Module-3

RTOS and IDE for Embedded System Design

Topics:

- Operating System basics
- Types of operating systems
- Task, process and threads (Only POSIX Threads with an example program)
- Thread preemption,
- Preemptive Task scheduling techniques
- Task Communication
- Task synchronization issues – Racing and Deadlock.
- How to choose an RTOS
- Integration and testing of Embedded hardware and firmware
- Embedded system Development Environment – Block diagram (excluding Keil).

OPERATING SYSTEM BASICS

The operating system acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services.
The OS manages the system resources and makes them available to the user applications/tasks on a need basis. A normal computing system is a collection of different I/O subsystems, working, and storage memory. The primary functions of an operating system is

- Make the system convenient to use
- Organise and manage the system resources effi ciently and correctly

Figure gives an insight into the basic components of an operating system and their interfaces with rest of the world.

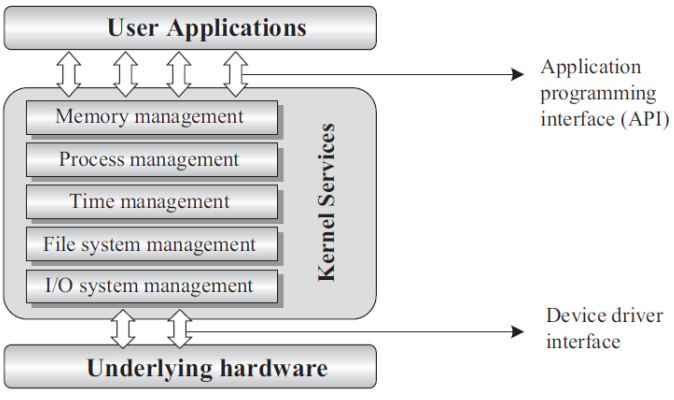


Fig. The Operating System Architecture

3.1 The Kernel

The kernel is the core of the operating system and is responsible for managing the system resources and the communication among the hardware and other system services. Kernel acts as the abstraction layer between system resources and user applications. Kernel contains a set of system libraries and services. For a general purpose OS, the kernel contains different services for handling the following.

Process Management deals with managing the processes/tasks. Process management includes setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, scheduling and managing the execution of the process, setting up and managing the Process Control Block (PCB), Inter Process Communication and synchronisation, process termination/ deletion, etc.

Primary Memory Management The term primary memory refers to the volatile memory (RAM) where processes are loaded and variables and shared data associated with each process are stored. The Memory Management Unit (MMU) of the kernel is responsible for

- Keeping track of which part of the memory area is currently used by which process
- Allocating and De-allocating memory space on a need basis (Dynamic memory allocation).

File System Management File is a collection of related information. A file could be a program (source code or executable), text files, image files, word documents, audio/video files, etc. Each of these files differ in the kind of information they hold and the way in which the information is stored. The file operation is a useful service provided by the OS. The file system management service of Kernel is responsible for

- The creation, deletion and alteration of files
- Creation, deletion and alteration of directories
- Saving of files in the secondary storage memory (e.g. Hard disk storage)
- Providing automatic allocation of file space based on the amount of free space available
- Providing a flexible naming convention for the files

The various file system management operations are OS dependent. For example, the kernel of Microsoft® DOS OS supports a specific set of file system management operations and they are not the same as the file system operations supported by UNIX Kernel.

I/O System (Device) Management Kernel is responsible for routing the I/O requests coming from different user applications to the appropriate I/O devices of the system. In a well-structured OS, the direct accessing of I/O devices are not allowed and the access to them are provided through a set of Application Programming Interfaces (APIs) exposed by the kernel.

The kernel maintains a list of all the I/O devices of the system. This list may be available in advance, at the time of building the kernel. Some kernels, dynamically updates the list of available devices as and when a new device is installed.

The service 'Device Manager' of the kernel is responsible for handling all I/O device related operations. The kernel talks to the I/O device through a set of low-level systems calls, which are implemented in a service, called device drivers. The device drivers are specific to a device or a class of devices. The Device

Manager is responsible for

- Loading and unloading of device drivers
- Exchanging information and the system specific control signals to and from the device

Secondary Storage Management deals with managing the secondary storage memory devices, if any, connected to the system. Secondary memory is used as backup medium for programs and data since the main memory is volatile. In most of the systems, the secondary storage is kept in disks (Hard Disk). The secondary storage management service of kernel deals with

- Disk storage allocation
- Disk scheduling (Time interval at which the disk is activated to backup data)
- Free Disk space management

Protection Systems Most of the modern operating systems are designed in such a way to support multiple users with different levels of access permissions (e.g. Windows 10 with user permissions like 'Administrator', 'Standard', 'Restricted', etc.). Protection deals with implementing the security policies

to restrict the access to both user and system resources by different applications or processes or users. In multiuser supported operating systems, one user may not be allowed to view or modify the whole/portions of another user's data or profile details. In addition, some application may not be granted with permission to make use of some of the system resources. This kind of protection is provided by the protection services running within the kernel.

Interrupt Handler Kernel provides handler mechanism for all external/internal interrupts generated by the system. These are some of the important services offered by the kernel of an operating system. It does not mean that a kernel contains no more than components/services explained above.

Depending on the type of the operating system, a kernel may contain lesser number of components/services or more number of components/ services. In addition to the components/services listed above, many operating systems offer a number of add on system components/services to the kernel.

Network communication, network management, user-interface graphics, timer services (delays, timeouts, etc.), error handler, database management, etc. are examples for such components/services.

Kernel exposes the interface to the various kernel applications/services, hosted by kernel, to the user applications through a set of standard Application Programming Interfaces (APIs). User applications can avail these API calls to access the various kernel application/services.

3.1.1 Kernel Space and User Space

The program code corresponding to the kernel applications/services are kept in a contiguous area (OS dependent) of primary (working) memory and is protected from the unauthorized access by user programs/applications. The memory space at which the kernel code is located is known as 'Kernel Space'.

Similarly, all user applications are loaded to a specific area of primary memory and this memory area is referred as 'User Space'. User space is the memory area where user applications are loaded and executed.

The partitioning of memory into kernel and user space is purely Operating System dependent. Some OS implements this kind of partitioning and protection whereas some OS do not segregate the kernel and user application code storage into two separate areas.

In an operating system with virtual memory support, the user applications are loaded into its corresponding virtual memory space with demand paging technique; Meaning, the entire code for the user application need not be loaded to the main (primary) memory at once; instead the user application code is split into different pages and these pages are loaded into and out of the main memory area on a need basis.

The act of loading the code into and out of the main memory is termed as 'Swapping'. Swapping happens between the main (primary) memory and secondary storage memory. Each process run in its own virtual memory space and are not allowed accessing the memory space corresponding to another processes, unless explicitly requested by the process.

Each process will have certain privilege levels on accessing the memory of other processes and based on the privilege settings, processes can request kernel to map another process's memory to its own or share through some other mechanism. Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory.

Monolithic Kernel and Microkernel

Based on the kernel design, kernels can be classified into 'Monolithic' and 'Micro'.

Monolithic Kernel In monolithic kernel architecture, all kernel services run in the kernel space. Here all kernel modules run within the same memory space under a single kernel thread. The tight internal

integration of kernel modules in monolithic kernel architecture allows the effective utilisation of the low-level features of the underlying system.

The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application. LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel. The architecture representation of a monolithic kernel is given in Fig.

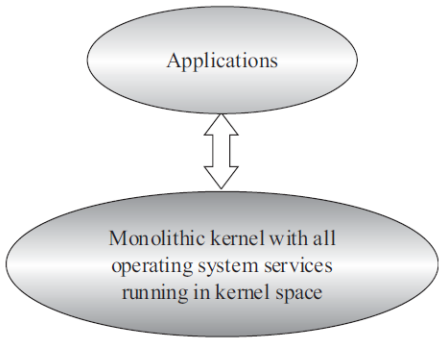


Fig. The Monolithic Kernel Model

Microkernel The microkernel design incorporates only the essential set of Operating System services into the kernel. The rest of the Operating System services are implemented in programs known as 'Servers' which runs in user space. This provides a highly modular design and OS-neutral abstraction to the kernel. Memory management, process management, timer systems and interrupt handlers are the essential services, which forms the part of the microkernel. Mach, QNX, Minix 3 kernels are examples for microkernel. The architecture representation of a microkernel is shown in Fig. 10.3.

Microkernel based design approach offers the following benefits

- Robustness: If a problem is encountered in any of the services, which runs as 'Server' application, the same can be reconfigured and re-started without the need for re-starting the entire OS. Thus, this approach is highly useful for systems, which demands high 'availability'. Refer Chapter 3 to get an understanding of 'availability'. Since the services which run as 'Servers' are running on a different memory space, the chances of corruption of kernel services are ideally zero.
- Configurability: Any services, which run as 'Server' application can be changed without the need to restart the whole system. This makes the system dynamically configurable.

3.2 TYPES OF OPERATING SYSTEMS

Depending on the type of kernel and kernel services, purpose and type of computing systems where the OS is deployed and the responsiveness to applications, Operating Systems are classified into different types.

3.2.1 General Purpose Operating System (GPOS)

The operating systems, which are deployed in general computing systems, are referred as General Purpose Operating Systems (GPOS).

The kernel of such an OS is more generalised and it contains all kinds of services required for executing generic applications. General-purpose operating systems are often quite non-deterministic in behaviour.

Their services can inject random delays into application software and may cause slow responsiveness of an application at unexpected times. GPOS are usually deployed in computing systems where deterministic behaviour is not an important criterion.

Personal Computer/Desktop system is a typical example for a system where GPOSs are deployed. Windows 10/8.x/XP/MS-DOS etc are examples for General Purpose Operating Systems.

3.2.2 Real-Time Operating System (RTOS)

There is no universal definition available for the term 'Real-Time' when it is used in conjunction with operating systems. In a broad sense, 'Real-Time' implies deterministic timing behaviour. Deterministic timing behaviour in RTOS context means the OS services consumes only known and expected amounts of time regardless the number of services.

A Real-Time Operating System or RTOS implements policies and rules concerning time-critical allocation of a system's resources. The RTOS decides which applications should run in which order and how much time needs to be allocated for each application. Predictable performance is the hallmark of a well-designed RTOS. This is best achieved by the consistent application of policies and rules. Policies guide the design of an RTOS. Rules implement those policies and resolve policy conflicts. Windows Embedded Compact, QNX, VxWorks MicroC/OS-II etc are examples of Real Time Operating Systems (RTOS).

