FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

INTRODUCTION TO EMBEDDED SYSTEMS

Shibu K V

Technical Architect
Mobility & Embedded Systems Practice
Infosys Technologies Ltd.,
Trivandrum Unit, Kerala

McGraw Hill Education (India) Private Limited
NEW DELHI

McGraw Hill Education Offices
New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

https://hemanthrajhemu.github.io

Keywords 221
Objective Questions 222
Review Questions 223
Lab Assignments 224

Part 2
Design and Development of Embedded Product

8. Embedded Hardware Design and Development 228
8.1 Analog Electronic Components *229*
8.2 Digital Electronic Components *230*
8.3 VLSI and Integrated Circuit Design *243*
8.4 Electronic Design Automation (EDA) Tools *248*
8.5 How to use the OrCAD EDA Tool? *249*
8.6 Schematic Design using Orcad Capture CIS *249*
8.7 The PCB Layout Design *267*
8.8 Printed Circuit Board (PCB) Fabrication *288*
Summary 294
Keywords 294
Objective Questions 296
Review Questions 298
Lab Assignments 299

9. Embedded Firmware Design and Development 302
9.1 Embedded Firmware Design Approaches *303*
9.2 Embedded Firmware Development Languages *306*
9.3 Programming in Embedded C *318*
Summary 371
Keywords 372
Objective Questions 373
Review Questions 378
Lab Assignments 380

10. Real-Time Operating System (RTOS) based Embedded System Design 381
10.1 Operating System Basics *382*
10.2 Types of Operating Systems *386*
10.3 Tasks, Process and Threads *390*
10.4 Multiprocessing and Multitasking *402*
10.5 Task Scheduling *404*
10.6 Threads, Processes and Scheduling: Putting them Altogether *422*
10.7 Task Communication *426*
10.8 Task Synchronisation *442*
10.9 Device Drivers *476*
10.10 How to Choose an RTOS *478*
Summary 480
Keywords 481
Objective Questions 483

https://hemanthrajhemu.github.io

Review Questions 492
Lab Assignments 496

11. An Introduction to Embedded System Design with VxWorks and MicroC/OS-II RTOS **498**
11.1 VxWorks *499*
11.2 MicroC/OS-II *514*
Summary 541
Keywords 542
Objective Questions 543
Review Questions 544
Lab Assignments 546

12. Integration and Testing of Embedded Hardware and Firmware **548**
12.1 Integration of Hardware and Firmware *549*
12.2 Board Power Up *553*
Summary 554
Keywords 554
Review Questions 555

13. The Embedded System Development Environment **556**
13.1 The Integrated Development Environment (IDE) *557*
13.2 Types of Files Generated on Cross-compilation *588*
13.3 Disassembler/Decompiler *597*
13.4 Simulators, Emulators and Debugging *598*
13.5 Target Hardware Debugging *606*
13.6 Boundary Scan *608*
Summary 610
Keywords 611
Objective Questions 612
Review Questions 612
Lab Assignments 613

14. Product Enclosure Design and Development **615**
14.1 Product Enclosure Design Tools *616*
14.2 Product Enclosure Development Techniques *616*
14.3 Summary *618*
Summary 618
Keywords 619
Objective Questions 620
Review Questions 620

15. The Embedded Product Development Life Cycle (EDLC) **621**
15.1 What is EDLC? *622*
15.2 Why EDLC *622*
15.3 Objectives of EDLC *622*
15.4 Different Phases of EDLC *625*

https://hemanthrajhemu.github.io

10 Real-Time Operating System (RTOS) based Embedded System Design

LEARNING OBJECTIVES

- ✓ *Learn the basics of an operating system and the need for an operating system*
- ✓ *Learn the internals of Real-Time Operating System and the fundamentals of RTOS based embedded firmware design*
- ✓ *Learn the basic kernel services of an operating system*
- ✓ *Learn about the classification of operating systems*
- ✓ *Learn about the different real-time kernels and the features that make a kernel Real-Time*
- ✓ *Learn about tasks, processes and threads in the operating system context*
- ✓ *Learn about the structure of a process, the different states of a process, process life cycle and process management*
- ✓ *Learn the concept of multithreading, thread standards and thread scheduling*
- ✓ *Understand the difference between multiprocessing and multitasking,*
- ✓ *Learn about the different types of multitasking (Co-operative, Preemptive and Non-preemptive)*
- ✓ *Learn about the FCFS/FIFO, LCFS/LIFO, SJF and priority based task/process scheduling*
- ✓ *Learn about the shortest remaining time (SRT), Round Robin and priority based preemptive task/process scheduling*
- ✓ *Learn about the different Inter Process Communication (IPC) mechanisms used by tasks/process to communicate and co-operate each other in a multitasking environment*
- ✓ *Learn the different types of shared memory techniques (Pipes, memory mapped object, etc.) for IPC*
- ✓ *Learn the different types of message passing techniques (Message queue, mailbox, signals, etc.) for IPC*
- ✓ *Learn the RPC based Inter Process Communication*
- ✓ *Learn the need for task synchronisation in a multitasking environment*
- ✓ *Learn the different issues related to the accessing of a shared resource by multiple processes concurrently*
- ✓ *Learn about 'Racing', 'Starvation', 'Livelock', 'Deadlock', 'Dining Philosopher's Problem', 'Producer-Consumer/Bounded Buffer Problem', 'Readers-Writers Problem' and 'Priority Inversion'*
- ✓ *Learn about the 'Priority Inheritance' and 'Priority Ceiling' based Priority avoidance mechanisms*
- ✓ *Learn the need for task synchronisation and the different mechanisms for task synchronisation in a multitasking environment*
- ✓ *Learn about mutual exclusion and the different policies for mutual exclusion implementation*
- ✓ *Learn about semaphores, different types of semaphores, mutex, critical section objects and events for task synchronisation*
- ✓ *Learn about device drivers, their role in an operating system based embedded system design, the structure of a device driver, and interrupt handling inside device drivers*
- ✓ *Understand the different functional and non-functional requirements that need to be addressed in the selection of a Real-Time Operating System*

https://hemanthrajhemu.github.io

In the previous chapter, we discussed about the *Super loop* based task execution model for firmware execution. The super loop executes the tasks sequentially in the order in which the tasks are listed within the loop. Here every task is repeated at regular intervals and the task execution is non-real time. As the number of task increases, the time intervals at which a task gets serviced also increases. If some of the tasks involve waiting for external events or I/O device usage, the task execution time also gets pushed off in accordance with the 'wait' time consumed by the task. The priority in which a task is to be executed is fixed and is determined by the task placement within the loop, in a super loop based execution. This type of firmware execution is suited for embedded devices where response time for a task is not time critical. Typical examples are electronic toys and video gaming devices. Here any response delay is acceptable and it will not create any operational issues or potential hazards. Whereas certain applications demand time critical response to tasks/events and any delay in the response may become catastrophic. Flight Control systems, Air bag control and Anti Locking Brake (ABS) systems for vehicles, Nuclear monitoring devices, etc. are typical examples of applications/devices demanding time critical task response.

How the increasing need for time critical response for tasks/events is addressed in embedded applications? Well the answer is

1. Assign priority to tasks and execute the high priority task when the task is ready to execute.
2. Dynamically change the priorities of tasks if required on a need basis.
3. Schedule the execution of tasks based on the priorities.
4. Switch the execution of task when a task is waiting for an external event or a system resource including I/O device operation.

The introduction of operating system based firmware execution in embedded devices can address these needs to a greater extent.

10.1 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 efficiently and correctly

Figure 10.1 gives an insight into the basic components of an operating system and their interfaces with rest of the world.

10.1.1 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 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

https://hemanthrajhemu.github.io

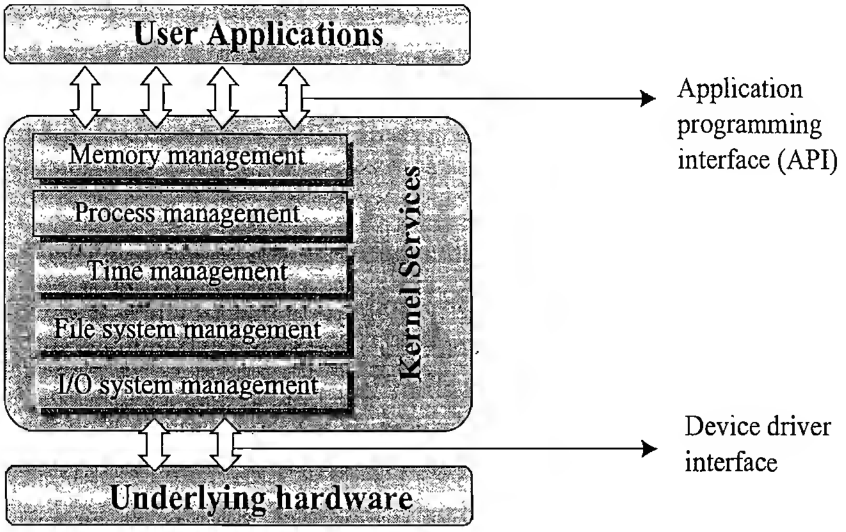


Fig. 10.1 The Operating System Architecture

up and managing the Process Control Block (PCB), Inter Process Communication and synchronisation, process termination/deletion, etc. We will look into the description of process and process management in a later section of this chapter.

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

https://hemanthrajhemu.github.io

(e.g. Windows XP kernel keeps the list updated when a new plug '*n*' play USB device is attached to the system). The service 'Device Manager' (Name may vary across different OS kernels) 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 The 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 XP 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.

10.1.1.1 Kernel Space and User Space As we discussed in the earlier section, the applications/ services are classified into two categories, namely: user applications and kernel applications. 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 un-authorised 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.

https://hemanthrajhemu.github.io

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.

10.1.1.2 Monolithic Kernel and Microkernel As we know, the kernel forms the heart of an operating system. Different approaches are adopted for building an Operating System kernel. 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. 10.2.

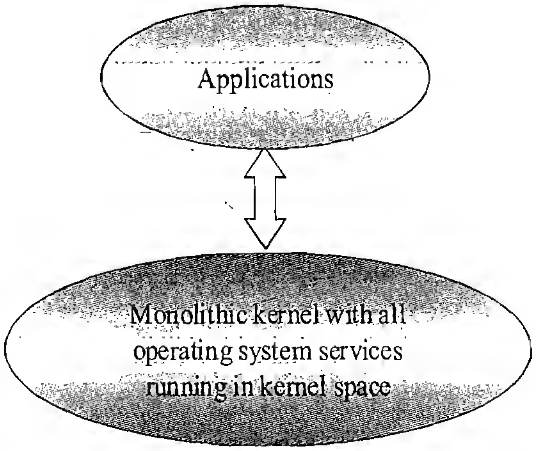

Applications
Monolithic kernel with all operating system services running in kernel space

Fig. 10.2 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

https://hemanthrajhemu.github.io

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.

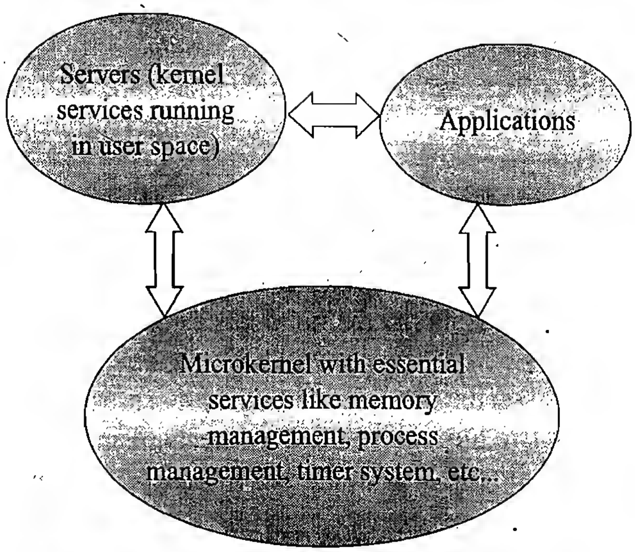

Servers (kernel services running in user space)
Applications
Microkernel with essential services like memory management, process management, timer system, etc...

Fig. 10.3 **The Microkernel model**

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.

10.2 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.

10.2.1 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 XP/MS-DOS etc. are examples for General Purpose Operating Systems.

10.2.2 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. What '*Real-Time*' means in Operating System context is still a debatable topic and there are many definitions available. 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

https://hemanthrajhemu.github.io

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 CE, QNX, VxWorks MicroC/OS-II, etc. are examples of Real-Time Operating Systems (RTOS).

10.2.2.1 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. We will discuss the various types of scheduling in a later section of this chapter.

https://hemanthrajhemu.github.io

Task/Process Synchronisation Deals with synchronising the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks. We will discuss the various synchronisation techniques and inter task /process communication in a later section of this chapter.

Error/Exception 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 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. (We will explore this technique under the MicroC/OS-II kernel in a

† *Virtual Memory* is an imaginary memory supported by certain operating systems. Virtual memory expands the address space available to a task beyond the actual physical memory (RAM) supported by the system. Virtual memory is implemented with the help of a Memory Management Unit (MMU) and 'memory paging'. The program memory for a task can be viewed as different pages and the page corresponding to a piece of code that needs to be executed is loaded into the main physical memory (RAM). When a memory page is no longer required, it is moved out to secondary storage memory and another page which contains the code snippet to be executed is loaded into the main memory. This memory movement technique is known as demand paging. The MMU handles the demand paging and converts the virtual address of a location in a page to corresponding physical address in the RAM.

https://hemanthrajhemu.github.io

latter chapter).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 (by asserting the interrupt line of the processor/controller to which the interrupt line of the device is connected) 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

$$2^{32} * 10^{-6} / (24 * 60 * 60) = 49700 \text{ Days} = \sim 0.0497 \text{ Days} = 1.19 \text{ Hours}$$

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

$$2^{32} * 10^{-3} / (24 * 60 * 60) = 497 \text{ Days} = 49.7 \text{ Days} = \sim 50 \text{ Days}$$

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 first 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.

https://hemanthrajhemu.github.io

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.

10.2.2.2 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'.

10.2.2.3 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 Real-time 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.

10.3 TASKS, PROCESS AND THREADS

The term '*task*' refers to something that needs to be done. In our day-to-day life, we are bound to the execution of a number of tasks. The task can be the one assigned by our managers or the one assigned by our professors/teachers or the one related to our personal or family needs. 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

https://hemanthrajhemu.github.io

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*'. The terms '*Task*', '*Job*' and '*Process*' refer to the same entity in the operating system context and most often they are used interchangeably.

10.3.1 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.

10.3.1.1 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 visualised as shown in Fig. 10.4.

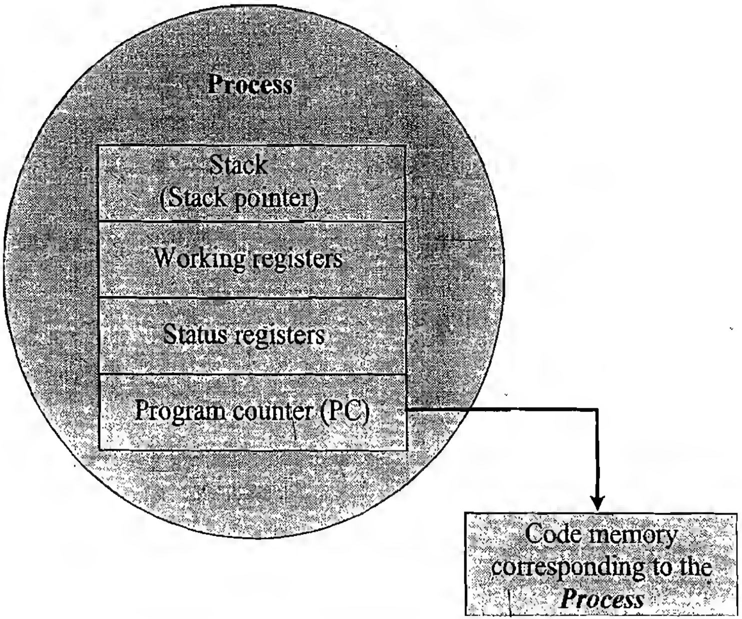


Fig. 10.4 **Structure of a *Process***

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. 10.5).

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

https://hemanthrajhemu.github.io

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.

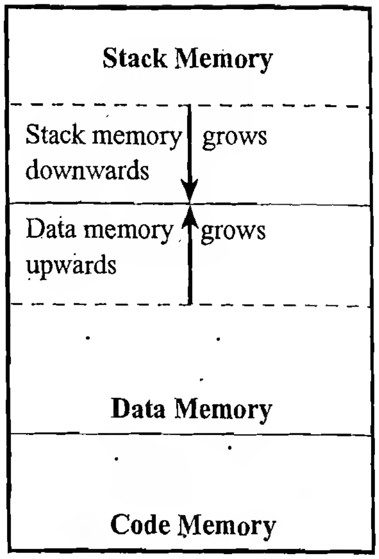


Fig. 10.5 **Memory organisation of a Process**

10.3.1.2 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 10.6 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. 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.

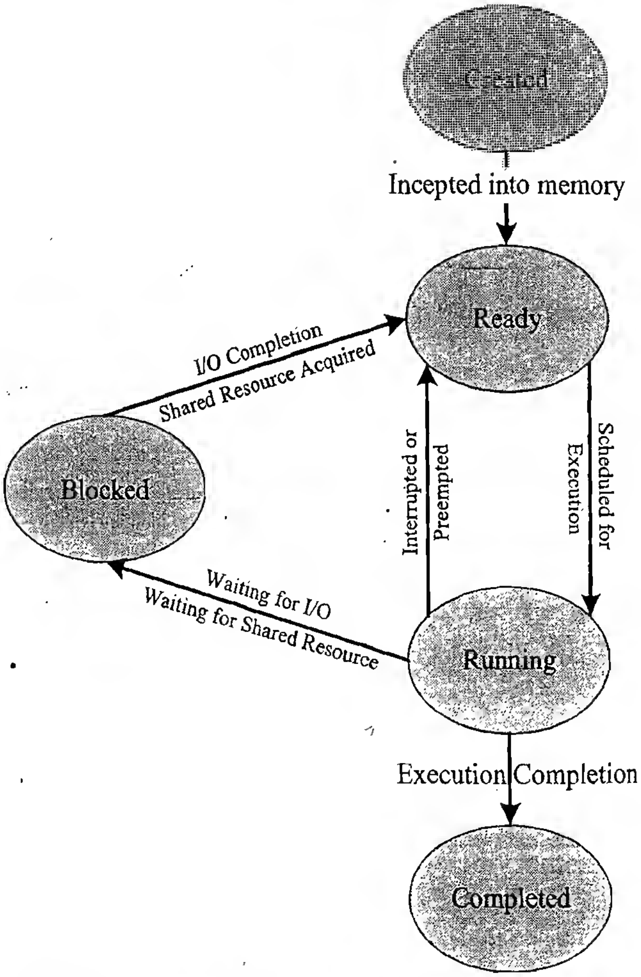


Fig. 10.6 **Process states and state transition representation**

It should be noted that the state representation for a process/task mentioned here is a generic rep-

https://hemanthrajhemu.github.io

resentation. 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 specific 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. We will discuss about the states and state transition for tasks under VxWorks and uC/OS-II kernel in a later chapter.

10.3.1.3 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. For more details on Process Management, refer to the section 'Task/Process management' given under the topic 'The Real-Time Kernel' of this chapter.

10.3.2 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. 10.7.

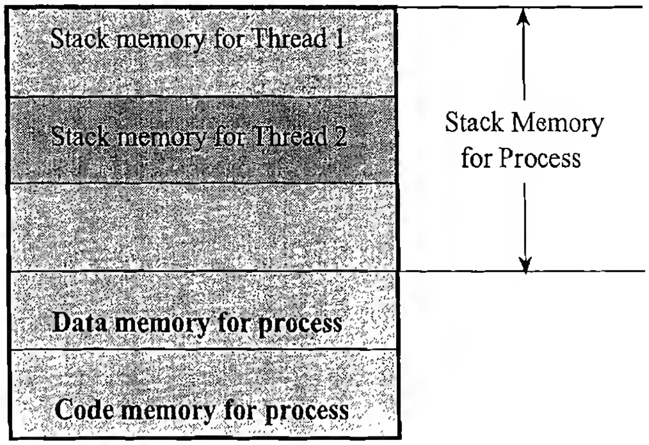
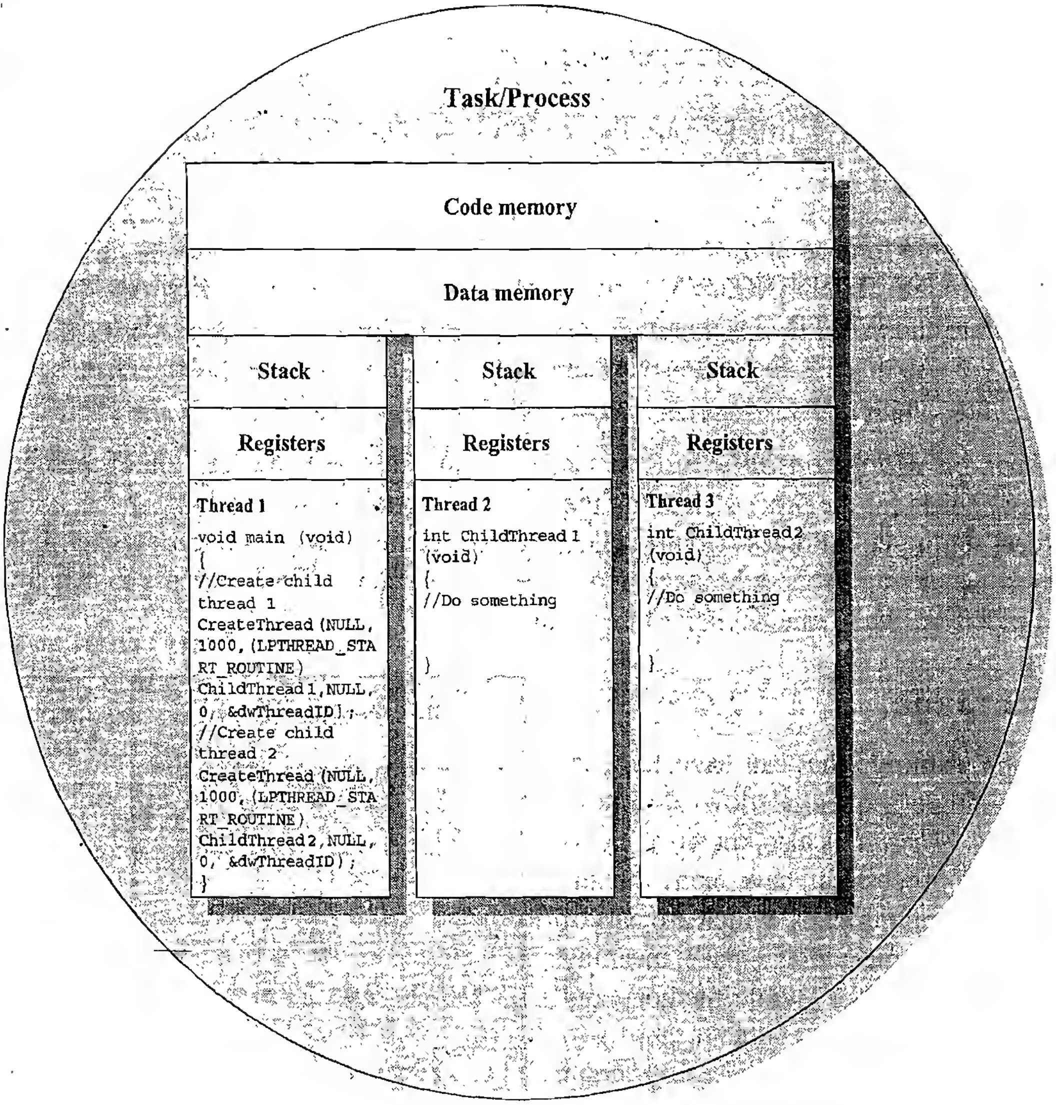


Fig. 10.7 **Memory organisation of a *Process* and its associated *Threads***

10.3.2.1 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 carrying out the different subfunctionalities 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. 10.8.

https://hemanthrajhemu.github.io



```
void main (void)
{
//Create child
thread 1
CreateThread (NULL,
1000, (LPTHREAD_STA
RT_ROUTINE)
ChildThread1,NULL,
0, &dwThreadID);
//Create child
thread 2
CreateThread (NULL,
1000, (LPTHREAD_STA
RT_ROUTINE)
ChildThread2,NULL,
0, &dwThreadID);
}
```

```
int ChildThread1
(void)
{
//Do something

}
```

```
int ChildThread2
(void)
{
//Do something

}
```

Fig. 10.8 ***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.

10.3.2.2 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.

https://hemanthrajhemu.github.io

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;
}
```

https://hemanthrajhemu.github.io

```
//*********************************************************************
//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;
}
```

You can compile this application using the *gcc* compiler. Examine the output to figure out the thread execution switching. The lines printed will give an idea of the order in which the thread execution is switched between. The *pthread_join* call forces the main thread to wait until the completion of the thread *tcb*, if the main thread finishes the execution first.

The termination of a thread can happen in different ways. The thread can terminate either by completing its execution (natural termination) or by a forced termination. In a natural termination, the thread completes its execution and returns back to the main thread through a simple *return* or by executing the *pthread_exit()* call. Forced termination can be achieved by the call *pthread_cancel()* or through the termination of the main thread with *exit* or *exec* functions. *pthread_cancel()* call is used by a thread to terminate another thread.

pthread_exit() call is used by a thread to explicitly exit after it completes its work and is no longer required to exist. If the main thread finishes before the threads it has created, and exits with *pthread_exit()*, the other threads continue to execute. If the main thread uses *exit* call to exit the thread, all threads created by the main thread is terminated forcefully. Exiting a thread with the call *pthread_exit()* will not perform a cleanup. It will not close any files opened by the thread and files will remain in the open status even after the thread terminates. Calling *pthread_join* at the end of the main thread is the best way to achieve synchronisation and proper cleanup. The main thread, after finishing its task waits for the completion of other threads, which were joined to it using the *pthread_join* call. With a *pthread_join* call, the main thread waits other threads, which were joined to it, and finally merges to the single main thread. If a new thread spawned by the main thread is still not joined to the main thread, it will be counted against the system's maximum thread limit. Improper cleanup will lead to the failure of new thread creation.

https://hemanthrajhemu.github.io

Win32 Threads Win32 threads are the threads supported by various flavours of Windows Operating Systems. The Win32 Application Programming Interface (Win32 API) libraries provide the standard set of Win32 thread creation and management functions. Win32 threads are created with the API

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,DWORD dwStack-
Size, LPTHREAD_START_ROUTINE lpStartAddress, LPVOID lpParameter, DWORD dwCre-
ationFlags, LPDWORD lpThreadId );
```

The parameter *lpThreadAttributes* defines the security attributes for the thread and *dwStackSize* defines the stack size for the thread. These two parameters are not supported by the Windows CE Real-Time Operating Systems and it should be kept as NULL and 0 respectively in a *CreateThread* API Call. The other parameters are

lpStartAddress: Pointer to the function which is to be executed by the thread.

lpParameter: Parameter specifying an application-defined value that is passed to the thread routine.

dwCreationFlags: Defines the state of the thread when it is created. Usually it is kept as 0 or CREATE_SUSPENDED implying the thread is created and kept at the suspended state.

lpThreadId: Pointer to a DWORD that receives the identifier for the thread.

On successful creation of the thread, *CreateThread* returns the handle to the thread and the thread identifier.

The API *GetCurrentThread(void)* returns the handle of the current thread and *GetCurrentThreadId(void)* returns its ID. *GetThreadPriority (HANDLE hThread)* API returns an integer value representing the current priority of the thread whose handle is passed as *hThread.* Threads are always created with normal priority (THREAD_PRIORITY_NORMAL. Refer MSDN documentation for the different thread priorities and their meaning). *SetThreadPriority (HANDLE hThread, int nPriority)* API is used for setting the priority of a thread. The first parameter to this function represents the thread handle and the second one the thread priority.

For Win32 threads, the normal thread termination happens when an exception occurs in the thread, or when the thread's execution is completed or when the primary thread or the process to which the thread is associated is terminated. A thread can exit itself by calling the *ExitThread (DWORD dwExitCode)* API. The parameter *dwExitCode* sets the exit code for thread termination. Calling *ExitThread* API frees all the resources utilised by the thread. The exit code of a thread can be checked by other threads by calling the *GetExitCodeThread (HANDLE hThread, LPDWORD lpExitCode). TerminateThread (HANDLE hThread, DWORD dwExitCode)* API is used for terminating a thread from another thread. The handle *hThread* indicates which thread is to be terminated and *dwExitCode* sets the exit code for the thread. This API will not execute the thread termination and clean up code and may not free the resources occupied by the thread. *TerminateThread* is a potentially dangerous call and it should not be used in normal conditions as a mechanism for terminating a thread. Use this call only as a final choice. *SuspendThread(HANDLE hThread)* API can be used for suspending a thread from execution provided the *handle hThread* possesses *THREAD_SUSPEND_RESUME* access right. If the *SuspendThread* API call succeeds, the thread stops executing and increments its internal suspend count. The thread becomes suspended if its suspend count is greater than zero. The *SuspendThread* function is primarily designed for use by debuggers. One must be cautious in using this API for the reason it may cause *deadlock* condition if the thread is suspended at a stage where it acquired a mutex or shared resource and another thread tries to access the same. The *ResumeThread(HANDLE hThread)* API is used for resuming a suspended thread. The *ResumeThread* API checks the suspend count of the specified thread. A suspend count of

https://hemanthrajhemu.github.io

zero indicates that the specified thread is not currently in the suspended mode. If the count is not zero, the count is decremented by one and if the resulting count value is zero, the thread is resumed. The API *Sleep (DWORD dwMilliseconds)* can be used for suspending a thread for the duration specified in milliseconds by the *Sleep* function. The *Sleep* call is initiated by the thread.

Example 2

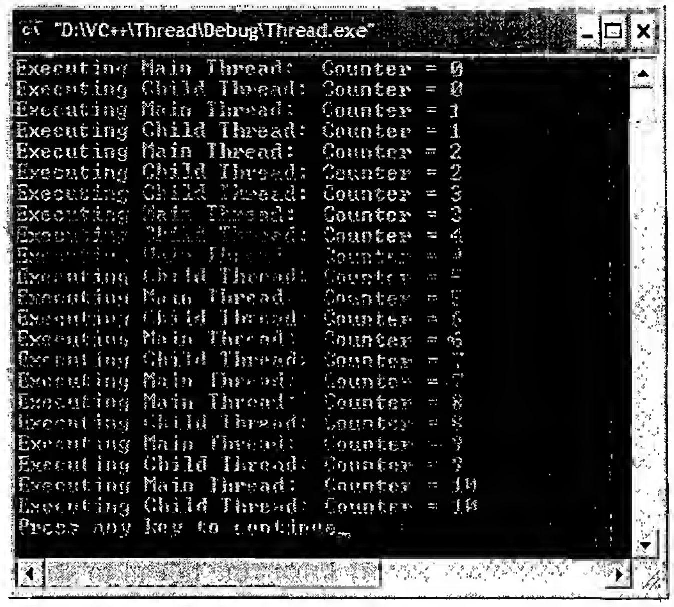
Write a multithreaded application using Win32 APIs to set up a counter in the main thread and secondary thread to count from 0 to 10 and print the counts from both the threads. Put a delay of 500 ms in between the successive printing in both the threads.

```
#include "windows.h"
#include "stdio.h"
//*******************************************************************
//Child thread
//*******************************************************************
void ChildThread(void)
    {
    char i;
    for(i=0;i<=10;++i)
        {
        printf("Executing Child Thread:Counter = %d\n",i);
        Sleep(500);
        }
    }
//*******************************************************************
//Primary thread
//*******************************************************************
int main(int argc, char* argv[])
{
       HANDLE hThread;
       DWORD dwThreadID;
       char i;
       hThread=CreateThread(NULL,1000,(LPTHREAD_START_ROUTINE)
       ChildThread, NULL, 0, &dwThreadID);
    if(hThread==NULL)
       {
       printf("Thread Creation Failed\nError No:
       %d\n",GetLastError());
       return 1;
       }
        for(i=0;i<=10;++i)
        {
        printf("Executing Main Thread: Counter = %d\n",i);
        Sleep(500);
        }
    return 0;
}
```

https://hemanthrajhemu.github.io

To execute this program, create a new Win32 Console Application using Microsoft Visual C++ and add the above piece of code to it and compile. The output obtained on running this application on a machine with Windows XP operating system is given in Fig. 10.9.

If you examine the output, you can see the switching between main and child threads. The output need not be the same always. The output is purely dependent on the scheduling policies implemented by the windows operating system for thread scheduling. You may get the same output or a different output each time you run the application.



```
Executing Main Thread:  Counter = 0
Executing Child Thread: Counter = 0
Executing Main Thread:  Counter = 1
Executing Child Thread: Counter = 1
Executing Main Thread:  Counter = 2
Executing Child Thread: Counter = 2
Executing Child Thread: Counter = 3
Executing Main Thread:  Counter = 3
Executing Child Thread: Counter = 4
Executing Main Thread:  Counter = 4
Executing Child Thread: Counter = 5
Executing Main Thread:  Counter = 5
Executing Child Thread: Counter = 6
Executing Main Thread:  Counter = 6
Executing Child Thread: Counter = 7
Executing Main Thread:  Counter = 7
Executing Main Thread:  Counter = 8
Executing Child Thread: Counter = 8
Executing Main Thread:  Counter = 9
Executing Child Thread: Counter = 9
Executing Main Thread:  Counter = 10
Executing Child Thread: Counter = 10
Press any key to continue_
```

Fig. 10.9 **Output of the Win32 Multithreaded application**

- **Java Threads** Java threads are the threads supported by Java programming Language. The java thread class '*Thread*' is defined in the package '*java.lang*'. This package needs to be imported for using the thread creation functions supported by the Java thread class. There are two ways of creating threads in Java: Either by extending the base '*Thread*' class or by implementing an interface. Extending the thread class allows inheriting the methods and variables of the parent class (Thread class) only whereas interface allows a way to achieve the requirements for a set of classes. The following piece of code illustrates the implementation of Java threads with extending the thread base class '*Thread*'.

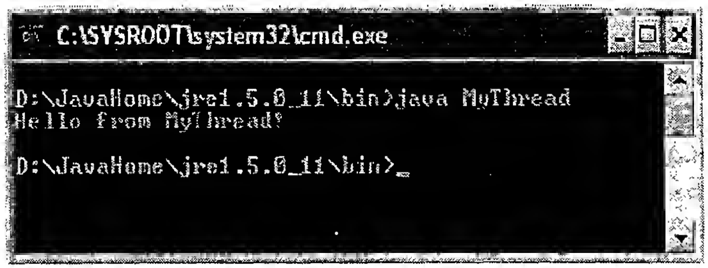
```
import java.lang.*;
public class MyThread extends Thread
{
      public void run()
      {
            System.out.println("Hello from MyThread!");
      }
      public static void main(String args[])
      {
            (new MyThread()).start();
      }
}
```

The above piece of code creates a new class *MyThread* by extending the base class *Thread*. It also overrides the *run()* method inherited from the base class with its own *run()* method. The *run()* method of *MyThread* implements all the task for the *MyThread* thread. The method *start()* moves the thread to a pool of threads waiting for their turn to be picked up for execution by the scheduler. The thread is said to be in the 'Ready' state at this stage. The scheduler picks the threads for execution from the pool based on the thread priorities.

```
E.g.  MyThread.start();
```

The output of the above piece of code when executed on Windows XP platform is given in Fig. 10.10.

https://hemanthrajhemu.github.io



```
D:\JavaHome\jre1.5.0_11\bin>java MyThread
Hello from MyThread!

D:\JavaHome\jre1.5.0_11\bin>_
```

Fig. 10.10 **Output of the Java Multithreaded application**

Invoking the static method *yield()* voluntarily give up the execution of the thread and the thread is moved to the pool of threads waiting to get their turn for execution, i.e. the thread enters the 'Ready' state.

```
E.g. MyThread.yield();
```

The static method *sleep()* forces the thread to sleep for the duration mentioned by the sleep call, i.e. the thread enters the 'Suspend' mode. Once the sleep period is expired, the thread is moved to the pool of threads waiting to get their turn for execution, i.e. the thread enters the 'Ready' state. The method *sleep()* only guarantees that the thread will sleep for the minimum period mentioned by the argument to the call. It will not guarantee anything on the resume of the thread after the sleep period. It is dependent on the scheduler.

```
E.g. MyThread.sleep(100); Sleep for 100 milliseconds.
```

Calling a thread Object's *wait()* method causes the thread object to wait. The thread will remain in the 'Wait' state until another thread invokes the *notify()* or *notifyAll()* method of the thread object which is waiting. The thread enters the 'Blocked' state when waiting for input from I/O devices or waiting for object lock in case of accessing shared resources. The thread is moved to the 'Ready' state on receiving the I/O input or on acquiring the object lock. The thread enters the 'Finished/Dead' state on completion of the task assigned to it or when the *stop()* method is explicitly invoked. The thread may also enter this state if it is terminated by an unrecoverable error condition.

For more information on Java threads, visit Sun Micro System's tutorial on Threads, available at http://java.sun.com/tutorial/applet/overview/threads.html

Summary So far we discussed about the various thread classes available for creation and management of threads in a multithreaded system in a General Purpose Operating System's perspective. From an RTOS perspective, POSIX threads and Win32 threads are the most commonly used thread class libraries for thread creation and management. Many non-standard, proprietary thread classes are also used by some proprietary RTOS. Portable threads (*Pth*), a very portable POSIX/ANSI-C based library from GNU, may be the "next generation" threads library. *Pth* provides non-preemptive priority based scheduling for multiple threads inside event driven applications. Visit http://www.gnu.org/software/pth/ for more details on GNU Portable threads.

10.3.2.3 Thread Pre-emption Thread pre-emption is the act of pre-empting the currently running thread (stopping the currently running thread temporarily). Thread pre-emption ability is solely dependent on the Operating System. Thread pre-emption is performed for sharing the CPU time among all the threads. The execution switching among threads are known as '*Thread context switching*'. Thread context switching is dependent on the Operating system's scheduler and the type of the thread. When we say 'Thread', it falls into any one of the following types.

https://hemanthrajhemu.github.io

User Level Thread User level threads do not have kernel/Operating System support and they exist solely in the running process. Even if a process contains multiple user level threads, the OS treats it as single thread and will not switch the execution among the different threads of it. It is the responsibility of the process to schedule each thread as and when required. In summary, user level threads of a process are non-preemptive at thread level from OS perspective.

Kernel/System Level Thread Kernel level threads are individual units of execution, which the OS treats as separate threads. The OS interrupts the execution of the currently running kernel thread and switches the execution to another kernel thread based on the scheduling policies implemented by the OS. In summary kernel level threads are pre-emptive.

For user level threads, the execution switching (thread context switching) happens only when the currently executing user level thread is voluntarily blocked. Hence, no OS intervention and system calls are involved in the context switching of user level threads. This makes context switching of user level threads very fast. On the other hand, kernel level threads involve lots of kernel overhead and involve system calls for context switching. However, kernel threads maintain a clear layer of abstraction and allow threads to use system calls independently. There are many ways for binding user level threads with system/kernel level threads. The following section gives an overview of various thread binding models.

Many-to-One Model Here many user level threads are mapped to a single kernel thread. In this model, the kernel treats all user level threads as single thread and the execution switching among the user level threads happens when a currently executing user level thread voluntarily blocks itself or relinquishes the CPU. Solaris Green threads and GNU Portable Threads are examples for this. The '*PThread*' example given under the POSIX thread library section is an illustrative example for application with Many-to-One thread model.

One-to-One Model In One-to-One model, each user level thread is bonded to a kernel/system level thread. Windows XP/NT/2000 and Linux threads are examples for One-to-One thread models. The modified '*PThread*' example given under the '*Thread Pre-emption*' section is an illustrative example for application with One-to-One thread model.

Many-to-Many Model In this model many user level threads are allowed to be mapped to many kernel threads. Windows NT/2000 with *ThreadFibre* package is an example for this.

10.3.2.4 Thread v/s Process I hope, by now you got a reasonably good knowledge of *process* and *threads*. Now let us summarise the properties of *process* and *threads*.

Thread	Process
Thread is a single unit of execution and is part of process.	Process is a program in execution and contains one or more threads.
A thread does not have its own data memory and heap memory. It shares the data memory and heap memory with other threads of the same process.	Process has its own code memory, data memory and stack memory.
A thread cannot live independently; it lives within the process.	A process contains at least one thread.
There can be multiple threads in a process. The first thread (main thread) calls the main function and occupies the start of the stack memory of the process.	Threads within a process share the code, data and heap memory. Each thread holds separate memory area for stack (shares the total stack memory of the process).

https://hemanthrajhemu.github.io

Threads are very inexpensive to create	Processes are very expensive to create. Involves many OS overhead.
Context switching is inexpensive and fast	Context switching is complex and involves lot of OS overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also dies.

10.4 MULTIPROCESSING AND MULTITASKING

The terms *multiprocessing* and *multitasking* are a little confusing and sounds alike. In the operating system context *multiprocessing* describes the ability to execute multiple processes simultaneously. Systems which are capable of performing multiprocessing, are known as *multiprocessor* systems. *Multiprocessor* systems possess multiple CPUs and can execute multiple processes simultaneously.

The ability of the operating system to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*. In a uniprocessor system, it is not possible to execute multiple processes simultaneously. However, it is possible for a uniprocessor system to achieve some degree of pseudo parallelism in the execution of multiple processes by switching the execution among different processes. The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as *multitasking*. Multitasking creates the illusion of multiple tasks executing in parallel. Multitasking involves the switching of CPU from executing one task to another. In an earlier section '*The Structure of a Process*' of this chapter, we learned that a Process is identical to the physical processor in the sense it has own register set which mirrors the CPU registers, stack and Program Counter (PC). Hence, a '*process*' is considered as a '*Virtual processor*', awaiting its turn to have its properties switched into the physical processor. In a multitasking environment, when task/process switching happens, the virtual processor (task/process) gets its properties converted into that of the physical processor. The switching of the virtual processor to physical processor is controlled by the scheduler of the OS kernel. Whenever a CPU switching happens, the current context of execution should be saved to retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching. The context saving and retrieval is essential for resuming a process exactly from the point where it was interrupted due to CPU switching. The act of switching CPU among the processes or changing the current execution context is known as '*Context switching*'. The act of saving the current context which contains the context details (Register details, memory details, system resource usage details, execution details, etc.) for the currently running process at the time of CPU switching is known as '*Context saving*'. The process of retrieving the saved context details for a process, which is going to be executed due to CPU switching, is known as '*Context retrieval*'. Multitasking involves '*Context switching*' (Fig. 10.11), '*Context saving*' and '*Context retrieval*'.

Toss juggling – The skilful object manipulation game is a classic real world example for the multitasking illusion. The juggler uses a number of objects (balls, rings, etc.) and throws them up and catches them. At any point of time, he throws only one ball and catches only one per hand. However, the speed at which he is switching the balls for throwing and catching creates the illusion, he is throwing and catching multiple balls or using more than two hands ☺ simultaneously, to the spectators.

https://hemanthrajhemu.github.io

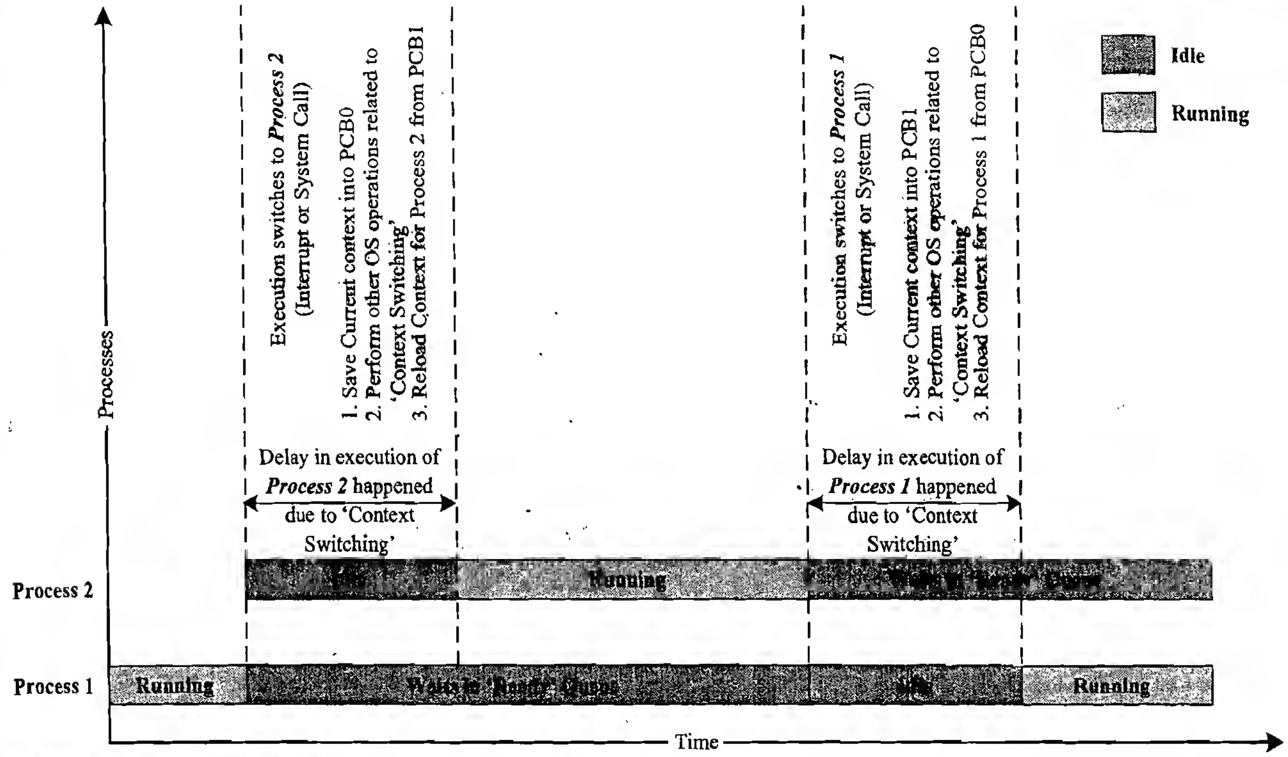

Idle
Running
Processes
Execution switches to Process 2 (Interrupt or System Call)
1. Save Current context into PCB0
2. Perform other OS operations related to 'Context Switching'
3. Reload Context for Process 2 from PCB1
Execution switches to Process 1 (Interrupt or System Call)
1. Save Current context into PCB1
2. Perform other OS operations related to 'Context Switching'
3. Reload Context for Process 1 from PCB0
Delay in execution of Process 2 happened due to 'Context Switching'
Delay in execution of Process 1 happened due to 'Context Switching'
Process 2
Idle
Running
Waits in 'Ready' Queue
Process 1
Running
Waits in 'Ready' Queue
Idle
Running
Time

Fig. 10.11 Context switching

10.4.1 Types of Multitasking

As we discussed earlier, multitasking involves the switching of execution among multiple tasks. Depending on how the switching act is implemented, multitasking can be classified into different types. The following section describes the various types of multitasking existing in the Operating System's context.

