISR it runs in the interrupt context, not the caller's context, so may have restrictions and privileges that a normal function does not.

**ISR:**

* Asynchronous event that can occur any time during the execution of the program.
* ISR cannot have arguments passed to it.
* Cannot return values.
* Generally small as they are taking the time of some other processes.

**Function call:**

* Occurs whenever there is a function call.
* Can have arguments.
* Can return values.
* No restriction on the size and duration of execution.

## Interrupt vs Polling

|  |  |  |
| --- | --- | --- |
|  | **Interrupt** | **Polling** |
| Basic | Devices notify CPU when they need attention. | CPU constantly checks device status whether it needs CPU's attention. |
| Mechanism | An interrupt is a hardware mechanism. | Polling is a *protocol*. |
| Servicing | ISR services the device. | CPU services the device. |
| Indication | Interrupt-request line indicates that device needs servicing. | Command-ready bit indicates that device needs servicing. |
| CPU | CPU is disturbed only when a device needs servicing, which saves CPU cycles. | CPU has to wait and check whether a device needs servicing which wastes lots of CPU cycles. |
| Occurrence | An interrupt can occur at any time. | CPU polls the devices at regular interval. |
| Efficiency | Interrupt becomes inefficient when devices keep on interrupting the CPU repeatedly. | Polling becomes inefficient when CPU rarely finds a device ready for service. |
| Example | Let the bell rings then open the door to check who has come. | Constantly keep on opening the door to check whether anybody has come. |

# Re-Entrant Function

## What Is Re-Entrant Function?

Imagine you call a function and it modifies some variables. In another thread or ISR, you then call that same function again, but before the first call is complete. An issue is that the new call might also modify the variables you have set up. When the thread or ISR finishes, your function resumes but now the variable has a different data. Thus, it fails.

This function is NOT re-entrant.

A re-entrant function is a function which can be **called safely in a threaded or interruptible environment**. It usually means it has no reliance on variables or properly uses mutexes to prevent accessing the same data object at the same time.

## When A Function Is Re-Entrant?

A function has to satisfy these conditions to be called as re-entrant:

1. Does not use or return **global and static data**. The interrupt may change certain global values and resuming the course of action of the reentrant function with the new data may give undesired results.
2. Does not **modify its own code**. This is important because the course of action of the function should remain the same throughout the code. But this may be allowed in case the ISR execute in its own unique thread, or uses a local copy of the reentrant function every time it uses different values or before and after the interrupt.
3. Does not **call another non-reentrant function**.
4. Does not rely on locks to singleton resources.

## When A Function Must Be Re-Entrant?

* Functions executed in **interrupt** context
* Functions called from **multi-threads** context

## Notes

Thread safety and reentrant functions: Reentrancy is distinct from, but closely related to, thread-safety. **A function can be thread-safe and still not reentrant**. For example, a function could be wrapped all around with a mutex (which avoids problems in multithreading environments), but if that function is used in an ISR, it could starve waiting for the first execution to release the mutex. **The key for avoiding confusion is that re-entrant refers to only one thread executing**. It is a concept from the time when no multitasking operating systems existed.

Q&As about re-entrant function: <https://stackoverflow.com/a/2799288>

# Synchronous vs Asynchronous Programing

## What Do "Synchronous" and "Asynchronous" Really Mean?

When you execute something *synchronously*, you **wait for it to finish before moving on to another task**.

When you execute something *asynchronously*, you can **move on to another task before it finishes**. This translates into executing a task on another "thread." A thread is a series of commands (a block of code) that exists as a *unit of work*.

Note: Technically, the concept of synchronous/asynchronous has nothing to do with threads. Although, in general, it is unusual to find asynchronous tasks running on the same thread, but it is common (and possible) to find two or more tasks executing synchronously on separate threads.

## When To Use Asynchronous?

**When Asyn is good?**

Rule: *Asynchrony is essential for activities that are potentially blocking*, such as:

1. Access to a web resource is sometimes slow or delayed. If such an activity is blocked within a synchronous process, the entire application must wait. In an asynchronous process, the application can continue with other work that doesn’t depend on the web resource until the potentially blocking task finishes.
2. Receiving/sending requests in a busy network server (like a web server) in a client-server environment. Each task represents one client request with I/O in the form of receiving the request and sending the reply. A network server implementation is a prime candidate for the asynchronous model, which is why Twisted and Node.js, among other asynchronous server libraries, have grown so much in popularity in recent years.
3. When an application asks for data (from a database, webservice, etc.), it has to wait until that service replies. All of the fancy gestures that are usually present in apps happen because of async calls to the services at the same time as entertaining the user with some logo or messaging.