3.2.2.1 The Real-Time Kernel

The kernel of a Real-Time Operating System is referred as Real. Time kernel. In complement to the conventional OS kernel, the Real-Time kernel is highly specialised and it contains only the minimal set of services required for running the user applications/tasks. The basic functions of a Real-Time kernel are listed below:

- Task/Process management
- Task/Process scheduling
- Task/Process synchronisation
- Error/Exception handling
- Memory management
- Interrupt handling
- Time management

Task/ Process management deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources, setting up a Task Control Block (TCB) for the task and task/process termination/deletion. A Task Control Block (TCB) is used for holding the information corresponding to a task. TCB usually contains the following set of information.

Task ID: Task Identification Number
Task State: The current state of the task (e.g. State = 'Ready' for a task which is ready to execute)
Task Type: Task type. Indicates what is the type for this task. The task can be a hard real time or soft real time or background task.
Task Priority: Task priority (e.g. Task priority = 1 for task with priority = 1)
Task Context Pointer: Context pointer. Pointer for context saving
Task Memory Pointers: Pointers to the code memory, data memory and stack memory for the task
Task System Resource Pointers: Pointers to system resources (semaphores, mutex, etc.) used by the task
Task Pointers: Pointers to other TCBs (TCBs for preceding, next and waiting tasks)
Other Parameters: Other relevant task parameters

The parameters and implementation of the TCB is kernel dependent. The TCB parameters vary across different kernels, based on the task management implementation. Task management service utilises the TCB of a task in the following way

- Creates a TCB for a task on creating a task
- Delete/remove the TCB of a task when the task is terminated or deleted
- Reads the TCB to get the state of a task

- Update the TCB with updated parameters on need basis (e.g. on a context switch)
- Modify the TCB to change the priority of the task dynamically

Task/ Process Scheduling Deals with sharing the CPU among various tasks/processes. A kernel application called 'Scheduler' handles the task scheduling. Scheduler is nothing but an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behaviour.

Task/ Process Synchronisation deals with synchronising the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.

Error/ Except on Handling Deals with registering and handling the errors occurred/exceptions raised during the execution of tasks. Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution, etc. are examples of errors/exceptions. Errors/Exceptions can happen at the kernel level services or at task level.

Deadlock is an example for kernel level exception, whereas timeout is an example for a task level exception. The OS kernel gives the information about the error in the form of a system call (API). GetLastError() API provided by Windows CE/Embedded Compact RTOS is an example for such a system call. Watchdog timer is a mechanism for handling the timeouts for tasks.

Certain tasks may involve the waiting of external events from devices. These tasks will wait infinitely when the external device is not responding and the task will generate a hang-up behaviour. In order to avoid these types of scenarios, a proper timeout mechanism should be implemented. A watchdog is normally used in such situations.

The watchdog will be loaded with the maximum expected wait time for the event and if the event is not triggered within this wait time, the same is informed to the task and the task is timed out. If the event happens before the timeout, the watchdog is resetted.

Memory Management Compared to the General Purpose Operating Systems, the memory management function of an RTOS kernel is slightly different. In general, the memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block (initialised memory block consumes more allocation time than un-initialised memory block).

Since predictable timing and deterministic behaviour are the primary focus of an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation. RTOS makes use of 'block' based memory allocation technique, instead of the usual dynamic memory allocation techniques used by the GPOS.

RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis. The blocks are stored in a 'Free Buffer Queue'. To achieve predictable timing and avoid the timing overheads, most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection. RTOS kernels assume that the whole design is proven correct and protection is unnecessary.

Some commercial RTOS kernels allow memory protection as optional and the kernel enters a fail-safe mode when an illegal memory access occurs.

A few RTOS kernels implement Virtual Memory* concept for memory allocation if the system supports secondary memory storage (like HDD and FLASH memory). In the 'block' based memory allocation, a block of fixed memory is always allocated for tasks on need basis and it is taken as a unit. Hence, there will not be any memory fragmentation issues.

The memory allocation can be implemented as constant functions and thereby it consumes fixed amount of time for memory allocation. This leaves the deterministic behaviour of the RTOS kernel untouched. The 'block' memory concept avoids the garbage collection overhead also.

The 'block' based memory allocation achieves deterministic behaviour with the trade-of limited choice of memory chunk size and suboptimal memory usage.

Interrupt Handling deals with the handling of various types of interrupts. Interrupts provide Real-Time behaviour to systems. Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.

Interrupts can be either Synchronous or Asynchronous. Interrupts which occurs in sync with the currently executing task is known as Synchronous interrupts.

Usually the software interrupts fall under the Synchronous Interrupt category. Divide by zero, memory segmentation error, etc. are examples of synchronous interrupts.

For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task. Asynchronous interrupts are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task.

The interrupts generated by external devices connected to the processor/controller, timer overflow interrupts, serial data reception/ transmission interrupts, etc. are examples for asynchronous interrupts. For asynchronous interrupts, the interrupt handler is usually written as separate task (Depends on OS kernel implementation) and it runs in a different context.

Hence, a context switch happens while handling the asynchronous interrupts. Priority levels can be assigned to the interrupts and each interrupts can be enabled or disabled individually. Most of the RTOS kernel implements 'Nested Interrupts' architecture. Interrupt nesting allows the pre-emption (interruption) of an Interrupt Service Routine (ISR), servicing an interrupt, by a high priority interrupt.

Time Management Accurate time management is essential for providing precise time reference for all applications. The time reference to kernel is provided by a high-resolution Real-Time Clock (RTC) hardware chip (hardware timer). The hardware timer is programmed to interrupt the processor/controller at a fixed rate.

This timer interrupt is referred as ' Timer tick'. The 'Timer tick' is taken as the timing reference by the kernel.

The 'Timer tick' interval may vary depending on the hardware timer. Usually the 'Timer tick' varies in the microseconds range. The time parameters for tasks are expressed as the multiples of the 'Timer tick'.

The System time is updated based on the 'Timer tick'. If the System time register is 32 bits wide and the 'Timer tick' interval is 1 microsecond, the System time register will reset in

$232 * 10^{-6} / (24 * 60 * 60) = 49700$ Days = ~ 0.0497 Days = 1.19 Hours

If the 'Timer tick' interval is 1 millisecond, the system time register will reset in

$232 * 10^{-3} / (24 * 60 * 60) = 497$ Days = 49.7 Days = ~ 50 Days

The 'Timer tick' interrupt is handled by the 'Timer Interrupt' handler of kernel. The 'Timer tick' interrupt can be utilised for implementing the following actions.

- Save the current context (Context of the currently executing task).
- Increment the System time register by one. Generate timing error and reset the System time register if the timer tick count is greater than the maximum range available for System time register.
- Update the timers implemented in kernel (Increment or decrement the timer registers for each timer depending on the count direction setting for each register. Increment registers with count direction setting = 'count up' and decrement registers with count direction setting = 'count down').
- Activate the periodic tasks, which are in the idle state.
- Invoke the scheduler and schedule the tasks again based on the scheduling algorithm.
- Delete all the terminated tasks and their associated data structures (TCBs)

- Load the context for the fi rst task in the ready queue. Due to the re-scheduling, the ready task might be changed to a new one from the task, which was preempted by the 'Timer Interrupt' task.

Apart from these basic functions, some RTOS provide other functionalities also (Examples are file management and network functions). Some RTOS kernel provides options for selecting the required kernel functions at the time of building a kernel. The user can pick the required functions from the set of available functions and compile the same to generate the kernel binary. Windows CE is a typical example for such an RTOS. While building the target, the user can select the required components for the kernel.

3.2.2.2 Hard Real-Time

Real-Time Operating Systems that strictly adhere to the timing constraints for a task is referred as 'Hard Real-Time' systems.

A Hard Real-Time system must meet the deadlines for a task without any slippage. Missing any deadline may produce catastrophic results for Hard Real-Time Systems, including permanent data lose and irrecoverable damages to the system/users.

Hard Real-Time systems emphasise the principle 'A late answer is a wrong answer'. A system can have several such tasks and the key to their correct operation lies in scheduling them so that they meet their time constraints. Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples for Hard Real-Time Systems.

The Air bag control system should be into action and deploy the air bags when the vehicle meets a severe accident. Ideally speaking, the time for triggering the air bag deployment task, when an accident is sensed by the Air bag control system, should be zero and the air bags should be deployed exactly within the time frame, which is predefined for the air bag deployment task. Any delay in the deployment of the air bags makes the life of the passengers under threat.

When the air bag deployment task is triggered, the currently executing task must be pre-empted, the air bag deployment task should be brought into execution, and the necessary I/O systems should be made readily available for the air bag deployment task.

To meet the strict deadline, the time between the air bag deployment event triggering and start of the air bag deployment task execution should be minimum, ideally zero.

As a rule of thumb, Hard Real-Time Systems does not implement the virtual memory model for handling the memory. This eliminates the delay in swapping in and out the code corresponding to the task to and from the primary memory.

In general, the presence of Human in the loop (HITL) for tasks introduces unexpected delays in the task execution. Most of the Hard Real-Time Systems are automatic and does not contain a 'human in the loop'.

3.2.2.3 Soft Real-Time

Real-Time Operating System that does not guarantee meeting deadlines, but offer the best effort to meet the deadline are referred as 'Soft Real-Time' systems.

Missing deadlines for tasks are acceptable for a Soft Realtime system if the frequency of deadline missing is within the compliance limit of the Quality of Service (QoS). A Soft Real-Time system emphasises the principle 'A late answer is an acceptable answer, but it could have done bit faster'. Soft Real-Time systems most often have a 'human in the loop (HITL)'.

Automatic Teller Machine (ATM) is a typical example for Soft-Real-Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens. An audio-video playback system is another example for Soft Real-Time system. No potential damage arises if a sample comes late by fraction of a second, for playback.

3.3 TASKS, PROCESS AND THREADS

The term 'task' refers to something that needs to be done. In addition, we will have an order of priority and schedule/timeline for executing these tasks. In the operating system context, a task is defined as the program in execution and the related information maintained by the operating system for the program. Task is also known as 'Job' in the operating system context. A program or part of it in execution is also called a 'Process'.

3.3.1 Process

A 'Process' is a program, or part of it, in execution. Process is also known as an instance of a program in execution. Multiple instances of the same program can execute simultaneously. A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange, etc. A process is sequential in execution.

3.3.1.1 The Structure of a Process

The concept of 'Process' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilisation of the CPU and other system resources. Concurrent execution is achieved through the sharing of CPU among the processes.

A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process. This can be visualized as shown in Fig.