10.4.1.1 Co-operative Multitasking Co-operative multitasking is the most primitive form of multitasking in which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU. In this method, any task/process can hold the CPU as much time as it wants. Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, it is known as co-operative multitasking. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU.

10.4.1.2 Preemptive Multitasking Preemptive multitasking ensures that every task/process gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling. As the name indicates, in preemptive multitasking, the currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/process priority.

10.4.1.3 Non-preemptive Multitasking In non-preemptive multitasking, the process/task, which is currently given the CPU time, is allowed to execute until it terminates (enters the '*Completed*' state) or enters the '*Blocked/Wait*' state, waiting for an I/O or system resource. The co-operative and non-preemptive multitasking differs in their behaviour when they are in the '*Blocked/Wait*' state. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the '*Blocked/Wait*' state,

https://hemanthrajhemu.github.io

waiting for an I/O, or a shared resource access or an event to occur whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O or system resource or an event to occur.

10.5 TASK SCHEDULING

As we already discussed, multitasking involves the execution switching among the different tasks. There should be some mechanism in place to share the CPU among the different tasks and to decide which process/task is to be executed at a given point of time. Determining which task/process is to be executed at a given point of time is known as task/process scheduling. Task scheduling forms the basis of multitasking. Scheduling policies forms the guidelines for determining which task is to be executed when. The scheduling policies are implemented in an algorithm and it is run by the kernel as a service. The kernel service/application, which implements the scheduling algorithm, is known as '*Scheduler*'. The process scheduling decision may take place when a process switches its state to

1. '*Ready*' state from '*Running*' state
2. '*Blocked/Wait*' state from '*Running*' state
3. '*Ready*' state from '*Blocked/Wait*' state
4. '*Completed*' state

A process switches to '*Ready*' state from the '*Running*' state when it is preempted. Hence, the type of scheduling in scenario 1 is pre-emptive. When a high priority process in the '*Blocked/Wait*' state completes its I/O and switches to the '*Ready*' state, the scheduler picks it for execution if the scheduling policy used is priority based preemptive. This is indicated by scenario 3. In preemptive/non-preemptive multitasking, the process relinquishes the CPU when it enters the '*Blocked/Wait*' state or the '*Completed*' state and switching of the CPU happens at this stage. Scheduling under scenario 2 can be either preemptive or non-preemptive. Scheduling under scenario 4 can be preemptive, non-preemptive or co-operative.

The selection of a scheduling criterion/algorithm should consider the following factors:

CPU Utilisation: The scheduling algorithm should always make the CPU utilisation high. CPU utilisation is a direct measure of how much percentage of the CPU is being utilised.

Throughput: This gives an indication of the number of processes executed per unit of time. The throughput for a good scheduler should always be higher.

Turnaround Time: It is the amount of time taken by a process for completing its execution. It includes the time spent by the process for waiting for the main memory, time spent in the ready queue, time spent on completing the I/O operations, and the time spent in execution. The turnaround time should be a minimal for a good scheduling algorithm.

Waiting Time: It is the amount of time spent by a process in the '*Ready*' queue waiting to get the CPU time for execution. The waiting time should be minimal for a good scheduling algorithm.

Response Time: It is the time elapsed between the submission of a process and the first response. For a good scheduling algorithm, the response time should be as least as possible.

> To summarise, a good scheduling algorithm has high CPU utilisation, minimum Turn Around Time (TAT), maximum throughput and least response time.

The Operating System maintains various queues† in connection with the CPU scheduling, and a process passes through these queues during the course of its admittance to execution completion.

† Queue is a special kind of arrangement of a collection of objects. In the operating system context queue is considered as a buffer.

https://hemanthrajhemu.github.io

The various queues maintained by OS in association with CPU scheduling are:

Job Queue: Job queue contains all the processes in the system

Ready Queue: Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution. The Ready queue is empty when there is no process ready for running.

Device Queue: Contains the set of processes, which are waiting for an I/O device.

A process migrates through all these queues during its journey from '*Admitted*' to '*Completed*' stage. The following diagrammatic representation (Fig. 10.12) illustrates the transition of a process through the various queues.

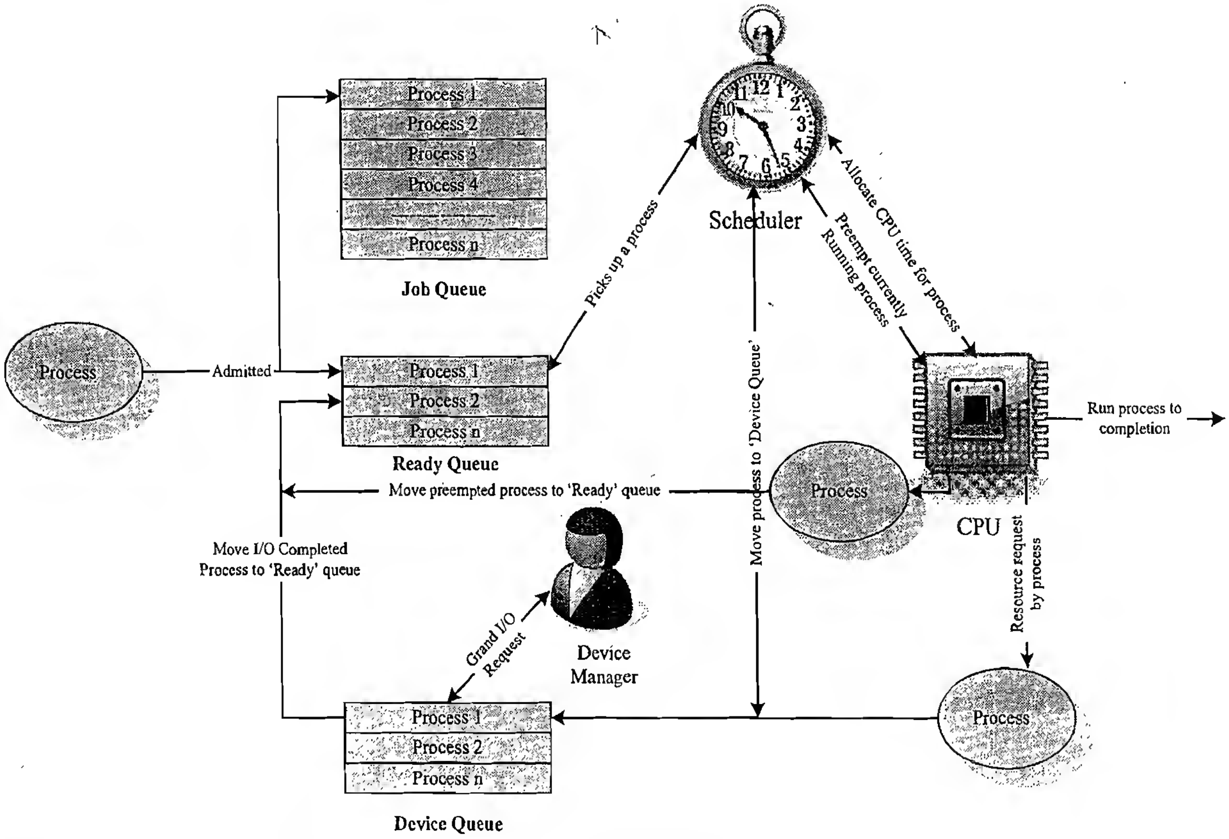

Process 1
Process 2
Process 3
Process 4
Process n
Job Queue
Picks up a process
Scheduler
Allocate CPU time for process
Preempt currently Running process
Process
Admitted
Process 1
Process 2
Process n
Ready Queue
Move preempted process to 'Ready' queue
Move process to 'Device Queue'
Process
CPU
Run process to completion
Resource request by process
Move I/O Completed Process to 'Ready' queue
Grand I/O Request
Device Manager
Process 1
Process 2
Process n
Device Queue
Process

Fig. 10.12 **Illustration of process transition through various queues**

Based on the scheduling algorithm used, the scheduling can be classified into the following categories.

10.5.1 Non-preemptive Scheduling

Non-preemptive scheduling is employed in systems, which implement non-preemptive multitasking model. In this scheduling type, the currently executing task/process is allowed to run until it terminates or enters the '*Wait*' state waiting for an I/O or system resource. The various types of non-preemptive scheduling adopted in task/process scheduling are listed below.

10.5.1.1 First-Come-First-Served (FCFS)/ FIFO Scheduling As the name indicates, the First-Come-First-Served (FCFS) scheduling algorithm allocates CPU time to the processes based on the

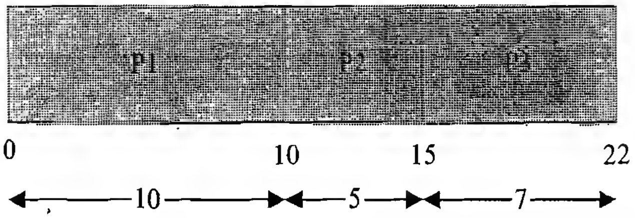
https://hemanthrajhemu.github.io

order in which they enter the '*Ready*' queue. The first entered process is serviced first. It is same as any real world application where queue systems are used; e.g. Ticketing reservation system where people need to stand in a queue and the first person standing in the queue is serviced first. FCFS scheduling is also known as First In First Out (FIFO) where the process which is put first into the '*Ready*' queue is serviced first.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 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).

The sequence of execution of the processes by the CPU is represented as



Assuming the CPU is readily available at the time of arrival of P1, P1 starts executing without any waiting in the '*Ready*' queue. Hence the waiting time for P1 is zero. The waiting time for all processes are given as

Waiting Time for P1 = 0 ms (P1 starts executing first).
Waiting Time for P2 = 10 ms (P2 starts executing after completing P1)
Waiting Time for P3 = 15 ms (P3 starts executing after completing P1 and P2)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P2+P3)) / 3
= (0+10+15)/3 = 25/3
= 8.33 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P2 = 15 ms (-Do-)
Turn Around Time (TAT) for P3 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P1+P2+P3)) / 3
= (10+15+22)/3 = 47/3
= 15.66 milliseconds

Average Turn Around Time (TAT) is the sum of average waiting time and average execution time.

Average Execution Time = (Execution time for all processes)/No. of processes
= (Execution time for (P1+P2+P3))/3
= (10+5+7)/3 = 22/3
= 7.33

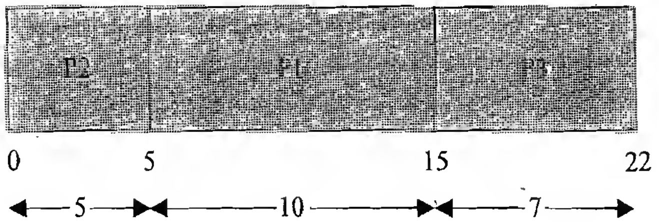
Average Turn Around Time = Average waiting time + Average execution time
= 8.33 + 7.33
= 15.66 milliseconds

https://hemanthrajhemu.github.io

Example 2

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) for the above example if the process enters the 'Ready' queue together in the order P2, P1, P3.

The sequence of execution of the processes by the CPU is represented as



Assuming the CPU is readily available at the time of arrival of P2, P2 starts executing without any waiting in the '*Ready*' queue. Hence the waiting time for P2 is zero. The waiting time for all processes is given as

Waiting Time for P2 = 0 ms (P2 starts executing first)
Waiting Time for P1 = 5 ms (P1 starts executing after completing P2)
Waiting Time for P3 = 15 ms (P3 starts executing after completing P2 and P1)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P2+P1+P3)) / 3
= (0+5+15)/3 = 20/3
= 6.66 milliseconds

Turn Around Time (TAT) for P2 = 5 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P1 = 15 ms (-Do-)
Turn Around Time (TAT) for P3 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P2+P1+P3)) / 3
= (5+15+22)/3 = 42/3
= 14 milliseconds

The Average waiting time and Turn Around Time (TAT) depends on the order in which the processes enter the '*Ready*' queue, regardless there estimated completion time.

From the above two examples it is clear that the Average waiting time and Turn Around Time improve if the process with shortest execution completion time is scheduled first.

The major drawback of FCFS algorithm is that it favours monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task. If the process contains any I/O operation, the CPU is relinquished by the process. In general, FCFS favours CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilisation. The average waiting time is not minimal for FCFS scheduling algorithm.

10.5.1.2 Last-Come-First Served (LCFS)/LIFO Scheduling The Last-Come-First Served (LCFS) scheduling algorithm also allocates CPU time to the processes based on the order in which they are entered in the '*Ready*' queue. The last entered process is serviced first. LCFS scheduling is also known as Last In First Out (LIFO) where the process, which is put last into the '*Ready*' queue, is serviced first.

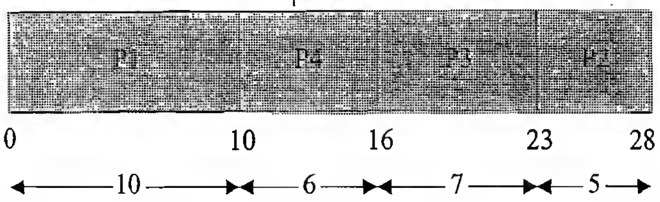
Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3 (Assume only P1 is present in the '*Ready*' queue when the scheduler picks

https://hemanthrajhemu.github.io

it up and P2, P3 entered '*Ready*' queue after that). Now a new process P4 with estimated completion time 6 ms enters the 'Ready' queue after 5 ms of scheduling P1. 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).Assume all the processes contain only CPU operation and no I/O operations are involved.

Initially there is only P1 available in the Ready queue and the scheduling sequence will be P1, P3, P2. P4 enters the queue during the execution of P1 and becomes the last process entered the '*Ready*' queue. Now the order of execution changes to P1, P4, P3, and P2 as given below.



The waiting time for all the processes is given as
Waiting Time for P1 = 0 ms (P1 starts executing first)
Waiting Time for P4 = 5 ms (P4 starts executing after completing P1. But P4 arrived after 5 ms of execution of P1. Hence its waiting time = Execution start time – Arrival Time = 10 – 5 = 5)
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 processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (0 + 5 + 16 + 23)/4 = 44/4
= 11 milliseconds
Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P4 = 11 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (10 – 5) + 6 = 5 + 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 processes) / No. of Processes
= (Turn Around Time for (P1+P4+P3+P2)) / 4
= (10+11+23+28)/4 = 72/4
= 18 milliseconds

LCFS scheduling is not optimal and it also possesses the same drawback as that of FCFS algorithm.

10.5.1.3 Shortest Job First (SJF) Scheduling

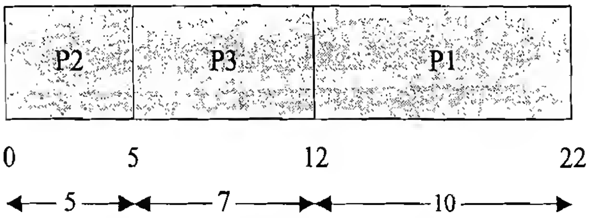
Shortest Job First (SJF) scheduling algorithm 'sorts the '*Ready*' queue' each time a process relinquishes the CPU (either the process terminates or enters the '*Wait*' state waiting for I/O or system resource) to pick the process with shortest (least) estimated completion/run time. In SJF, the process with the shortest estimated run time is scheduled first, followed by the next shortest process, and so on.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. 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 SJF algorithm.

The scheduler sorts the '*Ready*' queue based on the shortest estimated completion time and schedules the process with the least estimated completion time first and the next least one as second, and so on. The order in which the processes are scheduled for execution is represented as

https://hemanthrajhemu.github.io



The estimated execution time of P2 is the least (5 ms) followed by P3 (7 ms) and P1 (10 ms).
The waiting time for all processes are given as
Waiting Time for P2 = 0 ms (P2 starts executing first)
Waiting Time for P3 = 5 ms (P3 starts executing after completing P2)
Waiting Time for P1 = 12 ms (P1 starts executing after completing P2 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P2+P3+P1)) / 3
= (0+5+12)/3 = 17/3
= 5.66 milliseconds
Turn Around Time (TAT) for P2 = 5 ms (Time spent in Ready Queue + Execution Time)
Turn Around Time (TAT) for P3 = 12 ms (-Do-)
Turn Around Time (TAT) for P1 = 22 ms (-Do-)
Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P2+P3+P1)) / 3
= (5+12+22)/3 = 39/3
= 13 milliseconds
Average Turn Around Time (TAT) is the sum of average waiting time and average execution time.
The average Execution time = (Execution time for all processes)/No. of processes
= (Execution time for (P1+P2+P3))/3
= (10+5+7)/3 = 22/3 = 7.33
Average Turn Around Time = Average Waiting time + Average Execution time
= 5.66 + 7.33
= 13 milliseconds

From this example, it is clear that the average waiting time and turn around time is much improved with the SJF scheduling for the same processes when compared to the FCFS algorithm.

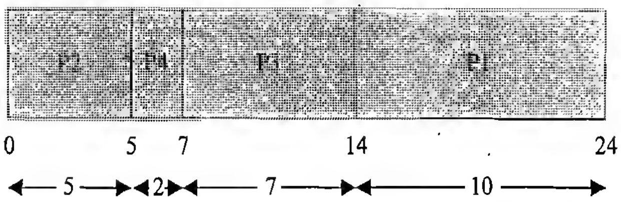
Example 2

Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time for the above example if a new process P4 with estimated completion time 2 ms enters the 'Ready' queue after 2 ms of execution of P2. Assume all the processes contain only CPU operation and no I/O operations are involved.

At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the SJF scheduler picks up the process with the least execution completion time (In this example P2 with execution completion time 5 ms) for scheduling. The execution sequence diagram for this is same as that of Example 1.

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 SJF algorithm is non-preemptive and process P2 does not contain any I/O operations, P2 continues its execution. After 5 ms of scheduling, P2 terminates and now the scheduler again sorts the '*Ready*' queue for process with least execution completion time. Since the execution completion time for P4 (2 ms) is less than that of P3 (7 ms), which was supposed to be run after the completion of P2 as per the '*Ready*' queue available at the beginning of execution scheduling, P4 is picked up for executing. Due to the arrival of the process P4 with execution time 2 ms, the '*Ready*' queue is re-sorted in the order P2, P4, P3, P1. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram

https://hemanthrajhemu.github.io



The waiting time for all the processes are given as

Waiting time for P2 = 0 ms (P2 starts executing first)

Waiting time for P4 = 3 ms (P4 starts executing after completing P2. But P4 arrived after 2 ms of execution of P2. Hence its waiting time = Execution start time – Arrival Time = 5 – 2 = 3)

Waiting time for P3 = 7 ms (P3 starts executing after completing P2 and P4)

Waiting time for P1 = 14 ms (P1 starts executing after completing P2, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P2+P4+P3+P1)) / 4
= (0 + 3 + 7 + 14)/4 = 24/4
= 6 milliseconds

Turn Around Time (TAT) for P2 = 5 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 5 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (5 – 2) + 2 = 3 + 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 Processes) / No. of Processes
= (Turn Around Time for (P2+P4+P3+P1)) / 4
= (5+5+14+24)/4 = 48/4
= 12 milliseconds

The average waiting time for a given set of process is minimal in SJF scheduling and so it is optimal compared to other non-preemptive scheduling like FCFS. The major drawback of SJF algorithm is that a process whose estimated execution completion time is high may not get a chance to execute if more and more processes with least estimated execution time enters the '*Ready*' queue before the process with longest estimated execution time started its execution (In non-preemptive SJF). This condition is known as '*Starvation*'. Another drawback of SJF is that it is difficult to know in advance the next shortest process in the '*Ready*' queue for scheduling since new processes with different estimated execution time keep entering the '*Ready*' queue at any point of time.

10.5.1.4 Priority Based Scheduling The Turn Around Time (TAT) and waiting time for processes in non-preemptive scheduling varies with the type of scheduling algorithm. Priority based non-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. The priority of a task/process can be indicated through various mechanisms. The Shortest Job First (SJF) algorithm can be viewed as a priority based scheduling where each task is prioritised in the order of the time required to complete the task. The lower the time required for completing a process the higher is its priority in SJF algorithm. Another way of priority assigning is associating a priority to the task/process at the time of creation of the task/process. The priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent. For Example, Windows CE supports 256 levels of priority (0 to 255 priority numbers). While creating the process/task, the priority can be assigned to it. The priority number associated with a task/process is the direct indication of its priority. The priority variation from high to low is represented by numbers from 0 to the maximum priority or by numbers from maximum priority to 0. For Windows CE operating system a priority number 0 indicates the highest priority and 255 indicates the lowest priority. This convention need not be universal and it depends on

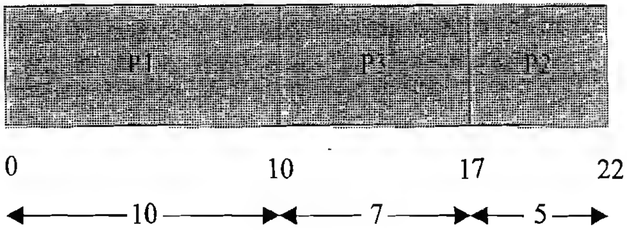
https://hemanthrajhemu.github.io

the kernel level implementation of the priority structure. The non-preemptive priority based scheduler sorts the 'Ready' queue based on priority and picks the process with the highest level of priority for execution.

Example 1

Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 0, 3, 2 (0—highest priority, 3—lowest priority) respectively enters the ready queue together. 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 priority based scheduling algorithm.

The scheduler sorts the '*Ready*' queue based on the priority and schedules the process with the highest priority (P1 with priority number 0) first and the next high priority process (P3 with priority number 2) as second, and so on. 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 ms (P1 starts executing first)

Waiting time for P3 = 10 ms (P3 starts executing after completing P1)

Waiting time for P2 = 17 ms (P2 starts executing after completing P1 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P3+P2)) / 3
= (0+10+17)/3 = 27/3
= 9 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P3 = 17 ms (-Do-)

Turn Around Time (TAT) for P2 = 22 ms (-Do-)

Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes
= (Turn Around Time for (P1+P3+P2)) / 3
= (10+17+22)/3 = 49/3
= 16.33 milliseconds

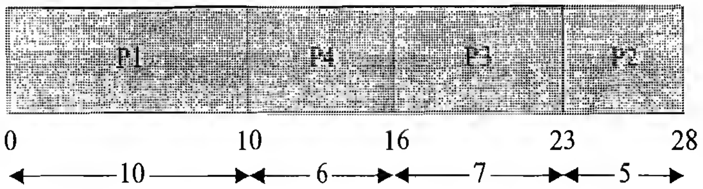
Example 2

Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time for the above example if a new process P4 with estimated completion time 6 ms and priority 1 enters the 'Ready' queue after 5 ms of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

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 0) for scheduling. The execution sequence diagram for this is same as that of Example 1. Now process P4 with estimated execution completion time 6 ms and priority 1 enters the '*Ready*' queue after 5 ms of execution of P1. Since the scheduling algorithm is non-preemptive and process P1 does not contain any I/O operations, P1 continues its execution. After 10 ms of scheduling, P1 terminates and now the scheduler again sorts the '*Ready*' queue for process with highest priority. Since the priority for P4 (priority 1) is higher than that of P3 (priority 2), which was supposed to be run after the completion of P1 as per the '*Ready*' queue available at the beginning of execution scheduling, P4 is picked up for executing. Due to the arrival of the process P4 with

https://hemanthrajhemu.github.io

priority 1, the '*Ready*' queue is resorted in the order P1, P4, 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 ms (P1 starts executing first)

Waiting time for P4 = 5 ms (P4 starts executing after completing P1. But P4 arrived after 5 ms of execution of P1. Hence its waiting time = Execution start time – Arrival Time = 10 – 5 = 5)

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 processes) / No. of Processes
= (Waiting time for (P1+P4+P3+P2)) / 4
= (0 + 5 + 16 + 23)/4 = 44/4
= 11 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 11 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (10 – 5) + 6 = 5 + 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 processes) / No. of Processes
= (Turn Around Time for (P2 + P4 + P3 + P1)) / 4
= (10 + 11 + 23 + 28)/4 = 72/4
= 18 milliseconds

Similar to SJF scheduling algorithm, non-preemptive priority based algorithm also possess the drawback of '*Starvation*' where a process whose priority is low may not get a chance to execute if more and more processes with higher priorities enter the '*Ready*' queue before the process with lower priority started its execution. '*Starvation*' can be effectively tackled in priority based non-preemptive scheduling by dynamically raising the priority of the low priority task/process which is under starvation (waiting in the ready queue for a longer time for getting the CPU time). The technique of gradually raising the priority of processes which are waiting in the 'Ready' queue as time progresses, for preventing '*Starvation*', is known as '*Aging*'.

10.5.2 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

https://hemanthrajhemu.github.io

'*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.

10.5.2.1 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. Preemptive SJF scheduling is also known as Shortest Remaining Time (SRT) scheduling.

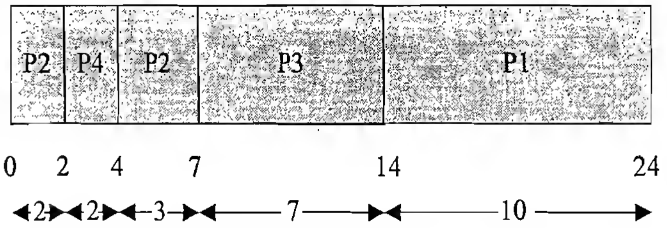
Now let us solve Example 2 given under the Non-preemptive SJF scheduling for preemptive SJF scheduling. The problem statement and solution is explained in the following example.

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.

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. The execution sequence diagram for this is same as that of example 1 under non-preemptive SJF 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. At the beginning it was P2, P3, P1. The execution sequence now changes as per the following diagram



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)

https://hemanthrajhemu.github.io

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

Now let's compare the Average Waiting time and Average Turn Around Time with that of the Average waiting time and Average Turn Around Time for non-preemptive SJF scheduling (Refer to Example 2 given under the section Non-preemptive SJF scheduling)

Average Waiting Time in non-preemptive SJF scheduling = 6 ms
Average Waiting Time in preemptive SJF scheduling = 5.75 ms

Average Turn Around Time in non-preemptive SJF scheduling = 12 ms
Average Turn Around Time in preemptive SJF scheduling = 11.75 ms

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

10.5.2.2 Round Robin (RR) Scheduling

The term *Round Robin* is very popular among the sports and games activities. You might have heard about 'Round Robin' league or 'Knock out' league associated with any football or cricket tournament. In the 'Round Robin' league each team in a group gets an equal chance to play against the rest of the teams in the same group whereas in the 'Knock out' league the losing team in a match moves out of the tournament ☺.

In the process scheduling context also, '*Round Robin*' brings the same 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 (see Fig. 10.13). 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.

The time slice is provided by the *timer tick* feature of the time management unit of the OS kernel (Refer the Time management section under the subtopic '*The Real-Time kernel*' for more details on Timer tick). Time slice is kernel dependent and it varies in the order of a few microseconds to milliseconds. Certain OS kernels may allow the time slice as user configurable. Round Robin scheduling ensures that

https://hemanthrajhemu.github.io

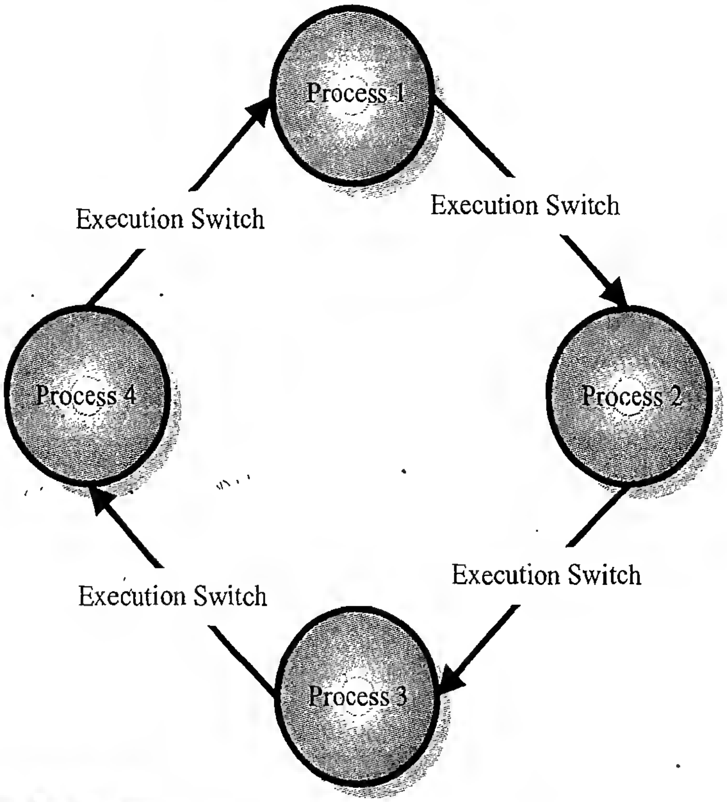


Fig. 10.13 **Round Robin Scheduling**

every process gets a fixed amount of ~~C~~PU time for execution. When the process gets its fixed time for execution is determined by the FCFS policy (That is, a process entering the Ready queue first gets its fixed execution time first and so on…). If a process terminates before the elapse of the time slice, the process releases the CPU voluntarily and the next process in the queue is scheduled for execution by the scheduler. The implementation of RR scheduling is kernel dependent. The following code snippet illustrates the RR scheduling implementation for RTX51 Tiny OS, an 8bit OS for *8051* microcontroller from Keil Software (www.keil.com), an ARM® Company.

```
#include <rtx51tny.h>       /* Definitions for RTX51 Tiny */
int counter0;
int counter1;

job0 () _task_ 0 {
  os_create_task (1);       /* Mark task 1 as "ready"      */

  while (1) {               /* Endless loop                */
    counter0++;             /* Increment counter 0         */
  }
}
job1 () _task_ 1 {
  while (1) {               /* Endless loop                */
    counter1++;             /* Increment counter 1         */
  }
}
```

https://hemanthrajhemu.github.io

RTX51 defines the tasks as simple C functions with void return type and void argument list. The attribute *_task_* is used for declaring a function as task. The general form of declaring a task is

```
void func (void) _task_ task_id
```

where *func* is the name of the task and *task_id* is the ID of the task. RTX51 supports up to 16 tasks and so *task_id* varies from 0 to 15. All tasks should be implemented as endless loops.

The two tasks in this program are counter loops. RTX51 Tiny starts executing task 0 which is the function named job0. This function creates another task called job1. After job0 executes for its time slice, RTX51 Tiny switches to job1. After job1 executes for its time slice, RTX51 Tiny switches back to job0. This process is repeated forever.

Now let's check how the RTX51 Tiny RR Scheduling can be implemented in an embedded device (A smart card reader) which addresses the following requirements.

- Check the presence of a card
- Process the data received from the card
- Update the Display
- Check the serial port for command/data
- Process the data received from serial port

These four requirements can be considered as four tasks. Implement them as four RTX51 tasks as explained below.

```
void check_card_task (void) _task_ 1
{
/* This task checks for the presence of a card */
/* Implement the necessary functionality here */
}
void process_card_task (void) _task_ 2
{
/* This task processes the data received from the card */
/* Implement the necessary functionality here */
}
void check_serial_io_task (void) _task_ 3
{
/* This task checks for serial I/O */
/* Implement the necessary functionality here */
}
void process_serial_data_task (void) _task_ 4
{
/* This task processes the data received from the serial port */
/* Implement the necessary functionality here */
}
```

Now the tasks are created. Next step is scheduling the tasks. The following code snippet illustrates the scheduling of tasks.

```
void startup_task (void) _task_ 0
{
os_create_task (1);    /* Create check_card_task Task */
os_create_task (2);    /* Create process_card_task Task */
os_create_task (3);    /* Create serial_io_task Task */
```

https://hemanthrajhemu.github.io

```
os_create_task (4);      /* Create serial_data_task Task */
os_delete_task (0);      /* Delete the Startup Task */
}
```

The *os_create_task* (task_ID) RTX51 Tiny kernel call puts the task with task ID *task_ID* in the 'Ready' state. All the ready tasks begin their execution at the next available opportunity. RTX51 Tiny does not have a *main ()* function to begin the code execution; instead it starts with executing task 0. Task 0 is used for creating other tasks. Once all the tasks are created, task 0 is stopped and removed from the task list with the *os_delete_task* kernel call. The RR scheduler selects each task based on the time slice and continues the execution. If we observe the tasks we can see that there is no point in executing the task *process_card_task* (Task 2) without detecting a card and executing the task *process_serial_data_task* (Task 4) without receiving some data in the serial port. In summary task 2 needs to be executed only when task 1 reports the presence of a card and task 4 needs to be executed only when task 3 reports the arrival of data at serial port. So these tasks (tasks 2 and 4) need to be put in the 'Ready' state only on satisfying these conditions. Till then these tasks can be put in the 'Wait' state so that the RR scheduler will not pick them for scheduling and the RR scheduling is effectively utilised among the other tasks. This can be achieved by implementing the wait and notify mechanism in the related tasks. Task 2 can be coded in a way that it waits for the card present event and task 1 signals the event 'card detected'. In a similar fashion Task 4 can be coded in such a way that it waits for the serial data received event and task 3 signals the reception of serial data on receiving serial data from serial port. The following code snippet explains the same.

```
void check_card_task (void) _task_ 1
{
/* This task checks for the presence of a card */
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the presence of card and card reading
//..............................
if (card is present)
//Signal card detected to task 2
os_send_signal (2)
}
}

void process_card_task (void) _task_ 2
{
/* This task processes the data received from the card */
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the signaling of card present event
os_wait1(K_SIG);
//Process card data
}
}
```

https://hemanthrajhemu.github.io

```
void check_serial_io_task (void) _task_ 3
{
/* This task checks for serial I/O */
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the reception of serial data
//.............................
if (data is received)
//Signal serial data reception to task 4
os_send_signal (4)
}
}

void process_serial_data_task (void) _task_ 4
{
/* This task processes the data received from the serial port */
/* Implement the necessary functionality here */
while (1)
{
//Function for checking the signaling of serial data received event
os_wait1(K_SIG);
//Process card data
}
}
```

The *os_send_signal* (*Task ID*) kernel call sends a signal to task *Task ID*. If the specified task is already waiting for a signal, this function call readies the task for execution but does not start it. The *os_wait1* (*event*) kernel call halts the current task and waits for an event to occur. The *event* argument specifies the event to wait for and may have only the value K_SIG which waits for a signal. RTX51 uses the Timer 0 of *8051* for time slice generation. The time slice can be configured by the user by changing the time slice related parameters in the RTX51 Tiny OS configuration file **CONF_TNY.A51** file which is located in the **\KEIL\C51\RTXTINY2\** folder. Configuration options in **CONF_TNY.A51** allow users to:

- Specify the Timer Tick Interrupt Register Bank.
- Specify the Timer Tick Interval (in *8051* machine cycles).
- Specify user code to execute in the Timer Tick Interrupt.
- Specify the Round-Robin Timeout.
- Enable or disable Round-Robin Task Switching.
- Specify that your application includes long duration interrupts.
- Specify whether or not code banking is used.
- Define the top of the RTX51 Tiny stack.
- Specify the minimum stack space required.
- Specify code to execute in the event of a stack error.
- Define idle task operations.

The RTX51 kernel provides a set of task management functions for managing the tasks. At any point of time each RTX51 task is exactly in any one of the following state.

https://hemanthrajhemu.github.io

Task State	State Description
RUNNING	The task that is currently running is in the RUNNING State. Only one task at a time may be in this state. The *os_running_task_id* kernel call returns the task number (ID) of the currently executing task.
READY	Tasks which are ready to run are in the READY State. Once the Running task has completed processing, RTX51 Tiny selects and starts the next Ready task. A task may be made ready immediately (even if the task is waiting for a timeout or signal) by setting its ready flag using the *os_set_ready* or *isr_set_ready* kernel functions.
WAITING	Tasks which are waiting for an event are in the WAITING State. Once the event occurs, the task is switched to the READY State. The *os_wait function* is used for placing a task in the WAITING State.
DELETED	Tasks which have not been started or tasks which have been deleted are in the DELETED State. The *os_delete_task* routine places a task that has been started (with *os_create_task*) into the DELETED State.
TIME - OUT	Tasks which were interrupted by a Round-Robin Time-Out are in the TIME-OUT State. This state is equivalent to the READY State for Round-Robin programs.

Refer the documentation available with RTX51 Tiny OS for more information on the various RTX51 task management kernel functions and their usage.

RR scheduling with interrupts is a good choice for the design of comparatively less complex *Real-Time Embedded Systems*. In this approach, the tasks which require less *Real-Time* attention can be scheduled with Round Robin scheduling and the tasks which require *Real-Time* attention can be scheduled through Interrupt Service Routines. RTX51 Tiny supports Interrupts with RR scheduling. For RTX51 the time slice for RR scheduling is provided by the Timer interrupt and if the interrupt is of high priority than that of the timer interrupt and if its service time (ISR) is longer than the timer tick interval, the RTX51 timer interrupt may be interrupted by the ISR and it may be reentered by a subsequent RX51 Tiny timer interrupt. Hence proper care must be taken to limit the ISR time within the timer tick interval or to protect the timer tick interrupt code from reentrancy. Otherwise unexpected results may occur. The limitations of RR with interrupt generic approach are the limited number of interrupts supported by embedded processors and the interrupt latency happening due to the context switching overhead.

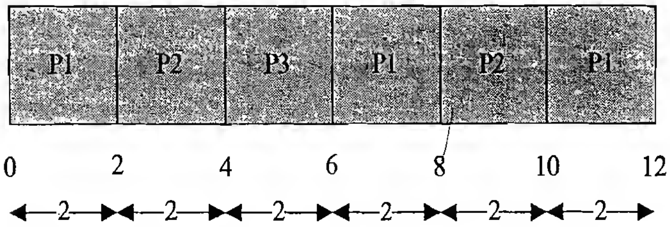
RR can also be used as technique for resolving the priority in scheduling among the tasks with same level of priority. We will discuss about how RR scheduling can be used for resolving the priority among equal tasks under the VxWorks kernel in a later chapter.

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.

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

https://hemanthrajhemu.github.io



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.

10.5.2.3 Priority Based Scheduling Priority based preemptive scheduling algorithm is same as that of the non-preemptive priority based scheduling except for the switching of execution between tasks. In preemptive scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the non-preemptive scheduling any high priority process entering the 'Ready' queue is scheduled only after the currently executing process completes its execution or only when it voluntarily relinquishes the CPU. The priority of a task/process in preemptive scheduling is indicated in the same way as that of the mechanism adopted for non-preemptive multitasking. Refer the non-preemptive priority based scheduling discussed in an earlier section of this chapter for more details.

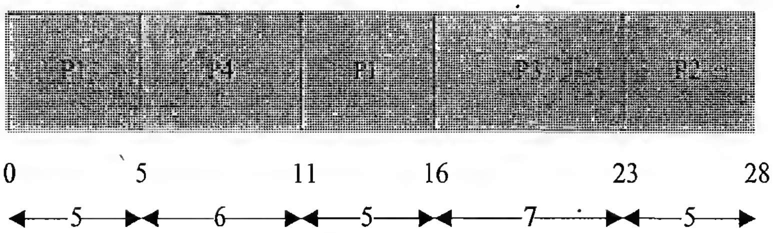
https://hemanthrajhemu.github.io

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.

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

https://hemanthrajhemu.github.io

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. Preemptive priority based scheduling also possesses the same drawback of non-preemptive priority based scheduling–'*Starvation*'. This can be eliminated by the '*Aging*' technique. Refer the section Non-preemptive priority based scheduling for more details on '*Starvation*' and '*Aging*'.

10.6 THREADS, PROCESSES AND SCHEDULING: PUTTING THEM ALTOGETHER

So far we discussed about threads, processes and process/thread scheduling. Now let us have a look at how these entities are addressed in a real world implementation. Let's examine the following pieces of code.

```
//*******************************************************************
//Process 1
//*******************************************************************
#include <windows.h>
#include <stdio.h>
//*******************************************************************
//Thread for executing Task
//*******************************************************************
void Task(void) {

while (1)
{
//Perform some task
//Task execution time is 7.5 units of execution
//Sleep for 17.5 units of execution
Sleep(17.5); //Parameter given is not in milliseconds
//Repeat task
}
}
//*******************************************************************
//Main Thread.
//*******************************************************************
void main(void) {
       DWORD id;
       HANDLE hThread;
//Create thread with normal priority
//*******************************************************************
       hThread = CreateThread(NULL,0,
                       (LPTHREAD_START_ROUTINE)Task,
                       (LPVOID) 0, 0, &id);
       if (NULL==hThread)
       {//Thread Creation failed. Exit process
       printf(“Creating thread failed: Error Code =
       %d”,GetLastError());
```

https://hemanthrajhemu.github.io

```
      return;
      }
      WaitForSingleObject(hThread, INFINITE);
      return;
}
//*********************************************************************
//Process 2
//*********************************************************************
#include <windows.h>
#include <stdio.h>
//*********************************************************************
//Thread for executing Task
//*********************************************************************
void Task(void) {
while (1)
{
//Perform some task
///Task execution time is 10 units of execution
//Sleep for 5 units of execution
Sleep(5); //Parameter given is not in milliseconds
//Repeat task
}
}
//*********************************************************************
//Main Thread.
//*********************************************************************
void main(void) {
        DWORD id;
        HANDLE hThread;
//Create thread with above normal priority
//*********************************************************************
       hThread = CreateThread(NULL,0,
                       (LPTHREAD_START_ROUTINE)Task,
                       (LPVOID) 0, CREATE_SUSPENDED, &id);
       if (NULL==hThread)
       {//Thread Creation failed. Exit process
       printf(“Creating thread failed: Error Code =
       %d”,GetLastError());
       return;
       }
       SetThreadPriority(hThread,THREAD_PRIORITY_ABOVE_NORMAL);
       ResumeThread(hThread);
       WaitForSingleObject(hThread, INFINITE);
       return;
}
```

The first piece of code represents a process (Process 1) with priority normal and it performs a task which requires 7.5 units of execution time. After performing this task, the process sleeps for 17.5 units of execution time and this is repeated forever. The second piece of code represents a process (Process

https://hemanthrajhemu.github.io

2) with priority above normal and it performs a task which requires 10 units of execution time. After performing this task, the process sleeps for 5 units of execution time and this is repeated forever. Process 2 is of higher priority compared to process 1, since its priority is above 'Normal'.

