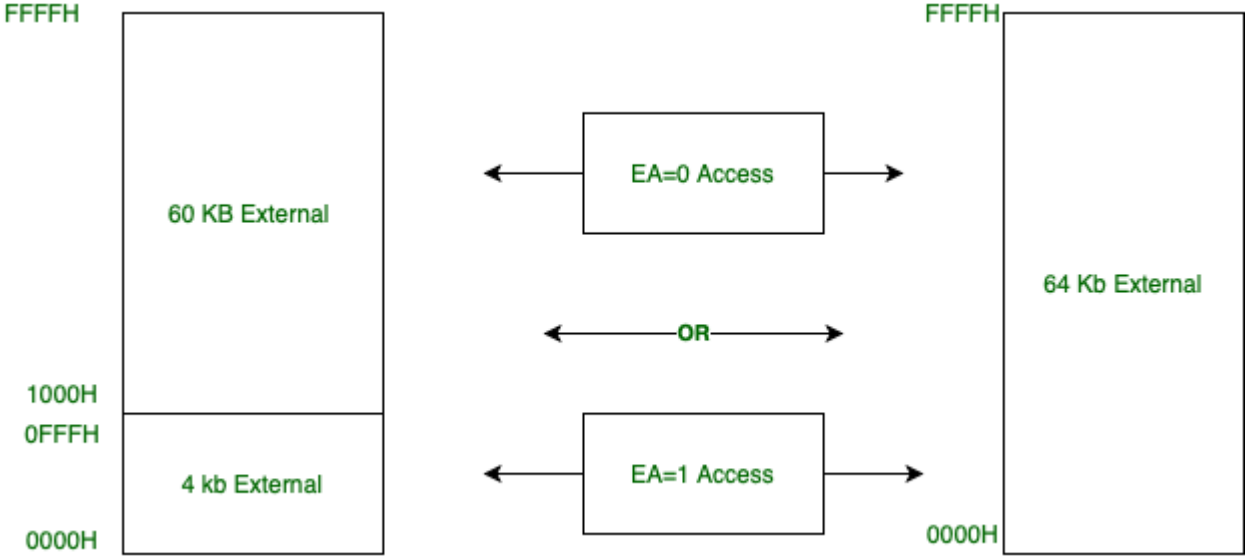
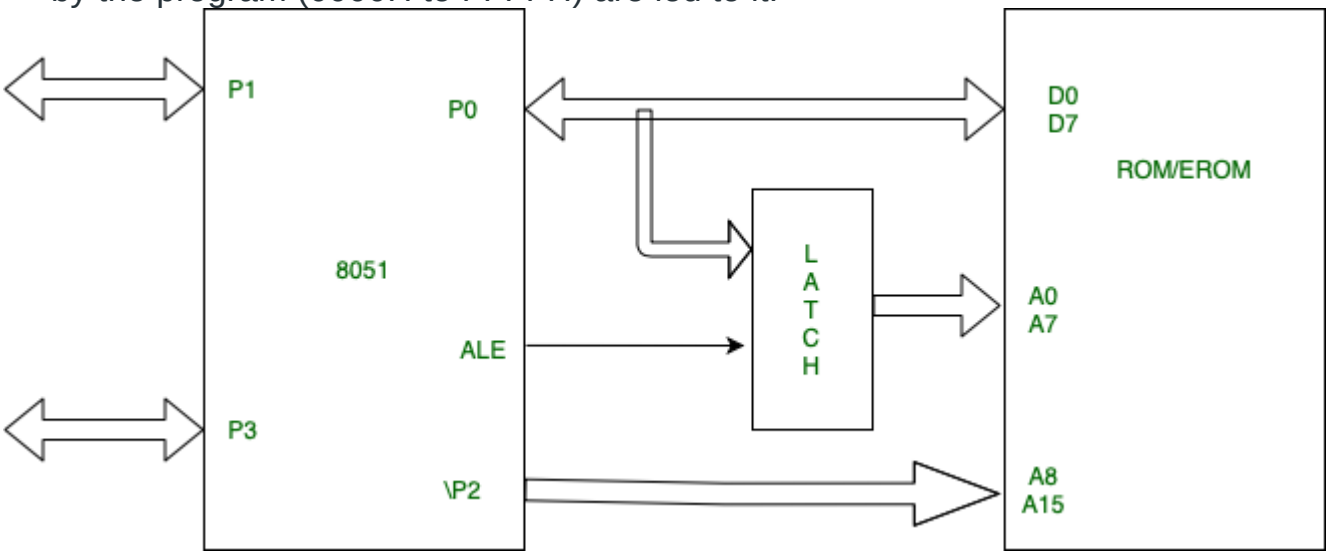
Memory and I/O Device Interfacing

The 8051 has internal data and code memory. In such a position. For certain applications, this memory capacity will not be adequate. To expand the memory space of the 8051 micro-controller, we must bind external ROM/EPROM and RAM. We also understand that ROM serves as program memory and RAM serves as data memory. Let's take a look at how the 8051 accesses these memories.