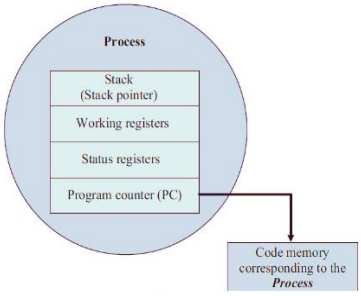


Fig. Structure of a Process

A process which inherits all the properties of the CPU can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor.

When the process gets its turn, its registers and the program counter register becomes mapped to the physical registers of the CPU.

From a memory perspective, the memory occupied by the process is segregated into three regions, namely, Stack memory, Data memory and Code memory

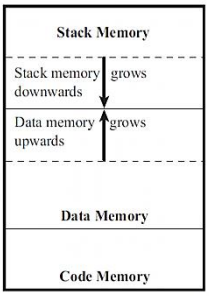


Fig. Memory organisation of a Process

The ‘ Stack’ memory holds all temporary data such as variables local to the process. Data memory holds all global data for the process.

The code memory contains the program code (instructions) corresponding to the process. On loading a process into the main memory, a specific area of memory is allocated for the process.

The stack memory usually starts (OS Kernel implementation dependent) at the highest memory address from the memory area allocated for the process.

Say for example, the memory map of the memory area allocated for the process is 2048 to 2100, the stack memory starts at address 2100 and grows downwards to accommodate the variables local to the process.

3.3.1.2 Process States and State Transition

The creation of a process to its termination is not a single step operation. The process traverses through a series of states during its transition from the newly created state to the terminated state. The cycle through which a process changes its state from ‘newly created’ to ‘execution completed’ is known as ‘Process Life Cycle’.

The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to do next.

Figure represents the various states associated with a process. The state at which a process is being created is referred as ‘Created State’. The Operating System recognises a process in the ‘Created State’ but no resources are allocated to the process.

The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as ‘Ready State’. At this stage, the process is placed in the ‘Ready list’ queue maintained by the OS.

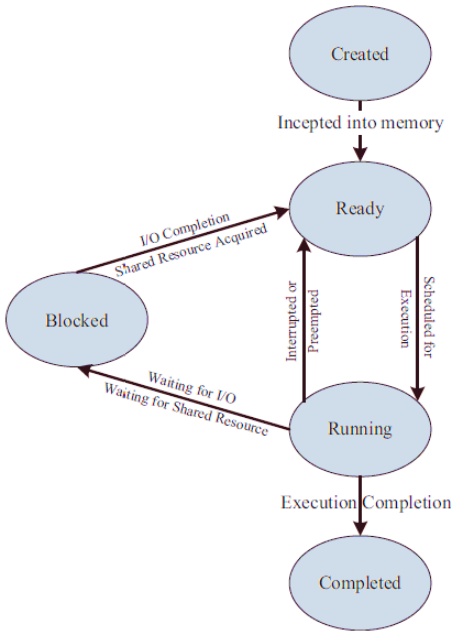


Fig. Process states and state transition representation

The state where in the source code instructions corresponding to the process is being executed is called ‘Running State’.

Running state is the state at which the process execution happens. ‘Blocked State/Wait State’ refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources.

The blocked state might be invoked by various conditions like: the process enters a wait state for an event to occur (e.g. Waiting for user inputs such as keyboard input) or waiting for getting access to a shared resource (will be discussed at a later section of this chapter).

A state where the process completes its execution is known as 'Completed State'. The transition of a process from one state to another is known as 'State transition'. When a process changes its state from Ready to running or from running to blocked or terminated or from blocked to running, the CPU allocation for the process may also change.

It should be noted that the state representation for a process/task mentioned here is a generic representation. The states associated with a task may be known with a different name or there may be more or less number of states than the one explained here under different OS kernel.

For example, under VxWorks' kernel, the tasks may be in either one or a specifi c combination of the states READY, PEND, DELAY and SUSPEND. The PEND state represents a state where the task/process is blocked on waiting for I/O or system resource. The DELAY state represents a state in which the task/process is sleeping and the SUSPEND state represents a state where a task/process is temporarily suspended from execution and not available for execution.

Under MicroC/OS-II kernel, the tasks may be in one of the states, DORMANT, READY, RUNNING, WAITING or INTERRUPTED. The DORMANT state represents the 'Created' state and WAITING state represents the state in which a process waits for shared resource or I/O access.

3.3.1.3 Process Management

Process management deals with the creation of a process, setting up the memory space for the process, loading the process's code into the memory space, allocating system resources, setting up a Process Control Block (PCB) for the process and process termination/deletion.

3.3.2 Threads

A thread is the primitive that can execute code. A thread is a single sequential flow of control within a process. 'Thread' is also known as lightweight process. A process can have many threads of execution. Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area. Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack. The memory model for a process and its associated threads are given in Fig.

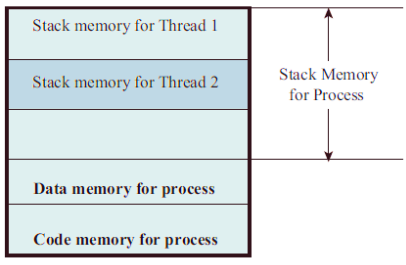


Fig. Memory organisation of a Process and its associated Threads

3.3.2.1 The Concept of Multithreading

A process/task in embedded application may be a complex or lengthy one and it may contain various suboperations like getting input from I/O devices connected to the processor, performing some internal calculations/operations, updating some I/O devices etc.

If all the subfunctions of a task are executed in sequence, the CPU utilisation may not be efficient. For example, if the process is waiting for a user input, the CPU enters the wait state for the event, and the process execution also enters a wait state.

Instead of this single sequential execution of the whole process, if the task/process is split into different threads carryingout the different sub-functionalities of the process, the CPU can be effectively utilised and when the thread corresponding to the I/O operation enters the wait state, another threads which do not require the I/O event for their operation can be switched into execution.

This leads to more speedy execution of the process and the efficient utilisation of the processor time and resources. The multithreaded architecture of a process can be better visualised with the thread-process diagram shown in Fig.

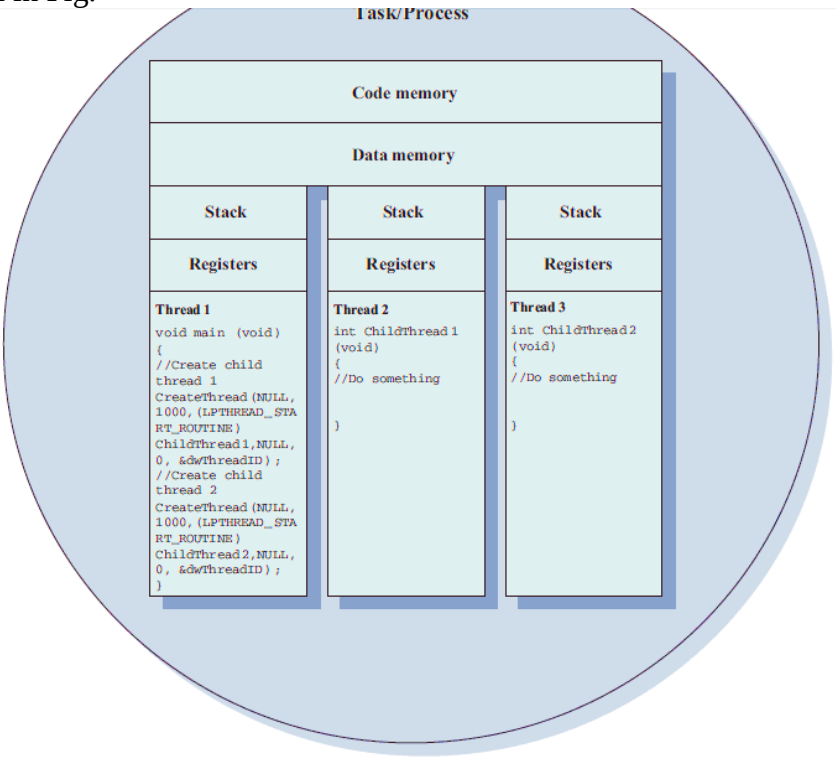


Fig. Process with multi-threads

If the process is split into multiple threads, which executes a portion of the process, there will be a main thread and rest of the threads will be created within the main thread. Use of multiple threads to execute a process brings the following advantage.

- Better memory utilisation. Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.
- Since the process is split into different threads, when one thread enters a wait state, the CPU can be utilised by other threads of the process that do not require the event, which the other thread is waiting, for processing. This speeds up the execution of the process.
- Efficient CPU utilisation. The CPU is engaged all time.

3.3.2.2 Thread Standards

Thread standards deal with the different standards available for thread creation and management. These standards are utilised by the operating systems for thread creation and thread management. It is a set of thread class libraries. The commonly available thread class libraries are explained below.

POSIX Threads POSIX stands for Portable Operating System Interface. The POSIX.4 standard deals with the Real-Time extensions and POSIX.4a standard deals with thread extensions. The POSIX standard library for thread creation and management is 'Pthreads'.

'Pthreads' library defines the set of POSIX thread creation and management functions in 'C' language. The primitive

```
int pthread_create(pthread_t *new_thread_ID, const pthread_attr_t
*attribute, void * (*start_function)(void *), void *arguments);
```

creates a new thread for running the function ***start_ function***. Here ***pthread_t*** is the handle to the newly created thread and ***pthread_attr_t*** is the data type for holding the thread attributes. ***'start_function'*** is the function the thread is going to execute and arguments is the arguments for 'start_function' (It is a void * in the above example). On successful creation of a Pthread, pthread_create() associates the Thread Control Block (TCB) corresponding to the newly created thread to the variable of type pthread_t (new_thread_ID in our example).
The primitive

int pthread_join(pthread_t new_thread,void * *thread_status);

blocks the current thread and waits until the completion of the thread pointed by it (In this example new_ thread)
All the POSIX 'thread calls' returns an integer. A return value of zero indicates the success of the call. It is always good to check the return value of each call.