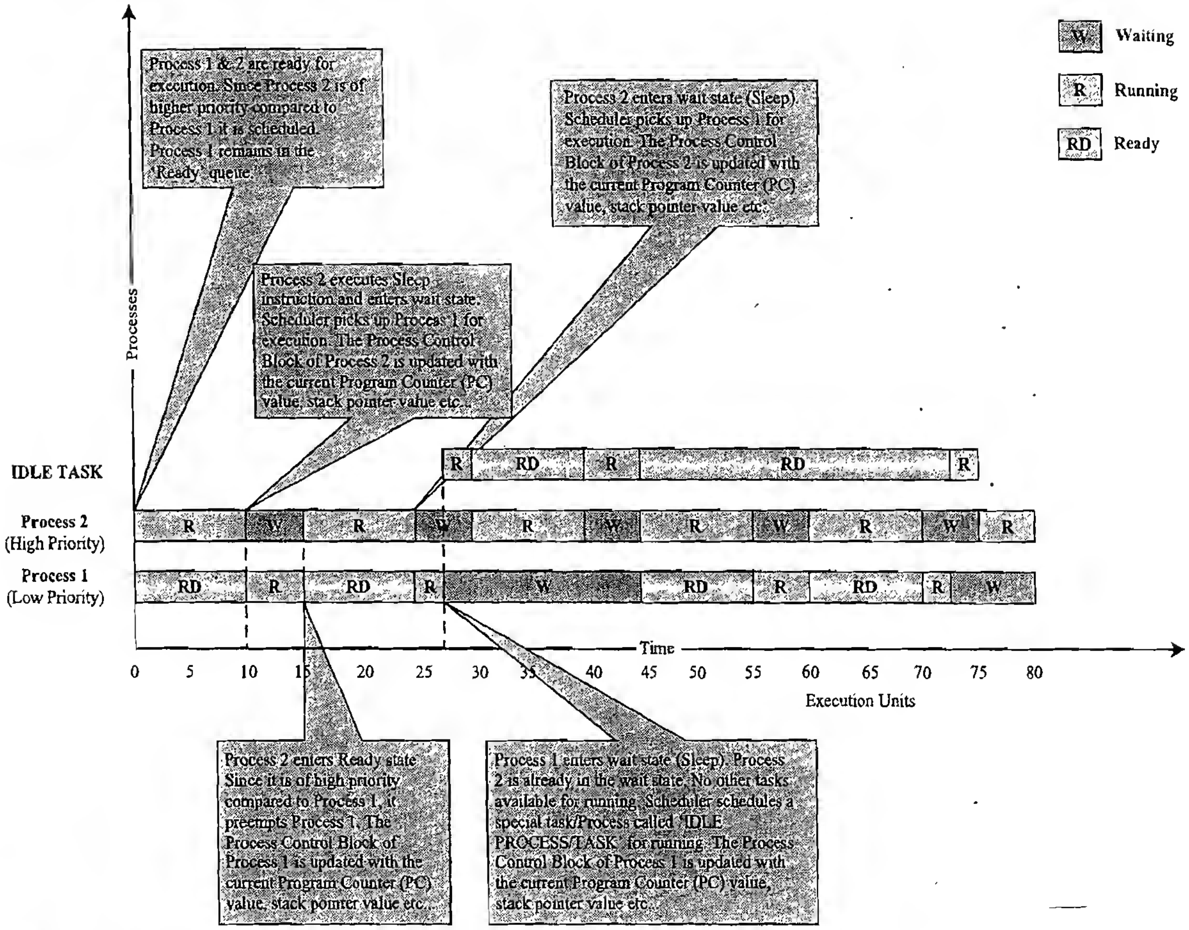
Now let us examine what happens if these processes are executed on a Real-Time kernel with pre-emptive priority based scheduling policy. Imagine Process 1 and Process 2 are ready for execution. Both of them enters the 'Ready' queue and the scheduler picks up Process 2 for execution since it is of higher priority (Assuming there is no other process running/ready for execution, when both the processes are 'Ready' for execution) compared to Process 1. Process 2 starts executing and runs until it executes the Sleep instruction (i.e. after10 units of execution time). When the Sleep instruction is executed, Process 2 enters the wait state. Since Process 1 is waiting for its turn in the 'Ready' queue, the scheduler picks up it for execution, resulting in a context switch. The Process Control Block (PCB) of Process 2 is updated with the values of the Program Counter (PC), stack pointer, etc. at the time of context switch. The estimated task execution time for Process 1 is 7.5 units of execution time and the sleeping time for Process 2 is 5 units of execution. After 5 units of execution time, Process 2 enters the 'Ready' state and moves to the 'Ready' queue. Since it is of higher priority compared to the running process, the running process (Process 1) is pre-empted and Process 2 is scheduled for execution. Process 1 is moved to the 'Ready' queue, resulting in context switching. The Process Control Block of Process 1 is updated with the current values of the Program Counter (PC), Stack pointer, etc. when the context switch is happened. The Program Counter (PC), Stack pointer, etc. for Process 2 is loaded with the values stored in the Process Control Block (PCB) of Process 2 and Process 2 continues its execution form where it was stopped earlier. Process 2 executes the Sleep instruction after 10 units of execution time and enters the wait state. At this point Process 1 is waiting in the 'Ready' queue and it requires 2.5 units of execution time for completing the task associated with it (The total time for completing the task is 7.5 units of time, out of this it has already completed 5 units of execution when Process 2 was in the wait state). The scheduler schedules Process 1 for execution. The Program Counter (PC), Stack pointer, etc. for Process 1 is loaded with the values stored in the Process Control Block (PCB) of Process 1 and Process 1 continues its execution form where it was stopped earlier. After 2.5 units of execution time, Process 1 executes the Sleep instruction and enters the wait state. Process 2 is already in the wait state and the scheduler finds no other process for scheduling. In order to keep the CPU always busy, the scheduler runs a dummy process (task) called '*IDLE PROCESS (TASK)*'. The '*IDLE PROCESS (TASK)*' executes some dummy task and keeps the CPU engaged. The execution diagram depicted in Fig. 10.24 explains the sequence of operations.

The implementation of the '*IDLE PROCESS (TASK)*' is dependent on the kernel and a typical implementation for a desktop OS may look like. It is simply an endless loop.

```
void Idle_Process (void)
{
//Simply wait.
//Do nothing...
While (1);
}
```

The Real-Time kernels deployed in embedded systems, where operating power is a big constraint (like systems which are battery powered); the '*IDLE TASK*' is used for putting the CPU into *IDLE* mode for saving the power. A typical example is the RTX51 Tiny Real-Time kernel, where the '*IDLE TASK*' sets the 8051 CPU to *IDLE* mode, a power saving mode. In the '*IDLE*' mode, the program execution is

https://hemanthrajhemu.github.io



Process 1 & 2 are ready for execution. Since Process 2 is of higher priority compared to Process 1 it is scheduled. Process 1 remains in the 'Ready' queue.

Process 2 enters wait state (Sleep). Scheduler picks up Process 1 for execution. The Process Control Block of Process 2 is updated with the current Program Counter (PC) value, stack pointer value etc.

Process 2 executes Sleep instruction and enters wait state. Scheduler picks up Process 1 for execution. The Process Control Block of Process 2 is updated with the current Program Counter (PC) value, stack pointer value etc.

Process 2 enters Ready state Since it is of high priority compared to Process 1, it preempts Process 1. The Process Control Block of Process 1 is updated with the current Program Counter (PC) value, stack pointer value etc.

Process 1 enters wait state (Sleep). Process 2 is already in the wait state. No other tasks available for running. Scheduler schedules a special task/Process called 'IDLE PROCESS/TASK' for running. The Process Control Block of Process 1 is updated with the current Program Counter (PC) value, stack pointer value etc.

Fig. 10.14 Process scheduling and context switch

halted and all peripherals and the interrupt system continues its operation. Once the CPU is put into the '*IDLE*' mode, it comes out of this mode when an Interrupt occurs or when the RTX51 Tiny Timer Tick Interrupt (The timer interrupt used for task scheduling in Round robin scheduling) occurs. It should be noted that the '*IDLE PROCESS (TASK)*' execution is not pre-emptive priority scheduling specific, it is applicable to all types of scheduling policies which demand 100% CPU utilisation/CPU power saving.

Back to the desktop OS environment, let's analyse the process, threads and scheduling in the Windows desktop environment. Windows provide a utility called *task manager* for monitoring the different process running on the system and the resources used by each process. A snapshot of the process details returned by the task manager for Windows XP kernel is shown in Fig. 10.15. It should be noted that this snapshot is purely machine dependent and it varies with the number of processes running on the machine.

'*Image Name*' represents the name of the process. '*PID*' represents the Process Identification Number (Process ID). As mentioned in the 'Threads and Process' section, when a process is created an ID is associated to it. CPU usage gives the % of CPU utilised by the process during an interval. '*CPU Time*' gives the total CPU time used by a process after its commencement. '*Mem Usage*' represents the total main memory, in kilobytes, used by the process. '*VM Size*' represents the total virtual memory (paged memory), in kilobytes, used by a process. '*Paged Pool*' represents the paged memory, in kilobytes, currently used by the system. '*NP Pool*' is the non-paged pool or system memory used by a process. The non-paged memory is not swapped to the secondary storage disk. '*Base Pri*' represents the priority of

https://hemanthrajhemu.github.io

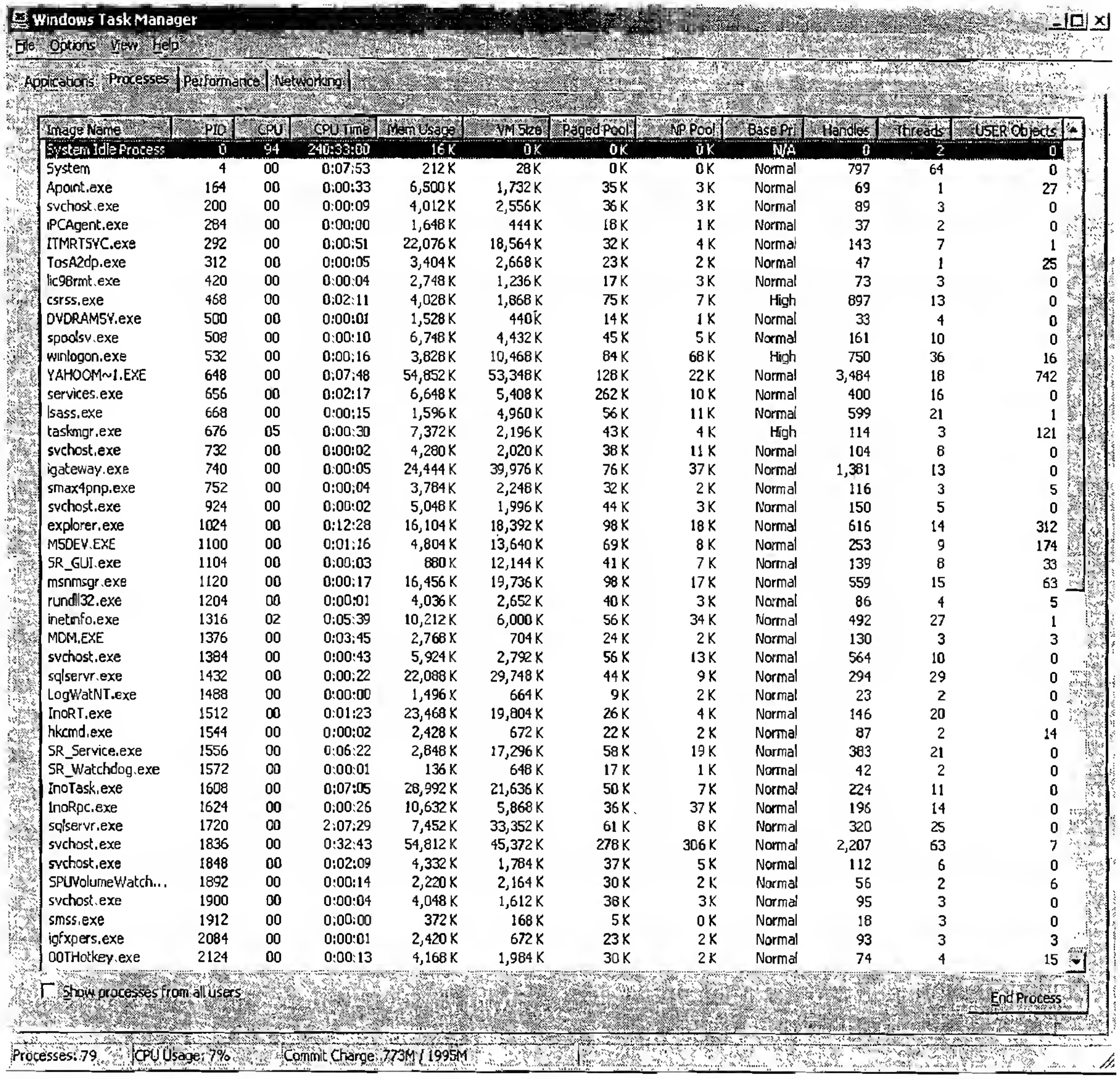

Windows Task Manager
File Options View Help
Applications Processes Performance Networking
Show processes from all users
End Process

Image Name	PID	CPU	CPU Time	Mem Usage	VM Size	Paged Pool	NP Pool	Base Pri	Handles	Threads	USER Objects
System Idle Process	0	94	240:33:00	16 K	0 K	0 K	0 K	N/A	0	2	0
System	4	00	0:07:53	212 K	28 K	0 K	0 K	Normal	797	64	0
Apoint.exe	164	00	0:00:33	6,500 K	1,732 K	35 K	3 K	Normal	69	1	27
svchost.exe	200	00	0:00:09	4,012 K	2,556 K	36 K	3 K	Normal	89	3	0
iPCAgent.exe	284	00	0:00:00	1,648 K	444 K	18 K	1 K	Normal	37	2	0
ITMRTSYC.exe	292	00	0:00:51	22,076 K	18,564 K	32 K	4 K	Normal	143	7	1
TosA2dp.exe	312	00	0:00:05	3,404 K	2,668 K	23 K	2 K	Normal	47	1	25
lic98rmt.exe	420	00	0:00:04	2,748 K	1,236 K	17 K	3 K	Normal	73	3	0
csrss.exe	468	00	0:02:11	4,028 K	1,868 K	75 K	7 K	High	897	13	0
DVDRAMSY.exe	500	00	0:00:01	1,528 K	440 K	14 K	1 K	Normal	33	4	0
spoolsv.exe	508	00	0:00:10	6,748 K	4,432 K	45 K	5 K	Normal	161	10	0
winlogon.exe	532	00	0:00:16	3,828 K	10,468 K	84 K	68 K	High	750	36	16
YAHOOM~1.EXE	648	00	0:07:48	54,852 K	53,348 K	128 K	22 K	Normal	3,484	18	742
services.exe	656	00	0:02:17	6,648 K	5,408 K	262 K	10 K	Normal	400	16	0
lsass.exe	668	00	0:00:15	1,596 K	4,960 K	56 K	11 K	Normal	599	21	1
taskmgr.exe	676	05	0:00:30	7,372 K	2,196 K	43 K	4 K	High	114	3	121
svchost.exe	732	00	0:00:02	4,280 K	2,020 K	38 K	11 K	Normal	104	8	0
igateway.exe	740	00	0:00:05	24,444 K	39,976 K	76 K	37 K	Normal	1,381	13	0
smax4pnp.exe	752	00	0:00:04	3,784 K	2,248 K	32 K	2 K	Normal	116	3	5
svchost.exe	924	00	0:00:02	5,048 K	1,996 K	44 K	3 K	Normal	150	5	0
explorer.exe	1024	00	0:12:28	16,104 K	18,392 K	98 K	18 K	Normal	616	14	312
MSDEV.EXE	1100	00	0:01:16	4,804 K	13,640 K	69 K	8 K	Normal	253	9	174
SR_GUI.exe	1104	00	0:00:03	880 K	12,144 K	41 K	7 K	Normal	139	8	33
msnmsgr.exe	1120	00	0:00:17	16,456 K	19,736 K	98 K	17 K	Normal	559	15	63
rundll32.exe	1204	00	0:00:01	4,036 K	2,652 K	40 K	3 K	Normal	86	4	5
inetinfo.exe	1316	02	0:05:39	10,212 K	6,000 K	56 K	34 K	Normal	492	27	1
MDM.EXE	1376	00	0:03:45	2,768 K	704 K	24 K	2 K	Normal	130	3	3
svchost.exe	1384	00	0:00:43	5,924 K	2,792 K	56 K	13 K	Normal	564	10	0
sqlservr.exe	1432	00	0:00:22	22,088 K	29,748 K	44 K	9 K	Normal	294	29	0
LogWatNT.exe	1488	00	0:00:00	1,496 K	664 K	9 K	2 K	Normal	23	2	0
InoRT.exe	1512	00	0:01:23	23,468 K	19,804 K	26 K	4 K	Normal	146	20	0
hkcmd.exe	1544	00	0:00:02	2,428 K	672 K	22 K	2 K	Normal	87	2	14
SR_Service.exe	1556	00	0:06:22	2,848 K	17,296 K	58 K	19 K	Normal	383	21	0
SR_Watchdog.exe	1572	00	0:00:01	136 K	648 K	17 K	1 K	Normal	42	2	0
InoTask.exe	1608	00	0:07:05	28,992 K	21,636 K	50 K	7 K	Normal	224	11	0
InoRpc.exe	1624	00	0:00:26	10,632 K	5,868 K	36 K	37 K	Normal	196	14	0
sqlservr.exe	1720	00	2:07:29	7,452 K	33,352 K	61 K	8 K	Normal	320	25	0
svchost.exe	1836	00	0:32:43	54,812 K	45,372 K	278 K	306 K	Normal	2,207	63	7
svchost.exe	1848	00	0:02:09	4,332 K	1,784 K	37 K	5 K	Normal	112	6	0
SPUVolumeWatch...	1892	00	0:00:14	2,220 K	2,164 K	30 K	2 K	Normal	56	2	6
svchost.exe	1900	00	0:00:04	4,048 K	1,612 K	38 K	3 K	Normal	95	3	0
smss.exe	1912	00	0:00:00	372 K	168 K	5 K	0 K	Normal	18	3	0
igfxpers.exe	2084	00	0:00:01	2,420 K	672 K	23 K	2 K	Normal	93	3	3
00THotkey.exe	2124	00	0:00:13	4,168 K	1,984 K	30 K	2 K	Normal	74	4	15

Processes: 79 | CPU Usage: 7% | Commit Charge: 773M / 1995M

Fig. 10.15 **Windows XP task manager for monitoring process and resource usage**

the process. As mentioned in an earlier section, a process may contain multiple threads. The '*Threads*' section gives the number of threads present in a process. '*Handles*' reflects the number of object handles owned by the process. This value is the reflection of the object handles present in the process's object table. '*User Objects*' reflects the number of objects active in the user mode for a process. Use '*Ctrl*' + '*Alt*' + '*Del*' key for accessing the task manager and select the '*View*' → '*Select Columns*' option to select the different monitoring parameters for a process.

10.7 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.

https://hemanthrajhemu.github.io

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 file, 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.

10.7.1 Shared Memory

Processes share some area of the memory to communicate among them (Fig. 10.16). 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).

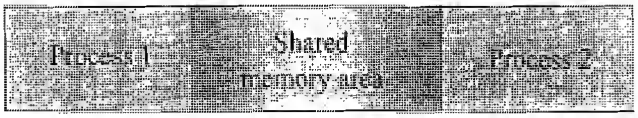


Fig. 10.16 **Concept of Shared Memory**

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

10.7.1.1 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

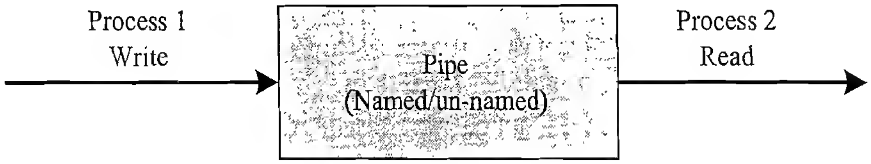


Fig. 10.17 **Concept of Pipe for IPC**

[†] Client Server is a software architecture containing a client application and a server application. The application which sends request is known as client and the application which receives the request process it and sends a response back to the client is known as server. A server is capable of receiving request from multiple clients.

https://hemanthrajhemu.github.io

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.

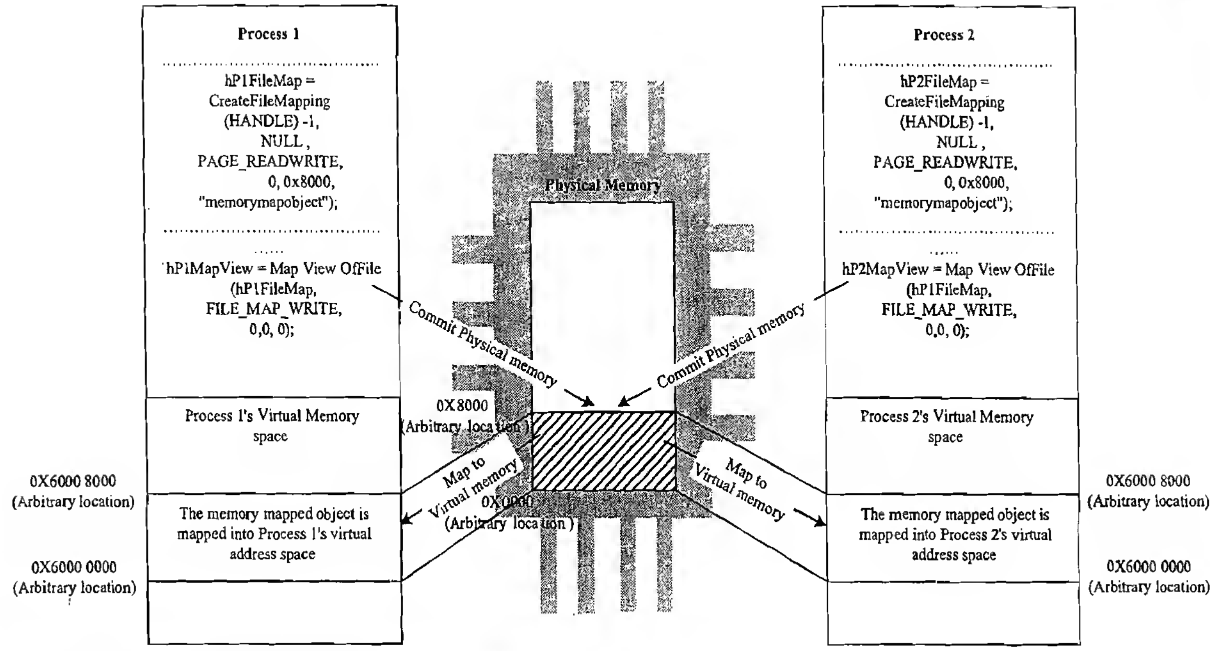
Please refer to the Online Learning Centre for details on the Pipe implementation under Windows Operating Systems.

Under VxWorks kernel, *pipe* is a special implementation of message queues. We will discuss the same in a latter chapter.

10.7.1.2 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.

Windows CE 5.0 RTOS uses the memory mapped object based shared memory technique for Inter Process Communication (Fig. 10.18). The *CreateFileMapping (HANDLE hFile, LPSECURITY_ATTRIBUTES lpFileMappingAttributes, DWORD flProtect, DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow, LPCTSTR lpName)* system call is used for sharing the memory. This API call is used for creating a mapping from a file. In order to create the mapping from the system paging memory, the handle parameter should be passed as *INVALID_HANDLE_VALUE (–1)*. The *lpFileMappingAttributes* parameter represents the security attributes and it must be *NULL*. The *flProtect* parameter represents the read write access for the shared memory area. A value of *PAGE_READONLY* makes the shared memory read only whereas the value *PAGE_READWRITE* gives read-write access to the shared memory. The parameter *dwMaximumSizeHigh* specifies the higher order 32 bits of the maximum size of the memory mapped object and *dwMaximumSizeLow* specifies the lower order 32 bits of the maximum size of the memory mapped object. The parameter *lpName* points to a null terminated string specifying the name of the memory mapped object. The memory mapped object is created as unnamed object if the parameter *lpName* is *NULL*. If *lpName* specifies the name of an existing memory mapped object, the function returns the handle of the existing memory mapped object to the caller process. The memory mapped object can be shared between the processes by either passing the handle of the object or by passing its name. If the handle of the memory mapped object created by a process is passed to another process for shared access, there is a possibility of closing the handle by the process which created the handle while it is in use by another process. This will throw OS level exceptions. If the name of the memory object is passed for shared access among processes, processes can use this name for creating a shared memory object which will open the shared memory object already existing with the given name. The OS will maintain a

https://hemanthrajhemu.github.io



Process 1

```
hP1FileMap =
CreateFileMapping
(HANDLE) -1,
NULL ,
PAGE_READWRITE,
0, 0x8000,
"memorymapobject");
```

```
hP1MapView = Map View OfFile
(hP1FileMap,
FILE_MAP_WRITE,
0,0, 0);
```

Process 2

```
hP2FileMap =
CreateFileMapping
(HANDLE) -1,
NULL ,
PAGE_READWRITE,
0, 0x8000,
"memorymapobject");
```

```
hP2MapView = Map View OfFile
(hP1FileMap,
FILE_MAP_WRITE,
0,0, 0);
```

Fig. 10.18 **Concept of memory mapped object**

usage count for the named object and it is incremented each time when a process creates/opens a memory mapped object with existing name. This will prevent the destruction of a shared memory object by one process while it is being accessed by another process. Hence passing the name of the memory mapped object is strongly recommended for memory mapped object based inter process communication. The *MapViewOfFile (HANDLE hFileMappingObject DWORD dwDesiredAccess, DWORD dwFileOffsetHigh, DWORD dwFileOffsetLow, DWORD dwNumberOfBytesToMap)* system call maps a view of the memory mapped object to the address space of the calling process. The parameter *hFileMappingObject* specifies the handle to an existing memory mapped object. The *dwDesiredAccess* parameter represents the read write access for the mapped view area. A value of *FILE_MAP_WRITE* makes the view access read-write, provided the memory mapped object *hFileMappingObject* is created with read-write access, whereas the value *FILE_MAP_READ* gives read only access to the shared memory, provided the memory mapped object *hFileMappingObject* is created with read-write/read only access. The parameter *dwFileOffsetHigh* specifies the higher order 32 bits and *dwFileOffsetLow* specifies the lower order 32 bits of the memory offset where mapping is to begin from the memory mapped object. A value of '0' for both of these maps the view from the beginning memory area of the memory object. *dwNumberOfBytesToMap* specifies the number of bytes of the memory object to map. If *dwNumberOfBytesToMap* is zero, the entire memory area owned by the memory mapped object is mapped. On successful execution, *MapViewOfFile* call returns the starting address of the mapped view. If the function fails it returns *NULL*. A mapped view of the memory mapped object is unmapped by the API call *UnmapViewOfFile (LPCVOID lpBaseAddress)*. The *lpBaseAddress* parameter specifies a pointer to the base address of the mapped view of a memory object that is to be unmapped. This value must be identical to the value returned by a previous call to the *MapViewOfFile* function. Calling *UnmapViewOfFile* cleans up the committed physical storage in a process's virtual address space. In other words, it frees the virtual address space of the mapping object. Under Windows NT/XP OS, a process can open an existing memory mapped object by calling the API *OpenFileMapping(DWORD dwDesiredAccess, BOOL bInheritHandle, LPCTSTR lpName)*. The parameter *dwDesiredAccess* specifies the read write access permissions foi

https://hemanthrajhemu.github.io

the memory mapped object. A value of *FILE_MAP_ALL_ACCESS* provides read-write access, whereas the value *FILE_MAP_READ* allocates only read access and *FILE_MAP_WRITE* allocates write only access. The parameter *bInheritHandle* specifies the handle inheritance. If this parameter is *TRUE*, the calling process inherits the handle of the existing object, otherwise not. The parameter *lpName* specifies the name of the existing memory mapped object which needs to be opened. Windows CE 5.0 does not support handle inheritance and hence the API call *OpenFileMapping* is not supported.

The following sample code illustrates the creation and accessing of memory mapped objects across multiple processes. The first piece of code illustrates the creation of a memory mapped object with name “memorymappedobject” and prints the address of the memory location where the memory is mapped within the virtual address space of Process 1.

```
#include <stdio.h>
#include <windows.h>
//************************************************************
//Process 1: Creates the memory mapped object and maps it to
//Process 1's Virtual Address space
//************************************************************
void main() {
      //Define the handle to Memory mapped Object
      HANDLE hFileMap;
      //Define the handle to the view of Memory mapped Object
      LPTSTR hMapView;
      printf(“//*****************************************\n”);
      printf(“               Process 1\n”);
      printf(“//*****************************************\n”);
      //Create an 8 KB memory mapped object
      hFileMap = CreateFileMapping( (HANDLE) -1,
       NULL,          // default security attributes
       PAGE_READWRITE, // Read-Write Access
      0, //Higher order 32 bits of the memory mapping object
      0x2000, //Lower order 32 bits of the memory mapping object
       TEXT(“memorymappedobject”));   // Memory mapped object name
if (NULL == hFileMap)
{
printf (“Memory mapped Object Creation Failed: Error Code: %d\n”, GetLastError
());
//Memory mapped Object Creation failed. Return
return;
}

//Map the memory mapped object to Process 1's address space
hMapView= (LPTSTR) MapViewOfFile(hFileMap,
              FILE_MAP_WRITE,
              0,      //Map the entire view
              0,
              0);
if (NULL == hMapView)
```

https://hemanthrajhemu.github.io

```
{
      printf (“Mapping of Memory mapped view Failed: Error Code:
      %d\n”, GetLastError ());
      //Memory mapped view Creation failed. Return
      return;
}
else
{
      //Successfully created the memory mapped view.
      //Print the start address of the mapped view
      printf (“The memory is mapped to the virtual address starting
      at 0x%08x\n”, hMapView);
}
//Wait for user input to exit. Run Process 2 before providing
//user input
printf (“Press any key to terminate Process 1”);
getchar();
//Unmap the view
UnmapViewOfFile(hMapView);
//Close memory mapped object handle
CloseHandle(hFileMap);
return;
}
```

The piece of code given below corresponds to Process 2. It illustrates the accessing of an existing memory mapped object (memory mapped object with name “memorymappedobject” created by Process 1) and prints the address of the memory location where the memory is mapped within the virtual address space of Process 2. To demonstrate the application, the program corresponding to Process 1 should be executed first and the program corresponding to Process 2 should be executed following Process 1.

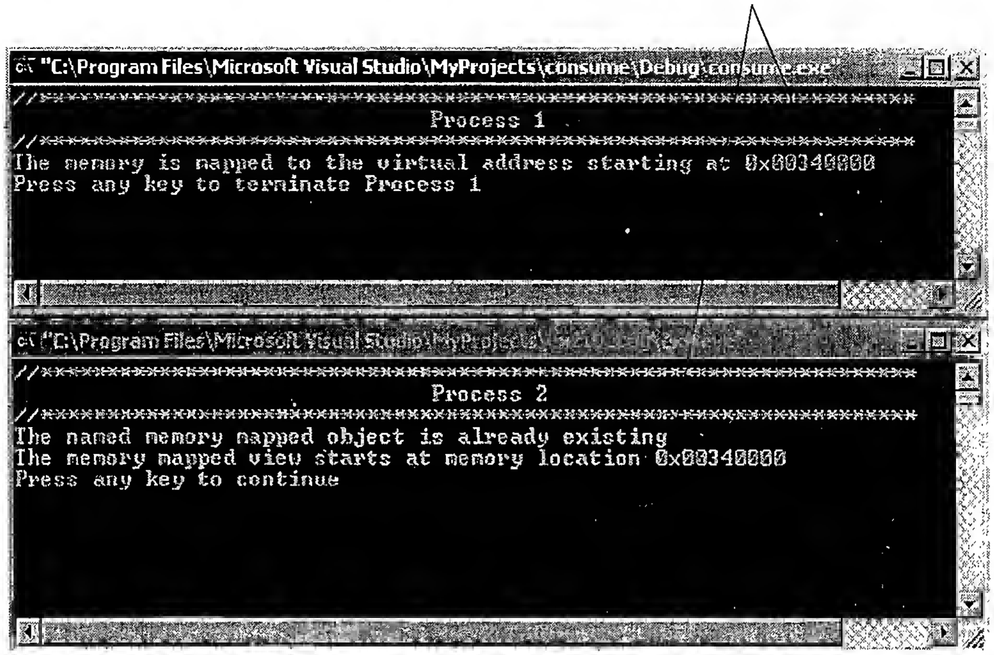
```
#include <stdio.h>
#include <windows.h>
//*******************************************************************
//Process 2: Opens the memory mapped object created by Process 1
//Maps the object to Process 2’s virtual address space.
//*******************************************************************
void main() {
      //Define the handle for the Memory mapped Object
      HANDLE hChildFileMap;
      //Define the handle for the view of Memory mapped Object
      LPTSTR hChildMapView;
      printf(“//**********************************************\n”);
      printf(“            Process 2\n”);
      printf(“//**********************************************\n”);
      //Create an 8 KB memory mapped object
      hChildFileMap = CreateFileMapping( INVALID_HANDLE_VALUE,
      NULL,           // default security attributes
      PAGE_READWRITE, // Read-Write Access
      0, //Higher order 32 bits of the memory mapping object
```

https://hemanthrajhemu.github.io

```
       0x2000, //Lower order 32 bits of the memory mapping object
       TEXT("memorymappedobject"));   // Memory mapped object name
if (NULL == hChildFileMap || INVALID_HANDLE_VALUE == hChildFileMap)
{
printf ("Memory mapped Object Creation Failed: Error Code: %d\n",
GetLastError ());
//Memory mapped Object Creation failed. Return
return;
}
else

       if (ERROR_ALREADY_EXISTS == GetLastError ())
       {
       //A memory mapped object with given name exists already.
        printf ("The named memory mapped object is already
       existing\n");
       }
//Map the memory mapped object to Process 2's address space

hChildMapView=(LPTSTR)MapViewOfFile(hChildFileMap,
                     FILE_MAP_WRITE, //Read-Write access
                     0,   //Map the entire view
                     0,
                     0);
if (NULL == hChildMapView)
{
printf ("Mapping of Memory mapped view Failed: Error Code: %d\n", GetLastError
());
//Memory mapped view Creation failed. Return
return;
}

else
{
//Successfully created the memory mapped view.
//Print the start address of the mapped view
printf ("The memory mapped view starts at memory location 0x%08x\n", hChild-
MapView);
}

//Unmap the view
UnmapViewOfFile(hChildMapView);
//Close memory mapped object handle
CloseHandle(hChildFileMap);
return;
}
```

https://hemanthrajhemu.github.io

The output of the above programs when run in the sequence, Process 1 is executed first and Process 2 is executed while Process 1 waits for the user input from the keyboard, is given in Fig. 10.19.



The Memory mapped object is mapped to the Virtual address 0x00340000 of both the Processes

"C:\Program Files\Microsoft Visual Studio\MyProjects\consume\Debug\consume.exe"

```
//******************************************************************************
                                   Process 1
//******************************************************************************
The memory is mapped to the virtual address starting at 0x00340000
Press any key to terminate Process 1
```

"C:\Program Files\Microsoft Visual Studio\MyProjects\ [illegible]

```
//******************************************************************************
                                   Process 2
//******************************************************************************
The named memory mapped object is already existing
The memory mapped view starts at memory location 0x00340000
Press any key to continue
```

Fig. 10.19 Output of Win32 memory mapped object illustration program

This reveals that using memory mapped objects with same name across multiple processes running on the same system maps the object to the same virtual address space of the processes.

Reading and writing to a memory mapped area is same as any read write operation using pointers. The pointer returned by the API call *MapViewOfFile* can be used for this. The exercise of Read and Write operation is left to the readers. Proper care should be taken to avoid any conflicts that may arise due to the simultaneous read/write access of the shared memory area by multiple processes. This can be handled by applying various synchronisation techniques like events, mutex, semaphore, etc.

For using a memory mapped object across multiple threads of a process, it is not required for all the threads of the process to create/open the memory mapped object and map it to the thread's virtual address space. Since the thread's address space is part of the process's virtual address space, which contains the thread, only one thread, preferably the parent thread (main thread) is required to create the memory mapped object and map it to the process's virtual address space. The thread which creates the memory mapped object can share the pointer to the mapped memory area as global pointer and other threads of the process can use this pointer for reading and writing to the mapped memory area. If one thread of a process tries to create a memory mapped object with the same name as that of an existing mapping object, which is created by another thread of the same process, a new view of the mapping object is created at a different virtual address of the process. This is same as one process trying to create two views of the same memory mapped object☺.

10.7.2 Message Passing

Message passing is an (a)synchronous information exchange mechanism used for Inter Process/Thread

https://hemanthrajhemu.github.io

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

10.7.2.1 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. 10.20). 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. The Windows XP OS kernel maintains a single system message queue and one process/thread (Process and threads are used interchangeably here, since thread is the basic unit of process in windows) specific message queue. 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. For posting a message to a thread's message queue, the kernel fills a message structure *MSG* and copies it to the message queue of the thread. The message structure *MSG* contains the handle of the process/thread for which the message is intended, the message parameters, the time at which the message is posted, etc. 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 classified 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. The thread which invoked the send message becomes blocked and the scheduler will not pick it up for scheduling. The *PostMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM*

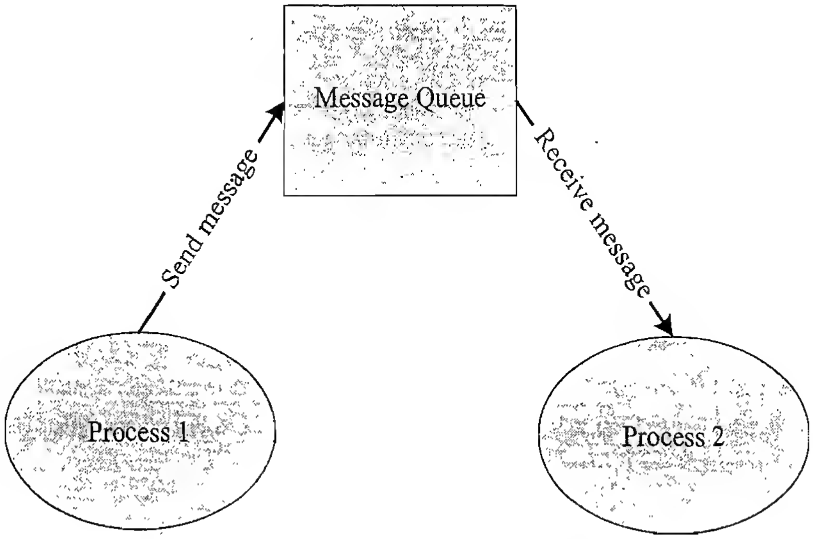


Fig. 10.20 **Concept of message queue based indirect messaging for IPC**

https://hemanthrajhemu.github.io

lParam) or *PostThreadMessage (DWORD idThread, UINT Msg, WPARAM wParam, LPARAM lParam)* API is used by a thread in Windows for posting a message to its own message queue or to the message queue of another thread. The *PostMessage* API does not always guarantee the posting of messages to message queue. The *PostMessage* API will not post a message to the message queue when the message queue is full. Hence it is recommended to check the return value of *PostMessage* API to confirm the posting of message. The *SendMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM lParam)* API call sends a message to the thread specified by the handle *hWnd* and waits for the callee thread to process the message. The thread which calls the *SendMessage* API enters waiting state and waits for the message result from the thread to which the message is posted. The thread which invoked the *SendMessage* API call becomes blocked and the scheduler will not pick it up for scheduling.

The Windows CE operating system supports a special Point-to-Point Message queue implementation. The OS maintains a First In First Out (FIFO) buffer for storing the messages and each process can access this buffer for reading and writing messages. The OS also maintains a special queue, with single message storing capacity, for storing high priority messages (Alert messages). The creation and usage of message queues under Windows CE OS is explained below.

The *CreateMsgQueue(LPCWSTR lpszName, LPMSGQUEUEOPTIONS lpOptions)* API call creates a message queue or opens a named message queue and returns a read only or write only handle to the message queue. A process can use this handle for reading or writing a message from/to of the message queue pointed by the handle. The parameter *lpszName* specifies the name of the message queue. If this parameter is *NULL*, an unnamed message queue is created. Processes can use the handle returned by the API call if the message queue is created without any name. If the message queue is created as named message queue, other processes can use the name of the message queue for opening the named message queue created by a process. Calling the *CreateMsgQueue* API with an existing named message queue as parameter returns a handle to the existing message queue. Under the Desktop Windows Operating Systems (Windows 9x/XP/NT/2K), each object type (viz. mutex, semaphores, events, memory maps, watchdog timers and message queues) share the same namespace and the same name is not allowed for creating any of this. Windows CE kernel maintains separate namespace for each and supports the same name across different objects. The *lpOptions* parameter points to a *MSGQUEUEOPTIONS* structure that sets the properties of the message queue. The member details of the *MSGQUEUEOPTIONS* structure is explained below.

```
typedef MSGQUEUEOPTIONS_OS{
  DWORD dwSize;
  DWORD dwFlags;
  DWORD dwMaxMessages;
  DWORD cbMaxMessage;
  BOOL bReadAccess;
} MSGQUEUEOPTIONS, FAR* LPMSGQUEUEOPTIONS, *PMSGQUEUEOPTIONS;
```

The members of the structure are listed below.

Member	Description
dwSize	Specifies the size of the structure in bytes
dwFlags	Describes the behaviour of the message queue. Set to MSGQUEUE_NOPRECOMMIT to allocate message buffers on demand and to free the message buffers after they are read, or set to MSGQUEUE_ALLOW_BROKEN to enable a read or write operation to complete even if there is no corresponding writer or reader present.

https://hemanthrajhemu.github.io

dwMaxMessages	Specifies the maximum number of messages to queue at any point of time. Set this value to zero to specify no limit on the number of messages to queue at any point of time.
cbMaxMessage	Specifies the maximum number of bytes in each message. This value must be greater than zero.
bReadAccess	Specifies the Read Write access to the message queue. Set to TRUE to request read access to the queue. Set to FALSE to request write access to the queue.

On successful execution, the *CreateMsgQueue* API call returns a 'Read Only' or 'Write Only' handle to the specified queue based on the *bReadAccess* member of the *MSGQUEUEOPTIONS* structure *lpOptions*. If the queue with specified name already exists, a new handle, which points to the existing queue, is created and a following call to *GetLastError* returns *ERROR_ALREADY_EXISTS*. If the function fails it returns *NULL*. A single call to the *CreateMsgQueue* creates the queue for either 'read' or 'write' access. The *CreateMsgQueue* API should be called twice with the *bReadAccess* member of the *MSGQUEUEOPTIONS* structure *lpOptions* set to TRUE and FALSE respectively in successive calls for obtaining 'Read only' and 'Write only' handles to the specified message queue. The handle returning by *CreateMsgQueue* API call is an *event* handle and, if it is a 'Read Only' access handle, it is signalled by the message queue if a new message is placed in the queue. The signal is reset on reading the message by *ReadMsgQueue* API call. A 'Write Only' access handle to the message queue is signalled when the queue is no longer full, i.e. when there is room for accommodating new messages. Processes can monitor the handles with the help of the wait functions, viz. *WaitForSingleObject* or *WaitForMultipleObjects*, supported by Windows CE. The *OpenMsgQueue(HANDLE hSrcProc, HANDLE hMsgQ, LPMSGQUEUEOPTIONS lpOptions)* API call opens an existing named or unnamed message queue. The parameter *hSrcProc* specifies the process handle of the process that owns the message queue and *hMsgQ* specifies the handle of the existing message queue (Handle to the message queue returned by the CreateMsgQueue function). As in the case of *CreateMsgQueue*, the *lpOptions* parameter points to a *MSGQUEUEOPTIONS* structure that sets the properties of the message queue. On successful execution the *OpenMsgQueue* API call returns a handle to the message queue and *NULL* if it fails. Normally the *OpenMsgQueue* API is used for opening an unnamed message queue. The *WriteMsgQueue(HANDLE hMsgQ, LPVOID lpBuffer, DWORD cbDataSize, DWORD dwTimeout, DWORD dwFlags)* API call is used for writing a single message to the message queue pointed by the handle *hMsgQ*. *lpBuffer* points to a buffer that contains the message to be written to the message queue. The parameter *cbDataSize* specifies the number of bytes stored in the buffer pointed by *lpBuffer*, which forms a message. The parameter *dwTimeout* specifies the timeout interval in milliseconds for the message writing operation. A value of zero specifies the write operation to return immediately without blocking if the write operation cannot succeed. If the parameter is set to *INFINITE*, the write operation will block until it succeeds or the message queue signals the 'write only' handle indicating the availability of space for posting a message. The *dwFlags* parameter sets the priority of the message. If it is set to *MSGQUEUE_MSGALERT*, the message is posted to the queue as high priority or alert message. The Alert message is always placed in the front of the message queue. This function returns *TRUE* if it succeeds and *FALSE* otherwise.

The *ReadMsgQueue(HANDLE hMsgQ, LPVOID lpBuffer, DWORD cbBufferSize, LPDWORD lpNumberOfBytesRead, DWORD dwTimeout, DWORD* pdwFlags)* API reads a single message from the message queue. The parameter *hMsgQ* specifies a handle to the message queue from which the message needs to be read. *lpBuffer* points to a buffer for storing the message read from the queue. The parameter *cbBufferSize* specifies the size of the buffer pointed by *lpBuffer*, in bytes. *lpNumberOfBytesRead* specifies the number of bytes stored in the buffer. This is same as the number of bytes present in

https://hemanthrajhemu.github.io

the message which is read from the message queue. *dwTimeout* specifies the timeout interval in milliseconds for the message reading operation. The timeout values and their meaning are same as that of the write message timeout parameter. The *dwFlags* parameter indicates the priority of the message. If the message read from the message queue is a high priority message (alert message), *dwFlags* is set to *MSGQUEUE_MSGALERT*. The function returns *TRUE* if it succeeds and *FALSE* otherwise. The *GetMsgQueueInfo (HANDLE hMsgQ, LPMSGQUEUEINFO lpInfo)* API call returns the information about a message queue specified by the handle *hMsgQ*. The message information is returned in a *MSGQUEUEINFO* structure pointed by *lpInfo*. The details of the *MSGQUEUEINFO* structure is explained below.

```
typedef MSGQUEUEINFO{
  DWORD dwSize;
  DWORD dwFlags;
  DWORD dwMaxMessages;
  DWORD cbMaxMessage;
  DWORD dwCurrentMessages;
  DWORD dwMaxQueueMessages;
  WORD wNumReaders;
  WORD wNumWriters;
} MSGQUEUEINFO, *PMSGQUEUEINFO, FAR* LPMSGQUEUEINFO;
```

The member variable details are listed below.

Member	Description
DwSize	Specifies the size of the buffer passed in.
dwFlags	Describes the behaviour of the message queue. It retrieves the *MSGQUEUEOPTIONS. dwFlags* passed when the message queue is created with *CreateMsgQueue* API call.
dwMaxMessages	Specifies the maximum number of messages to queue at any point of time. This reflects the *MSGQUEUEOPTIONS. dwMaxMessages* value passed when the message queue is created with *CreateMsgQueue* API call.
cbMaxMessage	Specifies the maximum number of bytes in each message. This reflects the *MSGQUEUEOPTIONS.cbMaxMessage* value passed when the message queue is created with *CreateMsgQueue* API call.
dwCurrentMessages	Specifies the number of messages currently existing in the specified message queue.
dwMaxQueueMessages	Specifies maximum number of messages that have ever been in the queue at one time.
wNumReaders	Specifies the number of readers (processes which opened the message queue for reading) subscribed to the message queue for reading.
wNumWriters	Specifies the number of writers (processes which opened the message queue for writing) subscribed to the message queue for writing.

The *GetMsgQueueInfo* API call returns *TRUE* if it succeeds and *FALSE* otherwise. The *CloseMsgQueue(HANDLE hMsgQ)* API call closes a message queue specified by the handle *hMsgQ*. If a process holds a 'read only' and 'write only' handle to the message queue, both should be closed for closing the message queue.

'Message queue' is the primary inter-task communication mechanism under VxWorks kernel. Message queues support two-way communication of messages of variable length. The two-way messaging

https://hemanthrajhemu.github.io

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. We will discuss about the VxWorks' message queue implementation in a separate chapter.

10.7.2.2 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). 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. The MicroC/OS-II implements mailbox as a mechanism for inter-task communication. We will discuss about the mailbox based IPC implementation under MicroC/OS-II in a latter chapter. Figure 10.21 given below illustrates the mailbox based IPC technique.