**When Asyn is abused?**

1. Make every loop asyn to avoid blocking the event loop. Although asyn loops are necessary when there is a large number of iterations or when the operations within the loop are complex. But for simple tasks like iterating through a small array, there is no reason to overcomplicate things by using a complex recursive function. A simple synchronous for/while loop works just fine, and will also be faster and more readable.

# ARM Compiler

## The '\_\_weak' Keyword

## The '\_\_root' Keyword

# Timer

# Memory

Generally, memory/storage is classified into two categories:

* Volatile Memory: This loses its data when power is off.
* Non-Volatile Memory: This is a permanent storage and does not lose any data when power is off.

From the moment you turn your computer on until you shut it down, your CPU is constantly using memory. Let's take a look at a typical scenario:

* You turn the computer on.
* The computer loads data from [**ROM**](#_gjdgxs)and performs a power-on self-test (POST) to make sure all the major components are functioning properly. As part of this test, the memory controller checks all of the memory addresses with a quick read/writeoperation to ensure that there are no errors in the memory chips.
* The computer loads the Basic Input/Output System ([BIOS](https://computer.howstuffworks.com/bios.htm)) from ROM. The BIOS provides the most basic information about storage devices, boot sequence, security, Plug and Play(auto device recognition) capability and a few other items.
* The computer loads the operating system (OS) from the hard drive into the system's RAM. Generally, the critical parts of the OS are maintained in RAM as long as the computer is on. This allows the CPU to have immediate access to the operating system, which enhances the performance and functionality of the overall system.
* When you open an application, it is loaded into [**RAM**](#_gjdgxs). To conserve RAM usage, many applications load only the essential parts of the program initially and then load other pieces as needed.
* After an application is loaded, any files that are opened for use in that application are loaded into RAM.
* When you save a file and close the application, the file is written to the specified storage device, and then it and the application are purged from RAM.

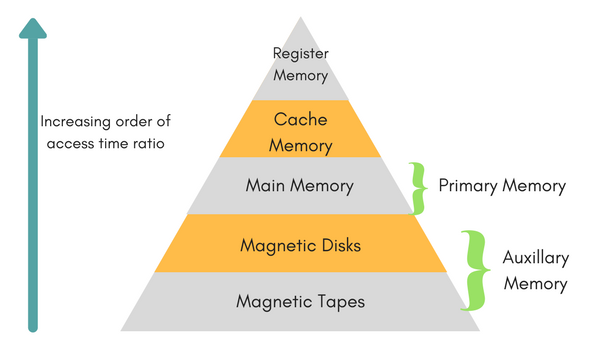
In the list above, every time something is loaded or opened, it is placed into RAM. This simply means that it has been put in the computer's temporary storage area so that the CPU can access that information more easily. The CPU requests the data it needs from RAM, processes it and writes new data back to RAM in a continuous cycle. In most computers, this shuffling of data between the CPU and RAM happens millions of times every second. When an application is closed, it and any accompanying files are usually purged (deleted) from RAM to make room for new data. If the changed files are not saved to a permanent storage device before being purged, they are lost.

## Memory Access Methods

Each memory is a collection of numerous memory locations. To access data from any memory, it must be first located and then the data is read from the memory location. Following are methods to access data from memory locations:

* **Random Access** (RAM): Each memory location has a unique address. Any memory location can be reached in the same amount of time in any order.
* **Sequential Access** (SAM): Memory locations are accessed in a sequence. SAM works very well for memory buffers, where the data is normally stored in the order in which it will be used (a good example is the texture buffer memory on a video card).
* **Direct Access**: Data is stored in tracks, with each track having a separate read/write head.