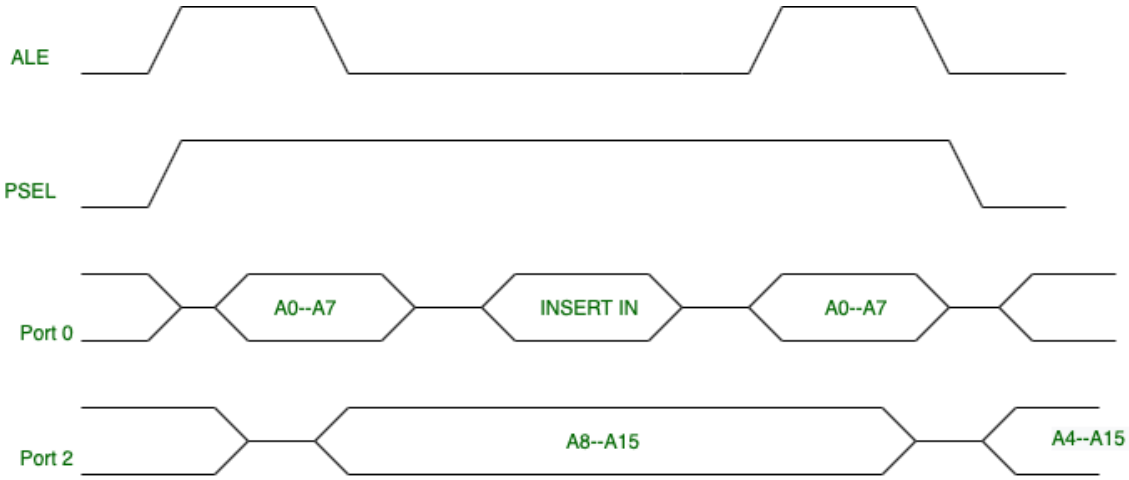
External Program Memory:



- The program fetches to addresses 0000H through OFFFH are directed to the internal ROM in the 8051 when the EA pin is attached to Vec, and program fetches to addresses 1000H through FFFFH are directed to the external ROM/EPROM. When the EA pin is grounded, all addresses fetched by the program (0000H to FFFFH) are led to it.



- ROM/EPROM that is external to the device. As seen in Fig. 1, the PSEN signal is used to trigger output e external ROM/EPROM.
- Port 0 is used as a multiplexed address/bus, as seen in Fig2. In the initial T-cycle, it provides a lower order 8-bit address, and later it is used as a data bus. The external latch and the ALE signal provided by the 8051 are used to latch the 8-bit address.

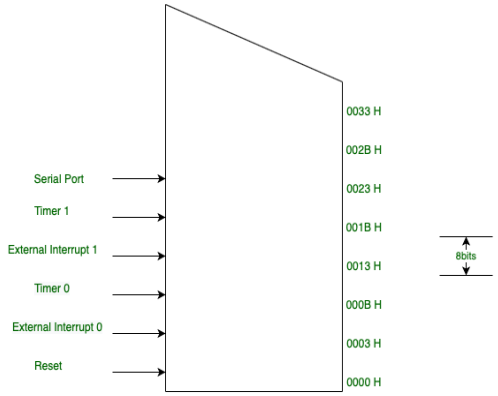


timing waveform for external data memory write cycle

Timing Waveform for external program memory

- Remote ROM/EPROM (Read Only Memory/Electronic Programmable ROM/Electronic Programmable ROM/Electronic Program The PSEN signal is used to activate output e external ROM/EPROM, as seen in Fig. 1.
- As seen in Fig2, port 0 is used as a multiplexed address/bus. It supplies a lower-order 8-bit address in the first T-cycle and later serves as a data bus. The 8-bit address is latched using the external latch and the ALE signal given by the 8051

.



- 1External Interrupt 0 is 0003H, Timer 0 is 000BH, External Interrupt 1 is 0013H, Timer 1 is 001BH, and so on. If an interrupt is to be used, the

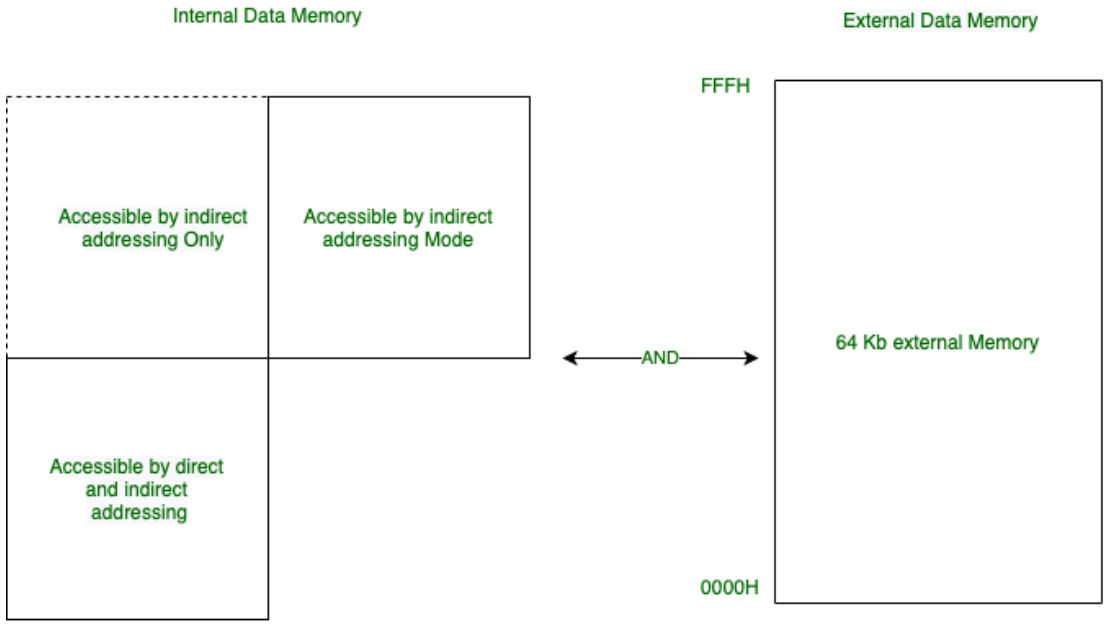
operation routine for it must be in the same place as the interrupt. If the interrupt isn't used, the service location may be used as general-purpose program memory.

Instructions to Access External ROM / Program Memory : This table is explaining the instructions to access external ROM/program memory.

Mnemonic	Operation
MOVC A, @ A+DPTR	Copy the contents of the external ROM address formed by adding A and the DPTR, to A
MOVC A, @ A + PC	This operation will do copy This operation contents of the external ROM address formed by adding A and the PC, to A.