10.7.2.3 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. Refer to the topic 'Round Robin Scheduling' under the section 'Priority based scheduling' for more details on Signalling in RTX51 Tiny OS. The VxWorks RTOS kernel also implements 'signals' for inter process

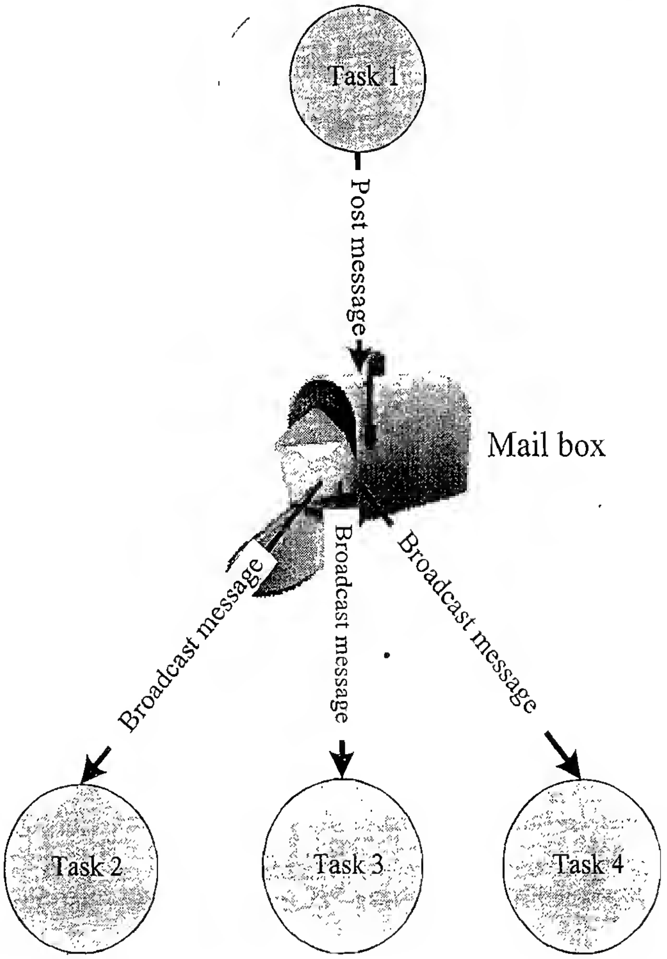


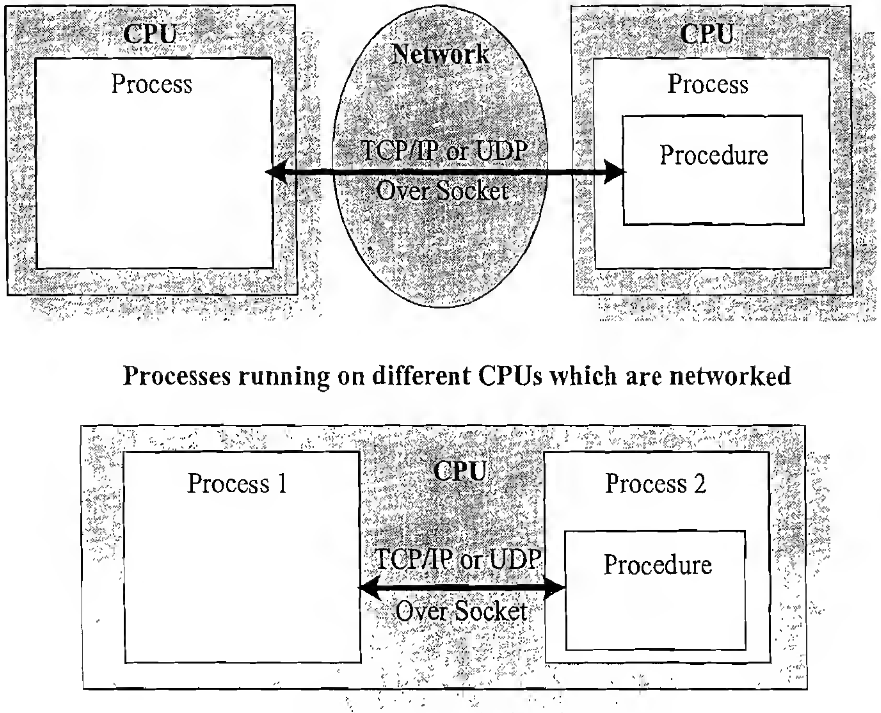
Fig. 10.21 **Concept of Mailbox based indirect messaging for IPC**

https://hemanthrajhemu.github.io

communication. Whenever a specified signal occurs it is handled in a signal handler associated with the signal. We will discuss about the signal based IPC mechanism for VxWorks' kernel in a later chapter.

10.7.3 Remote Procedure Call (RPC) and Sockets

Remote Procedure Call or RPC (Fig. 10.22) 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.



Processes running on different CPUs which are networked

Processes running on same CPU

Fig. 10.22 **Concept of Remote Procedure Call (RPC) for IPC**

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.

https://hemanthrajhemu.github.io

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. The following sample code illustrates the usage of socket for creating a client application under Windows OS. Winsock (Windows Socket 2) is the library implementing socket functions for Win32.

```
#include <stdio.h>
#include "winsock2.h"
//Specify the server address
#define SERVER        "172.168.0.1"
//Specify the server port
#define PORT 5000
int buflength = 100;
char *sendbuf = "Hi from Client";
char buffer[buflength];
void main() {
  //*********************************************************
  // Initialize Winsock
  WSADATA wsaData;
  if (WSAStartup(MAKEWORD(2,2), &wsaData) == NO_ERROR)
       printf("Winsock Initialisation succeeded...\n");
```

https://hemanthrajhemu.github.io

```
 else
 {
   printf("Winsock Initialisation failed...\n");
   return;
 }
 //***********************************************************
 // Create a SOCKET for connecting to server
 SOCKET MySocket;
 MySocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
 if (MySocket == INVALID_SOCKET)
 {
   printf("Socket Creation failed...\n");
   WSACleanup();
   return;
 }
 else
  {
   printf("Successfully created the socket...\n");
 //***********************************************************
 // Set the Socket type, IP address and port of the server
 sockaddr_in ServerParams;
 ServerParams.sin_family = AF_INET;
 ServerParams.sin_addr.s_addr = inet_addr( SERVER );
 ServerParams.sin_port = htons( PORT );
 //***********************************************************
 // Connect to server.
 if ( connect( MySocket, (SOCKADDR*) & ServerParams, sizeof(ServerParams) )
   == SOCKET_ERROR) {
   printf( "Connecting to Server failed...\n" );
   WSACleanup();
   return;
  }
  else
{
  printf("Successfully Connected to the server...\n");

  //***********************************************************
  // Send command to server
if (send(MySocket, sendbuf, (int)strlen(sendbuf), 0 )
      == SOCKET_ERROR) {
    printf("Sending data to server failed...\n");
    closesocket(MySocket);
    WSACleanup();
    return;
}

else
{
printf("Successfully sent command to server...\n");
```

https://hemanthrajhemu.github.io

```
 //*******************************************************
 // Receive a data packet
   if (recv(MySocket, recvbuf, recvbuflen, 0) > 0 )
       printf("Successfully Received a byte…\n
     The received byte is %s\n", recvbuf);
   else
     printf("No response from server…\n");
 //*******************************************************
 //Close Socket
   closesocket(MySocket);
   WSACleanup();
   return;
}
}
}
}
}
```

The above application tries to connect to a server machine with IP address 172.168.0.1 and port number 5000. Change the values of *SERVER* and *PORT* to connect to a machine with different IP address and port number. If the connection is success, it sends the data "Hi from Client" to the server and waits for a response from the server and finally terminates the connection.

Under Windows, the socket function library *Winsock* should be initiated before using the socket related functions. The function *WSAStartup* performs this initiation. The *socket()* function call creates a socket. The socket type, connection type and protocols for communication are the parameters for socket creation. Here the socket type is *INET* (*AF_INET*) and connection type is stream socket (*SOCK_STREAM*). The protocol selected for communication is *TCP/IP* (*IPPROTO_TCP*). After creating the socket it is connected to a server. For connecting to server, the server address and port number should be indicated in the connection request. The *sockaddr_in* structure specifies the socket type, IP address and port of the server to be connected to. The *connect ()* function connects the socket with a specified server. If the server grants the connection request, the *connect ()* function returns success. The *send()* function is used for sending data to a server. It takes the socket name and data to be sent as parameters. Data from server is received using the function call *recv()*. It takes the socket name and details of buffer to hold the received data as parameters. The TCP/IP network stack expects network byte order (Big Endian: Higher order byte of the data is stored in lower memory address location) for data. The function *htons()* converts the byte order of an unsigned short integer to the network order. The *closesocket()* function closes the socket connection. On the server side, the server creates a socket using the function *socket()* and binds the socket with a port using the *bind()* function. It listens to the port bonded to the socket for any incoming connection request. The function *listen()* performs this. Upon receiving a connection request, the server accepts it. The function *accept()* performs the accepting operation. Now the connectivity is established. Server can receive and transmit data using the function calls *recv()* and *send()* respectively. The implementation of the server application is left to the readers as an exercise.

10.8 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

https://hemanthrajhemu.github.io

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 synchronised properly. The following sections describe the major task communication synchronisation issues observed in multitasking and the commonly adopted synchronisation techniques to overcome these issues.

10.8.1 Task Communication/Synchronisation Issues

10.8.1.1 Racing Let us have a look at the following piece of code:

```
#include <windows.h>
#include <stdio.h>
//*******************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two processes Process A and Process B
char Buffer[10] = {1,2,3,4,5,6,7,8,9,10};
short int counter = 0;
//*******************************************************************
// Process A
void Process_A (void) {
int i;
  for (i =0; i<5; i++)
{
    if (Buffer[i] > 0)
       counter++;
}
}
//*******************************************************************
// Process B
void Process_B(void) {
int j;
  for (j =5; j<10; j++)
{
    if (Buffer[j] > 0)
       counter++;
}
}
//*******************************************************************
//Main Thread.
int main() {
       DWORD id;
```

https://hemanthrajhemu.github.io

```
    CreateThread(NULL,0,
                    (LPTHREAD_START_ROUTINE)Process_A,
                    (LPVOID)0,0,&id);
    CreateThread(NULL,0,
                    (LPTHREAD_START_ROUTINE)Process_B,
                    (LPVOID)0,0,&id);
    Sleep(100000);
    return 0;
}
```

From a programmer perspective the value of *counter* will be 10 at the end of execution of processes A & B. But 'it need not be always' in a real world execution of this piece of code under a multitasking kernel. The results depend on the process scheduling policies adopted by the OS kernel. Now let's dig into the piece of code illustrated above. The program statement *counter++;* looks like a single statement from a high level programming language ('C' language) perspective. The low level implementation of this statement is dependent on the underlying processor instruction set and the (cross) compiler in use. The low level implementation of the high level program statement *counter++;* under Windows XP operating system running on an *Intel Centrino Duo* processor is given below. The code snippet is compiled with Microsoft Visual Studio 6.0 compiler.

```
mov eax,dword ptr [ebp-4];Load counter in Accumulator
add eax,1 ; Increment Accumulator by 1
mov dword ptr [ebp-4],eax ;Store counter with Accumulator
```

At the processor instruction level, the value of the variable *counter* is loaded to the Accumulator register (***EAX*** register). The memory variable *counter* is represented using a pointer. The base pointer register (***EBP*** register) is used for pointing to the memory variable *counter*. After loading the contents of the variable *counter* to the Accumulator, the Accumulator content is incremented by one using the add instruction. Finally the content of Accumulator is loaded to the memory location which represents the variable counter. Both the processes Process A and Process B contain the program statement *counter++;* Translating this into the machine instruction.

Process A	**Process B**
mov eax,dword ptr [ebp-4]	mov eax,dword ptr [ebp-4]
add eax,1	add eax,1
mov dword ptr [ebp-4],eax	mov dword ptr [ebp-4],eax

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. 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++;*

https://hemanthrajhemu.github.io

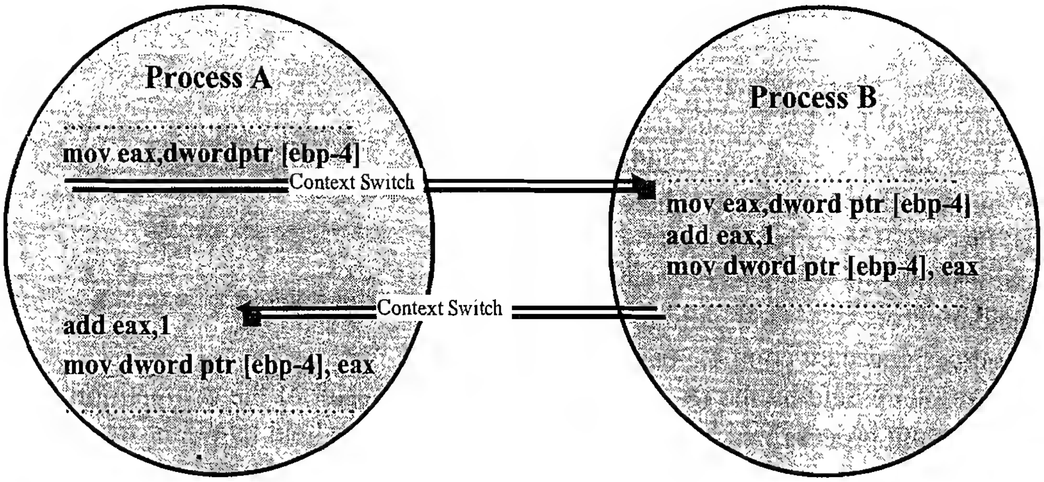


Fig. 10.23 **Race condition**

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. We will discuss this technique in more detail under the topic '*Task Synchronisation techniques*' in a later section of this chapter.

To summarise, *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.

10.8.1.2 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. A situation very similar to our traffic jam issues in a junction as illustrated in Fig. 10.24.

Fig. 10.24 **Deadlock visualisation**

† Atomic Operation: Operations which are non-interruptible.

https://hemanthrajhemu.github.io

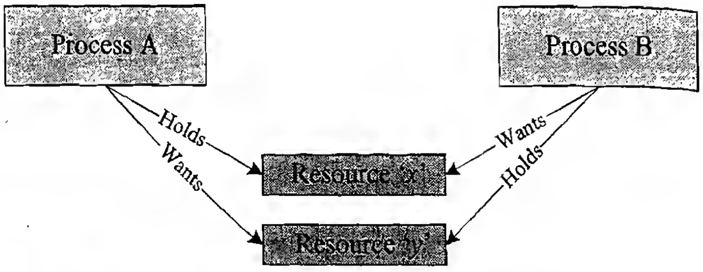


Fig. 10.25 **Scenarios leading to deadlock**

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). 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. When the vehicles from different directions compete to cross the junction, deadlock (traffic jam) condition is resulted. Once a

https://hemanthrajhemu.github.io

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 (Fig. 10.26).

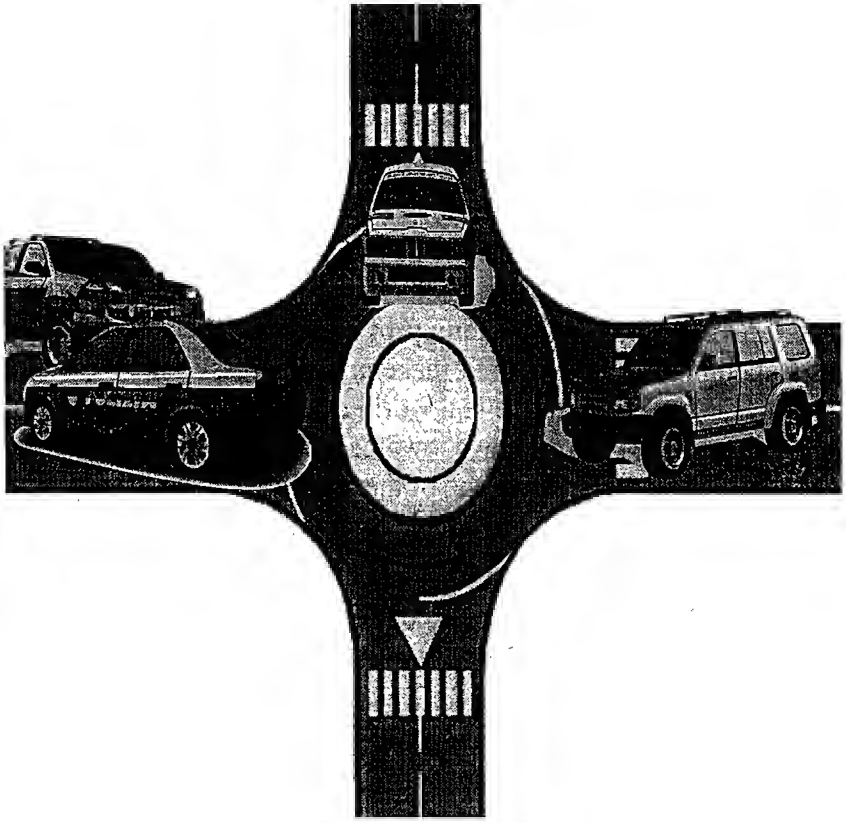


Fig. 10.26 ***'Back up cars'* technique for deadlock recovery**

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.

https://hemanthrajhemu.github.io

- 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 *Livelock* 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. We will make the livelock, the scenario more clear in a later section–*The Dining Philosophers' Problem*, of this chapter.

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 measures of deadlock, scheduling policies favouring high priority tasks and tasks with shortest execution time, etc.

10.8.1.3 The Dining Philosophers' Problem The '*Dining philosophers' problem*' is an interesting example for synchronisation issues in resource utilisation. The terms 'dining', 'philosophers', etc. may sound awkward in the operating system context, but it is the best way to explain technical things abstractly using non-technical terms. Now coming to the problem definition:

Five philosophers (It can be '*n*'. The number 5 is taken for illustration) are sitting around a round table, involved in eating and brainstorming (Fig. 10.37). At any point of time each philosopher will be in any one of the three states: eating, hungry or brainstorming. (While eating the philosopher is not involved in brainstorming and while brainstorming the philosopher is not involved in eating). For eating, each philosopher requires 2 forks. There are only 5 forks available on the dining table ('*n*' for '*n*' number of philosophers) and they are arranged in a fashion one fork in between two philosophers. The philosopher can only use the forks on his/her immediate left and right that too in the order pickup the left fork first and then the right fork. Analyse the situation and explain the possible outcomes of this scenario.

Let's analyse the various scenarios that may occur in this situation.

Scenario 1: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed since two forks are required for eating the spaghetti present in the plate. Philosopher 1 thinks that Philosopher 2 sitting to the right of him/her will put the fork down and waits for it. Philosopher 2 thinks that Philosopher 3 sitting to the right of him/her will

https://hemanthrajhemu.github.io

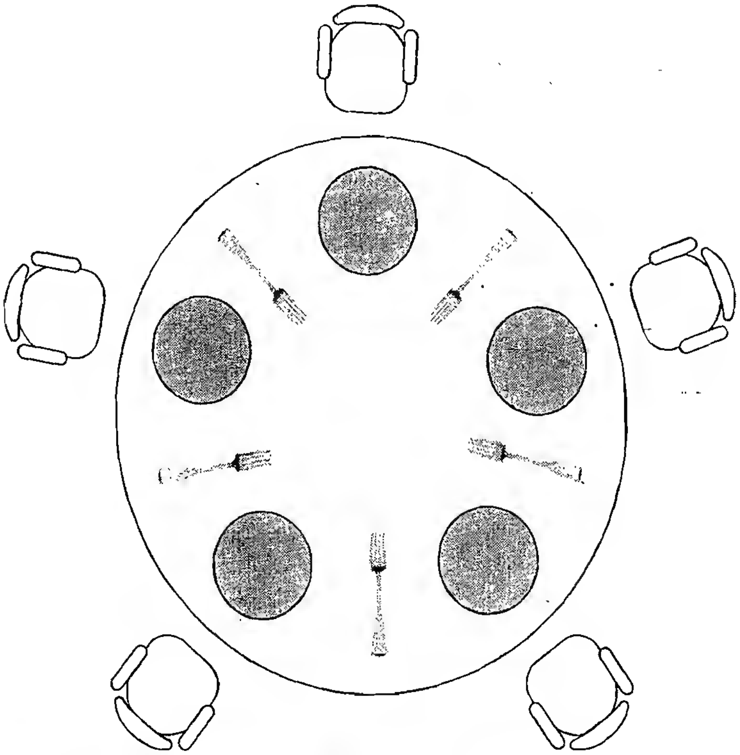


Fig. 10.27 **Visualisation of the '*Dining Philosophers problem*'**

put the fork down and waits for it, and so on. This forms a circular chain of un-granted requests. If the philosophers continue in this state waiting for the fork from the philosopher sitting to the right of each, they will not make any progress in eating and this will result in *starvation* of the philosophers and *deadlock*.

Scenario 2: All the philosophers start brainstorming together. One of the philosophers is hungry and he/she picks up the left fork. When the philosopher is about to pick up the right fork, the philosopher sitting to his right also become hungry and tries to grab the left fork which is the right fork of his neighbouring philosopher who is trying to lift it, resulting in a '*Race condition*'.

Scenario 3: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed, since two forks are required for eating the spaghetti present in the plate. Each of them anticipates that the adjacently sitting philosopher will put his/her fork down and waits for a fixed duration and after this puts the fork down. Each of them again tries to lift the fork after a fixed duration of time. Since all philosophers are trying to lift the fork at the same time, none of them will be able to grab two forks. This condition leads to *livelock* and *starvation* of philosophers, where each philosopher tries to do something, but they are unable to make any progress in achieving the target.

Figure 10.28 illustrates these scenarios.

Solution: We need to find out alternative solutions to avoid the *deadlock*, *livelock*, *racing* and *starvation* condition that may arise due to the concurrent access of forks by philosophers. This situation can be handled in many ways by allocating the forks in different allocation techniques including Round Robin allocation, FIFO allocation, etc. But the requirement is that the solution should be optimal, avoiding

https://hemanthrajhemu.github.io

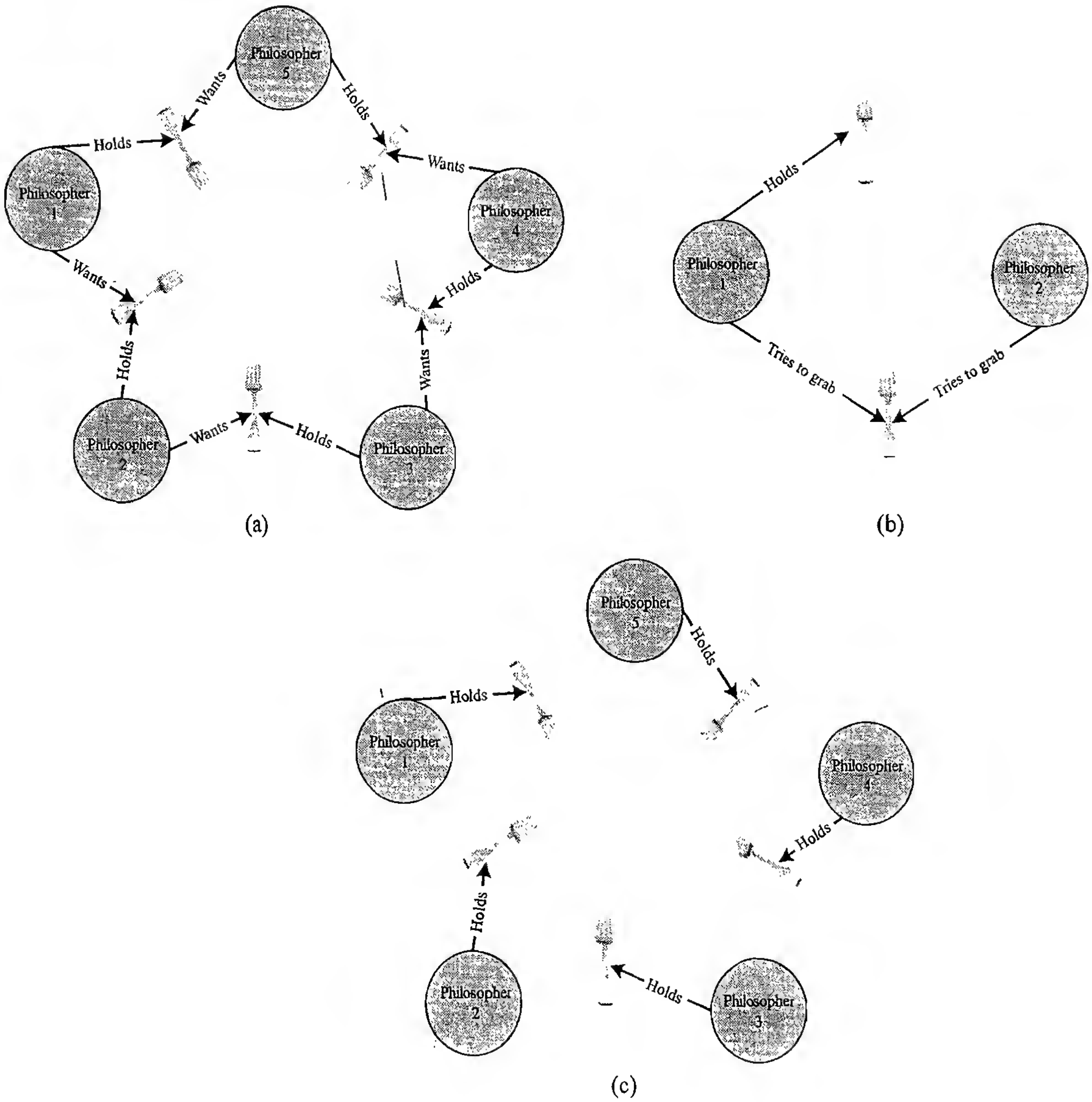


Fig. 10.28 **The 'Real Problems' in the '*Dining Philosophers problem*' (a) Starvation and Deadlock (b) Racing (c) Livelock and Starvation**

deadlock and starvation of the philosophers and allowing maximum number of philosophers to eat at a time. One solution that we could think of is:

- Imposing rules in accessing the forks by philosophers, like: The philosophers should put down the fork he/she already have in hand (left fork) after waiting for a fixed duration for the second fork (right fork) and should wait for a fixed time before making the next attempt.

This solution works fine to some extent, but, if all the philosophers try to lift the forks at the same time, a *livelock* situation is resulted.

Another solution which gives maximum concurrency that can be thought of is each philosopher acquires a semaphore (mutex) before picking up any fork. When a philosopher feels hungry he/she checks whether the philosopher sitting to the left and right of him is already using the fork, by checking the state

https://hemanthrajhemu.github.io

of the associated semaphore. If the forks are in use by the neighbouring philosophers, the philosopher waits till the forks are available. A philosopher when finished eating puts the forks down and informs the philosophers sitting to his/her left and right, who are hungry (waiting for the forks), by signalling the semaphores associated with the forks. We will discuss about semaphores and mutexes at a latter section of this chapter. In the operating system context, the dining philosophers represent the processes and forks represent the resources. The dining philosophers' problem is an analogy of processes competing for shared resources and the different problems like racing, deadlock, starvation and livelock arising from the competition.

10.8.1.4 Producer-Consumer/Bounded Buffer Problem Producer-Consumer problem is a common data sharing problem where two processes concurrently access a shared buffer with fixed size. A thread/process which produces data is called '*Producer thread/process*' and a thread/process which consumes the data produced by a producer thread/process is known as '*Consumer thread/process*'. Imagine a situation where the producer thread keeps on producing data and puts it into the buffer and the consumer thread keeps on consuming the data from the buffer and there is no synchronisation between the two. There may be chances where in which the producer produces data at a faster rate than the rate at which it is consumed by the consumer. This will lead to '*buffer overrun*' where the producer tries to put data to a full buffer. If the consumer consumes data at a faster rate than the rate at which it is produced by the producer, it will lead to the situation '*buffer under-run*' in which the consumer tries to read from an empty buffer. Both of these conditions will lead to inaccurate data and data loss. The following code snippet illustrates the producer-consumer problem

```
#include <windows.h>
#include <stdio.h>
#define N 20 //Define buffer size as 20
int buffer[N];//Shared buffer for producer & consumer
//**********************************************************************
//Producer thread
void producer_thread(void) {
       int x;
       while(true) {
               for(x=0;x<N;x++)
               {
               //Fill buffer with random data
               buffer[x]= rand()%1000;
               printf("Produced : Buffer [%d] = %4d\n", x,
               buffer[x]);
               Sleep(25);
               }
       }
}
//**********************************************************************
//Consumer thread
void consumer_thread(void) {
       int y=0,value;
       while(true) {
               for(y=0;y<N;y++)
```

https://hemanthrajhemu.github.io

```
                {
                value=buffer[y];
                printf("Consumed : Buffer [%d] = %4d\n", y, value);
                Sleep(20);
                }
        }
}
//**********************************************************************
//Main Thread
int main()
{
        DWORD thread_id;
        //Create Producer thread
        CreateThread(NULL,0,
                        (LPTHREAD_START_ROUTINE)producer_thread,
                        NULL,0,&thread_id);
        //Create Consumer thread
        CreateThread(NULL,0,
                        (LPTHREAD_START_ROUTINE)consumer_thread,
                        NULL,0,&thread_id);
        //Wait for some time and exit
        Sleep(500);
        return 0;
}
```

Here the 'producer thread' produces random numbers and puts it in a buffer of size 20. If the 'producer thread' fills the buffer fully it re-starts the filling of the buffer from the bottom. The 'consumer thread' consumes the data produced by the 'producer thread'. For consuming the data, the 'consumer thread' reads the buffer which is shared with the 'producer thread'. Once the 'consumer thread' consumes all the data, it starts consuming the data from the bottom of the buffer. These two threads run independently and are scheduled for execution based on the scheduling policies adopted by the OS. The different situations that may arise based on the scheduling of the 'producer thread' and 'consumer thread' is listed below.

1. 'Producer thread' is scheduled more frequently than the 'consumer thread': There are chances for overwriting the data in the buffer by the 'producer thread'. This leads to inaccurate data.
2. Consumer thread' is scheduled more frequently than the 'producer thread': There are chances for reading the old data in the buffer again by the 'consumer thread'. This will also lead to inaccurate data.

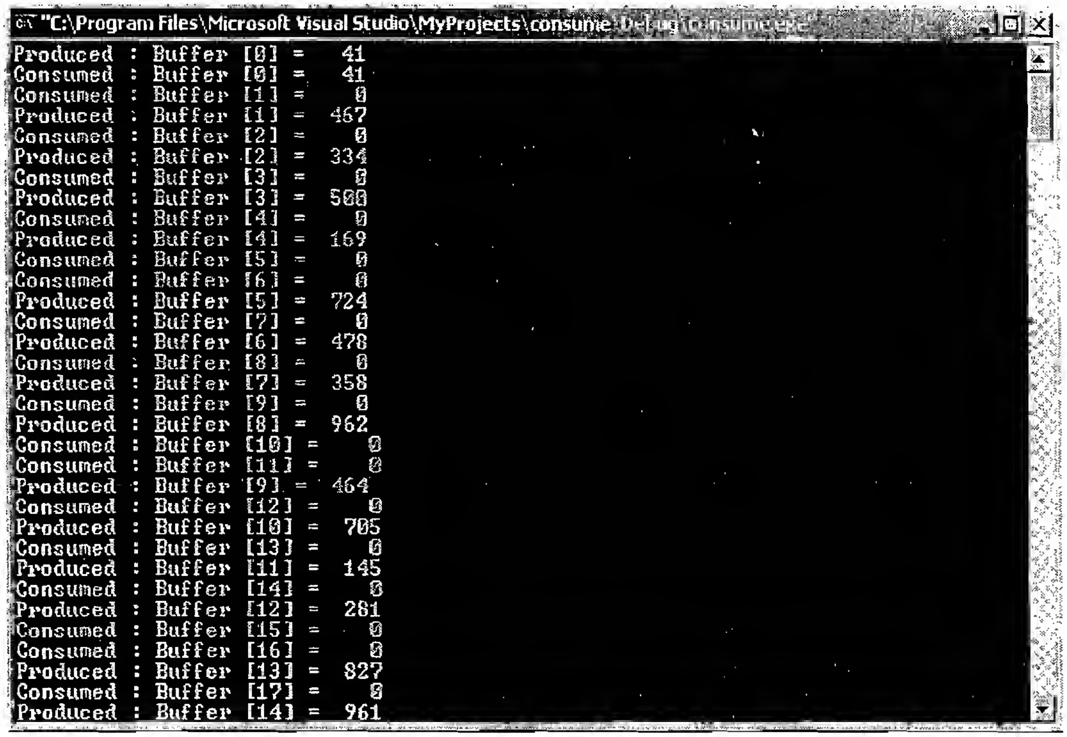
The output of the above program when executed on a Windows XP machine is shown in Fig. 10.29.

The output shows that the consumer thread runs faster than the producer thread and most often leads to buffer under-run and thereby inaccurate data.

> ***Note***
> It should be noted that the scheduling of the threads '*producer_thread*' and '*consumer_thread*' is OS kernel scheduling policy dependent and you may not get the same output all the time when you run this piece of code in Windows XP.

The producer-consumer problem can be rectified in various methods. One simple solution is the '*sleep and wake-up*'. The '*sleep and wake-up*' can be implemented in various process synchronisation techniques like semaphores, mutex, monitors, etc. We will discuss it in a latter section of this chapter.

https://hemanthrajhemu.github.io


"C:\Program Files\Microsoft Visual Studio\MyProjects\consume\Debug\consume.exe"

```
Produced : Buffer [0] =   41
Consumed : Buffer [0] =   41
Consumed : Buffer [1] =    0
Produced : Buffer [1] =  467
Consumed : Buffer [2] =    0
Produced : Buffer [2] =  334
Consumed : Buffer [3] =    0
Produced : Buffer [3] =  500
Consumed : Buffer [4] =    0
Produced : Buffer [4] =  169
Consumed : Buffer [5] =    0
Consumed : Buffer [6] =    0
Produced : Buffer [5] =  724
Consumed : Buffer [7] =    0
Produced : Buffer [6] =  478
Consumed : Buffer [8] =    0
Produced : Buffer [7] =  358
Consumed : Buffer [9] =    0
Produced : Buffer [8] =  962
Consumed : Buffer [10] =    0
Consumed : Buffer [11] =    0
Produced : Buffer [9] =  464
Consumed : Buffer [12] =    0
Produced : Buffer [10] =  705
Consumed : Buffer [13] =    0
Produced : Buffer [11] =  145
Consumed : Buffer [14] =    0
Produced : Buffer [12] =  281
Consumed : Buffer [15] =    0
Consumed : Buffer [16] =    0
Produced : Buffer [13] =  827
Consumed : Buffer [17] =    0
Produced : Buffer [14] =  961
```

Fig. 10.29 **Output of Win32 program illustrating producer consumer problem**

10.8.1.5 Readers-Writers Problem The Readers-Writers problem is a common issue observed in processes competing for limited shared resources. The Readers-Writers problem is characterised by multiple processes trying to read and write shared data concurrently. A typical real-world example for the Readers-Writers problem is the banking system where one process tries to read the account information like available balance and the other process tries to update the available balance for that account. This may result in inconsistent results. If multiple processes try to read a shared data concurrently it may not create any impacts, whereas when multiple processes try to write and read concurrently it will definitely create inconsistent results. Proper synchronisation techniques should be applied to avoid the readers-writers problem. We will discuss about the various synchronisation techniques in a later section of this chapter.

10.8.1.6 Priority Inversion Priority inversion is the byproduct of the combination of blocking based (lock based) process synchronisation and pre-emptive priority scheduling. '*Priority inversion*' is the condition in which a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task, and a medium priority task which doesn't require the shared resource continue its execution by preempting the low priority task (Fig. 10.30). Priority based preemptive scheduling technique ensures that a high priority task is always executed first, whereas the lock based process synchronisation mechanism (like mutex, semaphore, etc.) ensures that a process will not access a shared resource, which is currently in use by another process. The synchronisation technique is only interested in avoiding conflicts that may arise due to the concurrent access of the shared resources and not at all bothered about the priority of the process which tries to access the shared resource. In fact, the priority based preemption and lock based synchronisation are the two contradicting OS primitives. Priority inversion is better explained with the following scenario:

Let Process A, Process B and Process C be three processes with priorities High, Medium and Low respectively. Process A and Process C share a variable 'X' and the access to this variable is synchronised

https://hemanthrajhemu.github.io

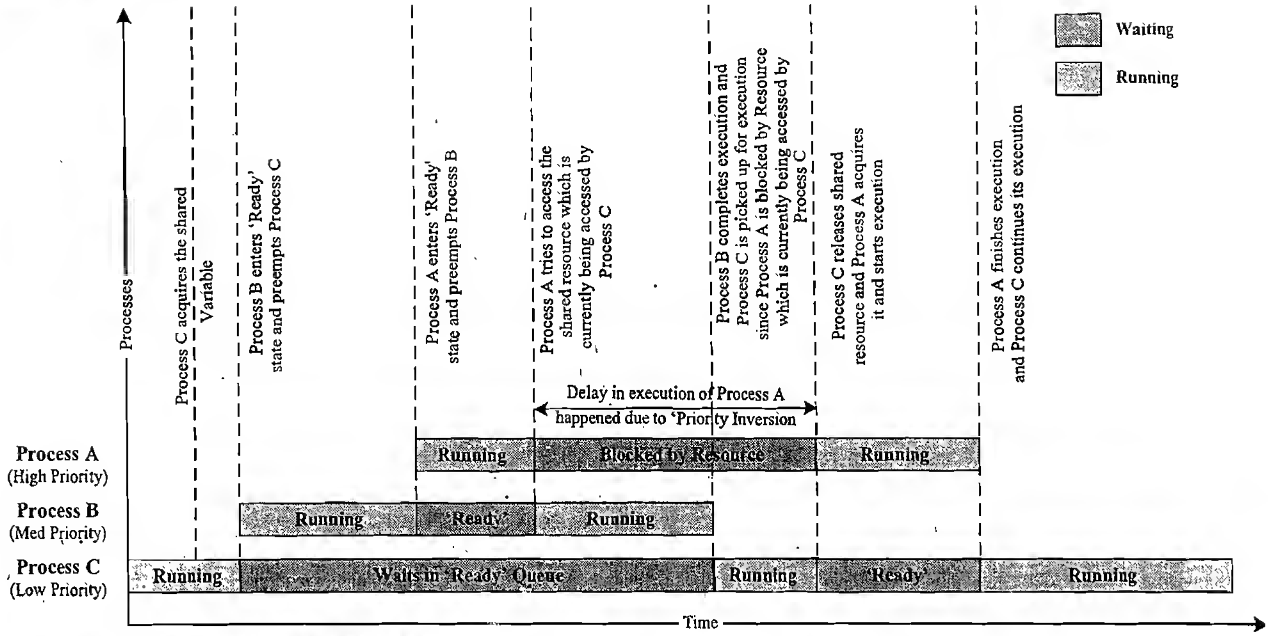

Processes
Process C acquires the shared Variable
Process B enters 'Ready' state and preempts Process C
Process A enters 'Ready' state and preempts Process B
Process A tries to access the shared resource which is currently being accessed by Process C
Process B completes execution and Process C is picked up for execution since Process A is blocked by Resource which is currently being accessed by Process C
Process C releases shared resource and Process A acquires it and starts execution
Process A finishes execution and Process C continues its execution
Waiting
Running
Delay in execution of Process A happened due to 'Priority Inversion
Process A (High Priority)
Process B (Med Priority)
Process C (Low Priority)
Running
Blocked by Resource
Running
Running
'Ready'
Running
Running
Waits in 'Ready' Queue
Running
'Ready'
Running
Time

Fig. 10.30 **Priority Inversion Problem**

through a mutual exclusion mechanism like *Binary Semaphore S*. Imagine a situation where Process C is ready and is picked up for execution by the scheduler and 'Process C' tries to access the shared variable 'X'. 'Process C' acquires the 'Semaphore S' to indicate the other processes that it is accessing the shared variable 'X'. Immediately after 'Process C' acquires the 'Semaphore S', 'Process B' enters the 'Ready' state. Since 'Process B' is of higher priority compared to 'Process C', 'Process C' is preempted and 'Process B' starts executing. Now imagine 'Process A' enters the 'Ready' state at this stage. Since 'Process A' is of higher priority than 'Process B', 'Process B' is preempted and 'Process A' is scheduled for execution. 'Process A' involves accessing of shared variable 'X' which is currently being accessed by 'Process C'. Since 'Process C' acquired the semaphore for signalling the access of the shared variable 'X', 'Process A' will not be able to access it. Thus 'Process A' is put into blocked state (This condition is called Pending on resource). Now 'Process B' gets the CPU and it continues its execution until it relinquishes the CPU voluntarily or enters a wait state or preempted by another high priority task. The highest priority process 'Process A' has to wait till 'Process C' gets a chance to execute and release the semaphore. This produces unwanted delay in the execution of the high priority task which is supposed to be executed immediately when it was 'Ready'.

Priority inversion may be sporadic in nature but can lead to potential damages as a result of missing critical deadlines. Literally speaking, priority inversion 'inverts' the priority of a high priority task with that of a low priority task. Proper workaround mechanism should be adopted for handling the priority inversion problem. The commonly adopted priority inversion workarounds are:

Priority Inheritance: A low-priority task that is currently accessing (by holding the lock) a shared resource requested by a high-priority task temporarily '*inherits*' the priority of that high-priority task, from the moment the high-priority task raises the request. Boosting the priority of the low priority task to that of the priority of the task which requested the shared resource holding by the low priority task eliminates the preemption of the low priority task by other tasks whose priority are below that of the task requested the shared resource and thereby reduces the delay in waiting to get the resource requested by the high priority task. The priority of the low priority task which is temporarily boosted to high is

https://hemanthrajhemu.github.io

brought to the original value when it releases the shared resource. Implementation of Priority inheritance workaround in the priority inversion problem discussed for Process A, Process B and Process C example will change the execution sequence as shown in Fig. 10.31.

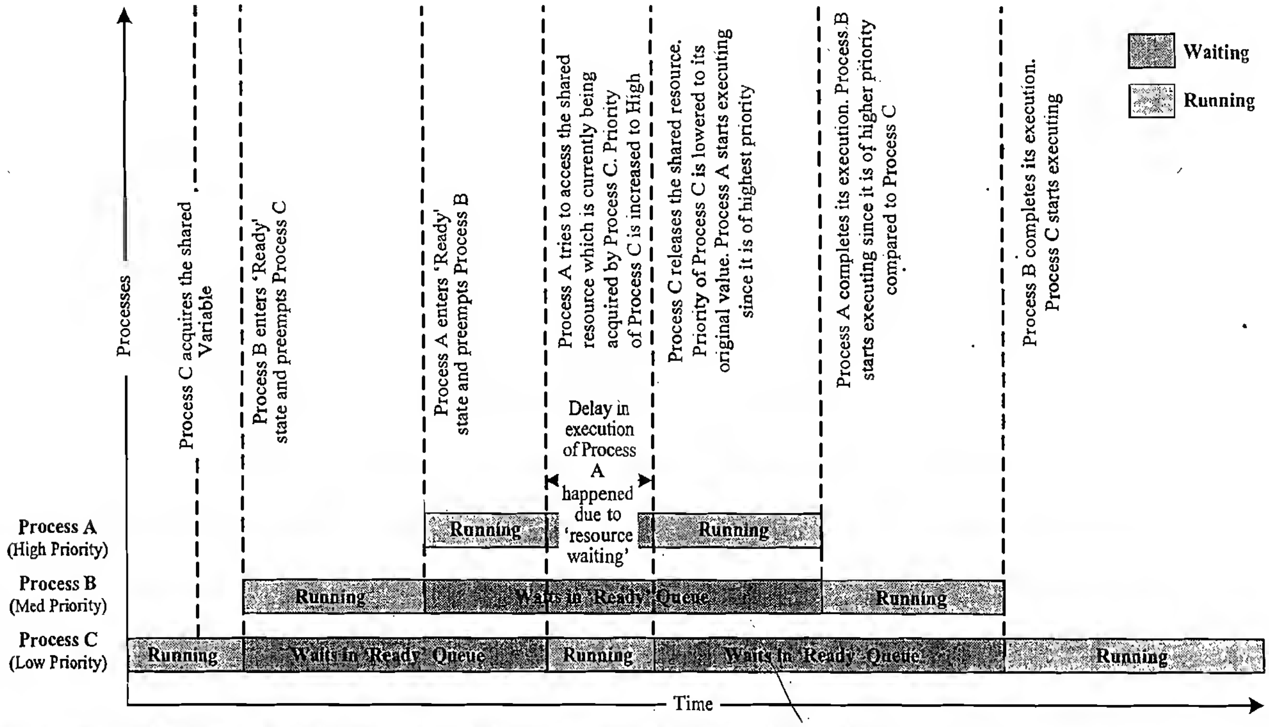

Processes
Process C acquires the shared Variable
Process B enters 'Ready' state and preempts Process C
Process A enters 'Ready' state and preempts Process B
Process A tries to access the shared resource which is currently being acquired by Process C. Priority of Process C is increased to High
Process C releases the shared resource. Priority of Process C is lowered to its original value. Process A starts executing since it is of highest priority
Process A completes its execution. Process B starts executing since it is of higher priority compared to Process C
Process B completes its execution. Process C starts executing
Waiting
Running
Delay in execution of Process A happened due to 'resource waiting'
Process A (High Priority)
Process B (Med Priority)
Process C (Low Priority)
Running
Running
Running
Waits in 'Ready' Queue
Running
Running
Waits in 'Ready' Queue
Running
Waits in 'Ready' Queue
Running
Time

Fig. 10.31 Handling Priority Inversion Problem with Priority Inheritance

Priority inheritance is only a work around and it will not eliminate the delay in waiting the high priority task to get the resource from the low priority task. The only thing is that it helps the low priority task to continue its execution and release the shared resource as soon as possible. The moment, at which the low priority task releases the shared resource, the high priority task kicks the low priority task out and grabs the CPU – A true form of selfishness☺. Priority inheritance handles priority inversion at the cost of run-time overhead at scheduler. It imposes the overhead of checking the priorities of all tasks which tries to access shared resources and adjust the priorities dynamically.