## Memory Hierarchy



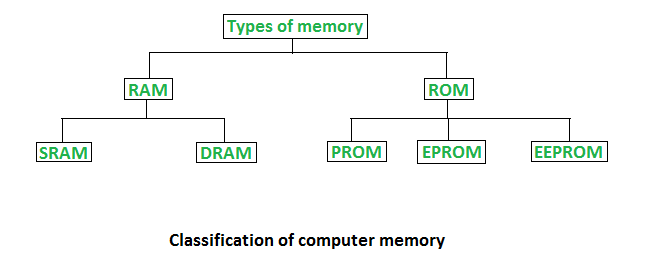
### Auxiliary Memory

Also called **secondary memory**, these devices provide permanent, large-capacity backup storage. For example: magnetic disks and tapes, magnetic bubble memory and optical disks.

It is not directly accessible to the CPU, and is accessed using the Input/Output channels.

### Main Memory

It is the central storage unit of a computer system. It’s a large and fast.

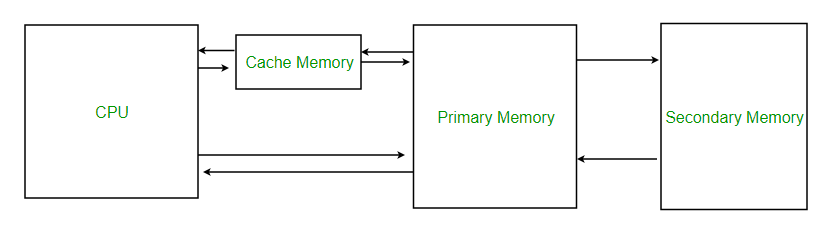


Main memory is made up of RAM and ROM:

|  |  |
| --- | --- |
| **RAM** (Random Access Memory)   * Volatile (temporary storage). * Used to store data during execution of a program. * Fast reading/writing data. | **DRAM** (Dynamic RAM)   * Made of capacitors and transistors (A transistor and a capacitor are paired to create a memory cell). * Must be recharged every 10-100 ms. * Slower and cheaper than SRAM. |
| **SRAM** (Static RAM)   * Made of circuits similar to D flip flop (takes 4 or 6 transistors). * Do not require recharging after a while. * Faster and costlier than DRAM. |
| **ROM** (Read Only Memory) or ***firmware***   * Non-volatile (permanent storage). * Used to store the bootstrap loader program, or to load and start the OS when the computer is turned on. * Low reading/writing data. | **PROM** (Programmable ROM)   * Can be programmed by user. * Once programmed, the data and instructions in it cannot be changed. |
| **EPROM** (Erasable PROM)   * Can be reprogrammed. * To erase data from it, expose it to ultra violet light. * To reprogram it, erase all the previous data. |
| **EEPROM** (Electrically Erasable PROM)   * The data can be erased by applying electric field, no need of ultra violet light. We can erase only portions of the chip. * Example: [flash memory](#_1fob9te). |

### Cache Memory

It is a high-speed memory, used to speed up and synchronize with high-speed CPU. It acts as a buffer between RAM and the CPU, holding frequently requested data and instructions so that they are immediately available to the CPU when needed.

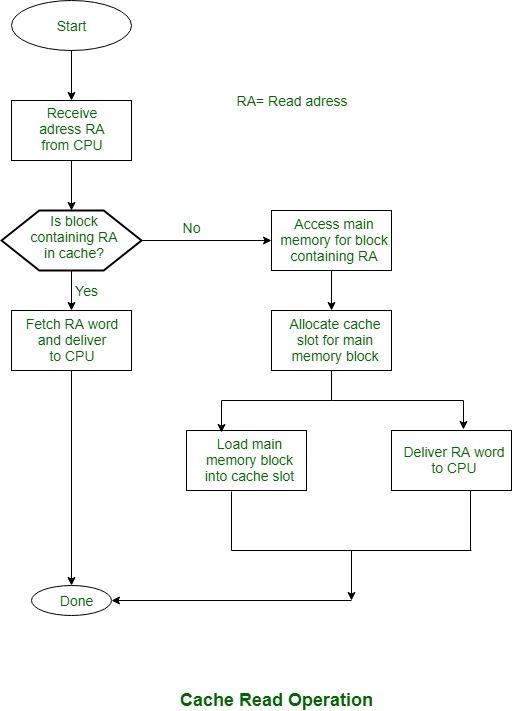


Cache memory is used to reduce the average time to access data from the main memory. There are various different independent caches in a CPU, which store instructions and data.