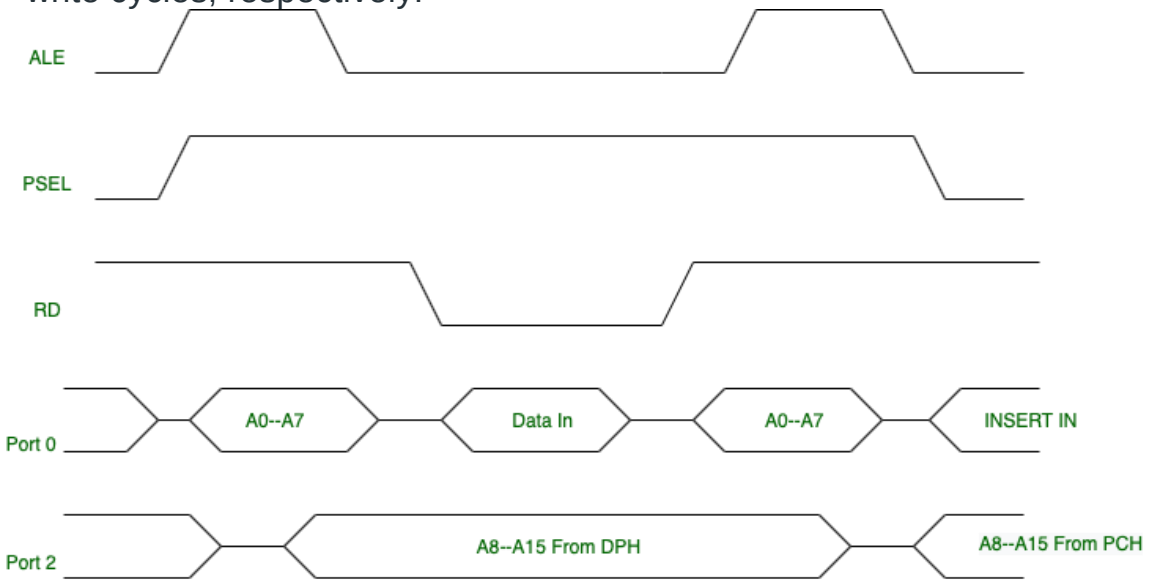
External Memory Interfacing:

- Up to 64 k-bytes of additional data memory can be addressed by the 8051. The external data memory is accessed using the "MOVX" instruction.
- The 8051's internal data memory is split into three sections: Lower 128 bytes, Upper 128 bytes, and SFRs. While they are physically distinct bodies, the upper addresses and SFRs share the same block of address space, 80H by FFH.
- The upper address space is only accessible via indirect addressing, and SFRs are only accessible via direct addressing, as seen in Higher address space, on the other hand, can be reached using either direct or indirect addressing.



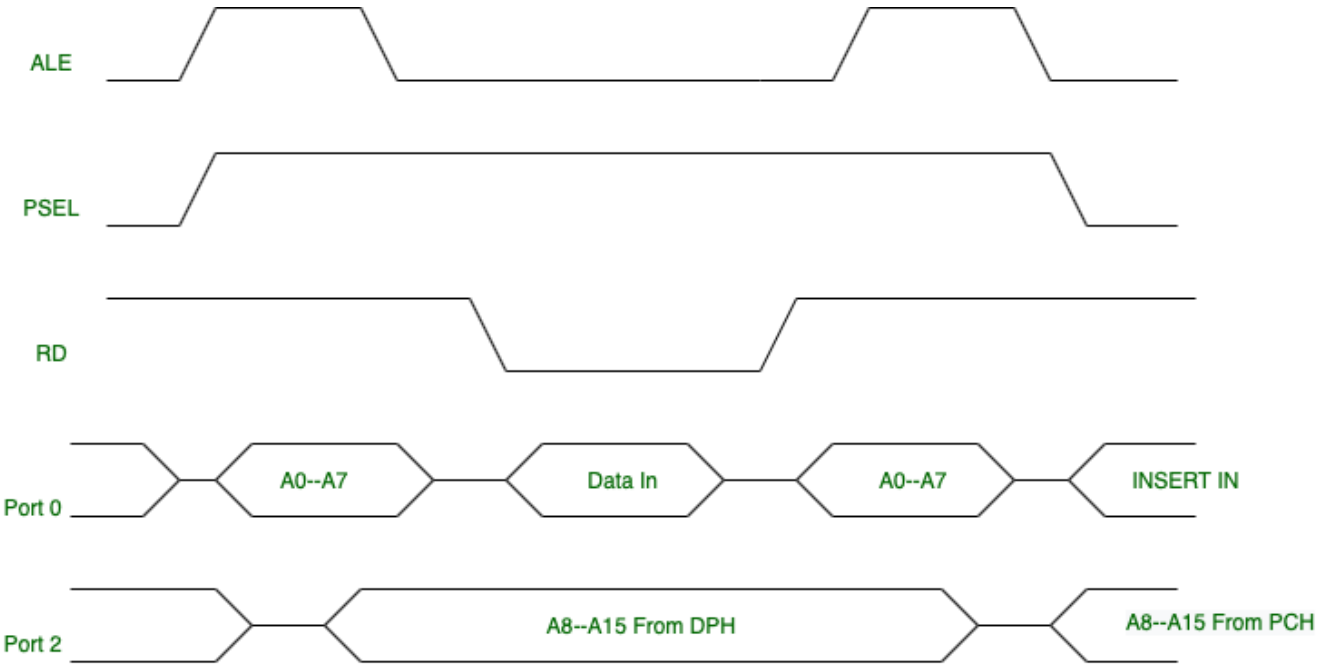
A map of the 8051 data memory Fig. 5

- Fig. 7 a and b show the timing waveform for external data memory read and write cycles, respectively.



timing waveform for external data memory read cycle

7(a)



timing waveform for external data memory write cycle

7(b)

Instructions to Access External Data Memory :

The Table explains the instruction to access external data memory.