Priority Ceiling: In '*Priority Ceiling*', a priority is associated with each shared resource. The priority associated to each resource is the priority of the highest priority task which uses this shared resource. This priority level is called 'ceiling priority'. Whenever a task accesses a shared resource, the scheduler elevates the priority of the task to that of the ceiling priority of the resource. If the task which accesses the shared resource is a low priority task, its priority is temporarily boosted to the priority of the highest priority task to which the resource is also shared. This eliminates the pre-emption of the task by other medium priority tasks leading to priority inversion. The priority of the task is brought back to the original level once the task completes the accessing of the shared resource. 'Priority Ceiling' brings the added advantage of sharing resources without the need for synchronisation techniques like locks. Since the priority of the task accessing a shared resource is boosted to the highest priority of the task among which the resource is shared, the concurrent access of shared resource is automatically handled. Another advantage of 'Priority Ceiling' technique is that all the overheads are at compile time instead of

https://hemanthrajhemu.github.io

run-time. Implementation of 'priority ceiling' workaround in the priority inversion problem discussed for Process A, Process B and Process C example will change the execution sequence as shown in Fig. 10.32.

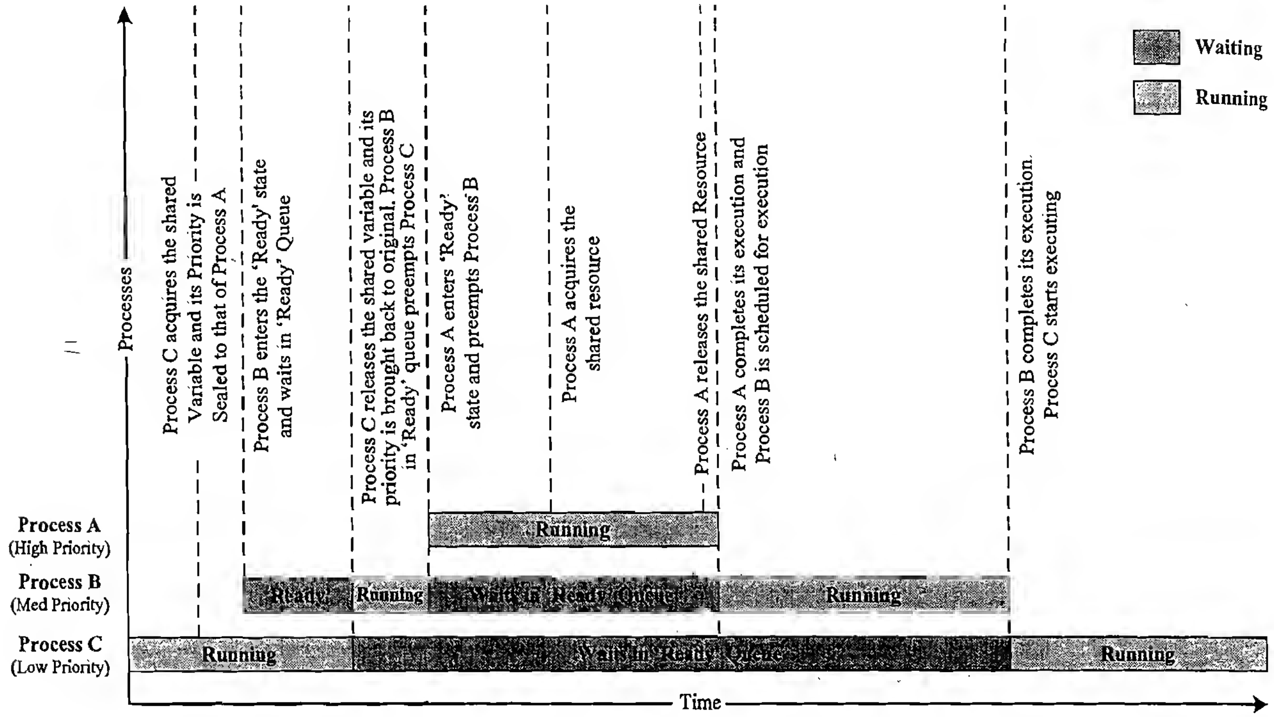

Processes
Process C acquires the shared Variable and its Priority is Sealed to that of Process A
Process B enters the 'Ready' state and waits in 'Ready' Queue
Process C releases the shared variable and its priority is brought back to original. Process B in 'Ready' queue preempts Process C
Process A enters 'Ready' state and preempts Process B
Process A acquires the shared resource
Process A releases the shared Resource
Process A completes its execution and Process B is scheduled for execution
Process B completes its execution. Process C starts executing
Waiting
Running
Process A (High Priority)
Process B (Med Priority)
Process C (Low Priority)
Running
Ready
Running
Wait in Ready Queue
Running
Running
Wait in Ready Queue
Running
Time

Fig. 10.32 **Handling Priority Inversion Problem with Priority Ceiling**

The biggest drawback of 'Priority Ceiling' is that it may produce *hidden priority inversion*. With 'Priority Ceiling' technique, the priority of a task is always elevated no matter another task wants the shared resources. This unnecessary priority elevation always boosts the priority of a low priority task to that of the highest priority tasks among which the resource is shared and other tasks with priorities higher than that of the low priority task is not allowed to preempt the low priority task when it is accessing a shared resource. This always gives the low priority task the luxury of running at high priority when accessing shared resources☺.

10.8.2 Task Synchronisation Techniques

So far we discussed about the various task/process synchronisation issues encountered in multitasking systems due to concurrent resource access. Now let's have a discussion on the various techniques used for synchronisation in concurrent access in multitasking. Process/Task synchronisation is essential for

1. Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.
2. Ensuring proper sequence of operation across processes. The producer consumer problem is a typical example for processes requiring proper sequence of operation. In producer consumer problem, accessing the shared buffer by different processes is not the issue, the issue is the writing process should write to the shared buffer only if the buffer is not full and the consumer thread

https://hemanthrajhemu.github.io

should not read from the buffer if it is empty. Hence proper synchronisation should be provided to implement this sequence of operations.

3. Communicating between processes.

The code memory area which holds the program instructions (piece of code) for accessing a shared resource (like shared memory, shared variables, etc.) is known as '*critical section*'. In order to synchronise the access to shared resources, the access to the critical section should be exclusive. The exclusive access to critical section of code is provided through mutual exclusion mechanism. Let us have a look at how mutual exclusion is important in concurrent access. Consider two processes *Process* A and *Process B* running on a multitasking system. *Process A* is currently running and it enters its critical section. Before *Process A* completes its operation in the critical section, the scheduler preempts *Process A* and schedules *Process B* for execution (*Process B* is of higher priority compared to *Process A*). *Process B* also contains the access to the critical section which is already in use by *Process A*. If *Process B* continues its execution and enters the critical section which is already in use by *Process A*, a racing condition will be resulted. A mutual exclusion policy enforces mutually exclusive access of critical sections.

Mutual exclusions can be enforced in different ways. Mutual exclusion blocks a process. Based on the behaviour of the blocked process, mutual exclusion methods can be classified into two categories. In the following section we will discuss them in detail.

10.8.2.1 Mutual Exclusion through Busy Waiting/Spin Lock '*Busy waiting*' is the simplest method for enforcing mutual exclusion. The following code snippet illustrates how 'Busy waiting' enforces mutual exclusion.

```
//Inside parent thread/main thread corresponding to a process
bool bFlag; //Global declaration of lock Variable.
bFlag= FALSE; //Initialise the lock to indicate it is available.
//...................................................................
//Inside the child threads/threads of a process
while(bFlag == TRUE); //Check the lock for availability
bFlag=TRUE; //Lock is available. Acquire the lock
//Rest of the source code dealing with shared resource access
```

The 'Busy waiting' technique uses a lock variable for implementing mutual exclusion. Each process/thread checks this lock variable before entering the critical section. The lock is set to '*1*' by a process/thread if the process/thread is already in its critical section; otherwise the lock is set to '*0*'. The major challenge in implementing the lock variable based synchronisation is the non-availability of a single atomic instruction† which combines the reading, comparing and setting of the lock variable. Most often the three different operations related to the locks, viz. the operation of Reading the lock variable, checking its present value and setting it are achieved with multiple low level instructions. The low level implementation of these operations are dependent on the underlying processor instruction set and the (cross) compiler in use. The low level implementation of the 'Busy waiting' code snippet, which we discussed earlier, under Windows XP operating system running on an *Intel Centrino Duo* processor is given below. The code snippet is compiled with Microsoft Visual Studio 6.0 compiler.

```
--- D:\Examples\counter.cpp ----------------------------------------
1:     #include <stdio.h>
2:     #include <windows.h>
```

† Atomic Instruction: Instruction whose execution is uninterruptible.

https://hemanthrajhemu.github.io

run-time. Implementation of 'priority ceiling' workaround in the priority inversion problem discussed for Process A, Process B and Process C example will change the execution sequence as shown in Fig. 10.32.

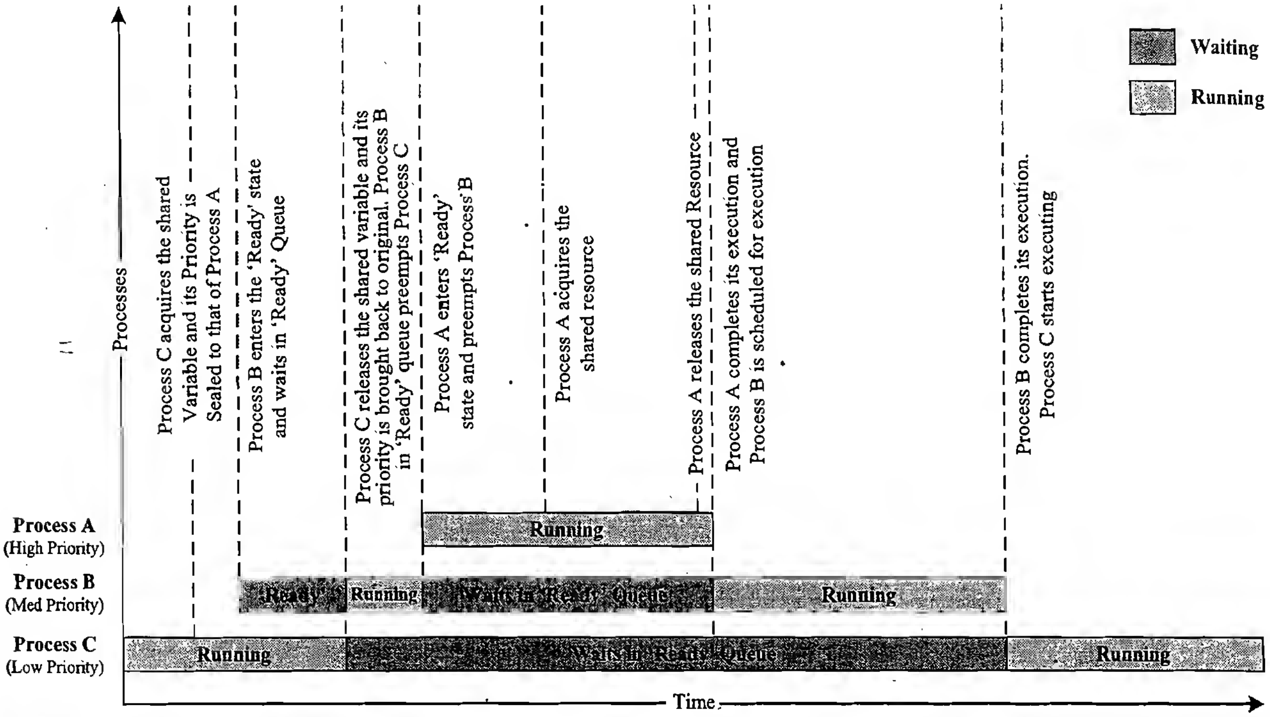

Processes
Process C acquires the shared Variable and its Priority is Sealed to that of Process A
Process B enters the 'Ready' state and waits in 'Ready' Queue
Process C releases the shared variable and its priority is brought back to original. Process B in 'Ready' queue preempts Process C
Process A enters 'Ready' state and preempts Process B
Process A acquires the shared resource
Process A releases the shared Resource
Process A completes its execution and Process B is scheduled for execution
Process B completes its execution. Process C starts executing
Waiting
Running
Process A (High Priority)
Process B (Med Priority)
Process C (Low Priority)
Running
Ready
Running
Waits in 'Ready' Queue
Running
Running
Waits in 'Ready' Queue
Running
Time

Fig. 10.32 **Handling Priority Inversion Problem with Priority Ceiling**

The biggest drawback of 'Priority Ceiling' is that it may produce *hidden priority inversion.* With 'Priority Ceiling' technique, the priority of a task is always elevated no matter another task wants the shared resources. This unnecessary priority elevation always boosts the priority of a low priority task to that of the highest priority tasks among which the resource is shared and other tasks with priorities higher than that of the low priority task is not allowed to preempt the low priority task when it is accessing a shared resource. This always gives the low priority task the luxury of running at high priority when accessing shared resources☺.

10.8.2 Task Synchronisation Techniques

So far we discussed about the various task/process synchronisation issues encountered in multitasking systems due to concurrent resource access. Now let's have a discussion on the various techniques used for synchronisation in concurrent access in multitasking. Process/Task synchronisation is essential for

1. Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.
2. Ensuring proper sequence of operation across processes. The producer consumer problem is a typical example for processes requiring proper sequence of operation. In producer consumer problem, accessing the shared buffer by different processes is not the issue, the issue is the writing process should write to the shared buffer only if the buffer is not full and the consumer thread

https://hemanthrajhemu.github.io

should not read from the buffer if it is empty. Hence proper synchronisation should be provided to implement this sequence of operations.

3. Communicating between processes.

The code memory area which holds the program instructions (piece of code) for accessing a shared resource (like shared memory, shared variables, etc.) is known as '*critical section*'. In order to synchronise the access to shared resources, the access to the critical section should be exclusive. The exclusive access to critical section of code is provided through mutual exclusion mechanism. Let us have a look at how mutual exclusion is important in concurrent access. Consider two processes *Process* A and *Process B* running on a multitasking system. *Process A* is currently running and it enters its critical section. Before *Process A* completes its operation in the critical section, the scheduler preempts *Process A* and schedules *Process B* for execution (*Process B* is of higher priority compared to *Process A*). *Process B* also contains the access to the critical section which is already in use by *Process A*. If *Process B* continues its execution and enters the critical section which is already in use by *Process A*, a racing condition will be resulted. A mutual exclusion policy enforces mutually exclusive access of critical sections.

Mutual exclusions can be enforced in different ways. Mutual exclusion blocks a process. Based on the behaviour of the blocked process, mutual exclusion methods can be classified into two categories. In the following section we will discuss them in detail.

10.8.2.1 Mutual Exclusion through Busy Waiting/Spin Lock '*Busy waiting*' is the simplest method for enforcing mutual exclusion. The following code snippet illustrates how 'Busy waiting' enforces mutual exclusion.

```
//Inside parent thread/main thread corresponding to a process
bool bFlag; //Global declaration of lock Variable.
bFlag= FALSE; //Initialise the lock to indicate it is available.
//..............................................................................
//Inside the child threads/threads of a process
while(bFlag == TRUE); //Check the lock for availability
bFlag=TRUE; //Lock is available. Acquire the lock
//Rest of the source code dealing with shared resource access
```

The 'Busy waiting' technique uses a lock variable for implementing mutual exclusion. Each process/thread checks this lock variable before entering the critical section. The lock is set to '*1*' by a process/thread if the process/thread is already in its critical section; otherwise the lock is set to '*0*'. The major challenge in implementing the lock variable based synchronisation is the non-availability of a single atomic instruction† which combines the reading, comparing and setting of the lock variable. Most often the three different operations related to the locks, viz. the operation of Reading the lock variable, checking its present value and setting it are achieved with multiple low level instructions. The low level implementation of these operations are dependent on the underlying processor instruction set and the (cross) compiler in use. The low level implementation of the 'Busy waiting' code snippet, which we discussed earlier, under Windows XP operating system running on an *Intel Centrino Duo* processor is given below. The code snippet is compiled with Microsoft Visual Studio 6.0 compiler.

```
--- D:\Examples\counter.cpp ----------------------------------------
1:      #include <stdio.h>
2:      #include <windows.h>
```

† Atomic Instruction: Instruction whose execution is uninterruptible.

https://hemanthrajhemu.github.io

```
3:
4:     int main()
5:     {
//Code memory           Opcode       Operand
00401010                push         ebp
00401011                mov          ebp,esp
00401013                sub          esp,44h
00401016                push         ebx
00401017                push         esi
00401018                push         edi
00401019                lea          edi,[ebp-44h]
0040101C                mov          ecx,11h
00401021                mov          eax,0CCCCCCCCh
00401026                rep stos     dword ptr [edi]
6:     //Inside parent thread/ main thread corresponding to a process
7:     bool bFlag; //Global declaration of lock Variable.
8:     bFlag= FALSE; //Initialise the lock to indicate it is //available.
00401028   mov        byte ptr [ebp-4],0
9:     //.................................................................
10:    //Inside the child threads/ threads of a process
11:    while(bFlag == TRUE); //Check the lock for availability
0040102C   mov        eax,dword ptr [ebp-4]
0040102F   and        eax,0FFh
00401034   cmp        eax,1
00401037   jne        main+2Bh (0040103b)
00401039   jmp        main+1Ch (0040102c)
12:    bFlag=TRUE; //Lock is available. Acquire the lock
0040103B   mov        byte ptr [ebp-4],1
```

The assembly language instructions reveals that the two high level instructions (*while(bFlag==false);* and *bFlag=true;*), corresponding to the operation of reading the lock variable, checking its present value and setting it is implemented in the processor level using six low level instructions. Imagine a situation where 'Process 1' read the lock variable and tested it and found that the lock is available and it is about to set the lock for acquiring the critical section (Fig. 10.33). But just before 'Process 1' sets the lock variable, 'Process 2' preempts 'Process 1' and starts executing. 'Process 2' contains a critical section code and it tests the lock variable for its availability. Since 'Process 1' was unable to set the lock variable, its state is still '0' and 'Process 2' sets it and acquires the critical section. Now the scheduler preempts 'Process 2' and schedules 'Process 1' before 'Process 2' leaves the critical section. Remember, 'Process 1' was preempted at a point just before setting the lock variable ('Process 1' has already tested the lock variable just before it is preempted and found that the lock is available). Now 'Process 1' sets the lock variable and enters the critical section☺. It violates the mutual exclusion policy and may produce unpredicted results.

The above issue can be effectively tackled by combining the actions of reading the lock variable, testing its state and setting the lock into a single step. This can be achieved with the combined hardware and software support. Most of the processors support a single instruction '*Test and Set Lock (TSL)*' for testing and setting the lock variable. The '*Test and Set Lock (TSL)*' instruction call copies the value of the lock variable and sets it to a nonzero value. It should be noted that the implementation and usage of

https://hemanthrajhemu.github.io

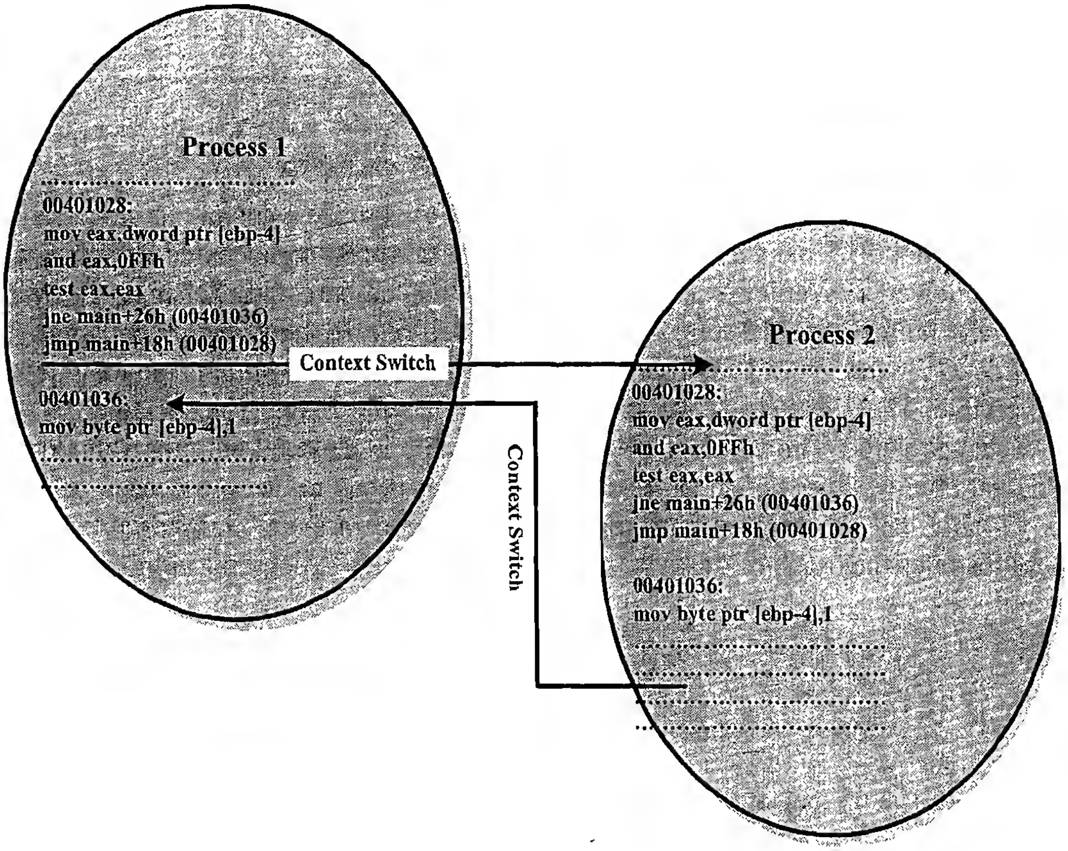


Fig. 10.33 **Issue with locks**

'*Test and Set Lock (TSL)*' instruction is processor architecture dependent. The *Intel 486* and the above family of processors support the '*Test and Set Lock (TSL)*' instruction with a special instruction *CMPXCHG*—Compare and Exchange. The usage of *CMPXCHG* instruction is given below.

```
CMPXCHG dest,src
```

This instruction compares the Accumulator (*EAX* register) with 'dest'. If the Accumulator and 'dest' contents are equal, 'dest' is loaded with 'src'. If not, the Accumulator is loaded with 'dest'. Executing this instruction changes the six status bits of the Program Control and Status register *EFLAGS*. The destination ('dest') can be a register or a memory location. The source ('src') is always a register. From a programmer's perspective the operation of *CMPXCHG* instruction can be viewed as:

```
if (accumulator == destination)
        {
        ZF =1; //Set the Zero Flag of EFLAGS Register
        destination =source;
        }
else
        {
        ZF = 0; //Reset the Zero Flag of EFLAGS Register
        accumulator = destination;
        }
```

https://hemanthrajhemu.github.io

The process/thread checks the lock variable to see whether its state is '0' and sets its state to '1' if its state is '0', for acquiring the lock. To implement this at the 486 processor level, load the accumulator with '0' and a general purpose register with '1' and compare the memory location holding the lock variable with accumulator using *CMPXCHG* instruction. This instruction makes the accessing, testing and modification of the lock variable a single atomic instruction. How the *CMPXCHG* instruction support provided by the Intel® family of processors (486 and above) is made available to processes/threads is OS kernel implementation dependent. Let us see how this feature is implemented by Windows Operating systems. Windows CE/Windows XP kernels support the compare and exchange hardware feature provided by Intel® family of processors, through the API call *InterlockedCompareExchange (LPLONG Destination, LONG Exchange, LONG Comperand)*. The variable *Destination* is the long pointer to the destination variable. The *Destination* variable should be of type '*long*'. The variable *Exchange* represents the exchange value. The value of *Destination* variable is replaced with the value of *Exchange* variable. The variable *Comperand* specifies the value which needs to be compared with the value of *Destination* variable. The function returns the initial value of the variable '*Destination*'. The following code snippet illustrates the usage of this API call for thread/process synchronisation.

```
//Inside parent thread/ main thread corresponding to a process
long bFlag; //Global declaration of lock Variable.
bFlag=0; //Initialise the lock to indicate it is available.
//..............................................................................
//Inside the child threads/ threads of a process
//Check the lock for availability & acquire the lock if available.
while (InterlockedCompareExchange (&bFlag, 1, 0) == 1);

//Rest of the source code dealing with shared resource access
```

The *InterlockedCompareExchange* function is implemented as '*Compiler intrinsic function*'. The 'code for *Compiler intrinsic functions*' are inserted inline while compiling the code. This avoids the function call overhead and makes use of the built-in knowledge of the optimisation technique for intrinsic functions. The compiler can be instructed to use the intrinsic implementation for a function using the compiler directive *#pragma intrinsic (intrinsic-function-name)*. A sample implementation of the *InterlockedCompareExchange* interlocked intrinsic function for Windows XP OS is given below.

```
#include "stdafx.h"
#include <intrin.h>
#include <windows.h>
long bFlag; //Global declaration of lock Variable.
//Declare InterlockedCompareExchange as intrinsic function
#pragma intrinsic(_InterlockedCompareExchange)
void child_thread(void)
{
//Inside the child thread of a process
//Check the lock for availability & acquire the lock if available.
//The lock can be set by any other threads
while (_InterlockedCompareExchange (&bFlag, 1, 0) == 1);
//Rest of the source code dealing with shared resource access
//...............................................................
return;
}
```

https://hemanthrajhemu.github.io

```
//.....................................................................
int _tmain(int argc, _TCHAR* argv[])
{
//Inside parent thread/ main thread corresponding to a process
DWORD thread_id;
//Define handle to the child thread
HANDLE tThread;
//Initialize the lock to indicate it is available.
bFlag =0;
//Create child thread
tThread = CreateThread    (NULL,0,
                          (LPTHREAD_START_ROUTINE) child_thread,
                          NULL, 0, &thread_id);
if(NULL== tThread)
{
//Child thread creation failed.
printf ("Creation of Child thread failed. Error Code =
%d",GetLastError());
return -1;
}
//Wait for the completion of the child thread.
WaitForSingleObject(tThread,INFINITE);
return 0;
}
```

Note: Visual Studio 2005 or a later version of the compiler, which supports interlocked intrinsic functions, is required for compiling this application. The assembly code generated for the intrinsic interlocked function *while (_InterlockedCompareExchange (&bFlag, 1, 0) == 1);* when compiled using Visual Studio 2005 compiler, on Windows XP platform with Service Pack 2 running on an Intel® Centrino® Duo processor is given below. It clearly depicts the usage of the *cmpxchg* instruction

```
//Inside the child threads/ threads of a process
//Check the lock for availability & acquire the lock if available.
//The lock can be set by any other threads
while (_InterlockedCompareExchange (&bFlag, 1, 0) == 1);
004113DE          mov  ecx,1
004113E3          mov  edx,offset bFlag (417178h)
004113E8          xor          eax,eax
004113EA          lock         cmpxchg dword ptr [edx],ecx
004113EE          cmp          eax,1
004113F1          jne          child_thread+35h (4113F5h)
004113F3          jmp          child_thread+1Eh (4113DEh)
//Rest of the source code dealing with shared resource access
//............................................................
```

The Intel 486 and above family of processors provide hardware level support for atomic execution of increment and decrement operations also. The *XADD* low level instruction implements atomic execution of increment and decrement operations. Windows CE/XP kernel makes these features available to the users through a set of *Interlocked* function API calls. The API call *InterlockedIncrement (LPLONG*

https://hemanthrajhemu.github.io

lpAddend) increments the value of the variable pointed by *lpAddend* and the API *InterlockedDecrement (LPLONG lpAddend)* decrements the value of the variable pointed by *lpAddend.*

The lock based mutual exclusion implementation always checks the state of a lock and waits till the lock is available. This keeps the processes/threads always busy and forces the processes/threads to wait for the availability of the lock for proceeding further. Hence this synchronisation mechanism is popularly known as '*Busy waiting*'. The '*Busy waiting*' technique can also be visualised as a lock around which the process/thread spins, checking for its availability. Spin locks are useful in handling scenarios where the processes/threads are likely to be blocked for a shorter period of time on waiting the lock, as they avoid OS overheads on context saving and process re-scheduling. Another drawback of Spin lock based synchronisation is that if the lock is being held for a long time by a process and if it is preempted by the OS, the other threads waiting for this lock may have to spin a longer time for getting it. The '*Busy waiting*' mechanism keeps the process/threads always active, performing a task which is not useful and leads to the wastage of processor time and high power consumption.

The interlocked operations are the most efficient synchronisation primitives when compared to the classic lock based synchronisation mechanism. Interlocked function based synchronisation technique brings the following value adds.

- The interlocked operation is free from waiting. Unlike the mutex, semaphore and critical section synchronisation objects which may require waiting on the object, if they are not available at the time of request, the interlocked function simply performs the operation and returns immediately. This avoids the blocking of the thread which calls the interlocked function.
- The interlocked function call is directly converted to a processor specific instruction and there is no user mode to kernel mode transition as in the case of mutex, semaphore and critical section objects. This avoids the user mode to kernel mode transition delay and thereby increases the overall performance.

The types of interlocked operations supported by an OS are underlying processor hardware dependent and so they are limited in functionality. Normally the bit manipulation (Boolean) operations are not supported by interlocked functions. Also the interlocked operations are limited to integer or pointer variables only. This limits the possibility of extending the interlocked functions to variables of other types. Under windows operating systems, each process has its own virtual address space and so the interlocked functions can only be used for synchronising the access to a variable that is shared by multiple threads of a process (Multiple threads of a process share the same address space) (Intra Process Synchronisation). The interlocked functions can be extended for synchronising the access of the variables shared across multiple processes if the variable is kept in shared memory.

10.8.2.2 Mutual Exclusion through Sleep & Wakeup The '*Busy waiting*' mutual exclusion enforcement mechanism used by processes makes the CPU always busy by checking the lock to see whether they can proceed. This results in the wastage of CPU time and leads to high power consumption. This is not affordable in embedded systems powered on battery, since it affects the battery backup time of the device. An alternative to '*busy waiting*' is the '*Sleep & Wakeup*' mechanism. When a process is not allowed to access the critical section, which is currently being locked by another process, the process undergoes '*Sleep*' and enters the '*blocked*' state. The process which is blocked on waiting for access to the critical section is awakened by the process which currently owns the critical section. The process which owns the critical section sends a wakeup message to the process, which is sleeping as a result of waiting for the access to the critical section, when the process leaves the critical section. The '*Sleep & Wakeup*' policy for mutual exclusion can be implemented in different ways. Implementation of

https://hemanthrajhemu.github.io

this policy is OS kernel dependent. The following section describes the important techniques for '*Sleep & Wakeup*' policy implementation for mutual exclusion by Windows XP/CE OS kernels.

Semaphore Semaphore is a sleep and wakeup based mutual exclusion implementation for shared resource access. Semaphore is a system resource and the process which wants to access the shared resource can first acquire this system object to indicate the other processes which wants the shared resource that the shared resource is currently acquired by it. The resources which are shared among a process can be either for exclusive use by a process or for using by a number of processes at a time. The display device of an embedded system is a typical example for the shared resource which needs exclusive access by a process. The Hard disk (secondary storage) of a system is a typical example for sharing the resource among a limited number of multiple processes. Various processes can access the different sectors of the hard-disk concurrently. Based on the implementation of the sharing limitation of the shared resource, semaphores are classified into two; namely '*Binary Semaphore*' and '*Counting Semaphore*'. The binary semaphore provides exclusive access to shared resource by allocating the resource to a single process at a time and not allowing the other processes to access it when it is being owned by a process. The implementation of binary semaphore is OS kernel dependent. Under certain OS kernel it is referred as *mutex*. Unlike a binary semaphore, the '*Counting Semaphore*' limits the access of resources by a fixed number of processes/threads. '*Counting Semaphore*' maintains a count between zero and a

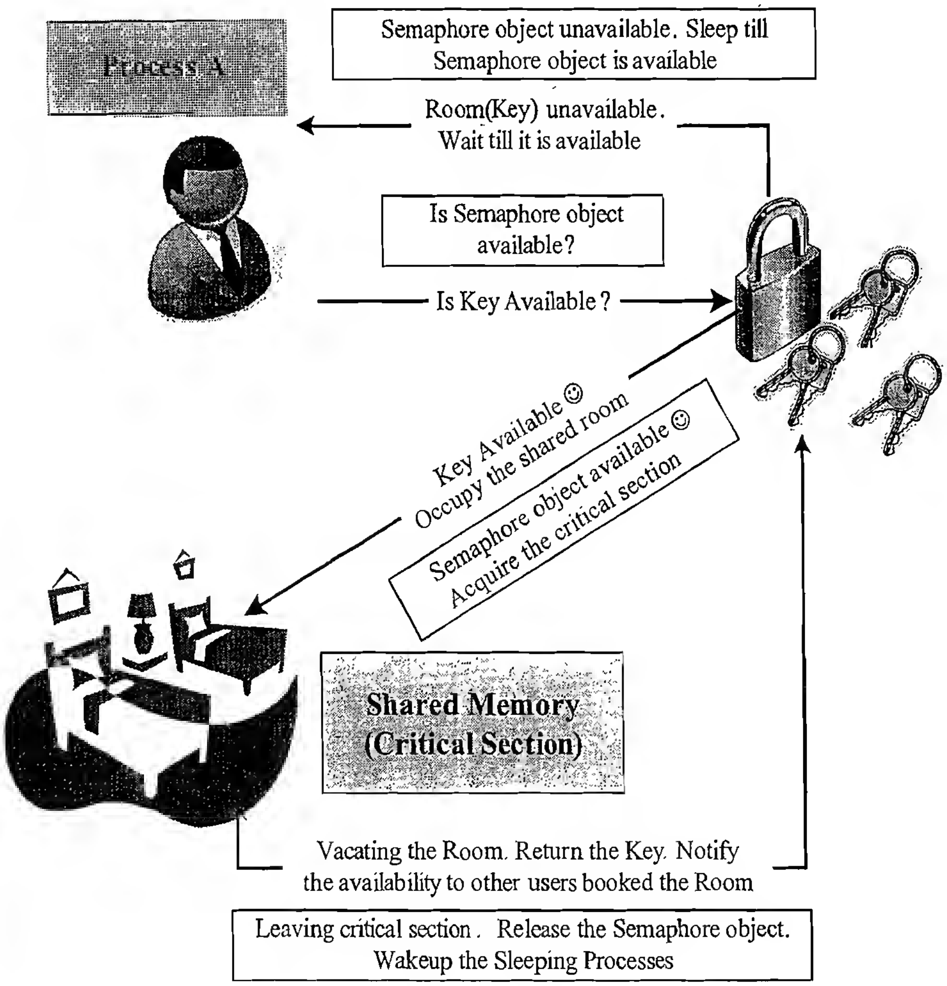

Process A
Semaphore object unavailable. Sleep till Semaphore object is available
Room(Key) unavailable. Wait till it is available
Is Semaphore object available?
Is Key Available ?
Key Available ☺ Occupy the shared room
Semaphore object available ☺ Acquire the critical section
Shared Memory (Critical Section)
Vacating the Room. Return the Key. Notify the availability to other users booked the Room
Leaving critical section. Release the Semaphore object. Wakeup the Sleeping Processes

Fig. 10.34 The Concept of Counting Semaphore

https://hemanthrajhemu.github.io

value. It limits the usage of the resource to the maximum value of the count supported by it. The state of the counting semaphore object is set to 'signalled' when the count of the object is greater than zero. The count associated with a '*Semaphore object*' is decremented by one when a process/thread acquires it and the count is incremented by one when a process/thread releases the '*Semaphore object*'. The state of the '*Semaphore object*' is set to non-signalled when the semaphore is acquired by the maximum number of processes/threads that the semaphore can support (i.e. when the count associated with the '*Semaphore object*' becomes zero). A real world example for the counting semaphore concept is the dormitory system for accommodation (Fig. 10.34). A dormitory contains a fixed number of beds (say 5) and at any point of time it can be shared by the maximum number of users supported by the dormitory. If a person wants to avail the dormitory facility, he/she can contact the dormitory caretaker for checking the availability. If beds are available in the dorm the caretaker will hand over the keys to the user. If beds are not available currently, the user can register his/her name to get notifications when a slot is available. Those who are availing the dormitory shares the dorm facilities like TV, telephone, toilet, etc. When a dorm user vacates, he/she gives the keys back to the caretaker. The caretaker informs the users, who booked in advance, about the dorm availability.

The creation and usage of '*counting semaphore object*' is OS kernel dependent. Let us have a look at how we can implement semaphore based synchronisation for the '*Racing*' problem we discussed in the beginning, under the Windows kernel. The following code snippet explains the same.

```
#include <stdio.h>
#include <windows.h>
#define MAX_SEMAPHORE_COUNT 1    //Make the semaphore object for //exclusive
use
#define thread_count 2               //No.of Child Threads
//********************************************************************
//counter is an integer variable and Buffer is a byte array shared //between
two threads Process_A and Process_B
char Buffer[10] = {1,2,3,4,5,6,7,8,9,10};
short int counter = 0;
//Define the handle to Semaphore object
HANDLE hSemaphore;
//********************************************************************
// Child Thread 1
void Process_A (void) {
int i;
  for (i =0; i<5; i++)
{
    if (Buffer[i] > 0)
       {
      //Wait for the signaling of Semaphore object
      WaitForSingleObject(hSemaphore,INFINITE);
      //Semaphore is acquired
      counter++;
      printf("Process A : Counter = %d\n",counter);
         //Release the Semaphore Object

             if (!ReleaseSemaphore(
                       hSemaphore, // handle to semaphore
```

https://hemanthrajhemu.github.io

```
                        1,         // increase count by one
                        NULL))    // not interested in previous count
              {
              //Semaphore Release failed. Print Error code &
              return.
              printf(“Release Semaphore Failed with Error Code:
              %d\n”, GetLastError());
              return;
              }
       }
}
  return;
}
//*********************************************************************
// Child Thread 2
void Process_B(void) {
int j;
  for (j =5; j<10; j++)
{
    if (Buffer[j] > 0)
       {
      //Wait for the signalling of Semaphore object
      WaitForSingleObject(hSemaphore,INFINITE);
      //Semaphore is acquired
      counter++;
      printf(“Process B : Counter = %d\n”,counter);
         //Release Semaphore
         if (!ReleaseSemaphore(
                        hSemaphore, // handle to semaphore
                        1,         // increase count by one
                        NULL) )   // not interested in previous count
                     {
              //Semaphore Release failed. Print Error code &
              //return.
              printf(“Release Semaphore Failed Error Code: %d\n”,
              GetLastError());
              return;
              }
       }
}
  return;
}
//*********************************************************************
// Main Thread
void main() {

       //Define HANDLE for child threads
       HANDLE child_threads[thread_count];
```

https://hemanthrajhemu.github.io

```
        DWORD thread_id;
        int i;

        //Create Semaphore object
        hSemaphore = CreateSemaphore(
         NULL,              // default security attributes
         MAX_SEMAPHORE_COUNT,  // initial count. Create as signaled
         MAX_SEMAPHORE_COUNT,  // maximum count
         "Semaphore");      // Semaphore object with name "Semaphore"

      if (NULL == hSemaphore)
         {
         printf ("Semaphore Object Creation Failed: Error Code: %d",
        GetLastError ());
         //Semaphore Object Creation failed. Return
        return;
         }
   //Create Child thread 1
   child_threads[0]=          CreateThread(NULL,0,
                               (LPTHREAD_START_ROUTINE)Process_A,
                               (LPVOID)0,0,&thread_id);
   //Create Child thread 2
   child_threads[1]=          CreateThread(NULL,0,
                               (LPTHREAD_START_ROUTINE)Process_B,
                               (LPVOID)0,0,&thread_id);
   //Check the success of creation of child threads
   for (i=0;i<thread_count; i++)
   {
         if(NULL==child_threads[i])
         {
         //Child thread creation failed.
         printf ("Child thread Creation failed with Error Code: %d",
         GetLastError ());
         return;
         }
   }
   // Wait for the termination of child threads
   WaitForMultipleObjects(thread_count, child_threads, TRUE,
   INFINITE);
   //Close handles of child threads
       for( i=0; i < thread_count; i++ )
           CloseHandle(child_threads[i]);
   //Close Semaphore object handle
         CloseHandle(hSemaphore);
         return;
   }
```

Please refer to the Online Learning Centre for details on the various Win32 APIs used in the program for counting semaphore creation, acquiring, signalling, and releasing. The VxWorks and MicroC/OS-II

https://hemanthrajhemu.github.io

Real-Time kernels also implements the Counting semaphore based task synchronisation/shared resource access. We will discuss them in detail in a later chapter.

Counting Semaphores are similar to *Binary Semaphores* in operation. The only difference between *Counting Semaphore* and *Binary Semaphore* is that *Binary Semaphore* can only be used for exclusive access, whereas *Counting Semaphores* can be used for both exclusive access (by restricting the maximum count value associated with the semaphore object to one (1) at the time of creation of the semaphore object) and limited access (by restricting the maximum count value associated with the semaphore object to the limited number at the time of creation of the semaphore object).

Binary Semaphore (Mutex) Binary Semaphore (Mutex) is a synchronisation object provided by OS for process/thread synchronisation. Any process/thread can create a '*mutex object*' and other processes/threads of the system can use this '*mutex object*' for synchronising the access to critical sections. Only one process/thread can own the '*mutex object*' at a time. The state of a mutex object is set to signalled when it is not owned by any process/thread, and set to non-signalled when it is owned by any process/thread. A real world example for the mutex concept is the hotel accommodation system (lodging system) Fig. 10.35. The rooms in a hotel are shared for the public. Any user who pays and follows the norms of the hotel can avail the rooms for accommodation. A person wants to avail the hotel room facility can contact the hotel reception for checking the room availability (see Fig. 10.35). If room is available the receptionist will handover the room key to the user. If room is not available currently, the user can book

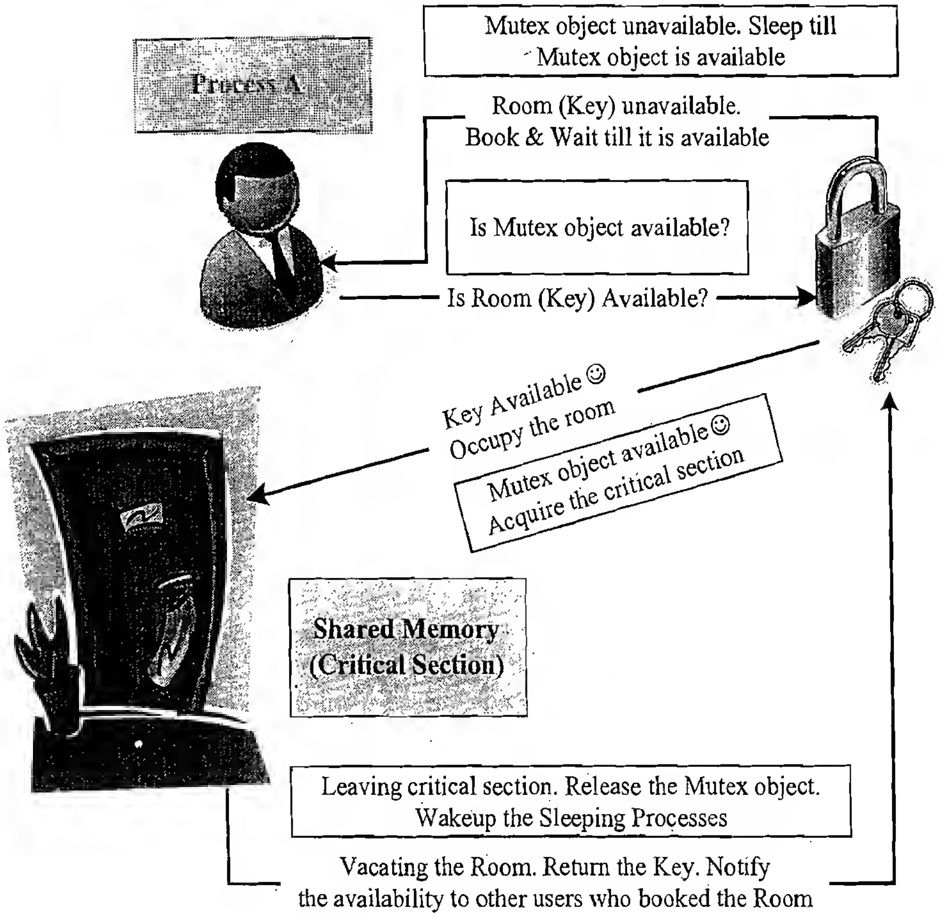

Mutex object unavailable. Sleep till Mutex object is available
Process A
Room (Key) unavailable. Book & Wait till it is available
Is Mutex object available?
Is Room (Key) Available?
Key Available ☺ Occupy the room
Mutex object available☺ Acquire the critical section
Shared Memory (Critical Section)
Leaving critical section. Release the Mutex object. Wakeup the Sleeping Processes
Vacating the Room. Return the Key. Notify the availability to other users who booked the Room

Fig. 10.35 **The Concept of Binary Semaphore (Mutex)**

https://hemanthrajhemu.github.io

the room to get notifications when a room is available. When a person gets a room he/she is granted the exclusive access to the room facilities like TV, telephone, toilet, etc. When a user vacates the room, he/she gives the keys back to the receptionist. The receptionist informs the users, who booked in advance, about the room's availability.