Whenever the program is ready to be executed, it is fetched from main memory and then copied to the cache memory. But, if its copy is already present in the cache memory then the program is directly executed.

When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache.

* If the processor finds that the memory location is in the cache, a *cache hit* has occurred and data is read from cache.
* If the processor does not find the memory location in the cache, a *cache miss* has occurred. For a cache miss, the cache allocates a new entry and copies in data from main memory, then the request is fulfilled from the contents of the cache.



### Register Memory

It is a type of memory in which data is stored and accepted that are immediately stored in CPU. Most commonly used register is accumulator, program counter, address register etc.

## Memory Characteristics

* **Capacity:** It is the global volume of information the memory can store. As we move from top to bottom in the hierarchy, the capacity increases.
* **Access Time:** It is the time interval between the read/write request and the availability of the data. As we move from top to bottom in the hierarchy, the access time increases.
* **Cost per bit:** As we move from bottom to top in the hierarchy, the cost per bit increases.

## Flash Memory

Flash memory is a type of **EEPROM**. It’s a portable storage device.

A few examples of flash memory are:

* Computer's [BIOS](https://computer.howstuffworks.com/bios.htm) chip
* SD memory cards
* Flash USB memory sticks
* Solid-state disks (SSDs replace the old disk technology of HDD)

## Virtual Memory

Must watch carefully first: <https://www.youtube.com/watch?v=qcBIvnQt0Bw&list=PLiwt1iVUib9s2Uo5BeYmwkDFUh70fJPxX&index=1>

1GB, or even 16GB, of RAM is not enough to run all of the programs that most users expect to run at once.

For example, if you load the OS while a playing heavy video game, 4GB of RAM maybe not enough to hold it all. If there were no **virtual memory**, then once you filled up the available RAM your computer would have to say, "Sorry, you cannot load any more applications. Please close another application to load a new one." With virtual memory, what the computer can do is looking at RAM for areas that have not been used recently and copy them onto the hard disk. This frees up space in RAM to load the new application.

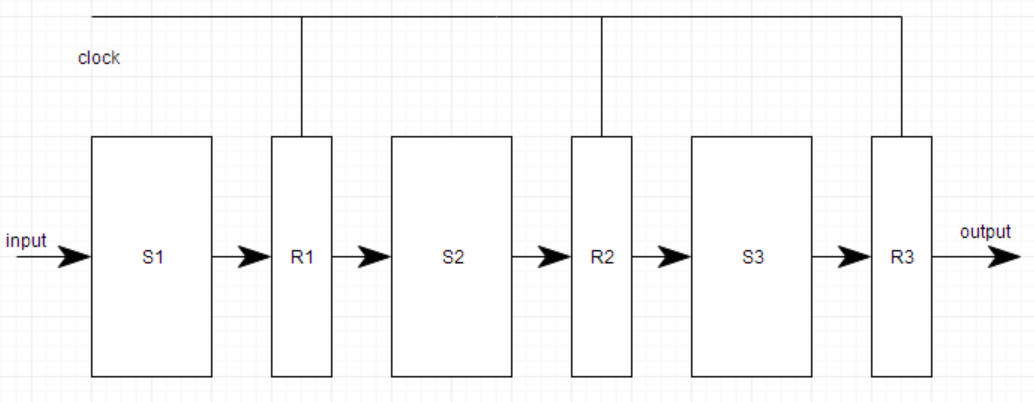
Because this copying happens automatically, you don't even know it’s happening. The read/write speed of a hard drive is much slower than RAM, and the technology of a hard drive is not geared toward accessing small pieces of data at a time. If your system has to rely too heavily on virtual memory, you will notice a significant performance drop.

# Pipeline

## What is Pipelining?

Pipelining is the process of overlapping operations by moving data or instructions into a conceptual pipe with all stages of the pipe performing simultaneously. It allows storing and executing data or instructions orderly, where the output of one element is the input of the next one. For example, while an instruction is being executed, the next one is being decoded. Data or instructions enter from one end and exit from another end.

In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment.



Pipeline system is like a modern assembly line in factories. For example, in a car manufacturing industry, huge assembly lines are setup and at each point, there are robotic arms to perform a certain task, and then the car moves on ahead to the next arm.