Mnemonic	Operation
MOV X A, @Rp	In this operation, it will copy the contents of the external address in Rp to A.
MOV X A. @DPTR	Copy the contents of the external address in DPTR to A.
MOV X @Rp. A	Copy data from A to the external address in Rp
MOV X DPTR, A	Copy data from A to the external address in DPTR.

Important Points to Remember in Accessing External Memory:

- In the Case of accessing external memory, all external data moves with external RAM or ROM involve the A register.
- While accessing external memory, R can address 256 bytes and DPTR can address 64 k-bytes
- MOV X instruction is used to access external RAM or 1/O addresses.

Note –

It must be noted that while the Program counter (PC) will be used then to access external ROM, it will be incremented by 1 (to point to the next instruction) before it is added to A to form the physical address of external ROM.

Embedded C is basically an extension to the Standard C Programming Language with additional features like Addressing I/O, multiple memory addressing and fixed-point arithmetic, etc.

C Programming Language is generally used for developing desktop applications, whereas Embedded C is used in the development of Microcontroller based applications.

Basics of Embedded C Program

Now that we have seen a little bit about Embedded Systems and Programming Languages, we will dive in to the basics of Embedded C Program. We will start with two of the basic features of the Embedded C Program: Keywords and Datatypes.

Keywords in Embedded C

A Keyword is a special word with a special meaning to the compiler (a C Compiler for example, is a software that is used to convert program written in C to Machine Code). For example, if we take the Keil's Cx51 Compiler (a popular C Compiler for 8051 based Microcontrollers) the following are some of the keywords:

bit

sbit

sfr

small

large

The following table lists out all the keywords associated with the Cx51 C Compiler.

_at_	alien	bdata
bit	code	compact
data	far	idata
interrupt	large	pdata
priority	reentrant	sbit
sfr	sfr16	small
task	using	xdata

Data Types in Embedded C

Data Types in Embedded C

Data Types in C Programming Language (or any programming language for that matter) help us declaring variables in the program. There are many data types in C Programming Language like signed int, unsigned int, signed char, unsigned char, float, double, etc. In addition to these there few more data types in Embedded C.

The following are the extra data types in Embedded C associated with the Keil's Cx51 Compiler.

bit

sbit

sfr

sfr16

The following table shows some of the data types in Cx51 Compiler along with their ranges.

Data Type	Bits (Bytes)	Range
bit	1	0 or 1 (bit addressable part of RAM)
signed int	16 (2)	-32768 to +32767
unsigned int	16 (2)	0 to 65535
signed char	8 (1)	-128 to +127
unsigned	8 (1)	0 to 255
float	32 (4)	±1.175494E-38 to ±3.402823E+38
double	32 (4)	±1.175494E-38 to ±3.402823E+38
sbit	1	0 or 1 (bit addressable part of RAM)
sfr	8 (1)	RAM Addresses (80h to FFh)
sfr16	16 (2)	0 to 65535

Basic Structure of an Embedded C Program (Template for Embedded C Program)

The next thing to understand in the Basics of Embedded C Program is the basic structure or Template of Embedded C Program. This will help us in understanding how an Embedded C Program is written.

The following part shows the basic structure of an Embedded C Program.

Multiline Comments . . . . . Denoted using /*……*/

Single Line Comments . . . . . Denoted using //

Preprocessor Directives . . . . . #include<…> or #define

Global Variables . . . . . Accessible anywhere in the program

Function Declarations . . . . . Declaring Function

Main Function . . . . . Main Function, execution begins here

{

Local Variables . . . . . Variables confined to main function

Function Calls . . . . . Calling other Functions

Infinite Loop . . . . . Like while(1) or for(;;)

Statements . . . . .

….

….

}

Function Definitions . . . . . Defining the Functions

{

Local Variables . . . . . Local Variables confined to this Function

Statements . . . . .

….

….

}

Before seeing an example with respect to 8051 Microcontroller, we will first see the different components in the above structure.

Different Components of an Embedded C Program

Comments: Comments are readable text that are written to help us (the reader) understand the code easily. They are ignored by the compiler and do not take up any memory in the final code (after compilation).

There are two ways you can write comments: one is the single line comments denoted by // and the other is multiline comments denoted by /*….*/.

Preprocessor Directive: A Preprocessor Directive in Embedded C is an indication to the compiler that it must look in to this file for symbols that are not defined in the program.

In C Programming Language (also in Embedded C), Preprocessor Directives are usually represented using # symbol like #include… or #define….

In Embedded C Programming, we usually use the preprocessor directive to indicate a header file specific to the microcontroller, which contains all the SFRs and the bits in those SFRs.

In case of 8051, Keil Compiler has the file “reg51.h”, which must be written at the beginning of every Embedded C Program.

Global Variables: Global Variables, as the name suggests, are Global to the program i.e., they can be accessed anywhere in the program.

Local Variables: Local Variables, in contrast to Global Variables, are confined to their respective function.

Main Function: Every C or Embedded C Program has one main function, from where the execution of the program begins.

Related Post: “EMBEDDED SYSTEM PROJECT IDEAS“.

Basic Embedded C Program

Till now, we have seen a few Basics of Embedded C Program like difference between C and Embedded C, basic structure or template of an Embedded C Program and different components of the Embedded C Program.

Continuing further, we will explore in to basics of Embedded C Program with the help of an example. In this example, we will use an 8051 Microcontroller to blink LEDs connected to PORT1 of the microcontroller.

Example of Embedded C Program

The following image shows the circuit diagram for the example circuit. It contains an 8051 based Microcontroller (AT89S52) along with its basic components (like RESET Circuit, Oscillator Circuit, etc.) and components for blinking LEDs (LEDs and Resistors).