Example 1
Write a multithreaded application to print "Hello I'm in main thread" from the main thread and "Hello I'm in new thread" 5 times each, using the pthread_create() and pthread_join() POSIX primitives.

```
//Assumes the application is running on an OS where POSIX library is
//available
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
//*********************************************************************
//New thread function for printing "Hello I'm in new thread"
void *new_thread( void *thread_args )
{
int i, j;
for( j= 0; j < 5; j++ )
{
printf("Hello I'm in new thread\n" );
//Wait for some time. Do nothing
//The following line of code can be replaced with
//OS supported delay function like sleep(), delay () etc…
for( i= 0; i < 10000; i++ );
}
return NULL;
}
```

```
//*****************************************************************
//Start of main thread
int main( void )
{
int i, j;
pthread_t tcb;
//Create the new thread for executing new_thread function
if (pthread_create( &tcb, NULL, new_thread, NULL ))
{
//New thread creation failed
printf("Error in creating new thread\n" );
return -1;
}
for( j= 0; j < 5; j++ )
{
printf("Hello I'm in main thread\n" );
//Wait for some time. Do nothing
//The following line of code can be replaced with
//OS supported delay function like sleep(), delay etc…
for( i= 0; i < 10000; i++ );
}
if (pthread_join(tcb, NULL ))
{
//Thread join failed
printf("Error in Thread join\n" );
return -1;
}
return 1;
}
```

3.4 Preemptive Scheduling

Preemptive scheduling is employed in systems, which implements preemptive multitasking model. In preemptive scheduling, every task in the 'Ready' queue gets a chance to execute.

When and how often each process gets a chance to execute (gets the CPU time) is dependent on the type of preemptive scheduling algorithm used for scheduling the processes. In this kind of scheduling, **the scheduler can preempt (stop temporarily)** the currently executing task/process and select another task from the 'Ready' queue for execution.

When to pre-empt a task and which task is to be picked up from the 'Ready' queue for execution after preempting the current task is purely dependent on the scheduling algorithm. A task which is preempted by the scheduler is moved to the 'Ready' queue.

The act of moving a 'Running' process/task into the 'Ready' queue by the scheduler, without the processes requesting for it is known as 'Preemption'. Preemptive scheduling can be implemented in different approaches.

The two important approaches adopted in preemptive scheduling are time-based preemption and priority-based preemption. The various types of preemptive scheduling adopted in task/process scheduling are explained below.

3.4.1 Preemptive SJF Scheduling/ Shortest Remaining Time (SRT)

The non-preemptive SJF scheduling algorithm sorts the 'Ready' queue only after completing the execution of the current process or when the process enters 'Wait' state, whereas the preemptive SJF scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated time for the currently executing process.

If the execution time of the new process is less, the currently executing process is preempted and the new process is scheduled for execution.

Thus preemptive SJF scheduling always compares the execution completion time (It is same as the remaining time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. A new process P4 with estimated completion time 2 ms enters the 'Ready' queue after 2 ms. Assume all the processes contain only CPU operation and no I/O operations are involved.

Solution :

At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the SRT scheduler picks up the process with the shortest remaining time for execution completion (In this example, P2 with remaining time 5 ms) for scheduling.

Now process P4 with estimated execution completion time 2 ms enters the 'Ready' queue after 2 ms of start of execution of P2.

Since the SRT algorithm is preemptive, the remaining time for completion of process P2 is checked with the remaining time for completion of process P4.

The remaining time for completion of P2 is 3 ms which is greater than that of the remaining time for completion of the newly entered process P4 (2 ms).

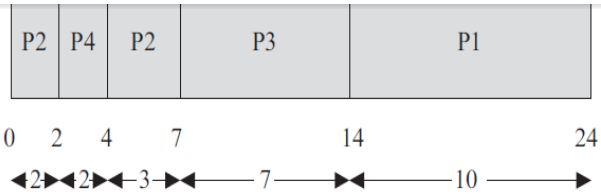
Hence P2 is preempted and P4 is scheduled for execution. P4 continues its execution to finish since there is no new process entered in the 'Ready' queue during its execution.

After 2 ms of scheduling P4 terminates and now the scheduler again sorts the 'Ready' queue based on the remaining time for completion of the processes present in the 'Ready' queue.

Since the remaining time for P2 (3 ms), which is preempted by P4 is less than that of the remaining time for other processes in the 'Ready' queue, P2 is scheduled for execution.

Due to the arrival of the process P4 with execution time 2 ms, the 'Ready' queue is re-sorted in the order P2, P4, P2, P3, P1.

This reveals that the Average waiting Time and Turn Around Time (TAT) improves significantly with preemptive SJF scheduling.



The waiting time for all the processes are given as

Waiting time for P2 = 0 ms + (4 – 2) ms = 2 ms (P2 starts executing first and is interrupted by P4 and has to wait till the completion of P4 to get the next CPU slot)
Waiting time for P4 = 0 ms (P4 starts executing by preempting P2 since the execution time for completion of P4 (2 ms) is less than that of the Remaining time for execution completion of P2 (Here it is 3 ms))
Waiting time for P3 = 7 ms (P3 starts executing after completing P4 and P2)
Waiting time for P1 = 14 ms (P1 starts executing after completing P4, P2 and P3)
Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P4+P2+P3+P1)) / 4
= (0 + 2 + 7 + 14)/4 = 23/4
= 5.75 milliseconds

Turn Around Time (TAT) for P2 = 7 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 2 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (2 – 2) + 2)
Turn Around Time (TAT) for P3 = 14 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P1 = 24 ms (Time spent in Ready Queue + Execution Time)
Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P2+P4+P3+P1)) / 4
= (7+2+14+24)/4 = 47/4
= 11.75 milliseconds

3.4.2 Round Robin (RR) Scheduling

In the process scheduling context, 'Round Robin' brings the message "Equal chance to all". In Round Robin scheduling, each process in the 'Ready' queue is executed for a pre-defined time slot. The execution starts with picking up the first process in the 'Ready' queue.

It is executed for a pre-defined time and when the pre-defined time elapses or the process completes (before the pre-defined time slice), the next process in the 'Ready' queue is selected for execution.

This is repeated for all the processes in the 'Ready' queue. Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the 'Ready' queue again for execution.

The sequence is repeated. This reveals that the Round Robin scheduling is similar to the FCFS scheduling and the only difference is that a time slice based preemption is added to switch the execution between the processes in the 'Ready' queue.

The 'Ready' queue can be considered as a circular queue in which the scheduler picks up the first process for execution and moves to the next till the end of the queue and then comes back to the beginning of the queue to pick up the first process.

Round Robin scheduling ensures that every process gets a fixed amount of CPU time for execution.

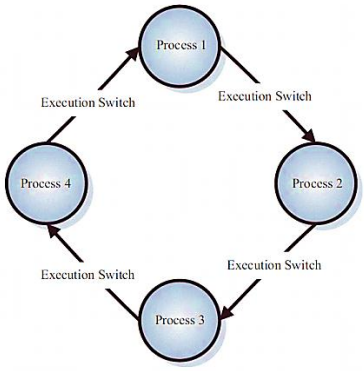


Fig. Round Robin Scheduling

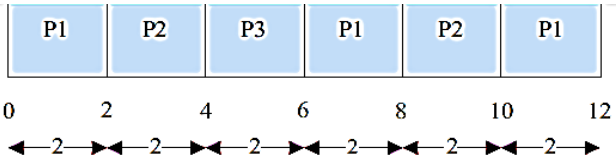
Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively, enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in RR algorithm with Time slice = 2 ms.

Solution: The scheduler sorts the 'Ready' queue based on the FCFS policy and picks up the first process P1 from the 'Ready' queue and executes it for the time slice 2 ms. When the time slice is expired, P1 is preempted and P2 is scheduled for execution.

The Time slice expires after 2ms of execution of P2. Now P2 is preempted and P3 is picked up for execution.

P3 completes its execution within the time slice and the scheduler picks P1 again for execution for the next time slice. This procedure is repeated till all the processes are serviced. The order in which the processes are scheduled for execution is represented as



The waiting time for all the processes are given as

Waiting time for P1 = 0 + (6 – 2) + (10 – 8) = 0 + 4 + 2 = 6 ms
(P1 starts executing first and waits for two time slices to get execution back and again 1 time slice for getting CPU time)

Waiting time for P2 = (2 – 0) + (8 – 4) = 2 + 4 = 6 ms
(P2 starts executing after P1 executes for 1 time slice and waits for two time slices to get the CPU time)

Waiting time for P3 = (4 – 0) = 4 ms
(P3 starts executing after completing the first time slices for P1 and P2 and completes its execution in a single time slice)

Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P1 + P2 + P3)) / 3
= (6 + 6 + 4)/3 = 16/3
= 5.33 milliseconds

Turn Around Time (TAT) for P1 = 12 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 10 ms (-Do-)
Turn Around Time (TAT) for P3 = 6 ms (-Do-)

Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P1 + P2 + P3))/3
= (12 + 10 + 6)/3 = 28/3
= 9.33 milliseconds
Average Turn Around Time (TAT) is the sum of average waiting time and average execution time.
Average Execution time = (Execution time for all the process)/No. of processes
= (Execution time for (P1 + P2 + P3))/3
= (6 + 4 + 2)/3 = 12/3 = 4
Average Turn Around Time = Average Waiting time + Average Execution time
= 5.33 + 4
= 9.33 milliseconds

RR scheduling involves lot of overhead in maintaining the time slice information for every process which is currently being executed.

3.4.3 Priority Based Scheduling

In preemptive scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution. Priority based preemptive scheduling algorithm ensures that a process with high priority is serviced at the earliest compared to other low priority processes in the 'Ready' queue.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0—highest priority, 3—lowest priority) respectively enters the ready queue together. A new process P4 with estimated completion time 6 ms and priority 0 enters the 'Ready' queue after 5 ms of start of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

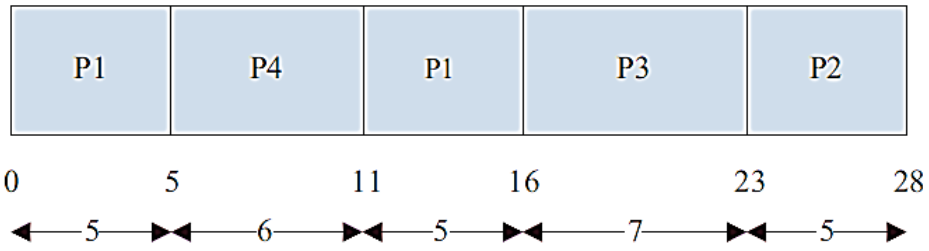
Solution: At the beginning, there are only three processes (P1, P2 and P3) available in the 'Ready' queue and the scheduler picks up the process with the highest priority (In this example P1 with priority 1) for scheduling.

Now process P4 with estimated execution completion time 6 ms and priority 0 enters the 'Ready' queue after 5 ms of start of execution of P1.

Since the scheduling algorithm is preemptive, P1 is preempted by P4 and P4 runs to completion. After 6 ms of scheduling, P4 terminates and now the scheduler again sorts the 'Ready' queue for process with highest priority.

Since the priority for P1 (priority 1), which is preempted by P4 is higher than that of P3 (priority 2) and P2 ((priority 3), P1 is again picked up for execution by the scheduler.