## Advantages and Disadvantages of Pipelining

Advantages:

* The cycle time of the processor is reduced.
* It increases the throughput of the system
* It makes the system reliable.

Disadvantages:

* The design of pipelined processor is complex and costly to manufacture.
* The instruction latency is more.

# Bootloader

## BIOS vs UEFI

<https://www.tecmint.com/linux-file-system-explained/>

# Configuration Time

## What Are Pre-Compile Time, Link Time and Post-Build Time?

**Pre-compile time**: All code and data are determined **before the compiler is run**. The code and data are then placed into the FLASH at the same time.

**Link time**: Code is compiled with **data added later by the linker**. The code and data are then placed into the FLASH at the same time.

**Post-build time**: Code is compiled with pointers into blank areas of FLASH. The compiled code is then placed into FLASH. The data is placed into FLASH at a different stage in the manufacturing process than the code. The data to be used at run-time can either be fixed (this is called post-build loadable) or selected at run-time (this is called post-build selectable). Note that these two classes are mutually exclusive. You cannot have a mixture of post build loadable and post build selectable in a particular variant in a particular module.

## When To Use Pre-Compile Time, Link Time and Post-Build Time?

**Pre-compile time is the most efficient possible implementation**

In this time, all parameters are determined before compilation. There is no possibility to change any parameters once compilation has taken place (other than by recompiling). Therefore, this time is the most efficient, but also the least flexible.

Pre-compile time is typically implemented with the pre-processor to enable or disable code generation for specific areas of the module's source. However, if you change your mind, you'll need to recompile.

For example:

void f(void)

{

    unsigned int count;

#ifdef DEVELOPMENT\_ERROR\_CHECKING

    if(!module\_initialised) {

        /\* do something with the DET \*/

        return;

    }

#endif

    /\* set CAN Baud rate \*/

    \*BAUD\_RATE\_REGISTER = BAUD\_RATE;

    /\* initialise the buffers in RAM from data in ROM\*/

    for(count=0; count<NBUFFS; count++)

    {

#ifdef CACHING\_IS\_BETTER

        /\* this code caches the pointer \*/

        struct buffer \*tmp = &my\_configuration[count];

        xx\_memcopy(tmp->size, tmp->init\_data, tmp->ram\_data);

#else

        /\* Caching is not better \*/

        xx\_memcopy(my\_configuration[count]->size,

        my\_configuration[count]->init\_data,

        my\_configuration[count]->ram\_data);

#endif

    }

}

**Link time has great flexibility**

In this time, flexibility is added by choosing some parameters that will be determined at link time without the need to recompile the module. Therefore, typically, a mixture of pre-compile and link time parameters will exist.

For example:

// In link\_time.h

extern const int fred;

// In user.c

#include "link\_time.h"

int get\_fred(void) {

    return fred;

}

// In init.c

#include "link\_time.h"

const int fred = 0x1234;

The file user.c uses fred but does not know in advance what its value is. The file is compiled by the supplier of the module and, therefore, cannot by changed by the user of the module.

The file link\_time.h is also supplied as part of the module, because it is necessary to compile a file in the user's domain, and cannot be changed by the user.

The file init.c is generated and compiled by the user. When user.o and init.o are linked the reference by user.c to fred is resolved so that the function get\_fred() can work correctly.

From a tools point-of-view this works as follows:

1. Module supplier uses the code generator to generate link\_time.h
2. Module supplier compiles user.c
3. Module supplier provides user.o and link\_time.h to user.

There can be a significant gap between steps 3 and 4 because they may take place at different times and in different divisions or companies.

1. User uses the code generator to generate init.c
2. User compiles init.c
3. User links user.o and init.o

More details: <https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_TR_CImplementationRules.pdf>

# UART - I2C - SPI - Serial Communications

<https://www.youtube.com/watch?v=IyGwvGzrqp8&ab_channel=Electronoobs>

# RPMsg

## What Is RPMsg?

RPMsg, standing for Remote Processor Messaging, is a protocol for transferring messages between processor cores. It is present in the Linux kernel and also available as a stand-alone component for microcontroller-based systems.

Most of the RPMsg implementations are developed as open source.

## How Does RPMsg Work?

<https://github.com/OpenAMP/open-amp/wiki/RPMsg-Messaging-Protocol>