Basics of Embedded C Program Image 2

In order to write the Embedded C Program for the above circuit, we will use the Keil C Compiler. This compiler is a part of the Keil µVision IDE. The program is shown below.

Sample Code

```
#include<reg51.h> // Preprocessor Directive

void delay (int); // Delay Function Declaration


void main(void) // Main Function

{

P1 = 0x00;

/* Making PORT1 pins LOW. All the LEDs are OFF.

 * (P1 is PORT1, as defined in reg51.h) */


while(1) // infinite loop

{

P1 = 0xFF; // Making PORT1 Pins HIGH i.e. LEDs are ON.

delay(1000);

/* Calling Delay function with Function parameter as 1000.

 * This will cause a delay of 1000mS i.e. 1 second */


P1 = 0x00; // Making PORT1 Pins LOW i.e. LEDs are OFF.

delay(1000);

}
```

```
}

void delay (int d) // Delay Function Definition
{
unsigned int i=0; // Local Variable. Accessible only in this function.

/* This following step is responsible for causing delay of 1000mS
 * (or as per the value entered while calling the delay function) */

for(; d>0; d–)
{
for(i=250; i>0; i – -);
for(i=248; i>0; i – -);
}
}
```

What is a Real-Time Operating System (RTOS)?

Real-time operating system (RTOS) is an operating system intended to serve real time application that process data as it comes in, mostly without buffer delay. The full form of RTOS is Real time operating system.

In a RTOS, Processing time requirement are calculated in tenths of seconds increments of time. It is time-bound system that can be defined as fixed time constraints. In this type of system, processing must be done inside the specified constraints. Otherwise, the system will fail.

Why use an RTOS?

Here are important reasons for using RTOS:

It offers priority-based scheduling, which allows you to separate analytical processing from non-critical processing.

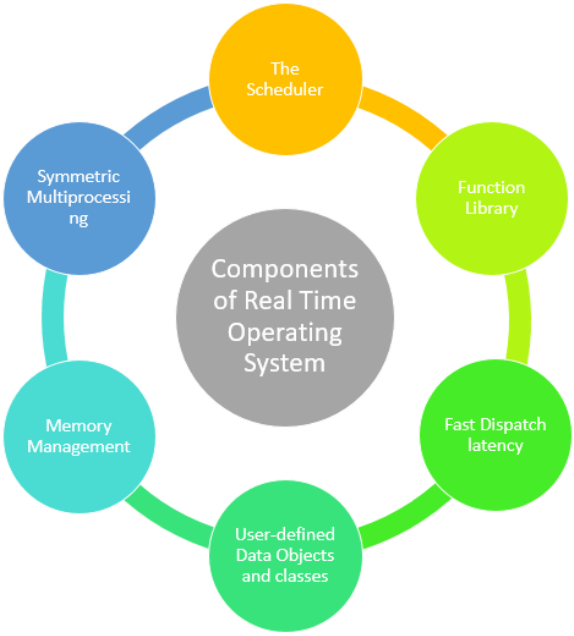
The Real time OS provides API functions that allow cleaner and smaller application code.

Abstracting timing dependencies and the task-based design results in fewer interdependencies between modules.

RTOS offers modular task-based development, which allows modular task-based testing.

The task-based API encourages modular development as a task, will typically have a clearly defined role. It allows designers/teams to work independently on their parts of the project.

An RTOS is event-driven with no time wastage on processing time for the event which is not occur



Components of RTOS

The Scheduler: This component of RTOS tells that in which order, the tasks can be executed which is generally based on the priority.

Symmetric Multiprocessing (SMP): It is a number of multiple different tasks that can be handled by the RTOS so that parallel processing can be done.

Function Library: It is an important element of RTOS that acts as an interface that helps you to connect kernel and application code. This application allows you to send the requests to the Kernel using a function library so that the application can give the desired results.

Memory Management: this element is needed in the system to allocate memory to every program, which is the most important element of the RTOS.

Fast dispatch latency: It is an interval between the termination of the task that can be identified by the OS and the actual time taken by the thread, which is in the ready queue, that has started processing.

User-defined data objects and classes: RTOS system makes use of programming languages like C or C++, which should be organized according to their operation.

Types of RTOS

Three types of RTOS systems are:

Hard Real Time :

In Hard RTOS, the deadline is handled very strictly which means that given task must start executing on specified scheduled time, and must be completed within the assigned time duration.

Example: Medical critical care system, Aircraft systems, etc.

Firm Real time:

These type of RTOS also need to follow the deadlines. However, missing a deadline may not have big impact but could cause undesired affects, like a huge reduction in quality of a product.

Example: Various types of Multimedia applications.

Soft Real Time:

Soft Real time RTOS, accepts some delays by the Operating system. In this type of RTOS, there is a deadline assigned for a specific job, but a delay for a small amount of time is acceptable. So, deadlines are handled softly by this type of RTOS.

Features of RTOS

Here are important features of RTOS:

Occupy very less memory

Consume fewer resources

Response times are highly predictable

Unpredictable environment

The Kernel saves the state of the interrupted task ad then determines which task it should run next.

The Kernel restores the state of the task and passes control of the CPU for that task.

Factors for selecting an RTOS

Here, are essential factors that you need to consider for selecting RTOS:

Performance: Performance is the most important factor required to be considered while selecting for a RTOS.

Middleware: if there is no middleware support in Real time operating system, then the issue of time-taken integration of processes occurs.

Error-free: RTOS systems are error-free. Therefore, there is no chance of getting an error while performing the task.

Embedded system usage: Programs of RTOS are of small size. So we widely use RTOS for embedded systems.

Maximum Consumption: we can achieve maximum Consumption with the help of RTOS.

Task shifting: Shifting time of the tasks is very less.

Unique features: A good RTS should be capable, and it has some extra features like how it operates to execute a command, efficient protection of the memory of the system, etc.