Let's see how we can implement mutual exclusion with mutex object in the '*Racing*' problem example given under the section '*Racing*', under Windows kernel.

```
#include <stdio.h>
#include <windows.h>
#define thread_count 2                  //No.of Child Threads
//*******************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two
//threads Process_A and Process_B
char Buffer[10] = {1,2,3,4,5,6,7,8,9,10};
short int counter = 0;
//Define the handle to Mutex Object
HANDLE hMutex;
//*******************************************************************
// Child Thread 1
void Process_A (void) {
int i;
  for (i =0; i<5; i++)
{
    if (Buffer[i] > 0)
       {
       //Wait for signalling of the Mutex object
       WaitForSingleObject(hMutex,INFINITE);
       //Mutex is acquired
       counter++;
       printf("Process A : Counter = %d\n",counter);
       //Release the Mutex Object

       if (!ReleaseMutex(hMutex))  // handle to Mutex Object
       {
       //Mutex object Releasing failed. Print Error code & return.
       printf("Release Mutex Failed with Error Code: %d\n",
       GetLastError());
       return;
       }

       }
}
  return;
}
//*******************************************************************
// Child Thread 2
void Process_B(void) {
int j;
```

https://hemanthrajhemu.github.io

```
  for (j =5; j<10; j++)
{
    if (Buffer[j] > 0)
       {
       //Wait for signalling of the Mutex object
       WaitForSingleObject(hMutex,INFINITE);
       //Mutex object is acquired
       counter++;
       printf("Process B : Counter = %d\n",counter);
       //Release Mutex object
       if (!ReleaseMutex(hMutex))  // handle to Mutex Object
          {
          //Mutex object Release failed. Print Error code & return.
          printf("Release Mutex Failed with Error Code: %d\n",
          GetLastError());
          return;
          }
       }
}
  return;
}
//*****************************************************************
// Main Thread
void main() {

       //Define HANDLE for child threads
       HANDLE child_threads[thread_count];
       DWORD thread_id;
       int i;

       //Create Mutex object
       hMutex = CreateMutex(
        NULL,            // default security attributes
        FALSE,  // Not initial ownership
        "Mutex");   // Mutex object with name "Mutex"
    if (NULL == hMutex)
       {
       printf ("Mutex Object Creation Failed: Error Code: %d",
       GetLastError ());
       //Mutex Object Creation failed. Return
       return;
        }
//Create Child thread 1
child_threads[0]=        CreateThread(NULL,0,
                         (LPTHREAD_START_ROUTINE)Process_A,
                         (LPVOID)0,0,&thread_id);
```

https://hemanthrajhemu.github.io

```
//Create Child thread 2
child_threads[1]=        CreateThread(NULL,0,
                         (LPTHREAD_START_ROUTINE)Process_B,
                         (LPVOID)0,0,&thread_id);
//Check the success of creation of child threads
for (i=0;i<thread_count; i++)
{
      if(NULL==child_threads[i])
      {
      //Child thread creation failed.
      printf (“Child thread Creation failed with Error Code: %d”,
      GetLastError ());
      return;
      }
}
// Wait for the termination of child threads

WaitForMultipleObjects(thread_count, child_threads, TRUE,
INFINITE);
    //Close child thread handles
      for( i=0; i < thread_count; i++ )
        CloseHandle(child_threads[i]);
      //Close Mutex object handle
      CloseHandle(hMutex);
      return;
}
```

Please refer to the Online Learning Centre for details on the various Win32 APIs used in the program for mutex creation, acquiring, signalling, and releasing.

The mutual exclusion semaphore is a special implementation of the binary semaphore by certain real-time operating systems like VxWorks and MicroC/OS-II to prevent priority inversion problems in shared resource access. The mutual exclusion semaphore has an option to set the priority of a task owning it to the highest priority of the task which is being pended while attempting to acquire the semaphore which is already in use by a low priority task. This ensures that the low priority task which is currently holding the semaphore, when a high priority task is waiting for it, is not pre-empted by a medium priority task. This is the mechanism supported by the mutual exclusion semaphore to prevent priority inversion.

VxWorks kernel also supports binary semaphores for synchronising shared resource access. We will discuss about it in detail in a later chapter.

Critical Section Objects In Windows CE, the ‘*Critical Section object*’ is same as the ‘*mutex object*’ except that ‘*Critical Section object*’ can only be used by the threads of a single process (Intra process). The piece of code which needs to be made as ‘*Critical Section*’ is placed at the ‘*Critical Section*’ area by the process. The memory area which is to be used as the ‘*Critical Section*’ is allocated by the process. The process creates a ‘*Critical Section*’ area by creating a variable of type *CRITICAL_SECTION*. The *Critical Section*’ must be initialised before the threads of a process can use it for getting exclusive access. The *InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection)* API initialises the critical section pointed by the pointer *lpCriticalSection* to the critical section. Once the critical section

https://hemanthrajhemu.github.io

is initialized, all threads in the process can use it. Threads can usc the API call *EnterCriticalSection (LPCRITICAL_SECTION lpCriticalSection)* for getting the exclusive ownership of the critical section pointed by the pointer *lpCriticalSection*. Calling the *EnterCriticalSection()* API blocks the execution of the caller thread if the critical section is already in use by other threads and the thread waits for the critical section object. Threads which are blocked by the *EnterCriticalSection()* call, waiting on a critical section are added to a wait queue and are woken when the critical section is available to the requested thread. The API call *TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)* attempts to enter the critical section pointed by the pointer *lpCriticalSection* without blocking the caller thread. If the critical section is not in use by any other thread, the calling thread gets the ownership of the critical section. If the critical section is already in use by another thread, the *TryEnterCriticalSection()* call indicates it to the caller thread by a specific return value and the thread resumes its execution. A thread can release the exclusive ownership of a critical section by calling the API *LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection)*. The threads of a process can use the API *DeleteCriticalSection (LPCRITICAL_SECTION lpCriticalSection)* to release all resources used by a critical section object which was created by the process with the *CRITICAL_SECTION* variable.

Now let's have a look at the '*Racing*' problem we discussed under the section '*Racing*'. The racing condition can be eliminated by using a critical section object for synchronisation. The following code snippet illustrates the same.

```
#include <stdio.h>
#include <windows.h>
//*****************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two threads
char Buffer[10] = {1,2,3,4,5,6,7,8,9,10};
short int counter = 0;
//Define the critical section
CRITICAL_SECTION CS;
//*****************************************************************
// Child Thread 1
void Process_A (void) {
int i;
   for (i =0; i<5; i++)
 {
     if (Buffer[i] > 0)
        {
        //Use critical section object for synchronisation
        EnterCriticalSection(&CS);
        counter++;
        LeaveCriticalSection(&CS);
        }
       printf("Process A : Counter = %d\n",counter);
 }
}
//*****************************************************************
// Child Thread 2
void Process_B(void) {
```

https://hemanthrajhemu.github.io

```
int j;
  for (j =5; j<10; j++)
{
    if (Buffer[j] > 0)
       {
       //Use critical section object for synchronisation
       EnterCriticalSection(&CS);
       counter++;
       LeaveCriticalSection(&CS);
       }
      printf("Process B : Counter = %d\n",counter);
}
}
//********************************************************************
// Main Thread

int main() {

       DWORD id;

       //Initialize critical section object
       InitializeCriticalSection(&CS);
       CreateThread(NULL,0,
                           (LPTHREAD_START_ROUTINE) Process_A,
                           (LPVOID)0,0,&id);
       CreateThread(NULL,0,
                           (LPTHREAD_START_ROUTINE) Process_B,
                           (LPVOID)0,0,&id);
       Sleep(100000);
       return 0;
}
```

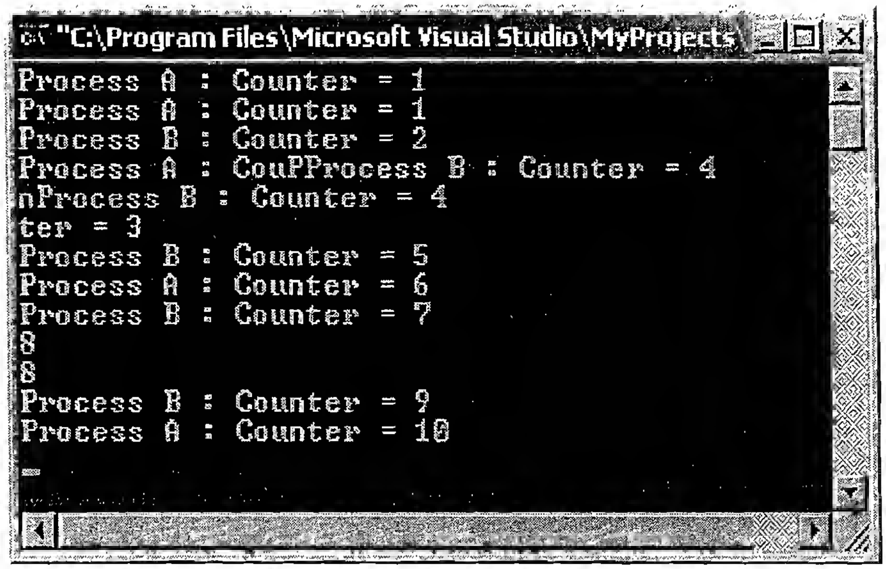
Here the shared resource is the shared variable '*counter*'. The concurrent access to this variable by the threads '*Process_A*' and '*Process_B*' may create race condition and may produce incorrect results. The critical section object '*CS*' holds the piece of code corresponding to the access of the shared variable '*counter*' by each threads. This ensures that the memory area containing the low level instructions corresponding to the high level instruction '*counter++*' is accessed exclusively by threads '*Process_A*' and '*Process_B*' and avoids a race condition. The output of the above piece of eode when executed on Windows XP machine is given in Fig. 10.36.

The final value of '*counter*' is obtained as 10, which is the expected result for this piece of code. If you observe this output window you can see that the text is not outputted to the o/p window in the expected manner. The *printf ()* library routine used in this sample code is re-entrant and it can be preempted while in execution. That is why the outputting of text happened in a non expected way.

Note

It should be noted that the scheduling of the threads '*Process_A*' and '*Process_B*' is OS kernel scheduling policy dependent and you may not get the same output all the time when you run this piece of code under Windows XP.

https://hemanthrajhemu.github.io



```
Process A : Counter = 1
Process A : Counter = 1
Process B : Counter = 2
Process A : CouPProcess B : Counter = 4
nProcess B : Counter = 4
ter = 3
Process B : Counter = 5
Process A : Counter = 6
Process B : Counter = 7
8
8
Process B : Counter = 9
Process A : Counter = 10
_
```

Fig. 10.36 **Output of the Win32 application resolving racing condition through critical section object**

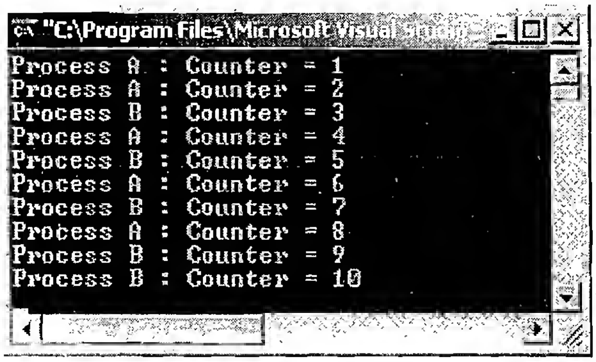
The critical section object makes the piece of code residing inside it non-reentrant. Now let's try the above piece of code by putting the *printf ()* library routine in the critical section object.

```
#include <stdio.h>
#include <windows.h>
//*******************************************************************
//counter is an integer variable and Buffer is a byte array shared
//between two threads Process A and Process B
char Buffer[10] = {1,2,3,4,5,6,7,8,9,10};
short int counter = 0;
//Define the critical section
CRITICAL_SECTION CS;
//*******************************************************************
// Child Thread 1
void Process_A (void) {

int i;
  for (i =0; i<5; i++)
{
    if (Buffer[i] > 0)
       {
       //Use critical section object for synchronisation
       EnterCriticalSection(&CS);
       counter++;
       printf("Process A : Counter = %d\n",counter);
       LeaveCriticalSection(&CS);
       }
}
}
```

https://hemanthrajhemu.github.io

```
//********************************************************************
// Child Thread 2
void Process_B(void) {

int j;
 for (j = 5; j<10; j++)
{
  if (Buffer[j] > 0)
   {
   //Use critical section object for synchronisation
   EnterCriticalSection(&CS);
   counter++;
   printf(“Process B : Counter = %d\n”,counter);
   LeaveCriticalSection(&CS);
   }
}
}
//********************************************************************
// Main Thread

int main() {

   DWORD id;

   //Initialize critical section object
   InitializeCriticalSection(&CS);
   CreateThread(NULL,0,
              (LPTHREAD_START_ROUTINE)Process_A,
              (LPVOID)0,0,&id);
   CreateThread(NULL,0,
              (LPTHREAD_START_ROUTINE)Process_B,
              (LPVOID)0,0,&id);
   Sleep(100000);
   return 0;
}
```

The output of the above piece of code when executed on a Windows XP machine is given below.

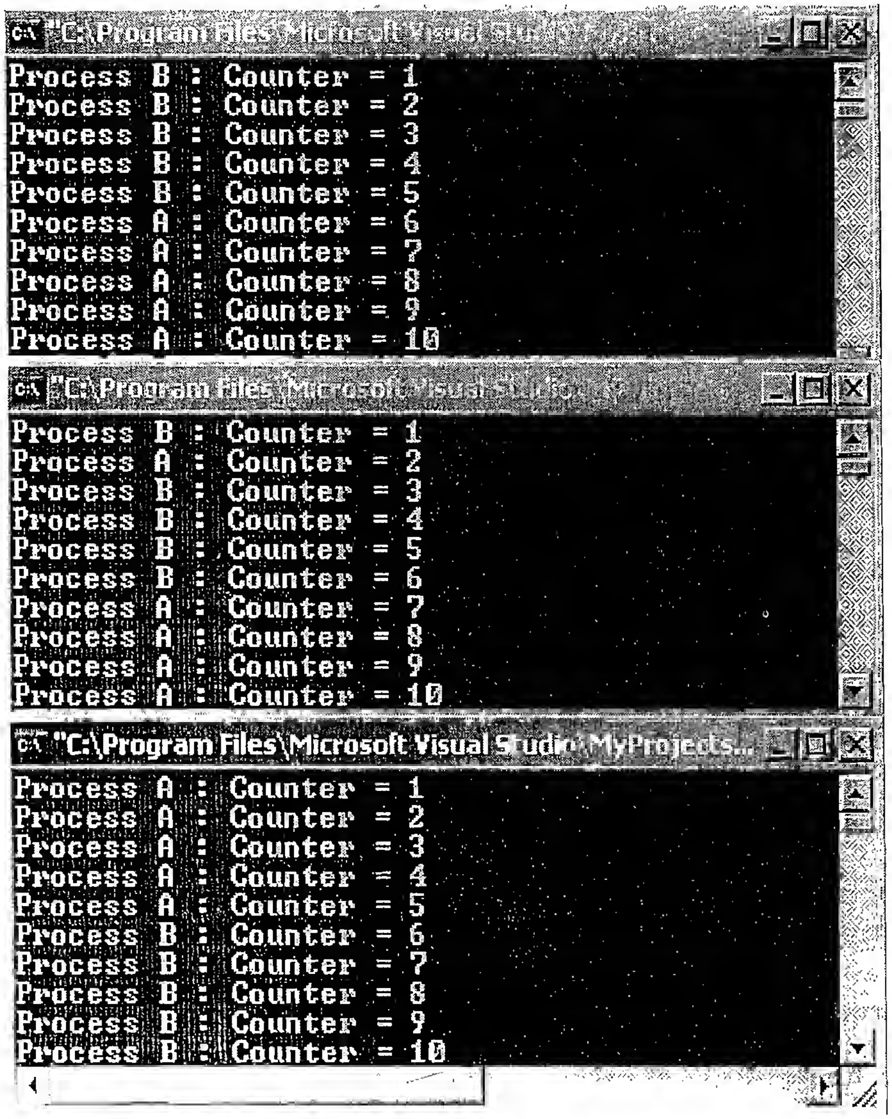


```
Process A : Counter = 1
Process A : Counter = 2
Process B : Counter = 3
Process A : Counter = 4
Process B : Counter = 5
Process A : Counter = 6
Process B : Counter = 7
Process A : Counter = 8
Process B : Counter = 9
Process B : Counter = 10
```

Fig. 10.37 **Output of the Win32 application resolving racing condition through critical section object**

https://hemanthrajhemu.github.io

> ***Note***
>
> It should be noted that the scheduling of the threads '*Process_A*' and '*Process_B*' is OS kernel scheduling policy dependent and you may not get the same output all the time when you run this piece of code in Windows XP. The output of the above program when executed at three different instances of time is given shown in Fig. 10.38.



```
Process B : Counter = 1
Process B : Counter = 2
Process B : Counter = 3
Process B : Counter = 4
Process B : Counter = 5
Process A : Counter = 6
Process A : Counter = 7
Process A : Counter = 8
Process A : Counter = 9
Process A : Counter = 10
```

```
Process B : Counter = 1
Process A : Counter = 2
Process B : Counter = 3
Process B : Counter = 4
Process B : Counter = 5
Process B : Counter = 6
Process A : Counter = 7
Process A : Counter = 8
Process A : Counter = 9
Process A : Counter = 10
```

```
Process A : Counter = 1
Process A : Counter = 2
Process A : Counter = 3
Process A : Counter = 4
Process A : Counter = 5
Process B : Counter = 6
Process B : Counter = 7
Process B : Counter = 8
Process B : Counter = 9
Process B : Counter = 10
```

Fig. 10.38 **Illustration of scheduler behaviour under Windows XP kernel**

Events Event object is a synchronisation technique which uses the notification mechanism for synchronisation. In concurrent execution we may come across situations which demand the processes to wait for a particular sequence for its operations. A typical example of this is the producer consumer threads, where the consumer thread should wait for the consumer thread to produce the data and producer thread should wait for the consumer thread to consume the data before producing fresh data. If this sequence is not followed it will end up in producer-consumer problem. Notification mechanism is used for handling this scenario. Event objects are used for implementing notification mechanisms.

https://hemanthrajhemu.github.io

A thread/process can wait for an event and another thread/process can set this event for processing by the waiting thread/process. The creation and handling of event objects for notification is OS kernel dependent. Please refer to the Online Learning Centre for information on the usage of 'Events' under Windows Kernel for process/thread synchronisation.

The MicroC/OS-II kernel also uses 'events' for task synchronisation. We will discuss it in a later chapter.

10.9 DEVICE DRIVERS

Device driver is a piece of software that acts as a bridge between the operating system and the hardware. In an operating system based product architecture, the user applications talk to the Operating System kernel for all necessary information exchange including communication with the hardware peripherals. The architecture of the OS kernel will not allow direct device access from the user application. All the device related access should flow through the OS kernel and the OS kernel routes it to the concerned hardware peripheral. OS provides interfaces in the form of Application Programming Interfaces (APIs) for accessing the hardware. The device driver abstracts the hardware from user applications. The topology of user applications and hardware interaction in an RTOS based system is depicted in Fig. 10.39.

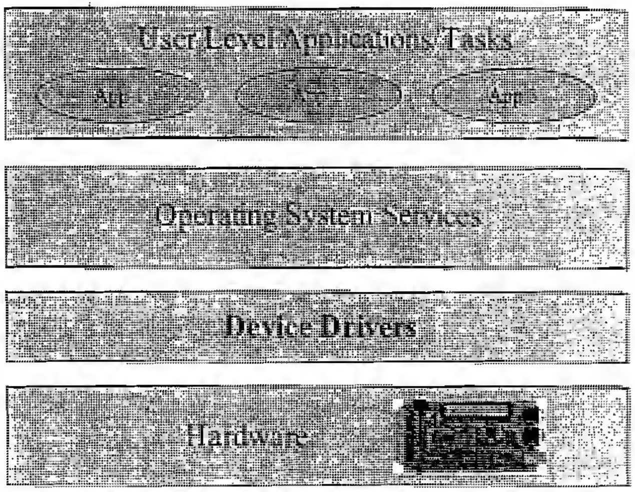


Fig. 10.39 **Role of Device driver in Embedded OS based products**

Device drivers are responsible for initiating and managing the communication with the hardware peripherals. They are responsible for establishing the connectivity, initialising the hardware (setting up various registers of the hardware device) and transferring data. An embedded product may contain different types of hardware components like Wi-Fi module, File systems, Storage device interface, etc. The initialisation of these devices and the protocols required for communicating with these devices may be different. All these requirements are implemented in drivers and a single driver will not be able to satisfy all these. Hence each hardware (more specifically each class of hardware) requires a unique driver component.

Certain drivers come as part of the OS kernel and certain drivers need to be installed on the fly. For example, the program storage memory for an embedded product, say NAND Flash memory requires a NAND Flash driver to read and write data from/to it. This driver should come as part of the OS kernel image. Certainly the OS will not contain the drivers for all devices and peripherals under the Sun. It contains only the necessary drivers to communicate with the onboard devices (Hardware devices which are part of the platform) and for certain set of devices supporting standard protocols and device class (Say USB Mass storage device or HID devices like Mouse/keyboard). If an external device, whose driver software is not available with OS kernel image, is connected to the embedded device (Say a medical device with custom USB class implementation is connected to the USB port of the embedded product), the OS prompts the user to instal its driver manually. Device drivers which are part of the OS image are known as 'Built-in drivers' or 'On-board drivers'. These drivers are loaded by the OS at the

https://hemanthrajhemu.github.io

time of booting the device and are always kept in the RAM. Drivers which need to be installed for accessing a device are known as 'Installable drivers'. These drivers are loaded by the OS on a need basis. Whenever the device is connected, the OS loads the corresponding driver to memory. When the device is removed, the driver is unloaded from memory. The Operating system maintains a record of the drivers corresponding to each hardware.

The implementation of driver is OS dependent. There is no universal implementation for a driver. How the driver communicates with the kernel is dependent on the OS structure and implementation. Different Operating Systems follow different implementations.

It is very essential to know the hardware interfacing details like the memory address assigned to the device, the Interrupt used, etc. of on-board peripherals for writing a driver for that peripheral. It varies on the hardware design of the product. Some Real-Time operating systems like 'Windows CE' support a layered architecture for the driver which separates out the low level implementation from the OS specific interface. The low level implementation part is generally known as Platform Dependent Device (PDD) layer. The OS specific interface part is known as Model Device Driver (MDD) or Logical Device Driver (LDD). For a standard driver, for a specific operating system, the MDD/LDD always remains the same and only the PDD part needs to be modified according to the target hardware for a particular class of devices.

Most of the time, the hardware developer provides the implementation for all on board devices for a specific OS along with the platform. The drivers are normally shipped in the form of *Board Support Package*. The *Board Support Package* contains low level driver implementations for the onboard peripherals and OEM Adaptation Layer (OAL) for accessing the various chip level functionalities and a bootloader for loading the operating system. The OAL facilitates communication between the Operating System (OS) and the target device and includes code to handle interrupts, timers, power management, bus abstraction, generic I/O control codes (IOCTLs), etc. The driver files are usually in the form of a dll file. Drivers can run on either user space or kernel space. Drivers which run in user space are known as *user mode drivers* and the drivers which run in kernel space are known as *kernel mode drivers*. User mode drivers are safer than kernel mode drivers. If an error or exception occurs in a user mode driver, it won't affect the services of the kernel. On the other hand, if an exception occurs in the kernel mode driver, it may lead to the kernel crash. The way how a device driver is written and how the interrupts are handled in it are operating system and target hardware specific. However regardless of the OS types, a device driver implements the following:

1. Device (Hardware) Initialisation and Interrupt configuration
2. Interrupt handling and processing
3. Client interfacing (Interfacing with user applications)

The Device (Hardware) initialisation part of the driver deals with configuring the different registers of the device (target hardware). For example configuring the I/O port line of the processor as Input or output line and setting its associated registers for building a General Purpose IO (GPIO) driver. The interrupt configuration part deals with configuring the interrupts that needs to be associated with the hardware. In the case of the GPIO driver, if the intention is to generate an interrupt when the Input line is asserted, we need to configure the interrupt associated with the I/O port by modifying its associated registers. The basic Interrupt configuration involves the following.

1. Set the interrupt type (Edge Triggered (Rising/Falling) or Level Triggered (Low or High)), enable the interrupts and set the interrupt priorities.
2. Bind the Interrupt with an Interrupt Request (IRQ). The processor identifies an interrupt through IRQ. These IRQs are generated by the Interrupt Controller. In order to identify an interrupt the interrupt needs to be bonded to an IRQ.

https://hemanthrajhemu.github.io

3. Register an Interrupt Service Routine (ISR) with an Interrupt Request (IRQ). ISR is the handler for an Interrupt. In order to service an interrupt, an ISR should be associated with an IRQ. Registering an ISR with an IRQ takes care of it.

With these the interrupt configuration is complete. If an interrupt occurs, depending on its priority, it is serviced and the corresponding ISR is invoked. The processing part of an interrupt is handled in an ISR. The whole interrupt processing can be done by the ISR itself or by invoking an Interrupt Service Thread (IST). The IST performs interrupt processing on behalf of the ISR. To make the ISR compact and short, it is always advised to use an IST for interrupt processing. The intention of an interrupt is to send or receive command or data to and from the hardware device and make the received data available to user programs for application specific processing. Since interrupt processing happens at kernel level, user applications may not have direct access to the drivers to pass and receive data. Hence it is the responsibility of the Interrupt processing routine or thread to inform the user applications that an interrupt is occurred and data is available for further processing. The client interfacing part of the device driver takes care of this. The client interfacing implementation makes use of the Inter Process communication mechanisms supported by the embedded OS for communicating and synchronising with user applications and drivers. For example, to inform a user application that an interrupt is occurred and the data received from the device is placed in a shared buffer, the client interfacing code can signal (or set) an event. The user application creates the event, registers it and waits for the driver to signal it. The driver can share the received data through shared memory techniques. IOCTLs, shared buffers, etc. can be used for data sharing. The story line is incomplete without performing an interrupt done (Interrupt processing completed) functionality in the driver. Whenever an interrupt is asserted, while vectoring to its corresponding ISR, all interrupts of equal and low priorities are disabled. They are re-enable only on executing the interrupt done function (Same as the Return from Interrupt RETI instruction execution for *8051*) by the driver. The interrupt done function can be invoked at the end of corresponding ISR or IST.

We will discuss more about device driver development in a dedicated book coming under this book—series.

10.10 HOW TO CHOOSE AN RTOS

The decision of choosing an RTOS for an embedded design is very crucial. 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 non-functional. The following section gives a brief introduction to the important functional and non-functional requirements that needs to be analysed in the selection of an RTOS for an embedded design.

10.10.1 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 files 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-time 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

https://hemanthrajhemu.github.io

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 modularisation 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.

10.10.2 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.

https://hemanthrajhemu.github.io

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.

Summary

- ✓ The *Operating System* is responsible for making the system convenient to use, organise and manage system resources efficiently and properly.
- ✓ Process/Task management, Primary memory management, File system management, I/O system (Device) management, Secondary Storage Management, protection implementation, Time management, Interrupt handling, etc. are the important services handled by the OS kernel.
- ✓ The core of the operating system is known as *kernel*. Depending on the implementation of the different kernel services, the kernel is classified as *Monolithic* and *Micro*. *User Space* is the memory area in which user applications are confined to run, whereas *kernel space* is the memory area reserved for kernel applications.
- ✓ Operating systems with a generalised kernel are known as *General Purpose Operating Systems* (*GPOS*), whereas operating systems with a specialised kernel with deterministic timing behaviour are known as *Real-Time Operating Systems* (*RTOS*).
- ✓ In the operating system context a task/process is a program, or part of it, in execution. The process 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.
- ✓ The different states through which a process traverses through during its journey from the newly created state to finished state is known as *Process Life Cycle*
- ✓ 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.
- ✓ A thread is the primitive that can execute code. It is a single sequential flow of control within a process. A process may contain multiple threads. The act of concurrent execution of multiple threads under an operating system is known as *multithreading*.
- ✓ Thread standards are the different standards available for thread creation and management. POSIX, Win32, Java, etc. are the commonly used thread creation and management libraries.
- ✓ The ability of a system to execute multiple processes simultaneously is known as *multiprocessing*, whereas the ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process is known as *multitasking*. Multitasking involves *Context Switching*, *Context Saving* and *Context Retrieval*.
- ✓ *Co-operative* multitasking, *Preemptive* multitasking and *Non-preemptive* multitasking are the three important types of multitasking which exist in the Operating system context.
- ✓ CPU utilisation, Throughput, Turn Around Time (TAT), Waiting Time and Response Time are the important criterions that need to be considered for the selection of a scheduling algorithm for task scheduling.
- ✓ Job queue, Ready queue and Device queue are the important queues maintained by an operating system in association with CPU scheduling.
- ✓ First Come First Served (FCFS)/First in First Out (FIFO), Last Come First Served (LCFS)/Last in First Out (LIFO), Shortest Job First (SJF), priority based scheduling, etc. are examples for Non-preemptive scheduling, whereas Preemptive SJF Scheduling/Shortest Remaining Time (SRT), Round Robin (RR) scheduling and priority based scheduling are examples for preemptive scheduling.
- ✓ Processes in a multitasking system falls into either *Co-operating* or *Competing*. The co-operating processes share data for communicating among the processes through *Inter Process Communication* (IPC), whereas competing

https://hemanthrajhemu.github.io

processes do not share anything among themselves but they share the system resources like display devices, keyboard, etc.

✓ *Shared memory*, *message passing* and *Remote Procedure Calls* (RPC) are the important IPC mechanisms through which the co-operating processes communicate in an operating system environment. The implementation of the IPC mechanism is OS kernel dependent.

✓ Racing, deadlock, livelock, starvation, producer-consumer problem, Readers-Writers problem and priority inversion are some of the problems involved in shared resource access in task communication through sharing.

✓ The 'Dining Philosophers' 'Problem' is a real-life reprcsentation of the deadlock, starvation, livelock and 'Racing' issues in shared resource access in operating system context.

✓ *Priority inversion* is the condition in which a medium priority task gets the CPU for execution, when a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task.

✓ *Priority inheritance* and *Priority ceiling* are the two mechanisms for avoiding *Priority Inversion* in a multitasking environment.

✓ The act of preventing the access of a shared resource by a task/process when it is currently being held by another task/process is known as *mutual exclusion*. Mutual exclusion can be implemented through either *busy waiting* (*spin lock*) or *sleep and wakeup* technique.

✓ *Test and Set*, *Flags*, etc. are examples of *Busy waiting* based *mutual exclusion* implementation, whereas *Semaphores*, *mutex*, *Critical Section Objects* and *events* are examples for *Sleep and Wakeup* based mutual exclusion.

✓ *Binary semaphore* implements exclusive shared resource access, whereas *counting semaphore* limits the concurrent access to a shared resource, and *mutual exclusion semaphore* prevents priority inversion in shared resource access.

✓ *Device driver* is a piece of software that acts as a bridge between the operating system and the hardware. Device drivers are responsible for initiating and managing the communication with the hardware peripherals.

✓ Various functional and non-functional requirements need to be evaluated before the selection of an RTOS for an embedded design.

Keywords

Operating System : A piece of software designed to manage and allocate system resources and execute other pieces of the software

Kernel : The core of the operating system which is responsible for managing the system resources and the communication among the hardware and other system services

Kernel space : The primary memory area where the kernel applications are confined to run

User space : The primary memory area where the user applications are confined to run

Monolithic kernel : A kernel with all kernel services run in the kernel space under a single kernel thread

Microkernel : A kernel which incorporates only the essential services within the kernel space and the rest is installed as loadable modules called *servers*

Real-Time Operating System (RTOS) : Operating system with a specialised kernel with a deterministic timing behaviour

Scheduler : OS kernel service which deals with the scheduling of processes/tasks

Hard Real-Time : Real-time operating systems that strictly adhere to the timing constraints for a task

Soft Real-Time : Real-time operating systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline

Task/Job/Process : In the operating system context a task/process is a program, or part of it, in execution

https://hemanthrajhemu.github.io

Process Life Cycle : The different *states* through which a process traverses through during its journey from the newly created state to completed state

Thread : The primitive that can execute code. It is a single sequential flow of control within a process

Multiprocessing systems : Systems which contain multiple CPUs and are capable of executing multiple processes simultaneously

Multitasking : The ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process

Context switching : The act of switching CPU among the processes and changing the current execution context

Co-operative multitasking : Multitasking model in which a task/process gets a chance when the currently executing task relinquishes the CPU voluntarily

Preemptive multitasking : Multitasking model in which a currently running task/process is preempted to execute another task/process

Non-preemptive multitasking : Multitasking model in which a task gets a chance to execute when the currently executing task relinquishes the CPU or when it enters a wait state

First Come First Served (FCFS)/First in First Out (FIFO) : Scheduling policy which sorts the *Ready Queue* with FCFS model and schedules the first arrived process from the *Ready queue* for execution

Last Come First Served (LCFS)/Last in First Out (LIFO) : Scheduling policy which sorts the *Ready Queue* with LCFS model and schedules the last arrived process from the *Ready queue* for execution

Shortest Job First (SJF) : Scheduling policy which sorts the *Ready queue* with the order of the shortest execution time for process and schedules the process with least estimated execution completion time from the *Ready queue* for execution

Priority based Scheduling : Scheduling policy which sorts the *Ready queue* based on priority and schedules the process with highest priority from the *Ready queue* for execution

Shortest Remaining Time (SRT) : Preemptive scheduling policy which sorts the *Ready queue* with the order of the shortest remaining time for execution completion for process and schedules the process with the least remaining time for estimated execution completion from the *Ready queue* for execution

Round Robin : Preemptive scheduling policy in which the currently running process is preempted based on time slice

Co-operating processes : Processes which share data for communicating among them

Inter Process/Task Communication (IPC) : Mechanism for communicating between co-operating processes of a system

Shared memory : A memory sharing mechanism used for inter process communication

Message passing : IPC mechanism based on exchanging of messages between processes through a message queue or mailbox

Message queue : A queue for holding messages for exchanging between processes of a multitasking system

Mailbox : A special implementation of message queue under certain OS kernel, which supports only a single message

Signal : A form of asynchronous message notification

Remote Procedure Call or RPC : The IPC mechanism used by a process to invoke a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network

Racing : The situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently

https://hemanthrajhemu.github.io

Deadlock : A situation where none of the processes are able to make any progress in their execution. 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

Livelock : A condition where a process always does something but is unable to make any progress in the execution completion

Starvation : The condition in which a process does not get the CPU or system resources required to continue its execution for a long time

Dining Philosophers' Problem : A real-life representation of the *deadlock*, *starvation*, *livelock* and *racing* issues in shared resource access in operating system context

Producer-Consumer problem : A common data sharing problem where two processes concurrently access a shared buffer with fixed size

Readers-Writers problem : A data sharing problem characterised by multiple processes trying to read and write shared data concurrently

Priority inversion : The condition in which a medium priority task gets the CPU for execution, when a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task

Priority inheritance : A mechanism by which the priority of a low-priority task which is currently holding a resource requested by a high priority task, is raised to that of the high priority task to avoid priority inversion

Priority Ceiling : The mechanism in which a priority is associated with a shared resource (The priority of the highest priority task which uses the shared resource) and the priority of the task is temporarily boosted to the priority of the shared resource when the resource is being held by the task, for avoiding priority inversion

Task/Process synchronisation : The act of synchronising the access of shared resources by multiple processes and enforcing proper sequence of operation among multiple processes of a multitasking system

Mutual Exclusion : The act of preventing the access of a shared resource by a task/process when it is being held by another task/process

Semaphore : A system resource for implementing mutual exclusion in shared resource access or for restricting the access to the shared resource

Mutex : The *binary semaphore* implementation for exclusive resource access under certain OS kernel

Device driver : A piece of software that acts as a bridge between the operating system and the hardware

Objective Questions

Operating System Basics

1. Which of the following is true about a kernel?
   (a) The kernel is the core of the operating system
   (b) It is responsible for managing the system resources and the communication among the hardware and other system services
   (c) It acts as the abstraction layer between system resources and user applications.
   (d) It contains a set of system libraries and services
   (e) All of these
2. The user application and kernel interface is provided through
   (a) System calls (b) Shared memory (d) None of these

https://hemanthrajhemu.github.io

3. The process management service of the kernel is responsible for
   (a) Setting up the memory space for the process
   (b) Allocating system resources
   (c) Scheduling and managing the execution of the process
   (d) Setting up and managing the Process Control Block (PCB), inter-process communication and synchronisation
   (e) All of these
4. The Memory Management Unit (MMU) of the kernel is responsible for
   (a) Keeping track of which part of the memory area is currently used by which process
   (b) Allocating and de-allocating memory space on a need basis (Dynamic memory allocation)
   (c) Handling all virtual memory operations in a kernel with virtual memory support
   (d) All of these
5. The memory area which holds the program code corresponding to the core OS applications/services is known as
   (a) User space (b) Kernel space (c) Shared memory (d) All of these
6. Which of the following is true about *Privilege separation*?
   (a) The user applications/processes runs at user space and kernel applications run at kernel space
   (b) Each user application/process runs on its own virtual memory space
   (c) A process is not allowed to access the memory space of another process directly
   (d) All of these
7. Which of the following is true about monolithic kernel?
   (a) All kernel services run in the kernel space under a single kernel thread.
   (b) The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilisation of the low-level features of the underlying system
   (c) Error prone. Any error or failure in any one of the kernel modules may lead to the crashing of the entire kernel
   (d) All of these
8. Which of the following is true about microkernel?
   (a) 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.
   (b) Highly modular and OS neutral
   (c) Less Error prone. Any 'Server' where error occurs can be restarted without restarting the entire kernel
   (d) All of these

Real-Time Operating System (RTOS)

1. Which of the following is true for Real-Time Operating Systems (RTOSes)?
   (a) Possess specialised kernel (b) Deterministic in behaviour
   (c) Predictable performance (d) All of these
2. Which of the following is (are) example(s) for RTOS?
   (a) Windows CE (b) Windows XP (c) Windows 2000 (d) QNX
   (e) (a) and (d)
3. Interrupts which occur in sync with the currently executing task are known as
   (a) Asynchronous interrupts (b) Synchronous interrupts
   (c) External interrupts (d) None of these
4. Which of the following is an example of a synchronous interrupt?
   (a) TRAP (b) External interrupt (c) Divide by zero (d) Timer interrupt
5. Which of the following is true about 'Timer tick' for RTOS?
   (a) The high resolution hardware timer interrupt is referred as 'Timer tick'
   (b) The 'Timer tick' is taken as the timing reference by the kernel
   (c) The time parameters for tasks are expressed as the multiples of the 'Timer tick'
   (d) All of these

https://hemanthrajhemu.github.io

6. Which of the following is true about hard real-time systems?
   (a) Strictly adhere to the timing constraints for a task
   (b) Missing any deadline may produce catastrophic results
   (c) Most of the hard real-time systems are automatic and may not contain a human in the loop
   (d) May not implement virtual memory based memory management
   (e) All of these.
7. Which of the following is true about soft real-time systems?
   (a) Does not guarantee meeting deadlines, but offer the best effort to meet the deadline are referred
   (b) Missing deadlines for tasks are acceptable
   (c) Most of the soft real-time systems contain a human in the loop
   (d) All of these

Tasks, Process and Threads

1. Which of the following is true about *Process* in the operating system context?
   (a) A '*Process*' is a program, or part of it, in execution
   (b) It can be an instance of a program in execution
   (c) 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.
   (d) A process is sequential in execution
   (e) All of these
2. A process has
   (a) Stack memory (b) Program memory (c) Working Registers (d) Data memory
   (e) All of these
3. The 'Stack' memory of a process holds all temporary data such as variables local to the process. State 'True' or 'False'
   (a) True (b) False
4. The data memory of a process holds
   (a) Local variables (b) Global variables (c) Program instructions (d) None of these
5. A process has its own memory space, when residing at the main memory. State 'True' or 'False'
   (a) True (b) False
6. A process when loaded to the memory is allocated a virtual memory space in the range 0x08000 to 0x08FF8. What is the content of the Stack pointer of the process when it is created?
   (a) 0x07FFF (b) 0x08000 (c) 0x08FF7 (d) 0x08FF8
7. What is the content of the program counter for the above example when the process is loaded for the first time?
   (a) 0x07FFF (b) 0x08000 (c) 0x08FF7 (d) 0x08FF8
8. The state where a process is incepted into the memory and awaiting the processor time for execution, is known as
   (a) Created state (b) Blocked state (c) Ready state (d) Waiting state
   (e) Completed state
9. The CPU allocation for a process may change when it changes its state from _______?
   (a) 'Running' to 'Ready' (b) 'Ready' to 'Running'
   (c) 'Running' to 'Blocked' (d) 'Running' to 'Completed'
   (e) All of these
10. Which of the following is true about threads?
    (a) A thread is the primitive that can execute code
    (b) A thread is a single sequential flow of control within a process
    (c) '*Thread*' is also known as lightweight process
    (d) All of these (e) None of these.
11. A process can have many threads of execution. State 'True' or 'False'
    (a) True (b) False

https://hemanthrajhemu.github.io

12. Different threads, which are part of a process, share the same address space. State 'True' or 'False'
    (a) True (b) False
13. Multiple threads of the same process share ________?
    (a) Data memory (b) Code memory
    (c) Stack memory (d) All of these
    (e) only (a) and (b)
14. Which of the following is true about multithreading?
    (a) Better memory utilisation
    (b) Better CPU utilisation
    (c) Reduced complexity in inter-thread communication
    (d) Faster process execution (e) All of these
15. Which of the following is a thread creation and management library?
    (a) POSIX (b) Win32
    (c) Java Thread Library (d) All of these
16. Which of the following is the POSIX standard library for thread creation and management
    (a) Pthreads (b) Threads (c) Jthreads (d) None of these
17. What happens when a thread object's *wait()* method is invoked in Java?
    (a) Causes the thread object to wait
    (b) The thread will remain in the 'Wait' state until another thread invokes the *notify()* or *notifyAll()* method of the thread object which is waiting
    (c) Both of these
    (d) None of these
18. Which of the following is true about user level threads?
    (a) Even if a process contains multiple user level threads, the OS treats it as a single thread
    (b) The user level threads are executed non-preemptively in relation to each other
    (c) User level threads follow co-operative execution model
    (d) All of these
19. Which of the following are the valid thread binding models for user level to kernel level thread binding?
    (a) One to Many (b) Many to One (c) One to One (d) Many to Many
    (e) All of these (f) only (b), (c) and (d)
20. If a thread expires, the stack memory allocated to it is reclaimed by the process to which the thread belongs. State 'True' or 'False'
    (a) True (b) False

Multiprocessing and Multitasking

1. Multitasking and multiprocessing refers to the same entity in the operating system context. State 'True' or 'False'
   (a) True (b) False
2. Multiprocessor systems contain
   (a) Single CPU (b) Multiple CPUs (c) No CPU
3. The ability of the operating system to have multiple programs in memory, which are ready for execution, is referred as
   (a) Multitasking (b) Multiprocessing (c) Multiprogramming
4. In a multiprocessing system
   (a) Only a single process can run at a time (b) Multiple processes can run simultaneously
   (c) Multiple processes run in pseudo parallelism
5. In a multitasking system
   (a) Only a single process can run at a time (b) Multiple processes can run simultaneously
   (c) Multiple processes run in pseudo parallelism (d) Only (a) and (c)
6. Multitasking involves
   (a) CPU execution switching of processes (b) CPU halting (c) No CPU operation

https://hemanthrajhemu.github.io

7. Multitasking involves
   (a) Context switching (b) Context saving (c) Context retrieval (d) All of these
   (e) None of these
8. What are the different types of multitasking present in operating systems?
   (a) Co-operative (b) Preemptive (c) Non-preemptive (d) All of these
9. In Co-operative multitasking, a process/task gets the CPU time when
   (a) The currently executing task terminates its execution
   (b) The currently executing task enters 'Wait' state
   (c) The currently executing task relinquishes the CPU before terminating
   (d) Never get a chance to execute (e) Either (a) or (c)
10. In Preemptive multitasking
    (a) Each process gets an equal chance for execution
    (b) The execution of a process is preempted based on the scheduling policy
    (c) Both of these (d) None of these
11. In Non-preemptive multitasking, a process/task gets the CPU time when
    (a) The currently executing task terminates its execution
    (b) The currently executing task enters 'Wait' state
    (c) The currently executing task relinquishes the CPU before terminating
    (d) All of these
    (e) None of these
12. MSDOS Operating System supports
    (a) Single user process with single thread
    (b) Single user process with multiple threads
    (c) Multiple user process with single thread per process
    (d) Multiple user process with multiple threads per process