Due to the arrival of the process P4 with priority 0, the 'Ready' queue is resorted in the order P1, P4, P1, P3, P2. At the beginning it was P1, P3, P2. The execution sequence now changes as per the following diagram



The waiting time for all the processes are given as
Waiting time for P1 = 0 + (11 – 5) = 0 + 6 = 6 ms
(P1 starts executing first and gets preempted by P4 after 5 ms and again gets the CPU time after completion of P4)
Waiting time for P4 = 0 ms
(P4 starts executing immediately on entering the 'Ready' queue, by preempting P1)
Waiting time for P3 = 16 ms (P3 starts executing after completing P1 and P4)
Waiting time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)
Average waiting time = (Waiting time for all the processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (6 + 0 + 16 + 23)/4 = 45/4
= 11.25 milliseconds
Turn Around Time (TAT) for P1 = 16 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 6 ms
(Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (5 – 5) + 6 = 0 + 6)
Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)
Average Turn Around Time = (Turn Around Time for all the processes) / No. of Processes
= (Turn Around Time for (P2 + P4 + P3 + P1)) / 4
= (16 + 6 + 23 + 28)/4 = 73/4
= 18.25 milliseconds

Priority based preemptive scheduling gives Real-Time attention to high priority tasks. Thus priority based preemptive scheduling is adopted in systems which demands 'Real-Time' behaviour. Most of the RTOSs make use of the preemptive priority based scheduling algorithm for process scheduling.

3.5 TASK COMMUNICATION

In a multitasking system, multiple tasks/processes run concurrently (in pseudo parallelism) and each process may or may not interact between. Based on the degree of interaction, the processes running on an OS are classified as

Co-operating Processes: In the co-operating interaction model one process requires the inputs from other processes to complete its execution.

Competing Processes: The competing processes do not share anything among themselves but they share the system resources. The competing processes compete for the system resources such as fi le, display device, etc. Co-operating processes exchanges information and communicate through the following methods.

Co-operation through Sharing: The co-operating process exchange data through some shared resources.

Co-operation through Communication: No data is shared between the processes. But they communicate for synchronisation.

The mechanism through which processes/tasks communicate each other is known as Inter Process/Task Communication (IPC).

Inter Process Communication is essential for process co-ordination. The various types of Inter Process Communication (IPC) mechanisms adopted by process are kernel (Operating System) dependent. Some of the important IPC mechanisms adopted by various kernels are explained below.

3.5.1 Shared Memory

Processes share some area of the memory to communicate among them (Fig.). Information to be communicated by the process is written to the shared memory area. Other processes which require this

information can read the same from the shared memory area. It is same as the real world example where 'Notice Board' is used by corporate to publish the public information among the employees (The only exception is; only corporate have the right to modify the information published on the Notice board and employees are given 'Read' only access, meaning it is only a one way channel).

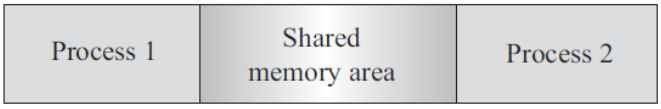


Fig. Concept of Shared Memory

The implementation of shared memory concept is kernel dependent. Different mechanisms are adopted by different kernels for implementing this. A few among them are:

3.5.1.1 Pipes

'Pipe' is a section of the shared memory used by processes for communicating. Pipes follow the client-server architecture. A process which creates a pipe is known as a pipe server and a process which connects to a pipe is known as pipe client. A pipe can be considered as a conduit for information flow and has two conceptual ends.

It can be unidirectional, allowing information flow in one direction or bidirectional allowing bi-directional information flow. A unidirectional pipe allows the process connecting at one end of the pipe to write to the pipe and the process connected at the other end of the pipe to read the data, whereas a bi-directional pipe allows both reading and writing at one end. The unidirectional pipe can be visualised as

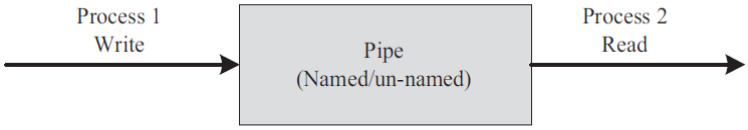


Fig. Concept of Pipe for IPC

The implementation of 'Pipes' is also OS dependent. Microsoft® Windows Desktop Operating Systems support two types of 'Pipes' for Inter Process Communication. They are:

Anonymous Pipes: The anonymous pipes are unnamed, unidirectional pipes used for data transfer between two processes.

Named Pipes: Named pipe is a named, unidirectional or bi-directional pipe for data exchange between processes. Like anonymous pipes, the process which creates the named pipe is known as pipe server.

A process which connects to the named pipe is known as pipe client. With named pipes, any process can act as both client and server allowing point-to-point communication.

Named pipes can be used for communicating between processes running on the same machine or between processes running on different machines connected to a network.

3.5.1.2 Memory Mapped Objects

Memory mapped object is a shared memory technique adopted by certain Real-Time Operating Systems for allocating a shared block of memory which can be accessed by multiple process simultaneously (of course certain synchronisation techniques should be applied to prevent inconsistent results).

In this approach a mapping object is created and physical storage for it is reserved and committed. A process can map the entire committed physical area or a block of it to its virtual address space.

All read and write operation to this virtual address space by a process is directed to its committed physical area.

Any process which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data.

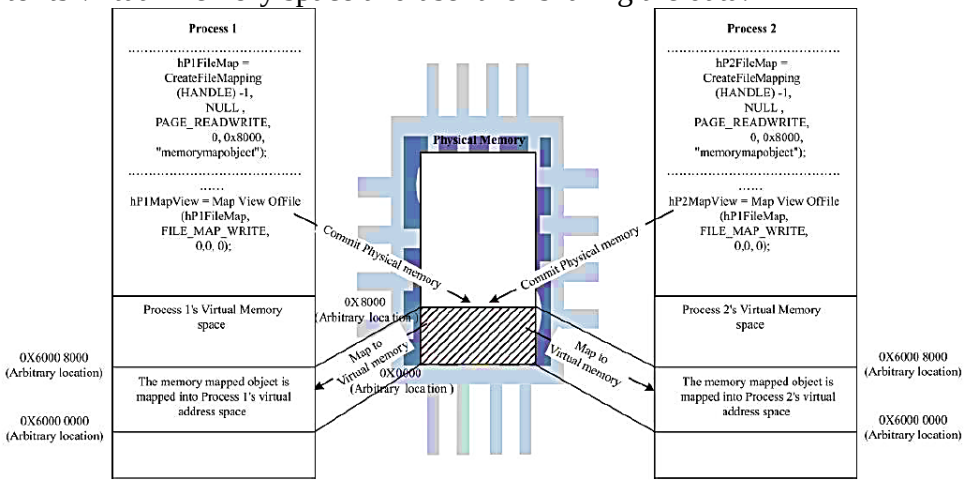


Fig. Concept of memory mapped object

3.5.2 Message Passing

Message passing is an (a)synchronous information exchange mechanism used for Inter Process/Thread Communication. The major difference between shared memory and message passing technique is that, through shared memory lots of data can be shared whereas only limited amount of info/data is passed through message passing. Also message passing is relatively fast and free from the synchronisation overheads compared to shared memory. Based on the message passing operation between the processes, message passing is classified into

3.5.2.1 Message Queue

Usually the process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called 'Message queue', which stores the messages temporarily in a system defined memory object, to pass it to the desired process (Fig.).

Messages are sent and received through send (Name of the process to which the message is to be sent, message) and receive (Name of the process from which the message is to be received, message) methods. The messages are exchanged through a message queue.

The implementation of the message queue, send and receive methods are OS kernel dependent.A thread which wants to communicate with another thread posts the message to the system message queue.

The kernel picks up the message from the system message queue one at a time and examines the message for finding the destination thread and then posts the message to the message queue of the corresponding thread.

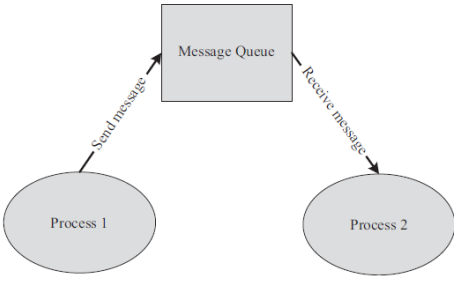


Fig. Concept of message queue based indirect messaging for IPC

A thread can simply post a message to another thread and can continue its operation or it may wait for a response from the thread to which the message is posted. The messaging mechanism is classifi ed into synchronous and asynchronous based on the behaviour of the message posting thread.

In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance (return) from the thread to which the message is posted, whereas in synchronous messaging, the thread which posts a message enters waiting state and waits for the message result from the thread to which the message is posted.

Message queues support two-way communication of messages of variable length. The two-way messaging between tasks can be implemented using one message queue for incoming messages and another one for outgoing messages. Messaging mechanism can be used for task-to task and task to Interrupt Service Routine (ISR) communication.

3.5.2.2 Mailbox

Mailbox is an alternate form of 'Message queues' and it is used in certain Real-Time Operating Systems for IPC. Mailbox technique for IPC in RTOS is usually used for one way messaging. The task/thread which wants to send a message to other tasks/threads creates a mailbox for posting the messages.

The threads which are interested in receiving the messages posted to the mailbox by the mailbox creator thread can subscribe to the mailbox. The thread which creates the mailbox is known as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients'.

The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox. The clients read the message from the mailbox on receiving the notification.

The mailbox creation, subscription, message reading and writing are achieved through OS kernel provided API calls.

Mailbox and message queues are same in functionality. The only difference is in the number of messages supported by them. Both of them are used for passing data in the form of message(s) from a task to another task(s).

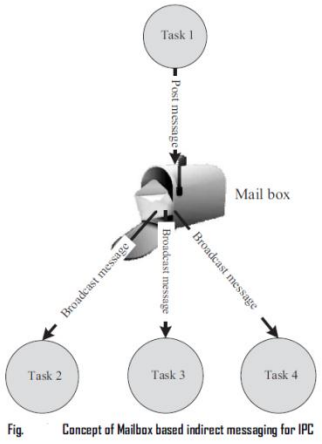


Fig. Concept of Mailbox based indirect messaging for IPC

Mailbox is used for exchanging a single message between two tasks or between an Interrupt Service Routine (ISR) and a task. Mailbox associates a pointer pointing to the mailbox and a wait list to hold the tasks waiting for a message to appear in the mailbox.

The implementation of mailbox is OS kernel dependent. Figure 10.21 given below illustrates the mailbox based IPC technique.