24/7 performance: RTOS is ideal for those applications which require to run 24/7.

Difference between in GPOS and RTOS

Here are important differences between GPOS and RTOS:

General-Purpose Operating System (GPOS)	Real-Time Operating System (RTOS)
It used for desktop PC and laptop.	It is only applied to the embedded application.
Process-based Scheduling.	Time-based scheduling used like round-robin scheduling.
Interrupt latency is not considered as important as in RTOS	Interrupt lag is minimal, which is measured in a few microseconds.
No priority inversion mechanism is present in the system.	The priority inversion mechanism is current. So it can not modify by the system.

Kernel's operation may or may not be preempted.	Kernel's operation can be preempted.
Priority inversion remain unnoticed	No predictability guarantees

Applications of Real Time Operating System

Real-time systems are used in:

Airlines reservation system.

Air traffic control system.

Systems that provide immediate updating.

Used in any system that provides up to date and minute information on stock prices.

Defense application systems like RADAR.

Networked Multimedia Systems

Command Control Systems

Internet Telephony

Anti-lock Brake Systems

Heart Pacemaker

Multitasking

[RTOS Fundamentals]

The **kernel** is the core component within an operating system. Operating systems such as Linux employ kernels that allow users access to the computer seemingly simultaneously. Multiple users can execute multiple programs apparently concurrently.

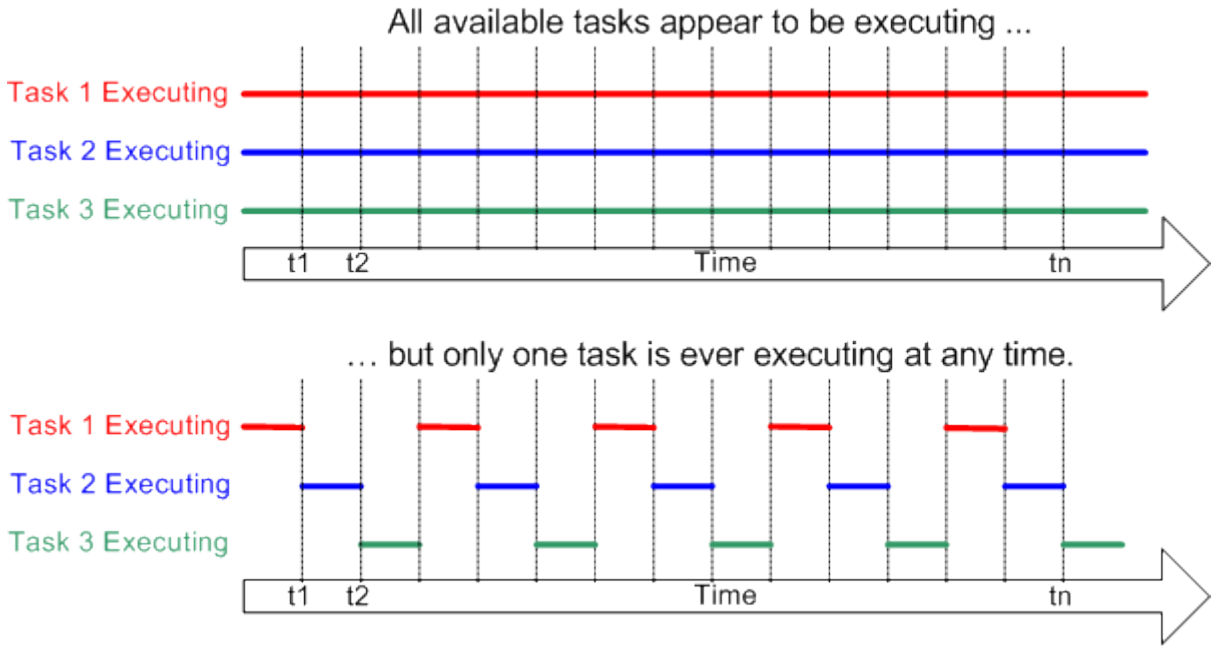
Each executing program is a **task** (or thread) under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be **multitasking**.

The use of a multitasking operating system can simplify the design of what would otherwise be a complex software application:

- The multitasking and inter-task communications features of the operating system allow the complex application to be partitioned into a set of smaller and more manageable tasks.
- The partitioning can result in easier software testing, work breakdown within teams, and code reuse.
- Complex timing and sequencing details can be removed from the application code and become the responsibility of the operating system.