Task Scheduling

1. Who determines which task/process is to be executed at a given point of time?
   (a) Process manager (b) Context manager (c) Scheduler (d) None of these
2. Task scheduling is an essential part of multitasking.
   (a) True (b) False
3. The process scheduling decision may take place when a process switches its state from
   (a) 'Running' to 'Ready' (b) 'Running' to 'Blocked'
   (c) 'Blocked' to 'Ready' (d) 'Running' to 'Completed'
   (e) All of these
   (f) Any one among (a) to (d) depending on the type of multitasking supported by OS
4. A process switched its state from 'Running' to 'Ready' due to scheduling act. What is the type of multitasking supported by the OS?
   (a) Co-operative (b) Preemptive (c) Non-preemptive (d) None of these
5. A process switched its state from 'Running' to 'Wait' due to scheduling act. What is the type of multitasking supported by the OS?
   (a) Co-operative (b) Preemptive (c) Non-preemptive (d) (b) or (c)
6. Which one of the following criteria plays an important role in the selection of a scheduling algorithm?
   (a) CPU utilisation (b) Throughput (c) Turnaround time (d) Waiting time
   (e) Response time (f) All of these
7. For a good scheduling algorithm, the CPU utilisation is
   (a) High (b) Medium (c) Non-defined

https://hemanthrajhemu.github.io

8. Under the process scheduling context, 'Throughput' is
   (a) The number of processes executed per unit of time
   (b) The time taken by a process to complete its execution
   (c) None of these
9. Under the process scheduling context, 'Turnaround Time' for a process is
   (a) The time taken to complete its execution (b) The time spent in the 'Ready' queue
   (c) The time spent on waiting on I/O (d) None of these
10. Turnaround Time (TAT) for a process includes
   (a) The time spent for waiting for the main memory (b) The time spent in the ready queue
   (c) The time spent on completing the I/O operations (d) The time spent in execution
   (e) All of these
11. For a good scheduling algorithm, the Turn Around Time (TAT) for a process should be
   (a) Minimum (b) Maximum (c) Average (d) Varying
12. Under the process scheduling context, 'Waiting time' for a process is
   (a) The time spent in the 'Ready queue'
   (b) The time spent on I/O operation (time spent in wait state)
   (c) Sum of (a) and (b)
   (d) None of these
13. For a good scheduling algorithm, the waiting time for a process should be
   (a) Minimum (b) Maximum (c) Average (d) Varying
14. Under the process scheduling context, 'Response time' for a process is
   (a) The time spent in 'Ready queue'
   (b) The time between the submission of a process and the first response
   (c) The time spent on I/O operation (time spent in wait state)
   (d) None of these
15. For a good scheduling algorithm, the response time for a process should be
   (a) Maximum (b) Average (c) Least (d) Varying
16. What are the different queues associated with process scheduling?
   (a) Ready Queue (b) Process Queue (c) Job Queue (d) Device Queue
   (e) All of the Above (f) (a), (c) and (d)
17. The 'Ready Queue' contains
   (a) All the processes present in the system (b) All the processes which are 'Ready' for execution
   (c) The currently running processes (d) Processes which are waiting for I/O
18. Which among the following scheduling is (are) Non-preemptive scheduling
   (a) First In First Out (FIFO/FCFS) (b) Last In First Out (LIFO/LCFS)
   (c) Shortest Job First (SJF) (d) All of these
   (e) None of these
19. Which of the following is true about FCFS scheduling
   (a) Favours CPU bound processes (b) The device utilisation is poor
   (c) Both of these (d) None of these
20. The average waiting time for a given set of process is ______ in SJF scheduling compared to FIFO scheduling
   (a) Minimal (b) Maximum (c) Average
21. Which among the following scheduling is (are) preemptive scheduling
   (a) Shortest Remaining Time First (SRT) (b) Preemptive Priority based
   (c) Round Robin (RR) (d) All of these
   (e) None of these
22. The Shortest Job First (SJF) algorithm is a priority based scheduling. State 'True' or 'False'
   (a) True (b) False

https://hemanthrajhemu.github.io

23. Which among the following is true about preemptive scheduling
    (a) A process is moved to the 'Ready' state from 'Running' state (preempted) without getting an explicit request from the process
    (b) A process is moved to the 'Ready' state from 'Running' state (preempted) on receiving an explicit request from the process
    (c) A process is moved to the 'Wait' state from the 'Running' state without getting an explicit request from the process
    (d) None of these
24. Which of the following scheduling technique(s) possess the drawback of 'Starvation'
    (a) Round Robin   (b) Priority based preemptive
    (c) Shortest Job First (SJF)   (d) (b) and (c)
    (e) None of these
25. Starvation describes the condition in which
    (a) A process is ready to execute and is waiting in the 'Ready' queue for a long time and is unable to get the CPU time due to various reasons
    (b) A process is waiting for a shared resource for a long time, and is unable to get it for various reasons.
    (c) Both of the above
    (d) None of these
26. Which of the scheduling policy offers equal opportunity for execution for all processes?
    (a) Priority based scheduling   (b) Round Robin (RR) scheduling
    (c) Shortest Job First (SJF)   (d) All of these
    (e) None of these
27. Round Robin (RR) scheduling commonly uses which one of the following policies for sorting the 'Ready' queue?
    (a) Priority   (b) FCFS (FIFO)   (c) LIFO   (d) SRT
    (e) SJF
28. Which among the following is used for avoiding 'Starvation' of processes in priority based scheduling?
    (a) Priority Inversion   (b) Aging   (c) Priority Ceiling   (d) All of these
29. Which of the following is true about 'Aging'
    (a) Changes the priority of a process at run time
    (b) Raises the priority of a process temporarily
    (c) It is a technique used for avoiding 'Starvation' of processes
    (d) All of these
    (e) None of these
30. Which is the most commonly used scheduling policy in Real-Time Operating Systems?
    (a) Round Robin (RR)   (b) Priority based preemptive
    (c) Priority based non-preemptive
    (d) Shortest Job First (SJF)
31. In the process scheduling context, the IDLE TASK is executed for
    (a) To handle system interrupts
    (b) To keep the CPU always engaged or to keep the CPU in idle mode depending on the system design
    (c) To keep track of the resource usage by a process
    (d) All of these

Task Communication and Synchronisation

1. Processes use IPC mechanisms for
    (a) Communicating between process   (b) Synchronising the access of shared resource
    (c) Both of these   (d) None of these
2. Which of the following techniques is used by operating systems for inter process communication?
    (a) Shared memory   (b) Messaging   (c) Signalling   (d) All of these

https://hemanthrajhemu.github.io

3. Under Windows Operating system, the input and output buffer memory for a named pipe is allocated in
   (a) Non-paged system memory (b) Paged system memory
   (c) Virtual memory (d) None of the above
4. Which among the following techniques is used for sharing data between processes?
   (a) Semaphores (b) Shared memory (c) Messages d) (b) and (c)
5. Which among the following is a shared memory technique for IPC?
   (a) Pipes (b) Memory mapped Object
   (c) Message blocks (d) Events (e) (a) and (b)
6. Which of the following is best advised for sharing a memory mapped object between processes under windows kernel?
   (a) Passing the handle of the shared memory object (b) Passing the name of the memory mapped object
   (c) None of these
7. Why is message passing relatively fast compared to shared memory based IPC?
   (a) Message passing is relatively free from synchronisation overheads
   (b) Message passing does not involve any OS intervention
   (c) All of these
   (d) None of these
8. In asynchronous messaging, the message posting thread just posts the message to the queue and will not wait for an acceptance (return) from the thread to which the message is posted. State 'True' or 'False'
   (a) True (b) False
9. Which of the following is a blocking message passing call in Windows?
   (a) PostMessage (b) PostThreadMessage (c) SendMessage (d) All of these
   (e) None of these
10. Under Windows operating system, the message is passed through _____ for Inter Process Communication (IPC) between processes?
   (a) Message structure (b) Memory mapped object
   (c) Semaphore (d) All of these
11. Which of the following is true about 'Signals' for Inter Process Communication?
   (a) Signals are used for asynchronous notifications (b) Signals are not queued
   (c) Signals do not carry any data (d) All of these
12. Which of the following is true about *Racing* or *Race condition*?
   (a) It is the condition in which multiple processes compete (race) each other to access and manipulate shared data concurrently
   (b) In a race condition the final value of the shared data depends on the process which acted on the data finally
   (c) Racing will not occur if the shared data access is atomic
   (d) All of these
13. Which of the following is true about *deadlock*?
   (a) 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
   (b) Is the situation in which none of the competing process will be able to access the resources held by other processes since they are locked by the respective processes
   (c) Is a result of chain of circular wait
   (d) All of these
14. What are the conditions favouring deadlock in multitasking?
   (a) Mutual Exclusion (b) Hold and Wait (c) No kernel resource preemption at kernel level
   (d) Chain of circular waits (e) All of these
15. Livelock describes the situation where
   (a) A process waits on a resource is not blocked on it and it makes frequent attempts to acquire the resource. But unable to acquire it since it is held by other process

https://hemanthrajhemu.github.io

(b) A process waiting in the 'Ready' queue is unable to get the CPU time for execution
(c) Both of these
(d) None of these

16. *Priority inversion* is
(a) The condition in which a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task
(b) The act of increasing the priority of a process dynamically
(c) The act of decreasing the priority of a process dynamically
(d) All of these

17. Which of the following is true about Priority inheritance?
(a) A low priority task which currently holds a shared resource requested by a high priority task temporarily inherits the priority of the high priority task
(b) The priority of the low priority task which is temporarily boosted to high is brought to the original value when it releases the shared resource
(c) All of these
(d) None of these

18. Which of the following is true about Priority Ceiling based Priority inversion handling?
(a) A priority is associated with each shared resource
(b) The priority associated to each resource is the priority of the highest priority task which uses this shared resource
(c) Whenever a task accesses a shared resource, the scheduler elevates the priority of the task to that of the ceiling priority of the resource
(d) The priority of the task is brought back to the original level once the task completes the accessing of the shared resource
(e) All of these

19. Process/Task synchronisation is essential for?
(a) Avoiding conflicts in resource access in multitasking environment
(b) Ensuring proper sequence of operation across processes.
(c) Communicating between processes
(d) All of these
(e) None of these

20. Which of the following is true about *Critical Section*?
(a) It is the code memory area which holds the program instructions (piece of code) for accessing a shared resource
(b) The access to the critical section should be exclusive
(c) All of these
(d) None of these

21. Which of the following is true about mutual exclusion?
(a) Mutual exclusion enforces mutually exclusive access of resources by processes
(b) Mutual exclusion may lead to deadlock
(c) Both of these
(d) None of these

22. Which of the following is an example of mutual exclusion enforcing policy?
(a) Busy Waiting (Spin lock) (b) Sleep & Wake up
(c) Both of these (d) None of these

23. Which of the following is true about lock based synchronisation mechanism?
(a) It is CPU intensive
(b) Locks are useful in handling situations where the processes is likely to be blocked for a shorter period of time on waiting the lock

https://hemanthrajhemu.github.io

(c) If the lock is being held for a long time by a process and if it is preempted by the OS, the other threads waiting for this lock may have to spin a longer time for getting

(d) All of these

(e) None of these

24. Which of the following synchronisation techniques follow the '*Sleep & Wakeup*' mechanism for mutual exclusion?

(a) Mutex (b) Semaphore (c) Critical Section (d) Spin lock

(e) (a), (b) and (c)

25. Which of the following is true about *mutex objects* for IPC synchronisation under Windows OS?

(a) Only one process/thread can own the '*mutex object*' at a time

(b) The state of a mutex object is set to non-signalled when it is not owned by any process/thread, and set to signalled when it is owned by any process/thread

(c) The state of a mutex object is set to signalled when it is not owned by any process/thread, and set to non-signalled when it is owned by any process/thread

(d) Both (a) & (b) (e) Both (a) & (c)

26. Which of the following is (are) the *wait functions* provided by windows for synchronisation purpose?

(a) WaitForSingleObject (b) WaitForMultipleObjects

(c) Sleep (d) Both (a) and (b)

27. Which of the following is true about *Critical Section object*?

(a) It can only be used by the threads of a single process (Intra process)

(b) The 'Critical Section' must be initialised before the threads of a process can use it

(c) Accessing Critical Section blocks the execution of the caller thread if the critical section is already in use by other threads

(d) Threads which are blocked by the Critical Section access call, waiting on a critical section, are added to a wait queue and are woken when the Critical Section is available to the requested thread

(e) All of these

28. Which of the following is a non-blocking Critical Section accessing call under windows?

(a) *EnterCriticalSection* (b) *TryEnterCriticalSection*

(c) Both of these (d) None of these

29. The Critical Section object makes the piece of code residing inside it ____?

(a) Non-reentrant (b) Re-entrant (c) Thread safe (d) Both (a) and (c)

30. Which of the following synchronisation techniques is exclusively used for synchronising the access of shared resources by the threads of a process (Intra Process Synchronisation) under Windows kernel?

(a) Mutex object (b) Critical Section object (c) Interlocked functions

(d) Both (c) and (d)

Review Questions

Operating System Basics

1. What is an Operating System? Where is it used and what are its primary functions?
2. What is kernel? What are the different functions handled by a general purpose kernel?
3. What is kernel space and user space? How is kernel space and user space interfaced?
4. What is monolithic and microkernel? Which one is widely used in real-time operating systems?
5. What is the difference between a General Purpose kernel and a Real-Time kernel? Give an example for both.

Real-Time Operating System (RTOS)

1. Explain the basic functions of a real-time kernel?
2. What is task control block (TCB)? Explain the structure of TCB

https://hemanthrajhemu.github.io

3. Explain the difference between the memory management of general purpose kernel and real-time kernel.
4. What is virtual memory? What are the advantages and disadvantages of virtual memory?
5. Explain how 'accurate time management' is achieved in real-time kernel
6. What is the difference between 'Hard' and 'Soft' real-time systems? Give an example for 'Hard' and 'Soft' Real-Time kernels

Tasks, Process and Threads

1. Explain *Task* in the operating system context
2. What is *Process* in the operating system context?
3. Explain the memory architecture of a process
4. What is *Process Life Cycle*?
5. Explain the various activities involved in the creation of process and threads
6. What is *Process Control Block* (*PCB*)? Explain the structure of *PCB*
7. Explain *Process Management* in the Operating System Context
8. What is *Thread* in the operating system context?
9. Explain how *Threads* and *Processes* are related? What are common to *Process* and *Threads*?
10. Explain the memory model of a 'thread'.
11. Explain the concept of 'multithreading'. What are the advantages of multithreading?
12. Explain how multithreading can improve the performance of an application with an illustrative example
13. Why is thread creation faster than process creation?
14. Explain the commonly used thread standards for thread creation and management by different operating systems
15. Explain *Thread context switch* and the various activities performed in thread context switching for user level and kernel level threads
16. What all information is held by the thread control data structure of a user/kernel thread?
17. What are the differences between user level and kernel level threads?
18. What are the advantages and disadvantages of using user level threads?
19. Explain the different thread binding models for user and kernel level threads
20. Compare threads and processes in detail

Multiprocessing and Multitasking

1. Explain multiprocessing, multitasking and multiprogramming
2. Explain context switching, context saving and context retrieval
3. What all activities are involved in context switching?
4. Explain the different multitasking models in the operating system context

Task Scheduling

1. What is *task scheduling* in the operating system context?
2. Explain the various factors to be considered for the selection of a scheduling criteria
3. Explain the different *queues* associated with process scheduling
4. Explain the different types of *non-preemptive* scheduling algorithms. State the merits and de-merits of each
5. Explain the different types of *preemptive* scheduling algorithms. State the merits and de-merits of each
6. Explain *Round Robin (RR)* process scheduling with interrupts
7. Explain *starvation* in the process scheduling context. Explain how starvation can be effectively tackled?
8. What is *IDLEPROCESS*? What is the significance of IDLEPROCESS in the process scheduling context?
9. Three processes with process IDs P1, P2, P3 with estimated completion time 5, 10, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3. Process P4 with estimated execution completion time 2 milliseconds enters the ready queue after 5 milliseconds. 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 the FIFO scheduling
10. Three processes with process IDs P1, P2, P3 with estimated completion time 12, 10, 2 milliseconds respectively enters the ready queue together in the order P2, P3, P1. Process P4 with estimated execution completion time

https://hemanthrajhemu.github.io

4 milliseconds enters the Ready queue after 8 milliseconds. 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 the FIFO scheduling

11. Three processes with process IDs P1, P2, P3 with estimated completion time 8, 4, 7 milliseconds respectively enters the ready queue together in the order P3, P1, P2. P1 contains an I/O waiting time of 2 milliseconds when it completes 4 milliseconds of its execution. P2 and P3 do not contain any I/O waiting. Calculate the waiting time and Turn Around Time (TAT) for each process and the average waiting time and Turn Around Time in the LIFO scheduling. All the estimated execution completion time is excluding I/O wait time
12. Three processes with process IDs P1, P2, P3 with estimated completion time 12, 10, 2 milliseconds respectively enters the ready queue together in the order P2, P3, P1. Process P4 with estimated execution completion time 4 milliseconds enters the Ready queue after 8 milliseconds. 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 the LIFO scheduling
13. Three processes with process IDs P1, P2, P3 with estimated completion time 6, 8, 2 milliseconds respectively enters the ready queue together. Process P4 with estimated execution completion time 4 milliseconds enters the Ready queue after 1 millisecond. 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 the non-preemptive SJF scheduling
14. Three processes with process IDs P1, P2, P3 with estimated completion time 4, 6, 5 milliseconds and priorities 1, 0, 3 (0—highest priority, 3 lowest priority) respectively enters the ready queue together. 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 non-preemptive priority based scheduling algorithm
15. Three processes with process IDs P1, P2, P3 with estimated completion time 4, 6, 5 milliseconds and priorities 1, 0, 3 (0—highest priority, 3 lowest priority) respectively enters the ready queue together. Process P4 with estimated execution completion time 6 milliseconds and priority 2 enters the 'Ready' queue after 5 milliseconds. 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 non-preemptive priority based scheduling algorithm
16. Three processes with process IDs P1, P2, P3 with estimated completion time 8, 4, 7 milliseconds respectively enters the ready queue together. P1 contains an I/O waiting time of 2 milliseconds when it completes 4 milliseconds of its execution. P2 and P3 do not contain any I/O waiting. Calculate the waiting time and Turn Around Time (TAT) for each process and the average waiting time and Turn Around Time in the SRT scheduling. All the estimated execution completion time is excluding I/O waiting time
17. Three processes with process IDs P1, P2, P3 with estimated completion time 12, 10, 6 milliseconds respectively enters the ready queue together. Process P4 with estimated execution completion time 2 milliseconds enters the Ready qieue after 3 milliseconds. 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 the SRT scheduling
18. Three processes with process IDs P1, P2, P3 with estimated completion time 10, 14, 20 milliseconds respectively, enters the ready queue together in the order P3, P2, P1. 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
19. Three processes with process IDs P1, P2, P3 with estimated completion time 12, 10, 12 milliseconds respectively enters the ready queue together in the order P2, P3, P1. Process P4 with estimated execution completion time 4 milliseconds enters the Ready queue after 8 milliseconds. 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 = 4 ms
20. Three processes with process IDs P1, P2, P3 with estimated completion time 4, 6, 5 milliseconds and priorities 1, 0, 3 (0—highest priority, 3 lowest priority) respectively enters the ready queue together. Calculate the waiting time

https://hemanthrajhemu.github.io

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 preemptive priority based scheduling algorithm

21. Three processes with process IDs P1, P2, P3 with estimated completion time 6, 2, 4 milliseconds respectively, enters the ready queue together in the order P1, P3, P2. Process P4 with estimated execution time 4 milliseconds entered the 'Ready' queue 3 milliseconds later the start of execution of P1. 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

Task Communication and Synchronisation

1. Explain the various process interaction models in detail.
2. What is Inter Process Communication (IPC)? Give an overview of different IPC mechanisms adopted by various operating systems.
3. Explain how multiple processes in a system co-operate.
4. Explain how multiple threads of a process co-operate.
5. Explain the shared memory based IPC.
6. Explain the concept of memory mapped objects for IPC.
7. Explain the handle sharing and name sharing based memory mapped object technique for IPC under Windows Operating System.
8. Explain the message passing technique for IPC. What are the merits and de-merits of message based IPC?
9. Explain the synchronous and asynchronous messaging mechanisms for IPC under Windows kernel.
10. Explain *Race condition* in detail, in relation to the shared resource access.
11. What is *deadlock*? What are the different conditions favouring deadlock?
12. Explain by *Coffman conditions*?
13. Explain the different methods of handling deadlocks.
14. Explain *livelock* in the resource sharing context.
15. Explain *starvation* in the resource sharing context.
16. Explain the *Dining Philosophers* problem in the process synchronisation context.
17. Explain the *Produces-consumer* problem in the inter process communication context.
18. Explain *bounded-buffer* problem in the interprocess communication context.
19. Explain *buffer overrun* and *buffer under-run.*
20. What is *priority inversion*? What are the different techniques adopted for handling priority inversion?
21. What are the merits and de-merits of *priority ceiling*?
22. Explain the different task communication synchronisation issues encountered in Interprocess Communication
23. What is *task (process) synchronisation*? What is the role of process synchronisation in IPC?
24. What is *mutual exclusion* in the process synchronisation context? Explain the different mechanisms for mutual exclusion
25. What are the merits and de-merits of *busy-waiting* (*spinlock*) based mutual exclusion?
26. Explain the *Test and Set Lock (TSL)* based mutual exclusion technique. Explain how *TSL* is implemented in Intel family of processors
27. Explain the *interlocked functions* for lock based mutual exclusion under Windows OS
28. Explain the advantages and limitations of *interlocked function* based synchronisation under Windows
29. Explain the *sleep & wakeup* mechanism for mutual exclusion
30. What are the merits and de-merits of *sleep & wakeup* mechanism based mutual exclusion?
31. What is *mutex*?
32. Explain the *mutex* based process synchronisation under Windows OS
33. What is *semaphore*? Explain the different types of semaphores. Where is it used?
34. What is binary *semaphore*? Where is it used?
35. What is the difference between *mutex* and *semaphore*?
36. What is the difference between *semaphore* and *binary semaphore*?
37. What is the difference between *mutex* and *binary semaphore*?

https://hemanthrajhemu.github.io

38. Explain the *semaphore* based process synchronisation under Windows OS
39. Explain the *critical section problem*?
40. What is *critical section*? What are the different techniques for controlling access to *critical section*?
41. Explain the *critical section object* for process synchronisation. Why is critical section object based synchronisation fast?
42. Explain the *critical section object* based process synchronisation under Windows OS.
43. Explain the *Event* based synchronisation mechanism for IPC.
44. Explain the *Event object* based synchronisation mechanism for IPC under Windows OS.
45. What is a *device driver*? Explain its role in the OS context.
46. Explain the architecture of device drivers.
47. Explain the different functional and non-functional requirements that needs to be evaluated in the selection of an RTOS.

Lab Assignments

1. Write a multithreaded Win32 console application satisfying:
   (a) The main thread of the application creates a child thread with name "child_thread" and passes the pointer of a buffer holding the data "Data passed from Main thread".
   (b) The main thread sleeps for 10 seconds after creating the child thread and then quits.
   (c) The child thread retrieves the message from the memory location pointed by the buffer pointer and prints the retrieved data to the console and sleeps for 100 milliseconds and then quits.
   (d) Use appropriate error handling mechanisms wherever possible.
2. Write a multithreaded Win32 console application for creating '*n*' number of child threads (*n* is configurable). Each thread prints the message "I'm in thread thread no" ('thread no' is the number passed to the thread when it is created. It varies from 0 to *n* – 1) and sleeps for 50 milliseconds and then quits. The main thread, after creating the child threads, wait for the completion of the execution of child threads and quits when all the child threads are completed their execution.
3. Write a multithreaded application using 'PThreads' for creating '*n*' number of child threads (*n* is configurable). The application should receive '*n*' as command line parameter. Each thread prints the message "I'm in thread thread no" ('thread no' is the number passed to the thread when it is created. It varies from 0 to *n* – 1) and sleeps for 1 second and then quits. The main thread, after creating the child threads, wait for the completion of the execution of child threads and quits when all the child threads are completed their execution. Compile and execute the application in Linux.
4. Write a multithreaded application in Win32 satisfying the following:
   (a) Two child threads are created with normal priority
   (b) Thread 1 retrieves and prints its priority and sleeps for 500 milliseconds and then quits
   (c) Thread 2 prints the priority of thread 1 and raises its priority to above normal and retrieves the new priority of thread 1, prints it and then quits
   (d) The main thread waits for the completion of both the child threads and then terminates.
5. Write a Win32 console application illustrating the usage of anonymous pipes for data sharing between a parent and child thread of a process. The application should satisfy the following conditions:
   (a) The main thread of the process creates an anonymous pipe with size 512KB and assigns the handle of the pipe to a global handle
   (b) The main thread creates an event object "synchronise" with state non-signalled and a child thread with name 'child_thread'.
   (c) The main thread waits for the signalling of the event object "synchronise" and reads data from the anonymous pipe when the event is signalled and prints the data read from the pipe on the console window.

https://hemanthrajhemu.github.io

(d) The main thread waits for the execution completion of the child thread and quits when the child thread completes its execution.

(e) The child thread writes the data "Hi from child thread" to the anonymous pipe and sets the event object "synchronise" and sleeps for 500 milliseconds and then quits.

Compile and execute the application using Visual Studio under Windows XP/NT OS.

6. Write a Win32 console application (Process 1) illustrating the creation of a memory mapped object of size 512KB with name "mysharedobject". Create an event object with name "synchronise" with state non-signalled. Read the memory mapped object when the event is signalled and display the contents on the console window. Create a second console application (Process 2) for opening the memory mapped object with name "mysharedobject" and event object with name "synchronise". Write the message "Message from Process 2" to the memory mapped object and set the event object "synchronise". Use appropriate error handling mechanisms wherever possible. Compile both the applications using Visual Studio and execute them in the order Process 1 followed by Process 2 under Windows XP/NT OS.

7. Write a multithreaded Win32 console application where:

(a) The main thread creates a child thread with default stack size and name 'Child_Thread'.

(b) The main thread sends user defined messages and the message 'WM_QUIT' randomly to the child thread.

(c) The child thread processes the message posted by the main thread and quits when it receives the 'WM_QUIT' message.

(d) The main thread checks the termination of the child thread and quits when the child thread completes its execution.

(e) The main thread continues sending random messages to the child thread till the WM_QUIT message is sent to child thread.

(f) The messaging mechanism between the main thread and child thread is synchronous.

Compile the application using Visual Studio and execute it under Windows XP/NT OS.

8. Write a Win32 console application illustrating the usage of anonymous pipes for data sharing between a parent and child processes using handle inheritance mechanism. Compile and execute the application using Visual Studio under Windows XP/NT OS.

9. Write a Win32 console application for creating an anonymous pipe with 512 bytes of size and pass the 'Read handle' of the pipe to a second process (another Win32 console application) using a memory mapped object. The first process writes a message "Hi from Pipe Server". The second process reads the data written by the pipe server to the pipe and displays it on the console window. Use event object for indicating the availability of data on the pipe and mutex objects for synchronising the access to the pipe.

10. Write a multithreaded Win32 Process addressing:

(a) The main thread of the process creates an unnamed memory mapped object with size 1K and shares the handle of the memory mapped object with other threads of the process

(b) The main thread writes the message "Hi from main thread" and informs the availability of data to the child thread by signalling an event object

(c) The main thread waits for the execution completion of the child thread after writing the message to the memory mapped area and quits when the child thread completes its execution

(d) The child thread reads the data from the memory mapped area and prints it on the console window when the event object is signalled by the main thread

(e) The read write access to the memory mapped area is synchronised using a mutex object

11. Write a multithreaded application using Java thread library satisfying:

(a) The first thread prints "Hello I'm going to the wait queue" and enters wait state by invoking the wait method.

(b) The second thread sleeps for 500 milliseconds and then prints "Hello I'm going to invoke first thread" and invokes the first thread.

(c) The first thread prints "Hello I'm invoked by the second thread" when invoked by the second thread.

https://hemanthrajhemu.github.io

12 Integration and Testing of Embedded Hardware and Firmware

LEARNING OBJECTIVES

- ✓ *Learn about the different techniques for embedding firmware into hardware*
- ✓ *Learn about the steps involved in the Out of system Programming*
- ✓ *Learn about the In System Programming (ISP) for firmware embedding*
- ✓ *Learn about the In Application Programming (IAP) for altering an area of the program storage memory for updating configuration data, tables, etc.*
- ✓ *Learn the technique used for embedding OS image and applications into the program storage memory of an embedded device*
- ✓ *Know the various things to be taken care in the 'board bring up'*

Integration testing of the embedded hardware and firmware is the immediate step following the embedded hardware and firmware development. Embedded hardware and firmware are developed in various steps as described in the earlier chapters. 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. As described in the earlier chapters, both embedded hardware and firmware should be independently tested (*Unit Tested*) to ensure their proper functioning. Functioning of individual hardware sections can be done by writing small utilities which checks the operation of the specified part. As far as the embedded firmware is concerned, its targeted functionalities can easily be checked by the simulator environment provided by the embedded firmware development tool's IDE. By simulating the firmware, the memory contents, register details, status of various flags and registers can easily be monitored and it gives an approximate picture of "What happens inside the processor/controller and what are the states of various peripherals" when the firmware is running on the target hardware. The IDE gives necessary support for simulating the various inputs required from the external world, like inputting data on ports, generating an interrupt condition,

https://hemanthrajhemu.github.io

etc. This really helps in debugging the functioning of the firmware without dumping the firmware in a real target board.

12.1 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 firmware embedding techniques for a non-OS based embedded system are explained below. The non-OS based embedded systems store the firmware either in the onchip processor/controller memory or offchip memory (FLASH/NVRAM, etc.).

12.1.1 Out-of-Circuit Programming

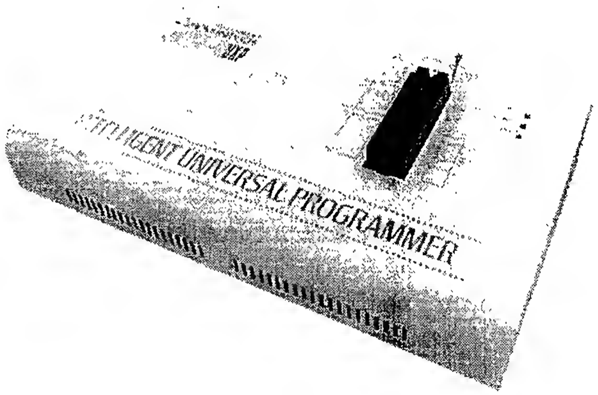


Fig. 12.1 **Firmware Embedding Tool-Device Programmer: LabTool-48UXP)**
(Courtesy of Burn Technology Limited www.burntec.com & Advantech Equipment Corp www.aec.com.tw)

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. 12.1). The programming device is a dedicated unit which contains the necessary hardware circuit to generate the programming signals. 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. 12.2).

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

https://hemanthrajhemu.github.io

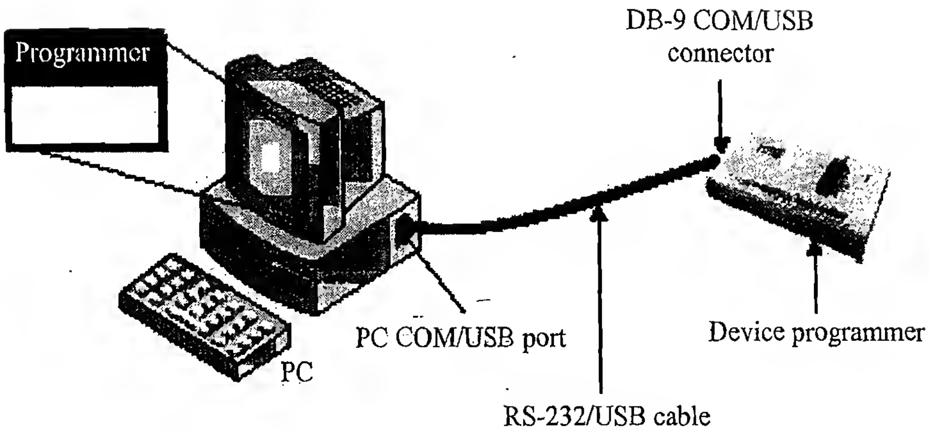


Fig. 12.2 Interfacing of Device Programmer with PC

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 file 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'

https://hemanthrajhemu.github.io

resolves this issue to certain extent. A gang programmer is similar to an ordinary programmer except that it contains multiple ZIF sockets (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.

12.1.2 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-configured so that it will operate normally by applying a reset or a re-power up.

12.1.2.1 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. What you need to do is just connect the pins as mentioned by the program requirement. As mentioned earlier, for ISP operations, the

https://hemanthrajhemu.github.io

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 first 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 verified 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 best advised 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.

12.1.3 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

https://hemanthrajhemu.github.io

and performing a call to the common entry point. Like any other subroutine call, after completion of the function, control will return to the end-user'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).

12.1.4 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 files 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.

12.1.5 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, TCP/IP etc. Bootloader implements menu options for selecting the source for OS image to load (Typical menu item examples are 1. 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. We will discuss about the role of '*Boot loader*', '*Boot loader*' development and embedding, firmware and OS embedding and booting process for an OS based embedded system in detail in a dedicated book coming under this series. Please be patient till then.

12.2 BOARD POWER UP

Now the firmware is embedded into the target board using one of the programming techniques described above. 'What Next?' The answer is power up the board. You may be expecting the device functioning exactly in a way as you designed. But in real scenario it need not be and if the board functions well in the first attempt itself you are very lucky. Sometimes the first power up may end up in a messy explosion leaving the smell of burned components behind. It may happen due to various reasons, like Proper care was not taken in applying the power and power applied in reverse polarity (+ve of supply connected to

https://hemanthrajhemu.github.io

–ve of the target board and vice versa), components were not placed in the correct polarity order (E.g. a capacitor on the target board is connected to the board with +ve terminal to –ve of the board and vice versa), etc... etc... *I would like to share a very interesting incident which happened in my development career during the power up of a new board. Though the board was well checked and extreme care was taken in applying the power, when I powered the board after embedding the firmware, I heard an explosion followed by the burning of a capacitor on the target board and I really struggled hard to stop the fire from spreading by a strong puff of air—It is not a joke, It's a true incident.* The reason was very simple, the power applied to the board was 9V and the filter capacitor placed at the regulator IC input side was with a rating of 6V (220MFD/6V). Since the capacitor voltage rating was below the input supply, the dielectric of capacitor got burned. Be cautious: Before you power up the board make sure everything is intact. We will discuss about the various tools used for troubleshooting the target hardware in a later chapter.

Summary

✓ Integration of hardware and firmware deals with the embedding of firmware into the target hardware board.

✓ 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, the firmware is held in an external dedicated EPROM/FLASH memory chip

✓ Out-of-circuit programming and In System Programming (ISP) are the two different methods for embedding firmware into a non Operating System based product

✓ In the out-of-circuit programming, the device is removed from the target board and is programmed using a 'Device Programmer', whereas in the In System Programming technique, the firmware is embedded into the controller memory/program memory chip without removing the chip from the target board

✓ JTAG, SPI, etc. are the commonly used serial protocols for transferring the embedded firmware into the target device

✓ 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

✓ IAP is normally used for updating calibration data, look-up tables, etc. which are stored in code memory

✓ Factory programmed chip embeds firmware into the chip at the time of chip fabrication

✓ In System Programming (ISP) is used for embedding the OS image and application program into the non-volatile storage memory of the embedded product. The bootloader program running in the target device implements the necessary routine for embedding the OS image into the non-volatile memory and booting the system

Keywords

JTAG : An Interface for hardware troubleshooting like boundary scan testing

In System Programming (ISP) : Firmware embedding technique in which the firmware is embedded into the program memory without removing the chip from the target board

Serial Peripheral Interface (SPI) : Serial interface for connecting devices

https://hemanthrajhemu.github.io

In Application Programming (IAP) : A technique used by the firmware running on the target device for modifying a selected portion of the code memory

BootROM : A program, which contains the libraries for implementing In System Programming, which is embedded into the memory at the time of chip fabrication

Bootloader : 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

Review Questions

1. Explain the different techniques for embedding the firmware into the target board for a non-OS based embedded system
2. Explain the major drawbacks of *out-of-circuit* programming
3. Explain the firmware embedding process for OS based embedded products
4. What is the difference between In System Programming (ISP) and In Application Programming (IAP)?

https://hemanthrajhemu.github.io

13 The Embedded System Development Environment

LEARNING OBJECTIVES

- ✓ Learn about the different entities of the embedded system development environment
- ✓ Learn about the Integrated Development Environments (IDEs) for embedded firmware development and debugging
- ✓ Learn the usage of µVision3 IDE from Keil software (www.keil.com) for embedded firmware development, simulation and debugging for 8051 family of microcontrollers
- ✓ Familiarise with the different IDEs for firmware development for different family of processors/controllers and Embedded Operating Systems
- ✓ Learn about the different types of files (List File, Preprocessor output file, Object File, Map File, Hex File, etc.) generated during the cross compilation of a source file written in high level language like Embedded C and during the cross assembling of a source file written in Assembly Language
- ✓ Learn about disassembler and decompiler, and their role in embedded firmware development
- ✓ Learn about Simulators, In Circuit Emulators (ICE), and Debuggers and their role in embedded firmware debugging
- ✓ Learn about the different tools and techniques used for embedded hardware debugging
- ✓ Learn the use of magnifying glass, multimeter, CRO, logic analyser and function generator in embedded hardware debugging
- ✓ Learn about the boundary scanning technique for testing the interconnection among the various chips in a complex hardware board

This chapter is designed to give you an insight into the embedded-system development environment. The various tools used and the various steps followed in embedded system development are explained here. It is a summary of the various design and development phases we discussed so far and an introduction to the various design/development/debug tools employed in embedded system development. A typical embedded system development environment is illustrated in Fig. 13.1.

As illustrated in the figure, 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

https://hemanthrajhemu.github.io

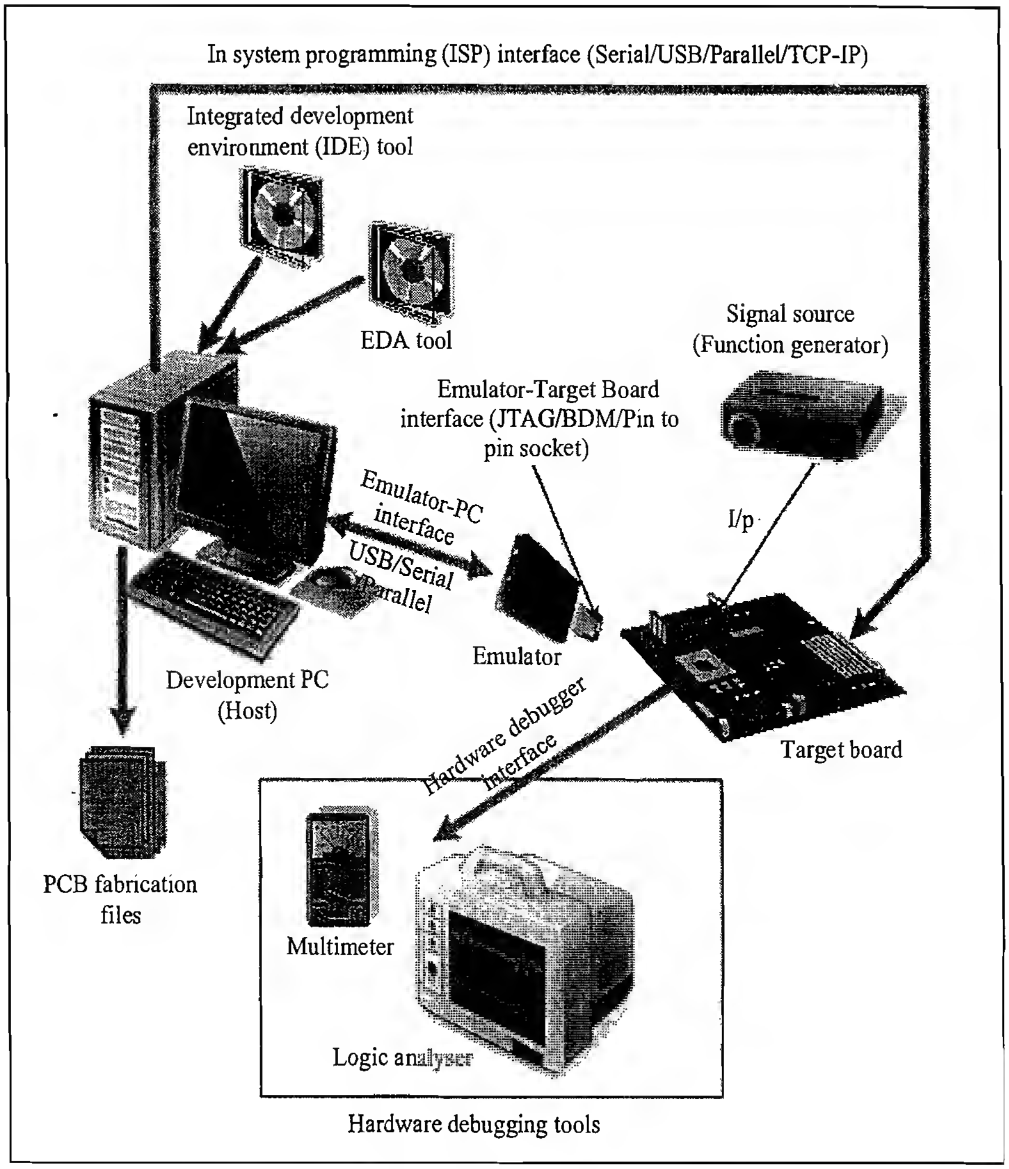

In system programming (ISP) interface (Serial/USB/Parallel/TCP-IP)
Integrated development environment (IDE) tool
EDA tool
Signal source (Function generator)
Emulator-Target Board interface (JTAG/BDM/Pin to pin socket)
Emulator-PC interface USB/Serial Parallel
I/p
Emulator
Development PC (Host)
Target board
Hardware debugger interface
PCB fabrication files
Multimeter
Logic analyser
Hardware debugging tools

Fig. 13.1 **The Embedded System Development Environment**

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 files in CDs 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.

13.1 THE INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

In embedded system development context, Integrated Development Environment (IDE) stands for an integrated environment for developing and debugging the target processor specific embedded firmware. IDE is a software package which bundles a 'Text Editor (Source Code Editor)', 'Cross-compiler (for

https://hemanthrajhemu.github.io

cross platform development and compiler for same platform development)', 'Linker' and a 'Debugger'. Some IDEs may provide interface to target board emulators, Target processor's/controller's Flash memory programmer, etc. and incorporate other software development utilities like 'Version Control Tool', 'Help File for the Development Language', etc. IDEs can be either command line based or GUI based. Command line based IDEs may include little or less GUI support. The old version of TURBO C IDE for developing applications in C/C++ for x86 processor on Windows platform is an example for a generic IDE with command line interface. GUI based IDEs provide a Visual Development Environment with mouse click support for each action. Such IDEs are generally known as Visual IDEs. Visual IDEs are very helpful in firmware development. A typical example for a Visual IDE is Microsoft® Visual Studio for developing Visual C++ and Visual Basic programs. Other examples are NetBeans and Eclipse.

IDEs used in embedded firmware development are slightly different from the generic IDEs used for high level language based development for desktop applications. In Embedded Applications, the IDE is either supplied by the target processor/controller manufacturer or by third party vendors or as Open Source. MPLAB is an IDE tool supplied by microchip for developing embedded firmware using their PIC family of microcontrollers. Keil µVision3 (spelt as micro vision three) from Keil software is an example for a third party IDE, which is used for developing embedded firmware for *8051* family microcontrollers. CodeWarrior by Metrowerks is an example of IDE for ARM family of processors. It should be noted that in embedded firmware development applications each IDE is designed for a specific family of controllers/processors and it may not be possible to develop firmware for all family of controllers/processors using a single IDE (as of now there is no known IDE with support for all family of processors/controllers). However there is a rapid move happening towards the open source IDE, Eclipse for embedded development. Most of the proccessor/control manufacturers and third party IDE providers are trying to build the IDE around the popular Eclipse open source IDE. This may lead to a single IDE based on Eclipse for embedded system development in the near future. Since this book is primarily focusing on *8051* based embedded firmware development, the IDE chosen for demonstration is Keil µVision3. A demo version of the tool for Microsoft Windows OS based development is available for free download from the Keil Software website. Please instal the same on your machine before proceeding to the next sections.

13.1.1 The Keil µVision3 IDE for *8051*

Keil µVision3 is a licensed IDE tool from Keil Software (www.keil.com), an ARM company, for *8051* family microcontroller based embedded firmware development. To start with the IDE (after installing the demo tool) execute the program Uv3.exe (or the short cut 'Keil µVision3' from desktop or 'All Programs' tab from 'Start Menu' – For Host machine with Microsoft® Windows Operating System). The IDE view is shown in Fig. 13.2.

The IDE looks very similar to the Microsoft® Visual Studio IDE and it contains various menu options, a project window showing files, Register view, Books and Functions Tab and an output window. To start a new project, go to the '*Project*' tab on the menu, select '*New Project*' option. Give a name to your workspace in the '*File Name*' section of the '*Create New Project*' Pop-up dialog Box (Let it be 'sample'). Choose the directory to save the project from the pop-up dialog. The default extension of a project workspace file is .uv2. On clicking the '***Save***' button of the '*Create New Project*' pop-up dialog, a device selection dialog as shown in Fig. 13.3 appears on the screen.