3.5.2.3 Signalling

Signalling is a primitive way of communication between processes/threads. Signals are used for asynchronous notifications where one process/thread fires a signal, indicating the occurrence of a

scenario which the other process(es)/thread(s) is waiting. Signals are not queued and they do not carry any data.

The communication mechanisms used in RTX51 Tiny OS is an example for Signalling. The os_send_signal kernel call under RTX 51 sends a signal from one task to a specified task. Similarly the os_wait kernel call waits for a specified signal.

The VxWorks RTOS kernel also implements 'signals' for inter process communication. Whenever a specified signal occurs it is handled in a signal handler associated with the signal.

3.5.3 Remote Procedure Call (RPC) and Sockets

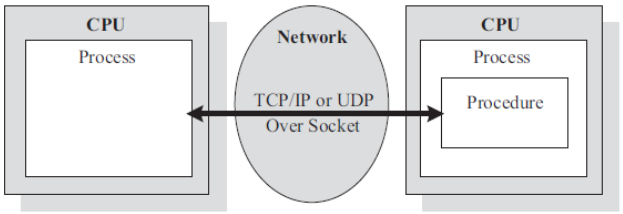
Remote Procedure Call or RPC (Fig.) is the Inter Process Communication (IPC) mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network.

In the object oriented language terminology RPC is also known as Remote Invocation or Remote Method Invocation (RMI). RPC is mainly used for distributed applications like client server applications. With RPC it is possible to communicate over a heterogeneous network (i.e. Network where Client and server applications are running on different Operating systems).

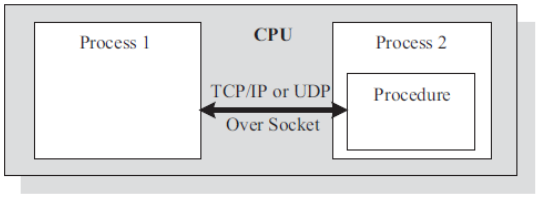
The CPU/process containing the procedure which needs to be invoked remotely is known as server. The CPU/process which initiates an RPC request is known as client. It is possible to implement RPC communication with different invocation interfaces. In order to make the RPC communication compatible across all platforms it should stick on to certain standard formats.

Interface Definition Language (IDL) defines the interfaces for RPC. Microsoft Interface Definition Language (MIDL) is the IDL implementation from Microsoft for all Microsoft platforms.

The RPC communication can be either Synchronous (Blocking) or Asynchronous (Non-blocking). In the Synchronous communication, the process which calls the remote procedure is blocked until it receives a response back from the other process. In asynchronous RPC calls, the calling process continues its execution while the remote process performs the execution of the procedure. The result from the remote procedure is returned back to the caller through mechanisms like callback functions.



Processes running on different CPUs which are networked



Processes running on same CPU

Fig. | Concept of Remote Procedure Call (RPC) for IPC

On security front, RPC employs authentication mechanisms to protect the systems against vulnerabilities. The client applications (processes) should authenticate themselves with the server for getting access. Authentication mechanisms like IDs, public key cryptography (like DES, 3DES), etc. are used by the client for authentication. Without authentication, any client can access the remote procedure. This may lead to potential security risks.

Sockets are used for RPC communication. Socket is a logical endpoint in a two-way communication link between two applications running on a network. A port number is associated with a socket so that the network layer of the communication channel can deliver the data to the designated application. Sockets are of different types, namely, Internet sockets (INET), UNIX sockets, etc. The INET socket works on internet communication protocol. TCP/IP, UDP, etc. are the communication protocols used by INET sockets. INET sockets are classified into:

1. Stream sockets
2. Datagram sockets

Stream sockets are connection oriented and they use TCP to establish a reliable connection. On the other hand, Datagram sockets rely on UDP for establishing a connection. The UDP connection is unreliable when compared to TCP. The client-server communication model uses a socket at the client side and a socket at the server side. A port number is assigned to both of these sockets. The client and server should be aware of the port number associated with the socket.

In order to start the communication, the client needs to send a connection request to the server at the specified port number. The client should be aware of the name of the server along with its port number.

The server always listens to the specified port number on the network. Upon receiving a connection request from the client, based on the success of authentication, the server grants the connection request and a communication channel is established between the client and server. The client uses the host name and port number of server for sending requests and server uses the client's name and port number for sending responses. If the client and server applications (both processes) are running on the same CPU, both can use the same host name and port number for communication.

The physical communication link between the client and server uses network interfaces like Ethernet or Wi-Fi for data communication. The underlying implementation of socket is OS kernel dependent. Different types of OSs provide different socket interfaces.

3.6 TASK SYNCHRONISATION

In a multitasking environment, multiple processes run concurrently (in pseudo parallelism) and share the system resources. Apart from this, each process has its own boundary wall and they communicate with each other with different IPC mechanisms including shared memory and variables.

Imagine a situation where two processes try to access display hardware connected to the system or two processes try to access a shared memory area where one process tries to write to a memory location when the other process is trying to read from this. What could be the result in these scenarios? Obviously unexpected results.

How these issues can be addressed? The solution is, make each process aware of the access of a shared resource either directly or indirectly. The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as 'Task/ Process Synchronisation'.

Various synchronisation issues may arise in a multitasking environment if processes are not synchronized properly. The following sections describe the major task communication synchronisation issues observed in multitasking and the commonly adopted synchronisation techniques to overcome these issues.

3.6.1 Task Communication/Synchronisation Issues

3.6.1.1 Racing

Racing or Race condition is the situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently. In a Race condition the final value of the shared data depends on the process which acted on the data finally.

- Imagine a situation where a process switching (context switching) happens from Process A to Process B when Process A is executing the counter++; statement.
- Process A accomplishes the counter++; statement through three different low level instructions.
- Now imagine that the process switching happened at the point where Process A executed the low level instruction, 'mov eax,dword ptr [ebp-4]' and is about to execute the next instruction 'add eax,1'.
- The scenario is illustrated in Fig. 10.23. Process B increments the shared variable 'counter' in the middle of the operation where Process A tries to increment it.

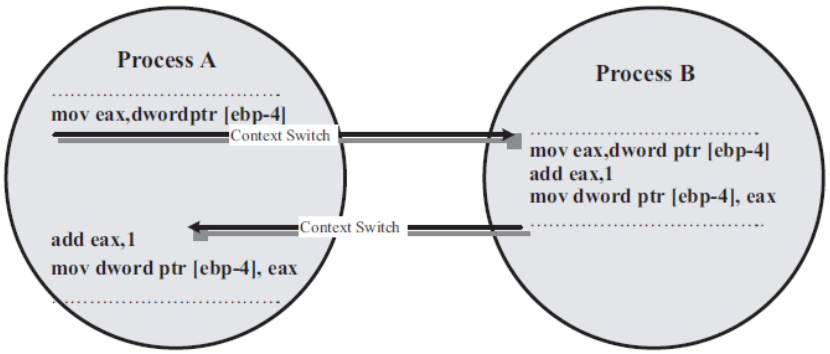


Fig. Race condition

- When Process A gets the CPU time for execution, it starts from the point where it got interrupted (If Process B is also using the same registers eax and ebp for executing counter++; instruction, the original content of these registers will be saved as part of the context saving and it will be retrieved back as part of context retrieval, when process A gets the CPU for execution. Hence the content of eax and ebp remains intact irrespective of context switching).
- Though the variable counter is incremented by Process B, Process A is unaware of it and it increments the variable with the old value.
- This leads to the loss of one increment for the variable counter. This problem occurs due to non-atomic§ operation on variables. This issue wouldn't have been occurred if the underlying actions corresponding to the program statement counter++; is finished in a single CPU execution cycle. The best way to avoid this situation is make the access and modification of shared variables mutually exclusive; meaning when one process accesses a shared variable, prevent the other processes from accessing it.

3.6.1.2 Deadlock

A race condition produces incorrect results whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a set of deadlocked processes.

In its simplest form 'deadlock' is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process (Fig. 10.25).

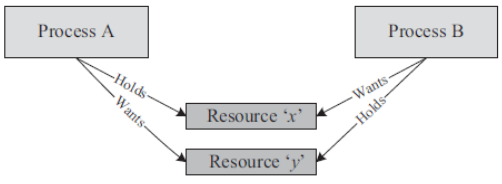


Fig. Scenarios leading to deadlock

To elaborate: Process A holds a resource x and it wants a resource y held by Process B. Process B is currently holding resource y and it wants the resource x which is currently held by Process A. Both hold the respective resources and they compete each other to get the resource held by the respective processes.

The result of the competition is 'deadlock'. None of the competing process will be able to access the resources held by other processes since they are locked by the respective processes (If a mutual exclusion policy is implemented for shared resource access, the resource is locked by the process which is currently accessing it).

The different conditions favouring a deadlock situation are listed below.

Mutual Exclusion: The criteria that only one process can hold a resource at a time. Meaning processes should access shared resources with mutual exclusion. Typical example is the accessing of display hardware in an embedded device.

Hold and Wait: The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.

No Resource Preemption: The criteria that operating system cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.

Circular Wait: A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process. In general, there exists a set of waiting process P0, P1 …Pn with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held by P0, …, Pn is waiting for a resource held by P0 and P0 is waiting for a resource held by Pn and so on… This forms a circular wait queue.
'Deadlock' is a result of the combined occurrence of these four conditions listed above. These conditions are first described by E. G. Coffman in 1971 and it is popularly known as Coffman conditions.

Deadlock Handling A smart OS may foresee the deadlock condition and will act proactively to avoid such a situation. Now if a deadlock occurred, how the OS responds to it? The reaction to deadlock condition by OS is nonuniform. The OS may adopt any of the following techniques to detect and prevent deadlock conditions.
Ignore Deadlocks: Always assume that the system design is deadlock free. This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of happening a deadlock. UNIX is an example for an OS following this principle. A life critical system cannot pretend that it is deadlock free for any reason.
Detect and Recover: This approach suggests the detection of a deadlock situation and recovery from it. This is similar to the deadlock condition that may arise at a traffic junction. Once a deadlock (traffic jam) is happened at the junction, the only solution is to back up the vehicles from one direction and allow the vehicles from opposite direction to cross the junction. If the traffic is too high, lots of vehicles may have to be backed up to resolve the traffic jam. This technique is also known as 'back up cars' technique.

Operating systems keep a resource graph in their memory. The resource graph is updated on each resource request and release. A deadlock condition can be detected by analysing the resource graph by graph analyser algorithms. Once a deadlock condition is detected, the system can terminate a process or preempt the resource to break the deadlocking cycle.

Avoid Deadlocks: Deadlock is avoided by the careful resource allocation techniques by the Operating System. It is similar to the traffic light mechanism at junctions to avoid the traffic jams.