Multitasking Vs Concurrency

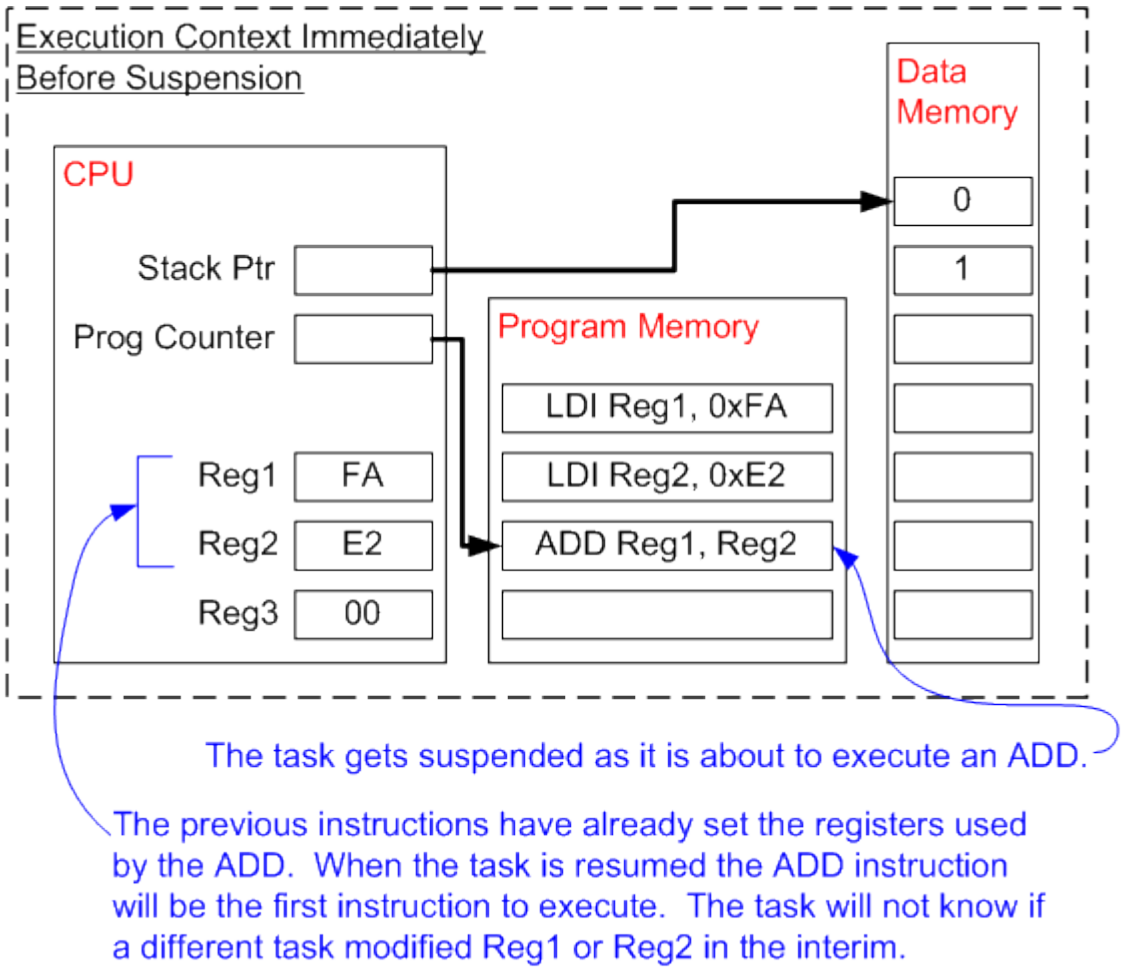
A conventional processor can only execute a single task at a time - but by rapidly switching between tasks a multitasking operating system can make it **appear** as if each task is executing concurrently. This is depicted by the diagram below which shows the execution pattern of three tasks with respect to time. The task names are color coded and written down the left hand. Time moves from left to right, with the colored lines showing which task is executing at any particular time. The upper diagram demonstrates the perceived concurrent execution pattern, and the lower the actual multitasking execution pattern.



Context Switching

[RTOS Fundamentals]

As a task executes it utilizes the processor / microcontroller registers and accesses RAM and ROM just as any other program. These resources together (the processor registers, stack, etc.) comprise the task execution context.



A task is a sequential piece of code - it does not know when it is going to get suspended (swapped out or switched out) or resumed (swapped in or switched in) by the kernel and does not even know when this has happened. Consider the example of a task being suspended immediately before executing an instruction that sums the values contained within two processor registers. While the task is suspended other tasks will execute and may modify the processor register values. Upon resumption the task will not know that the processor registers have been altered - if it used the modified values the summation would result in an incorrect value.

To prevent this type of error it is essential that upon resumption a task has a context identical to that immediately prior to its suspension. The operating system kernel is responsible for ensuring this is the case - and does so by saving the context of a task as it is suspended. When the task is resumed its saved context is restored by the operating system kernel prior to its execution. The process of saving the context of a task being suspended and restoring the context of a task being resumed is called context switching.

Scheduling

[RTOS Fundamentals]

The scheduler is the part of the kernel responsible for deciding which task should be executing at any particular time. The kernel can suspend and later resume a task many times during the task lifetime.

The scheduling policy is the algorithm used by the scheduler to decide which task to execute at any point in time. The policy of a (non real time) multi user system will most likely allow each task a "fair" proportion of processor time. The policy used in real time / embedded systems is described later.

In addition to being suspended involuntarily by the kernel a task can choose to suspend itself. It will do this if it either wants to delay (sleep) for a fixed period, or wait (block) for a resource to become available (eg a serial port) or an event to occur (eg a key press). A blocked or sleeping task is not able to execute, and will not be allocated any processing time.

suspending.gif

Referring to the numbers in the diagram above:

At (1) task 1 is executing.

At (2) the kernel suspends (swaps out) task 1 ...

... and at (3) resumes task 2.

While task 2 is executing (4), it locks a processor peripheral for its own exclusive access.

At (5) the kernel suspends task 2 ...

... and at (6) resumes task 3.

Task 3 tries to access the same processor peripheral, finding it locked task 3 cannot continue so suspends itself at (7).

At (8) the kernel resumes task 1.

Etc.

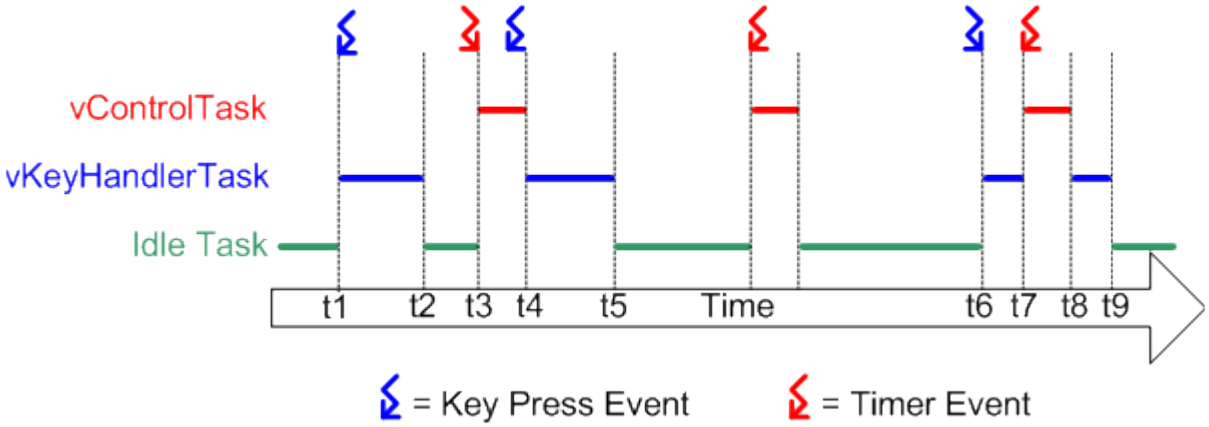
The next time task 2 is executing (9) it finishes with the processor peripheral and unlocks it.