Prevent Deadlocks: Prevent the deadlock condition by negating one of the four conditions favouring the deadlock situation.

- Ensure that a process does not hold any other resources when it requests a resource. This can be achieved by implementing the following set of rules/guidelines in allocating resources to processes.

  1. A process must request all its required resource and the resources should be allocated before the process begins its execution.

  2. Grant resource allocation requests from processes only if the process does not hold a resource currently.

- Ensure that resource preemption (resource releasing) is possible at operating system level. This can be achieved by implementing the following set of rules/guidelines in resources allocation and releasing.

  1. Release all the resources currently held by a process if a request made by the process for a new resource is not able to fulfil immediately.

  2. Add the resources which are preempted (released) to a resource list describing the resources which the process requires to complete its execution.

  3. Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

Imposing these criterions may introduce negative impacts like low resource utilisation and starvation of processes.

Livelock The condition is similar to the deadlock condition except that a process in livelock condition changes its state with time. While in deadlock a process enters in wait state for a resource and continues in that state forever without making any progress in the execution, in a livelock condition a process always does something but is unable to make any progress in the execution completion.

The livelock condition is better explained with the real world example, two people attempting to cross each other in a narrow corridor. Both the persons move towards each side of the corridor to allow the opposite person to cross. Since the corridor is narrow, none of them are able to cross each other. Here both of the persons perform some action but still they are unable to achieve their target, cross each other.

Starvation In the multitasking context, starvation is the condition in which a process does not get the resources required to continue its execution for a long time. As time progresses the process starves on resource. Starvation may arise due to various conditions like byproduct of preventive measure of deadlock, scheduling policies favouring high priority tasks and tasks with shortest execution time, etc.

3.7 HOW TO CHOOSE AN RTOS

A lot of factors needs to be analysed carefully before making a decision on the selection of an RTOS. These factors can be either functional or nonfunctional.

3.7.1 Functional Requirements

Processor Support It is not necessary that all RTOS's support all kinds of processor architecture. It is essential to ensure the processor support by the RTOS.

Memory Requirements The OS requires ROM memory for holding the OS fi les and it is normally stored in a non-volatile memory like FLASH. OS also requires working memory RAM for loading the

OS services. Since embedded systems are memory constrained, it is essential to evaluate the minimal ROM and RAM requirements for the OS under consideration.

Real- me Capabilities It is not mandatory that the operating system for all embedded systems need to be Real-time and all embedded Operating systems are 'Real-time' in behaviour. The task/process scheduling policies plays an important role in the 'Real-time' behaviour of an OS. Analyse the real-time capabilities of the OS under consideration and the standards met by the operating system for real-time capabilities.

Kernel and Interrupt Latency The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency. For an embedded system whose response requirements are high, this latency should be minimal.

Inter Process Communication and Task Synchronisation The implementation of Inter Process Communication and Synchronisation is OS kernel dependent. Certain kernels may provide a bunch of options whereas others provide very limited options. Certain kernels implement policies for avoiding priority inversion issues in resource sharing.

Modularisation Support Most of the operating systems provide a bunch of features. At times it may not be necessary for an embedded product for its functioning. It is very useful if the OS supports modularization where in which the developer can choose the essential modules and re-compile the OS image for functioning. Windows CE is an example for a highly modular operating system.

Support for Networking and Communication The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking. Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.

Development Language Support Certain operating systems include the run time libraries required for running applications written in languages like Java and C#. A Java Virtual Machine (JVM) customised for the Operating System is essential for running java applications. Similarly the .NET Compact Framework (.NETCF) is required for running Microsoft® .NET applications on top of the Operating System. The OS may include these components as built-in component, if not, check the availability of the same from a third party vendor for the OS under consideration.

3.7.2 Non-functional Requirements

Custom Developed or Off the Shelf Depending on the OS requirement, it is possible to go for the complete development of an operating system suiting the embedded system needs or use an off the shelf, readily available operating system, which is either a commercial product or an Open Source product, which is in close match with the system requirements. Sometimes it may be possible to build the required features by customising an Open source OS. The decision on which to select is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.

Cost The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.

Development and Debugging Tools Availability The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design. Certain Operating

Systems may be superior in performance, but the availability of tools for supporting the development may be limited. Explore the different tools available for the OS under consideration.

Ease of Use How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.

After Sales For a commercial embedded RTOS, after sales in the form of e-mail, on-call services, etc. for bug fixes, critical patch updates and support for production issues, etc. should be analysed thoroughly.

3.8 INTEGRATION AND TESTING OF EMBEDDED HARDWARE AND FIRMWARE

Integration testing of the embedded hardware and firmware is the immediate step following the embedded hardware and firmware development. The final embedded hardware constitute of a PCB with all necessary components affixed to it as per the original schematic diagram.

Embedded firmware represents the control algorithm and configuration data necessary to implement the product requirements on the product. Embedded firmware will be in a target processor/controller understandable format called machine language (sequence of 1s and 0s–Binary).

The target embedded hardware without embedding the firmware is a dumb device and cannot function properly. If you power up the hardware without embedding the firmware, the device may behave in an unpredicted manner.

3.8.1 INTEGRATION OF HARDWARE AND FIRMWARE

Integration of hardware and firmware deals with the embedding of firmware into the target hardware board. It is the process of 'Embedding Intelligence' to the product. The embedded processors/controllers used in the target board may or may not have built in code memory.

For non-operating system based embedded products, if the processor/controller contains internal memory and the total size of the firmware is fitting into the code memory area, the code memory is downloaded into the target controller/processor.

If the processor/controller does not support built in code memory or the size of the firmware is exceeding the memory size supported by the target processor/controller, an external dedicated EPROM/FLASH memory chip is used for holding the firmware.

This chip is interfaced to the processor/ controller. (The type of firmware storage, either processor storage or external storage is decided at the time of hardware design by taking the firmware complexity into consideration).

A variety of techniques are used for embedding the firmware into the target board. The commonly used fi rmware embedding techniques for a non-OS based embedded system. The non-OS based embedded systems store the firmware either in the onchip processor/controller memory or offchip memory (FLASH/NVRAM, etc.).

3.8.1.1 Out-of-Circuit Programming

Out-of-circuit programming is performed outside the target board. The processor or memory chip into which the firmware needs to be embedded is taken out of the target board and it is programmed with the

help of a programming device (Fig). The programming device is a dedicated unit which contains the necessary hardware circuit to generate the programming signals.

Fig. Firmware Embedding Tool-Device Programmer: LabTool-48UXP)
(Courtesy of Burn Technology Limited www.burntec.com & Advantech Equipment Corp www.aec.com.tw)

Most of the programmer devices available in the market are capable of programming different family of devices with different pin outs (Pin counts). The programmer contains a ZIF socket with locking pin to hold the device to be programmed. The programming device will be under the control of a utility program running on a PC. Usually the programmer is interfaced to the PC through RS-232C/USB/Parallel Port Interface. The commands to control the programmer are sent from the utility program to the programmer through the interface (Fig).

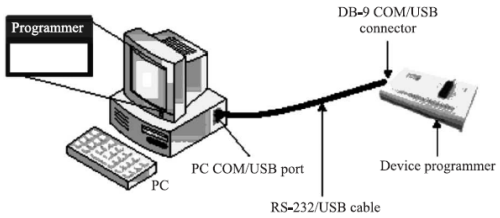


Fig. Interfacing of Device Programmer with PC

The sequence of operations for embedding the firmware with a programmer is listed below.
1. Connect the programming device to the specified port of PC (USB/COM port/parallel port)
2. Power up the device (Most of the programmers incorporate LED to indicate Device power up. Ensure that the power indication LED is ON)
3. Execute the programming utility on the PC and ensure proper connectivity is established between PC and programmer. In case of error, turn off device power and try connecting it again
4. Unlock the ZIF socket by turning the lock pin
5. Insert the device to be programmed into the open socket as per the insert diagram shown on the programmer
6. Lock the ZIF socket
7. Select the device name from the list of supported devices
8. Load the hex fi le which is to be embedded into the device
9. Program the device by 'Program' option of utility program
10. Wait till the completion of programming operation (Till busy LED of programmer is off)
11. Ensure that programming is successful by checking the status LED on the programmer (Usually 'Green' for success and 'Red' for error condition) or by noticing the feedback from the utility program
12. Unlock the ZIF socket and take the device out of programmer

Now the firmware is successfully embedded into the device. Insert the device into the board, power up the board and test it for the required functionalities. It is to be noted that the most of programmers

support only Dual Inline Package (DIP) chips, since its ZIF socket is designed to accommodate only DIP chips. Hence programming of chips with other packages is not possible with the current setup. Adaptor sockets which convert a non-DIP package to DIP socket can be used for programming such chips. One side of the Adaptor socket contains a DIP interface and the other side acts as a holder for holding the chip with a non- DIP package (say VQFP).

Option for setting firmware protection will be available on the programming utility. If you really want the firmware to be protected against unwanted external access, and if the device is supporting memory protection, enable the memory protection on the utility before programming the device.

The programmer usually erases the existing content of the chip before programming the chip. Only EEPROM and FLASH memory chips are erasable by the programmer. Some old embedded systems may be built around UVEPROM chips and such chips should be erased using a separate 'UV Chip Eraser' before programming.

The major drawback of out-of-circuit programming is the high development time. Whenever the firmware is changed, the chip should be taken out of the development board for re-programming. This is tedious and prone to chip damages due to frequent insertion and removal.

Better use a socket on the board side to hold the chip till the firmware modifications are over. The programmer facilitates programming of only one chip at a time and it is not suitable for batch production. Using a 'Gang Programmer' resolves this limitation to certain extent.

A gang programmer is similar to an ordinary programmer except that it contains multiple ZIF sockets (say 4 to 8) and capable of programming multiple devices at a time. But it is bit expensive compared to an ordinary programmer. Another big drawback of this programming technique is that once the product is deployed in the market in a production environment, it is very difficult to upgrade the firmware.

The out-of-system programming technique is used for firmware integration for low end embedded products which runs without an operating system. Out-of-circuit programming is commonly used for development of low volume products and Proof of Concept (PoC) product Development.

3.8.1.2 In System Programming (ISP)

With ISP, programming is done 'within the system', meaning the firmware is embedded into the target device without removing it from the target board. It is the most flexible and easy way of firmware embedding. The only pre-requisite is that the target device must have an ISP support.

Apart from the target board, PC, ISP cable and ISP utility, no other additional hardware is required for ISP. Chips supporting ISP generates the necessary programming signals internally, using the chip's supply voltage.