The next time task 3 is executing (10) it finds it can now access the processor peripheral and this time executes until suspended by the kernel.

Real Time Scheduling

[RTOS Fundamentals]

The diagram below demonstrates how the tasks defined on the previous page would be scheduled by a real time operating system. The RTOS has itself created a task - the idle task - which will execute only when there are no other tasks able to do so. The RTOS idle task is always in a state where it is able to execute.



Referring to the diagram above:

At the start neither of our two tasks are able to run - vControlTask is waiting for the correct time to start a new control cycle and vKeyHandlerTask is waiting for a key to be pressed. Processor time is given to the RTOS idle task.

At time t1, a key press occurs. vKeyHandlerTask is now able to execute - it has a higher priority than the RTOS idle task so is given processor time.

At time t2 vKeyHandlerTask has completed processing the key and updating the LCD. It cannot continue until another key has been pressed so suspends itself and the RTOS idle task is again resumed.

At time t3 a timer event indicates that it is time to perform the next control cycle. vControlTask can now execute and as the highest priority task is scheduled processor time immediately.

Between time t3 and t4, while vControlTask is still executing, a key press occurs. vKeyHandlerTask is now able to execute, but as it has a lower priority than vControlTask it is not scheduled any processor time.

At t4 vControlTask completes processing the control cycle and cannot restart until the next timer event - it suspends itself. vKeyHandlerTask is now the task with the highest priority that is able to run so is scheduled processor time in order to process the previous key press.

At t5 the key press has been processed, and vKeyHandlerTask suspends itself to wait for the next key event. Again neither of our tasks are able to execute and the RTOS idle task is scheduled processor time.

Between t5 and t6 a timer event is processed, but no further key presses occur.

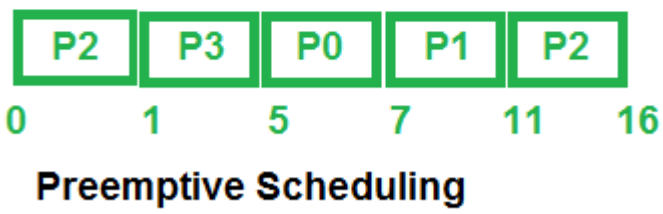
The next key press occurs at time t6, but before vKeyHandlerTask has completed processing the key a timer event occurs. Now both tasks are able to execute. As vControlTask has the higher priority vKeyHandlerTask is suspended before it has completed processing the key, and vControlTask is scheduled processor time.

At t8 vControlTask completes processing the control cycle and suspends itself to wait for the next. vKeyHandlerTask is again the highest priority task that is able to run so is scheduled processor time so the key press processing can be completed.

1. Preemptive Scheduling:

Preemptive scheduling is used when a process switches from running state to ready state or from the waiting state to ready state. The resources (mainly CPU cycles) are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue if that process still has CPU burst time remaining. That process stays in the ready queue till it gets its next chance to execute.

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



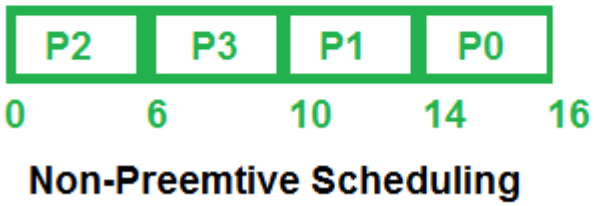
Preemptive Scheduling

2. Non-Preemptive Scheduling:

Non-preemptive Scheduling is used when a process terminates, or a process switches from running to the waiting state. In this scheduling, once the resources (CPU cycles) are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state. In the case of non-preemptive scheduling does not interrupt a process running CPU in the middle of the execution. Instead, it waits till the process completes its CPU burst time, and then it can allocate the CPU to another process.

Preemptive Scheduling has to maintain the integrity of shared data that's why it is cost associative which is not the case with Non-preemptive Scheduling.

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4



Non-Preemtive Scheduling

Key Differences Between Preemptive and Non-Preemptive Scheduling:

In preemptive scheduling, the CPU is allocated to the processes for a limited time whereas, in Non-preemptive scheduling, the CPU is allocated to the process till it terminates or switches to the waiting state.

The executing process in preemptive scheduling is interrupted in the middle of execution when higher priority one comes whereas, the executing process in non-preemptive scheduling is not interrupted in the middle of execution and waits till its execution.

In Preemptive Scheduling, there is the overhead of switching the process from the ready state to running state, vise-verse and maintaining the ready queue. Whereas in the case of non-preemptive scheduling has no overhead of switching the process from running state to ready state.

In preemptive scheduling, if a high-priority process frequently arrives in the ready queue then the process with low priority has to wait for a long, and it may have to starve. , in the non-preemptive scheduling, if CPU is allocated to the process having a larger burst time then the processes with small burst time may have to starve.

Preemptive scheduling attains flexibility by allowing the critical processes to access the CPU as they arrive into the ready queue, no matter what process is executing currently. Non-preemptive scheduling is called rigid as even if a critical process enters the ready queue the process running CPU is not disturbed.