The target board can be interfaced to the utility program running on PC through Serial Port/Parallel Port/USB. The communication between the target device and ISP utility will be in a serial format. The serial protocols used for ISP may be 'Joint Test Action Group (JTAG)' or ' Serial Peripheral Interface (SPI)' or any other proprietary protocol.

In order to perform ISP operations the target device (in most cases the target device is a microcontroller/microprocessor) should be powered up in a special 'ISP mode'. ISP mode allows the device to communicate with an external host through a serial interface, such as a PC or terminal. The device receives commands and data from the host, erases and reprograms code memory according to the

received command. Once the ISP operations are completed, the device is re-confi gured so that it will operate normally by applying a reset or a re-power up.

3.8.1.2.1 In System Programming with SPI Protocol

Devices with SPI In System Programming support contains a built-in SPI interface and the on-chip EEPROM or FLASH memory is programmed through this interface. The primary I/O lines involved in SPI – In System Programming are listed below.

MOSI – Master Out Slave In
MISO – Master In Slave Out
SCK – System Clock
RST – Reset of Target Device
GND – Ground of Target Device

PC acts as the master and target device acts as the slave in ISP. The program data is sent to the MOSI pin of target device and the device acknowledgement is originated from the MISO pin of the device. SCK pin acts as the clock for data transfer.

A utility program can be developed on the PC side to generate the above signal lines. Since the target device works under a supply voltage less than 5V (TTL/CMOS), it is better to connect these lines of the target device with the parallel port of the PC. Since Parallel port operations are also at 5V logic, no need for any other intermediate hardware for signal conversion.

The pins of parallel port to which the ISP pins of device needs to be connected are dependent on the program, which is used for generating these signals, or you can fix these lines first and then write the program according to the pin inter-connection assignments. Standard SPI-ISP utilities are feely available on the internet and there is no need for going for writing own program.

Connect the pins as mentioned by the program requirement. For ISP operations, the target device needs to be powered up in a pre-defined sequence. The power up sequence for In System Programming for Atmel's AT89S series microcontroller family is listed below.

1. Apply supply voltage between VCC and GND pins of target chip.
2. Set RST pin to "HIGH" state.
3. If a crystal is not connected across pins XTAL1 and XTAL2, apply a 3 MHz to 24 MHz clock to XTAL1 pin and wait for at least 10 milliseconds.
4. Enable serial programming by sending the Programming Enable serial instruction to pin MOSI/P1.5. The frequency of the shift clock supplied at pin SCK/P1.7 needs to be less than the CPU clock at XTAL1 divided by 40.
5. The Code or Data array is programmed one byte at a time by supplying the address and data together with the appropriate Write instruction. The selected memory location is fi rst erased before the new data is written. The write cycle is self-timed and typically takes less than 2.5 ms at 5V.
6. Any memory location can be verifi ed by using the Read instruction, which returns the content at the selected address at serial output MISO/P1.6.
7. After successfully programming the device, set RST pin low or turn off the chip power supply and turn it ON to commence the normal operation.

Note: This sequence is applicable only to Atmel AT89S Series microcontroller and it need not be the same for other series or family of microcontroller/ISP device. Please refer to the datasheet of the device which needs to be programmed using ISP technique for the sequence of operations.

The key player behind ISP is a factory programmed memory (ROM) called 'Boot ROM'. The Boot ROM normally resides at the top end of code memory space and it varies in the order of a few Kilo Bytes (For a controller with 64K code memory space and 1K Boot ROM, the Boot ROM resides at memory location FC00H to FFFFH). It contains a set of Low-level Instruction APIs and these APIs allow the processor/ controller to perform the FLASH memory programming, erasing and Reading operations. The contents of the Boot ROM are provided by the chip manufacturer and the same is masked into every device. The Boot ROM for different family or series devices is different. By default the Reset vector starts the code memory execution at location 0000H. If the ISP mode is enabled through the special ISP Power up sequence, the execution will start at the Boot ROM vector location. In System Programming technique is the most recommended programming technique for development work since the effort required to re-program the device in case of firmware modification is very little. Firmware upgrades for products supporting ISP is quite simple.

3.8.1.3 In Application Programming (IAP)

In Application Programming (IAP) is a technique used by the firmware running on the target device for modifying a selected portion of the code memory. It is not a technique for first time embedding of user written firmware.

It modifies the program code memory under the control of the embedded application. Updating calibration data, look-up tables, etc., which are stored in code memory, are typical examples of IAP. The Boot ROM resident API instructions which perform various functions such as programming, erasing, and reading the Flash memory during ISP mode, are made available to the end-user written firmware for IAP.

Thus it is possible for an end-user application to perform operations on the Flash memory. A common entry point to these API routines is provided for interfacing them to the end-user's application. Functions are performed by setting up specific registers as required by a specific operation and performing a call to the common entry point.

Like any other subroutine call, after completion of the function, control will return to the enduser's code. The Boot ROM is shadowed with the user code memory in its address range. This shadowing is controlled by a status bit. When this status bit is set, accesses to the internal code memory in this address range will be from the Boot ROM.

When cleared, accesses will be from the user's code memory. Hence the user should set the status bit prior to calling the common entry point for IAP operations (The IAP technique described here is for PHILIPS' 89C51RX series microcontroller. Though the underlying principle in IAP is the same, the shadowing technique used for switching access between Boot ROM and code memory may be different for other family of devices).

3.8.1.4 Use of Factory Programmed Chip

It is possible to embed the firmware into the target processor/controller memory at the time of chip fabrication itself. Such chips are known as ' Factory programmed chips'. Once the firmware design is over and the firmware achieved operational stability, the firmware fi les can be sent to the chip fabricator to embed it

into the code memory. Factory programmed chips are convenient for mass production applications and it greatly reduces the product development time. It is not recommended to use factory programmed chips for development purpose where the firmware undergoes frequent changes. Factory programmed ICs are bit expensive.

3.8.1.5 Firmware Loading for Operating System Based Devices

The OS based embedded systems are programmed using the In System Programming (ISP) technique. OS based embedded systems contain a special piece of code called 'Boot loader' program which takes control of the OS and application firmware embedding and copying of the OS image to the RAM of the system for execution.

The ' Boot loader' for such embedded systems comes as pre-loaded or it can be loaded to the memory using the various interface supported like JTAG.

The bootloader contains necessary driver initialisation implementation for initialising the supported interfaces like UART/I2C, TCP/IP etc.

Bootloader implements menu options for selecting the source for OS image to load (Typical menu item examples are Load from FLASH ROM, Load from Network, Load through UART etc).

In case of the network based loading, the bootloader broadcasts the target's presence over the network and the host machine on which the OS image resides can identify the target device by capturing this message.

Once a communication link is established between the host and target machine, the OS image can be directly downloaded to the FLASH memory of the target device.

3.8.2 BOARD BRING UP

Bring up of prototype/evaluation/production version is one of the most important steps in the embedded product design. The prototype/evaluation/production version must pass through a varied set of tests to verify that the embedded hardware is rock solid and the firmware functions as expected.

The term bring up in embedded product design deals with performing a series of validations in a controlled environment, in a well-defined sequence which is specifically tailored for the product. Bring up process usually includes basic hardware spot checks/validations to make sure that the individual components and buses/interconnects are operational – which involves checking power, clocks, and basic functional connectivity; basic firmware verification to make sure that the processor is fetching the code and the firmware execution is happening in the expected manner and then running advanced validations such as memory validations, signal integrity validation etc.

When the hardware PCB is assembled, most designers, in their eagerness to get the project rolling, will try to power it on immediately. But before that there are a few spot checks that can quickly identify problems that can damage the hardware when powering it on.

The first check is to look over the board and check for any missing parts, parts that are loose or partly soldered or shorts in the path, especially around the main components.

Now make sure that various power lines in the board (2.2V, 3.3V, 5.0V etc.) are up, are within the spec limits and are clean and ripple free. Certain controllers/processors which has multiple power input will have power sequencing requirement to apply the power in a predefined order. Make sure that the power is applied in the proper sequence.

Clocks play a critical role in the boot up of the system. In addition to power rails, CPUs, chipsets and other components need reference clocks in order to start up. Make sure that all the clocks are up and running and they are meeting the spec values. Monitor the different interconnect buses (Like I2C) to ensure that they are meeting the electrical specifications as well as the protocol requirements.

3.9 THE EMBEDDED SYSTEM DEVELOPMENT ENVIRONMENT

The various tools used and the various steps followed in embedded system development are explained here typical embedded system development environment is illustrated in Fig.

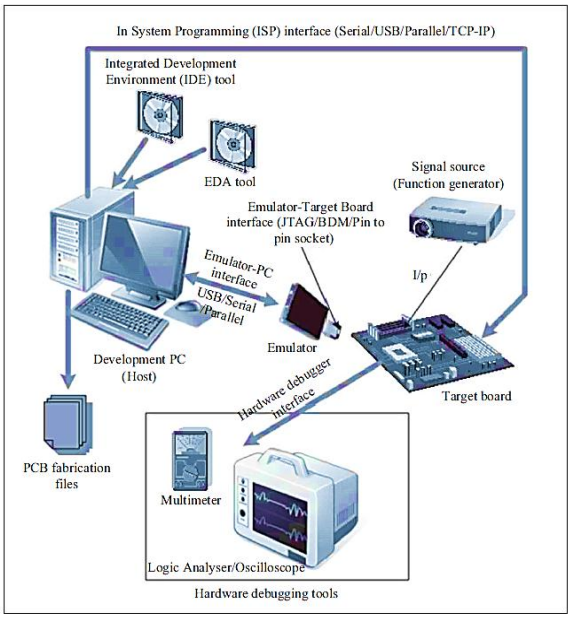


Fig. The Embedded System Development Environment

The development environment consists of a Development Computer (PC) or Host, which acts as the heart of the development environment, Integrated Development Environment (IDE) Tool for embedded firmware development and debugging, Electronic Design Automation (EDA) Tool for Embedded Hardware design, An emulator hardware for debugging the target board, Signal sources (like Function generator) for simulating the inputs to the target board, Target hardware debugging tools (Digital CRO, Multimeter, Logic Analyser, etc.) and the target hardware.

The Integrated Development Environment (IDE) and Electronic Design Automation (EDA) tools are selected based on the target hardware development requirement and they are supplied as Installable fi les in CDs/Online downloads by vendors. These tools need
to be installed on the host PC used for development activities. These tools can be either freeware or licensed copy or evaluation versions. Licensed versions of the tools are fully featured and fully functional whereas trial versions fall into two categories, tools with limited features, and full featured copies with limited period of usage.