This Dialog Box lists out all the vendors (manufacturers) for *8051* family microcontroller, supported by IDE. Choose the manufacturer of the chip for your design (Let it be 'Atmel' for our design). Atmel itself manufactures a variety of *8051* flavours. Choose the exact part number of the device used as the

https://hemanthrajhemu.github.io

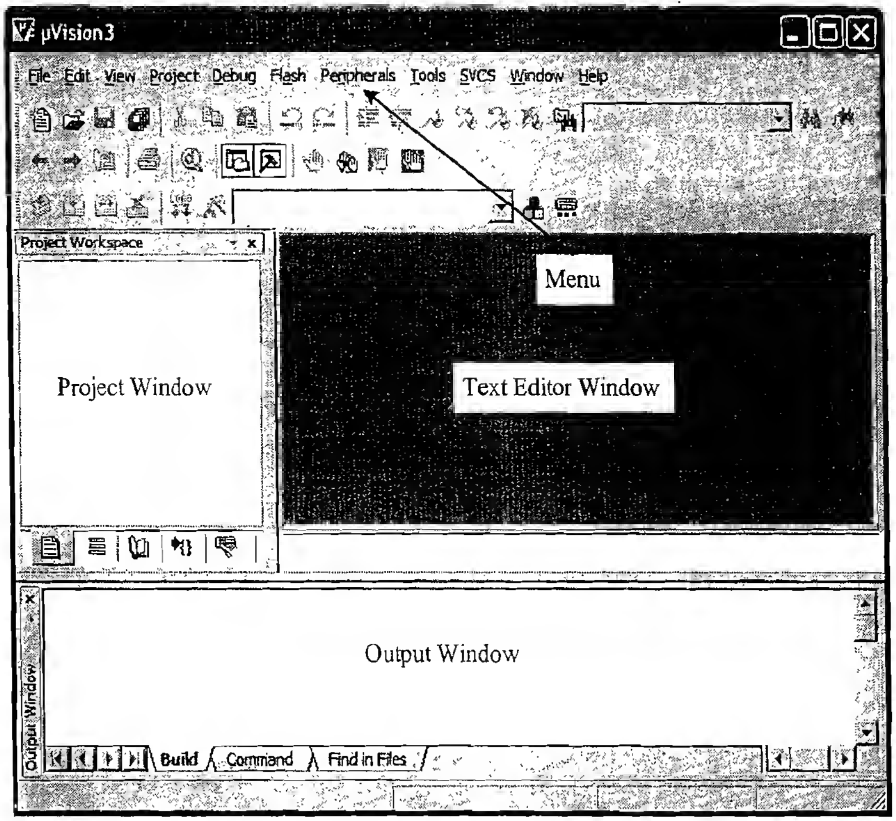
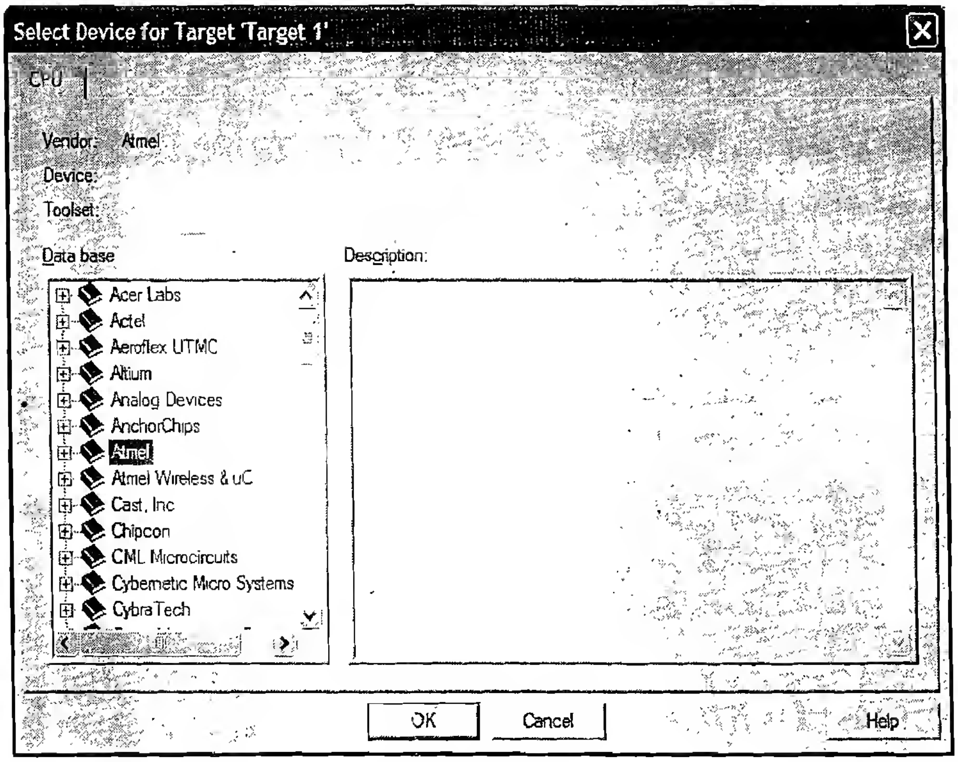

µVision3
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Project Workspace
Project Window
Menu
Text Editor Window
Output Window
Output Window
Build Command Find in Files

Fig. 13.2 **Keil µVision3 Integrated Development Environment (IDE)**

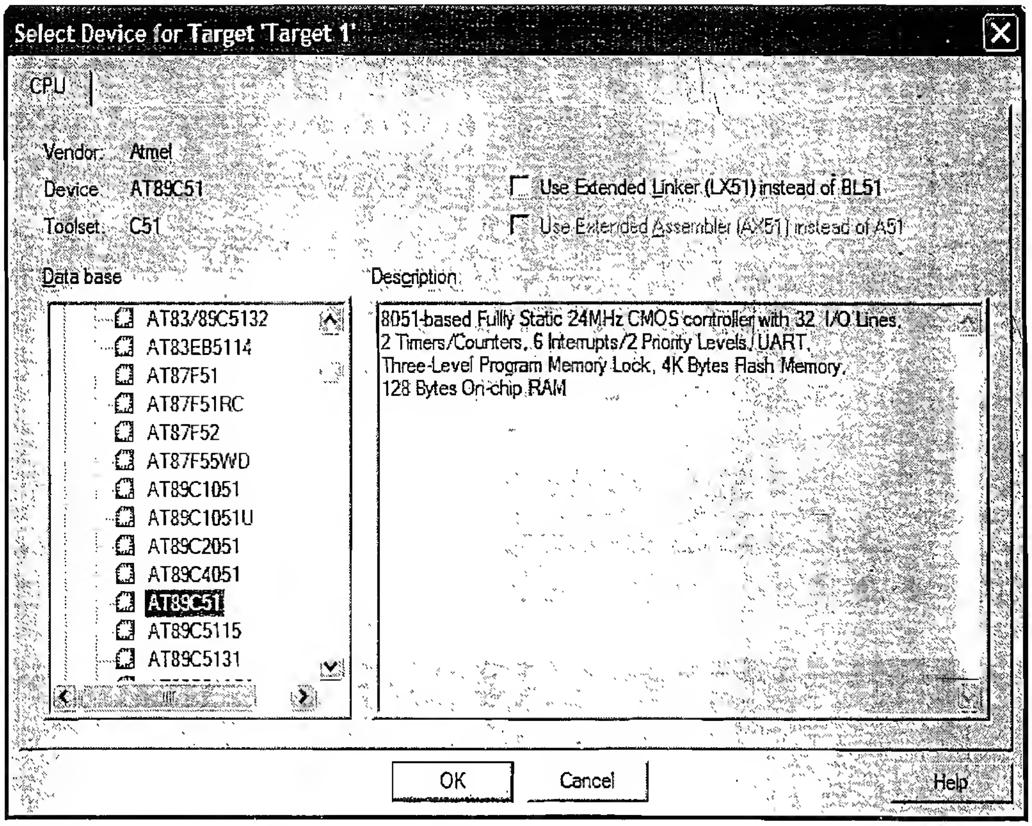


Vendor: Atmel
Device:
Toolset:

Fig. 13.3 **Target CPU Vendor selection for Keil µVision3 IDE**

https://hemanthrajhemu.github.io

target processor for the design, by expanding the vendor node. It will list out all supported CPUs by the selected vendor under the vendor node. On selecting the target processor's exact part number, the vendor name, device name and tool set supported for the device is displayed on the appropriate fields of the dialog box along with a small description of the target processor under the *Description* column on the right side of the pop-up dialog as shown below. Press 'OK' to proceed after selecting the target CPU (Let it be 'AT89C51' for our design).



Vendor: Atmel
Device: AT89C51
Toolset: C51

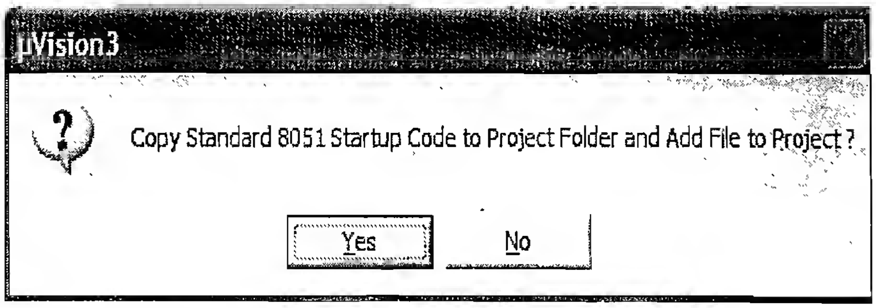
□ Use Extended Linker (LX51) instead of BL51
□ Use Extended Assembler (AX51) instead of A51

AT83/89C5132
AT83EB5114
AT87F51
AT87F51RC
AT87F52
AT87F55WD
AT89C1051
AT89C1051U
AT89C2051
AT89C4051
AT89C51
AT89C5115
AT89C5131

8051-based Fully Static 24MHz CMOS controller with 32 I/O Lines, 2 Timers/Counters, 6 Interrupts/2 Priority Levels, UART, Three-Level Program Memory Lock, 4K Bytes Flash Memory, 128 Bytes On-chip RAM

Fig. 13.4 **Target CPU selection for Keil μVision3 IDE**

Once the target processor is selected, the IDE automatically adds the required startup code for the firmware and it prompts you whether the standard startup code needs to be added to the project (Fig. 13.5). Press '*Yes*' to proceed. The startup code contains the required default initialisation like stack pointer setting and initialisation, memory clearing, etc. On cross-compiling, the code generated for the startup file is placed on top of the code generated for the function *main()*. Hence the reset vector



Copy Standard 8051 Startup Code to Project Folder and Add File to Project ?

Fig. 13.5 **Startup file addition to the project**

https://hemanthrajhemu.github.io

(0000H) always starts with the execution of startup code before the main code. For more details on the contents and code of startup file please go through the µVision help files which is listed under the '*Books*' section of the project workspace window.

A '*Target*' group with the name '*Target1*' is automatically generated under the '*Files*' section of the project Window. '*Target1*' contains a '*Source Group*' with the name '*Source Group1*' and the startup file (STARTUP.A51) is kept under this group (Fig. 13.6). All these groups are generated automatically. If you want you can rename these groups by clicking the respective group names.

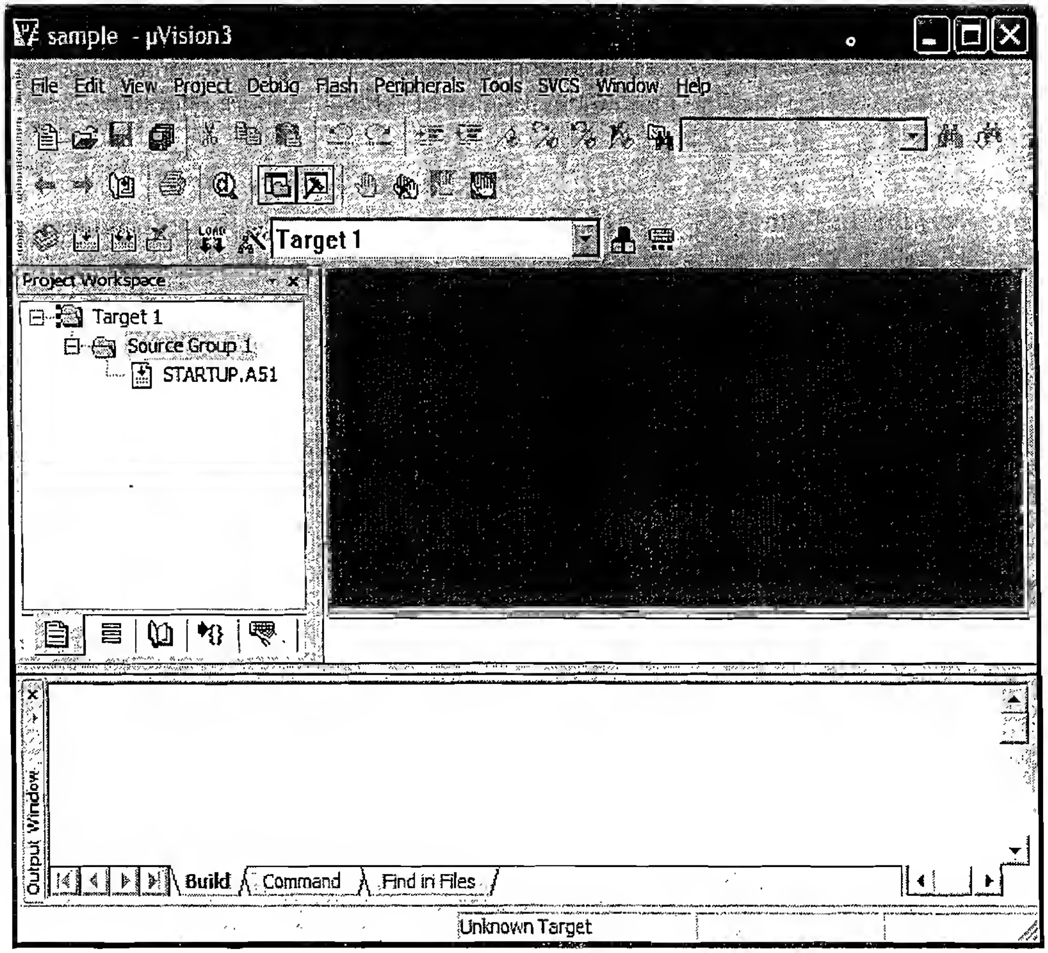


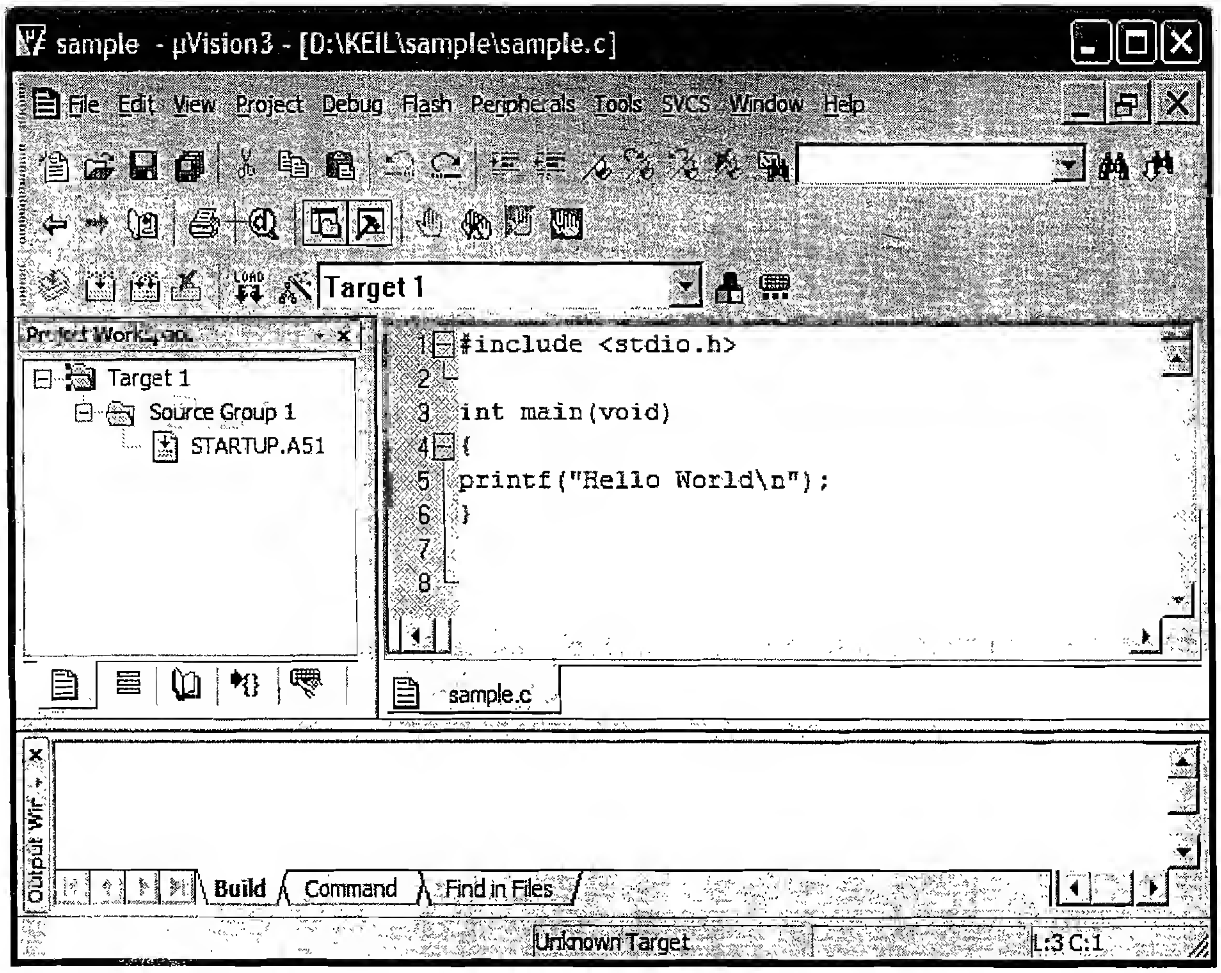
Fig. 13.6 **Startup file added to the project**

You can see that similar to the Visual Studio IDE's '*Project Window*' for VC++ development, Keil IDE's '*Project Window*' also contains multiple tabs. They are the '*Files*' tab, which gives the file details for the current project, '*Regs*' tab, giving the Register details while debugging the source code, '*Books*' tab showing all the available help and documentation files supplied by the IDE, '*Functions*' tab lists out the functions present in a '*C*' source file and finally a '*Templates*' tab which generate automatic code framework (function body) for *if*, *if else*, *switch case* etc and code documentation template (Header). These steps create a project workspace. To start with the firmware development we need to create a source file and then add that source file to the '*Source Group*' of the '*Target*'. Click on the '*File*' tab on the menu tool of the IDE and select the option '*New*'. A blank text editor will be visible on the IDE to the right of the '*Project Window*'. Start writing the code on the text editor as per your design (Refer to

https://hemanthrajhemu.github.io

the Keil help file for using Keil supported specific Embedded C instructions for *8051* family). You can write the program in ANSI C and *8051* specific codes (like Port Access, bit manipulation instruction etc) using Keil specific Embedded C codes. For using the Keil specific Embedded code, you need to add the specific header file to the text editor using the *#include* compiler directive. For example, *#include <reg51.h>* is the header file including all the target processor specific declarations for *8051* family processors for Keil C51 Compiler. Standard '*C*' programs (Desktop applications) calls the library routines for accessing the I/O and they are defined in the *stdio.h* file, whereas these library files cannot be used as such for embedded application development since the I/O medium is not a graphic console as in the *C* language based development on DOS Operating system and they are re-defined for the target processor I/Os for the cross compilers by the cross compiler developer. If you open the *stdio.h* file by ANSI C and Keil for its IDE, you can find that the implementation of I/O related functions (e.g. *printf()*) are entirely different.

The difference in the implementation is explained with typical *stdio.h* function-*printf()* (e.g. *printf("Hello World\n")).* With ANSI C & DOS the function outputs the string *Hello World* to the DOS console whereas with the C51 cross-compiler, the same function outputs the string *Hello World* to the serial port of the device with the default settings. Coming back to the firmware development, let's follow the universal unwritten law of first 'C' program- The "*Hello World*" program. Write a simple 'C' code to print the string Hello world.

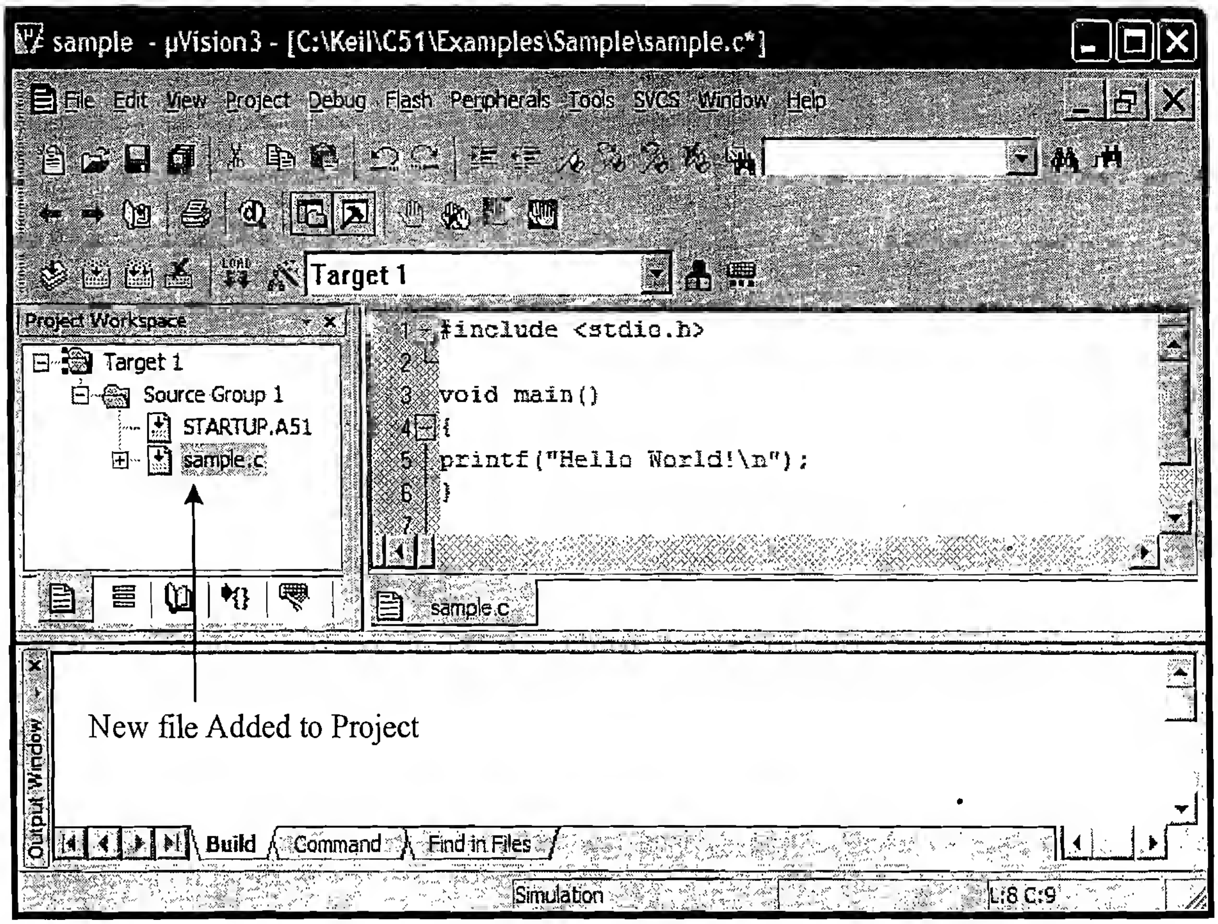


```
#include <stdio.h>

int main(void)
{
printf("Hello World\n");
}
```

Fig. 13.7 **Writing the first Embedded C code**

https://hemanthrajhemu.github.io

The code is written in the text editor which appears within the IDE on selecting the '*New*' tab from the '*File*' Menu. Write the code in *C* language syntax (Fig. 13.7). Add the necessary header files. You can make use of the standard template files available under the '*Templates*' tab of the '*Project Window*' for adding functions, loops, conditional instructions, etc. for writing the code. Once you are done with the code, save it with extension .c in a desired folder (Preferably in the current project directory). Let the name of the file be '*sample.c*'. At the moment you save the program with extension .c, all the keywords (like #include, int, void, etc.) appear in a different colour and the title bar of the IDE displays the name of the current .c file along with its path. By now we have created a 'c' source file. Next step is adding the created source file to the project. For this, right click on the '*Source Group*' from the '*Project Window*' and select the option '*Add Files to Group 'Source Group''*. Choose the file '*sample.c*' from the file selection Dialog Box and press '*Add*' button and exit the file selection dialog by pressing '*Close*' button. Now the file is added to the target project (Fig. 13.8). You can see the file in the '*Project Window*' under '*Files*' tab beneath the '*Source Group*'.



```
#include <stdio.h>

void main()
{
printf("Hello World!\n");
}
```

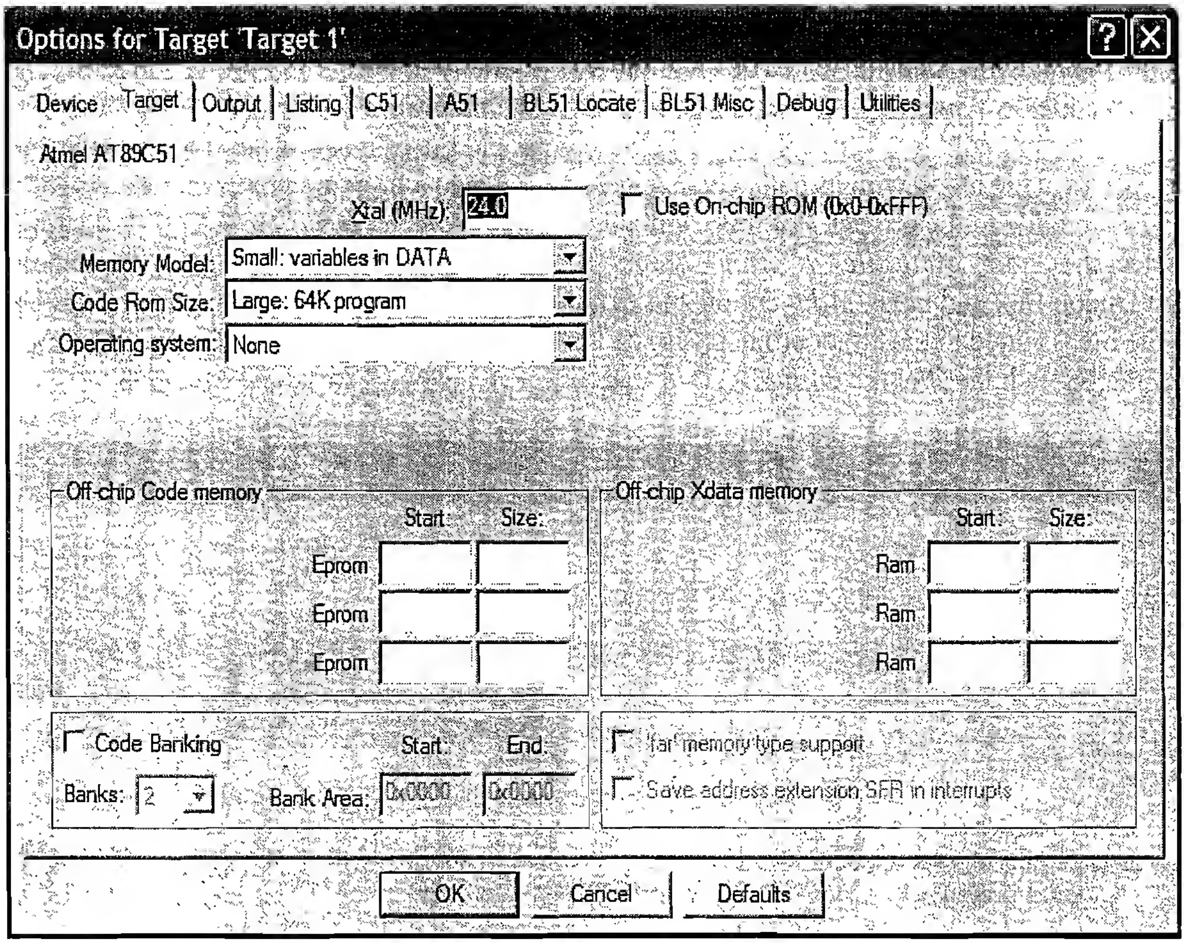
sample - µVision3 - [C:\Keil\C51\Examples\Sample\sample.c*]
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Source Group 1
STARTUP.A51
sample.c
sample.c
New file Added to Project
Output Window
Build
Command
Find in Files
Simulation
L:8 C:9

Fig. 13.8 **Adding Files to the Project**

If you are following the modular programming technique, you may have different source files for performing an intended operation. Add all those files to the project as described above. It should be noted that function *main()* is the only entry point and only one '.*c*' file among the files added to the project is allowed to contain the function *main()*. If more than one file contains a function with the name *main()*, compilation will end up in error. The next step is configuring the target. To configure the target,

https://hemanthrajhemu.github.io

go to '*Project*' tab on the Menu and select '*Options for Target*'. The configuration window as shown in Fig. 13.9 is displayed.



Atmel AT89C51

Xtal (MHz): 24.0

□ Use On-chip ROM (0x0-0xFFF)

Memory Model: Small: variables in DATA

Code Rom Size: Large: 64K program

Operating system: None

	Start:	Size:
Eprom		
Eprom		
Eprom		

	Start:	Size:
Ram		
Ram		
Ram		

□ Code Banking

Banks: 2

Bank Area: Start: 0x0000 End: 0x0000

□ 'far' memory type support

□ Save address extension SFR in interrupts

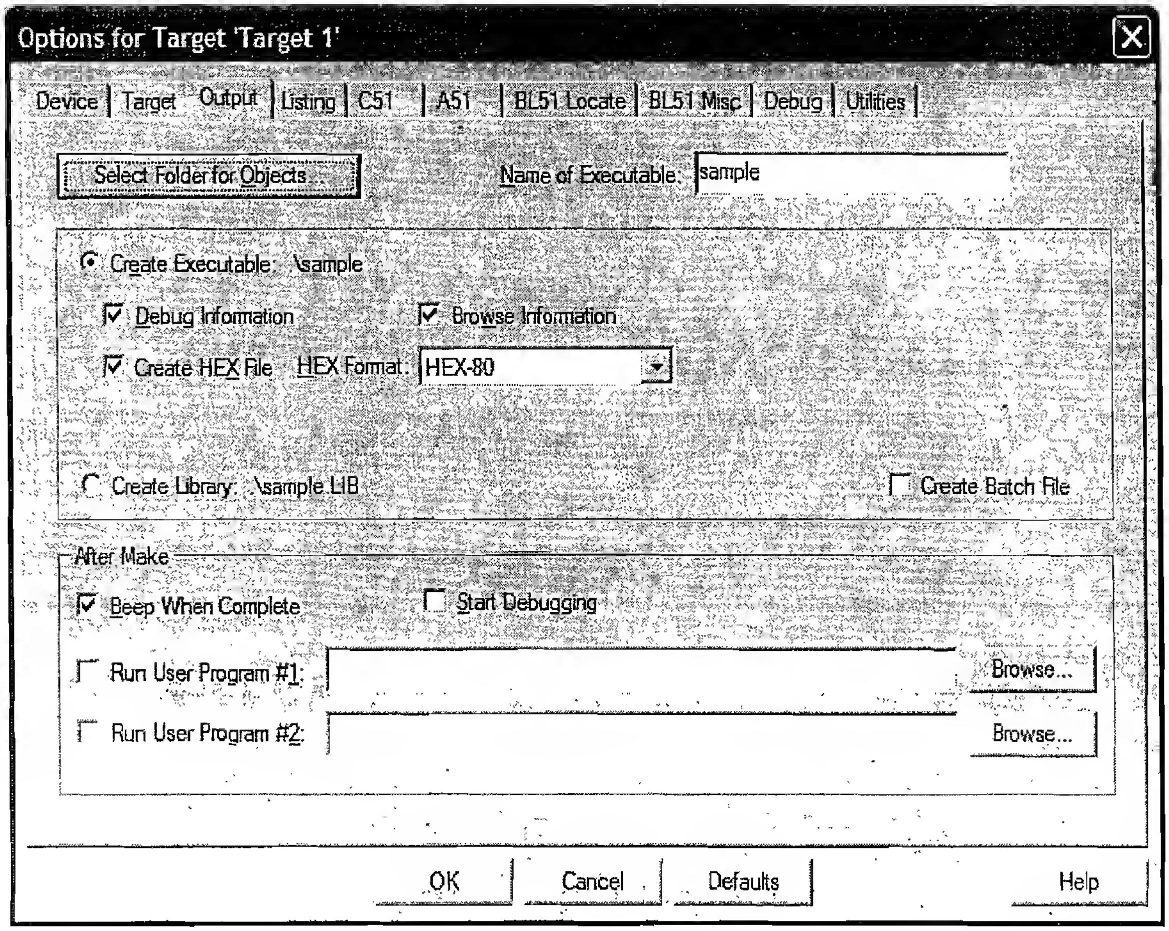
Fig. 13.9 **Target Configuration**

The target configuration window is a tabbed dialog box. The device is already configured at the time of creating a new project by selecting the target device (e.g. Atmel AT89C51). If you want to check it, select the '*Device*' tab and verify the same. Select '*Target*' tab and configure the following. Give the clock frequency for which the system is designed. e.g. 6MHz, 11.0592MHz, 12MHz, 24MHz, etc. This has nothing to do with the firmware creation but it is very essential while debugging the firmware to note the execution time since execution time is dependent on the clock frequency. If the system is designed to make use of the processor resident code memory, select the option Use On-chip ROM (For AT89C51 On-chip ROM is 4K only; 0x0000 to 0x0FFF). If external code memory is used, enter the start address and size of the code memory at the Off-chip Code memory column (e.g. Eprom Start: 0x0000 and Size 0x0FFF). The working memory (data memory or RAM) can also be either internal or external to the processor. If it is external, enter the memory map starting address of the external memory along with the size of external memory in the 'Off-chip Xdata memory section (e.g. Ram Start: 0x8000 and Size 0x 1000). Select the memory model for the target. Memory model refers to the data memory. Keil supports three types of data memory model; internal data memory (Small), external data memory in paged mode (Compact) and external data memory in non-paged mode (Large). Now select the Code memory size.

https://hemanthrajhemu.github.io

Code memory model is also classified into three; namely, Small (code less than 2K bytes), Compact (2K bytes functions and 64K bytes code memory) and Large (Plain 64K bytes memory). Choose the type depending on your target application and target hardware design. If your design is for an RTOS based system, select the supported RTOS by the IDE. Keil supports two 8 bit RTOS namely, RTX51 Tiny and RTX51 Full. Choose none for a non RTOS based design.

Move to the next Tab, '***Output***'. The output tab holds the settings for output file generation from the source code (Fig. 13.10). The source file can either be converted into an executable machine code or a library file.



Options for Target 'Target 1'

Device | Target | Output | Listing | C51 | A51 | BL51 Locate | BL51 Misc | Debug | Utilities

Select Folder for Objects... Name of Executable: sample

● Create Executable: .\sample

☑ Debug Information ☑ Browse Information

☑ Create HEX File HEX Format: HEX-80

○ Create Library: .\sample.LIB □ Create Batch File

After Make

☑ Beep When Complete □ Start Debugging

□ Run User Program #1: Browse...

□ Run User Program #2: Browse...

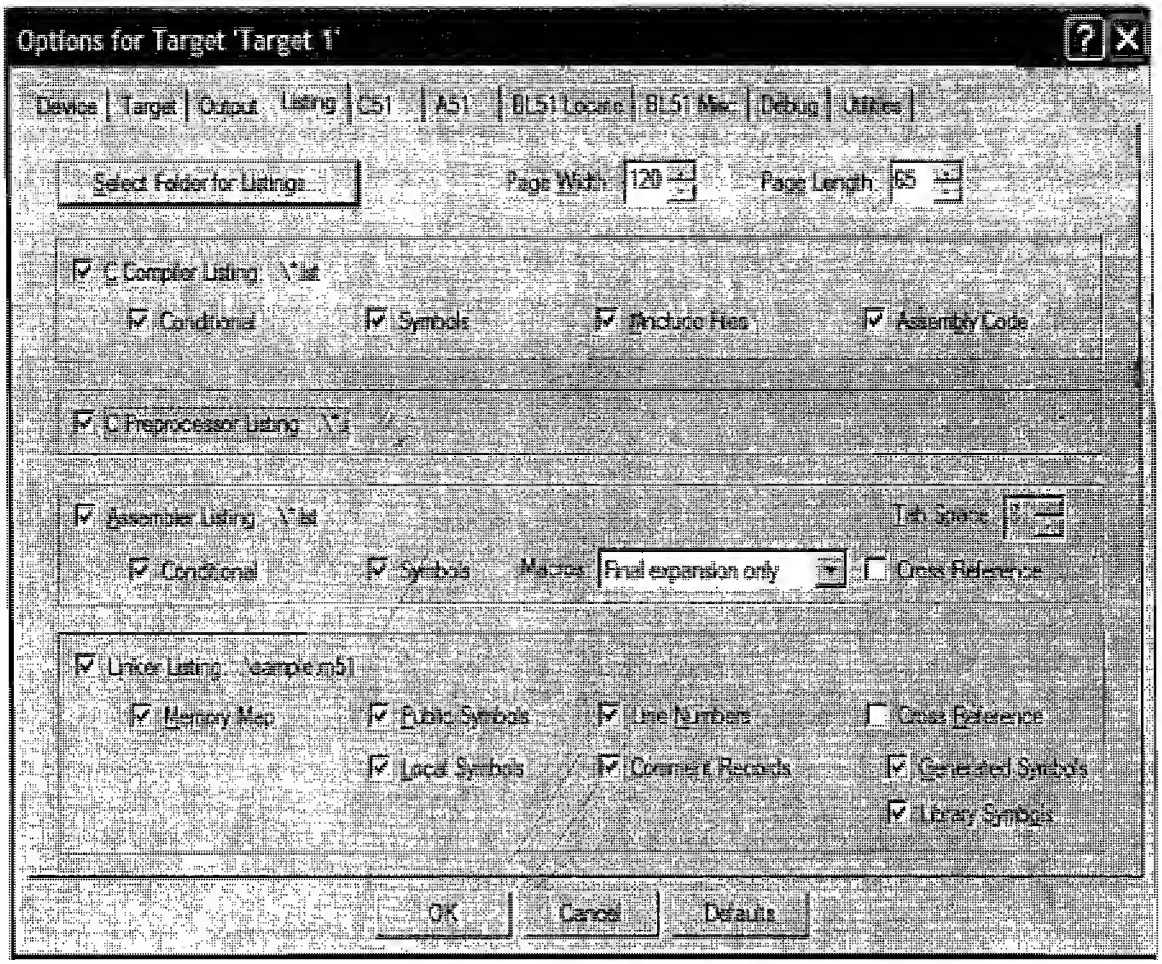
OK Cancel Defaults Help

Fig. 13.10 **Output File creation settings**

You can select one of the output settings (viz. executable binary file (hex) or library file (lib)). For executable file, tick the 'Create Hex File' option and select the target processor specific hex file format. Depending on the target processor architecture the hex file format may vary, e.g. Intel Hex file and Motorola hex file. For *8051*, only one choice is available and it is Intel hex File HEX-80. The list files section coming under the tab '*Listing*' tells what all listing files should be created during cross-compilation process (Fig. 13.11).

'*C51*' tab settings are used for cross compiler directives and settings like/Pre-processor symbols, code optimisation settings, type of optimisation (viz. code optimised for execution speed and code optimised for size), *include* file's path settings, etc. The '*A51*' tab settings are used for assembler direc-

https://hemanthrajhemu.github.io



Page Width: 120 Page Length: 65

☑ C Compiler Listing: .*.lst

☑ Conditional ☑ Symbols ☑ #include Files ☑ Assembly Code

☑ C Preprocessor Listing: .*.i

☑ Assembler Listing: .*.lst Tab Space: [illegible]

☑ Conditional ☑ Symbols Macros: Final expansion only ☐ Cross Reference

☑ Linker Listing: .\example.m51

☑ Memory Map ☑ Public Symbols ☑ Line Numbers ☐ Cross Reference

☑ Local Symbols ☑ Comment Records ☑ Generated Symbols

☑ Library Symbols

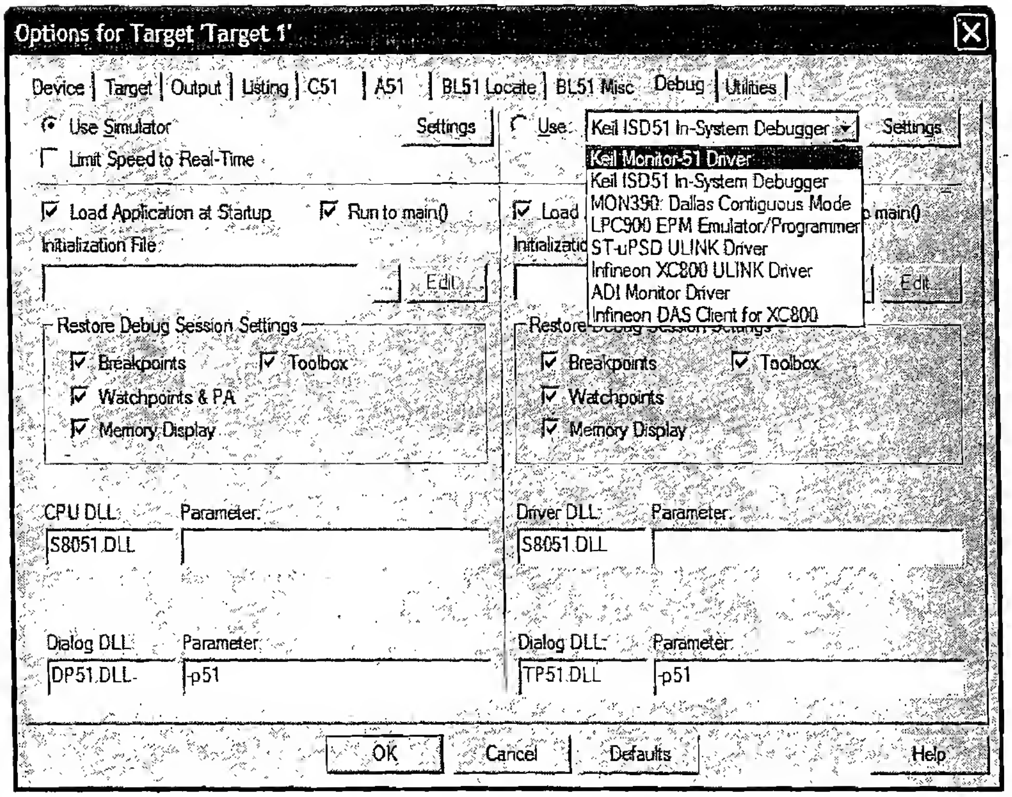
Fig. 13.11 **List File generation settings**

tives and settings like conditional assembly control symbols, *include* file's path settings etc. Another important option is '*Debug*'. The '*Debug*' tab is used for configuring the firmware debugging. '*Debug*' supports both simulation type firmware debugging and debugging the application while it is running on the target hardware (Fig. 13.12).

You can either select the Simulator based firmware debugging or a target firmware level debugging from the '*Debug*' option. If the Simulator is selected, the firmware need not be downloaded into the target machine. The IDE provides an application firmware debugging environment by simulating the target hardware in a software environment. It is most suitable for offline analysis and rapid code developments. If target level debugging is selected, the binary file created by the cross-compilation process needs to be downloaded into the target hardware and the debugging is done by single stepping the firmware. A physical link should be established between the target hardware and the PC on which the IDE is running for target level debugging. Target level hardware debugging is achieved using the Keil supported monitor programs or through an emulator interface. Select the same from the Drop-down list. Normally the link between target hardware and IDE is established through a Serial interface. Use the settings tab to configure the Serial interface. Select the '*Comm Port*' to which the target device is connected and the baudrate for communication (Fig. 13.13).

If the Debug mode is configured to use the Target level debugging using any one of the monitor program or the emulator interface supported by the Keil IDE, the created binary file is downloaded into

https://hemanthrajhemu.github.io



Options for Target 'Target 1'

Device | Target | Output | Listing | C51 | A51 | BL51 Locate | BL51 Misc | Debug | Utilities

● Use Simulator   Settings

□ Limit Speed to Real-Time

☑ Load Application at Startup   ☑ Run to main()

Initialization File:

Edit...

Restore Debug Session Settings

☑ Breakpoints   ☑ Toolbox

☑ Watchpoints & PA

☑ Memory Display

CPU DLL: S8051.DLL   Parameter:

Dialog DLL: DP51.DLL   Parameter: -p51

○ Use: Keil ISD51 In-System Debugger   Settings

- Keil Monitor-51 Driver
- Keil ISD51 In-System Debugger
- MON390: Dallas Contiguous Mode
- LPC900 EPM Emulator/Programmer
- ST-uPSD ULINK Driver
- Infineon XC800 ULINK Driver
- ADI Monitor Driver
- Infineon DAS Client for XC800

☑ Load   Run to main()

Initializatio

Edit...

Restore Debug Session Settings

☑ Breakpoints   ☑ Toolbox

☑ Watchpoints

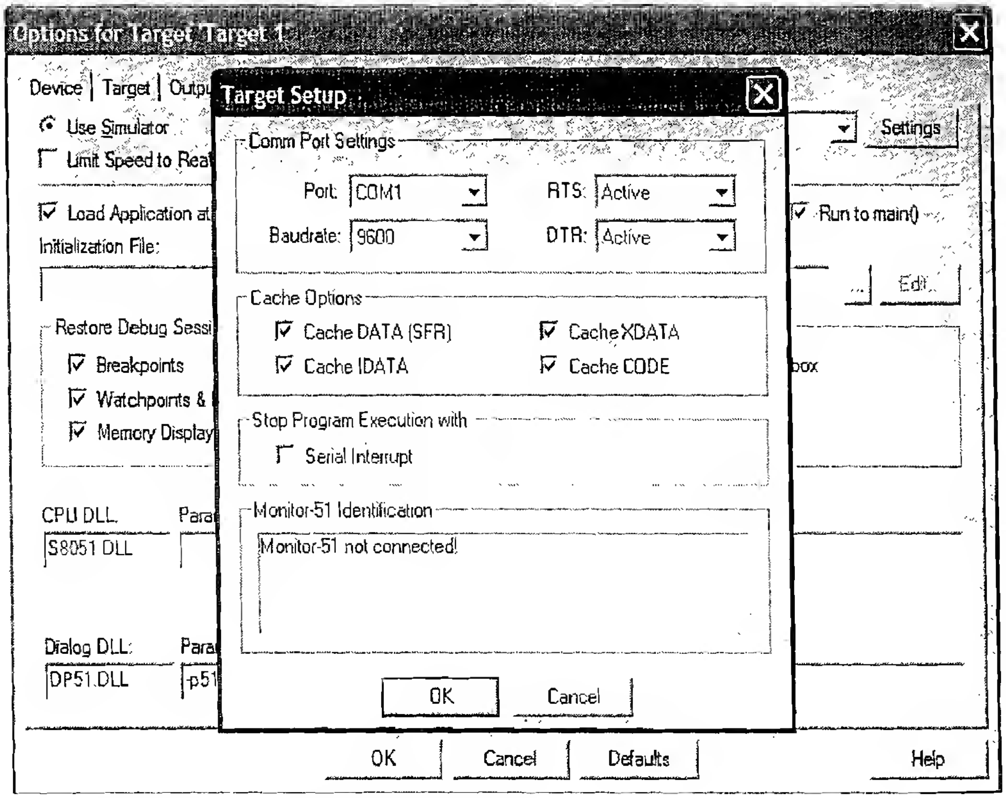
☑ Memory Display

Driver DLL: S8051.DLL   Parameter:

Dialog DLL: TP51.DLL   Parameter: -p51

OK   Cancel   Defaults   Help

Fig. 13.12 **Firmware debugging options**



Options for Target 'Target 1'

Device | Target | Outpu

● Use Simulator

□ Limit Speed to Real

☑ Load Application at   ☑ Run to main()

Initialization File:

Restore Debug Sessi

☑ Breakpoints

☑ Watchpoints &

☑ Memory Display

CPU DLL: S8051.DLL   Para

Dialog DLL: DP51.DLL   Para -p51

Target Setup

Comm Port Settings

Port: COM1   RTS: Active

Baudrate: 9600   DTR: Active

Cache Options

☑ Cache DATA (SFR)   ☑ Cache XDATA

☑ Cache IDATA   ☑ Cache CODE

Stop Program Execution with

□ Serial Interrupt

Monitor-51 Identification

Monitor-51 not connected!

OK   Cancel

Settings   Edit...   box

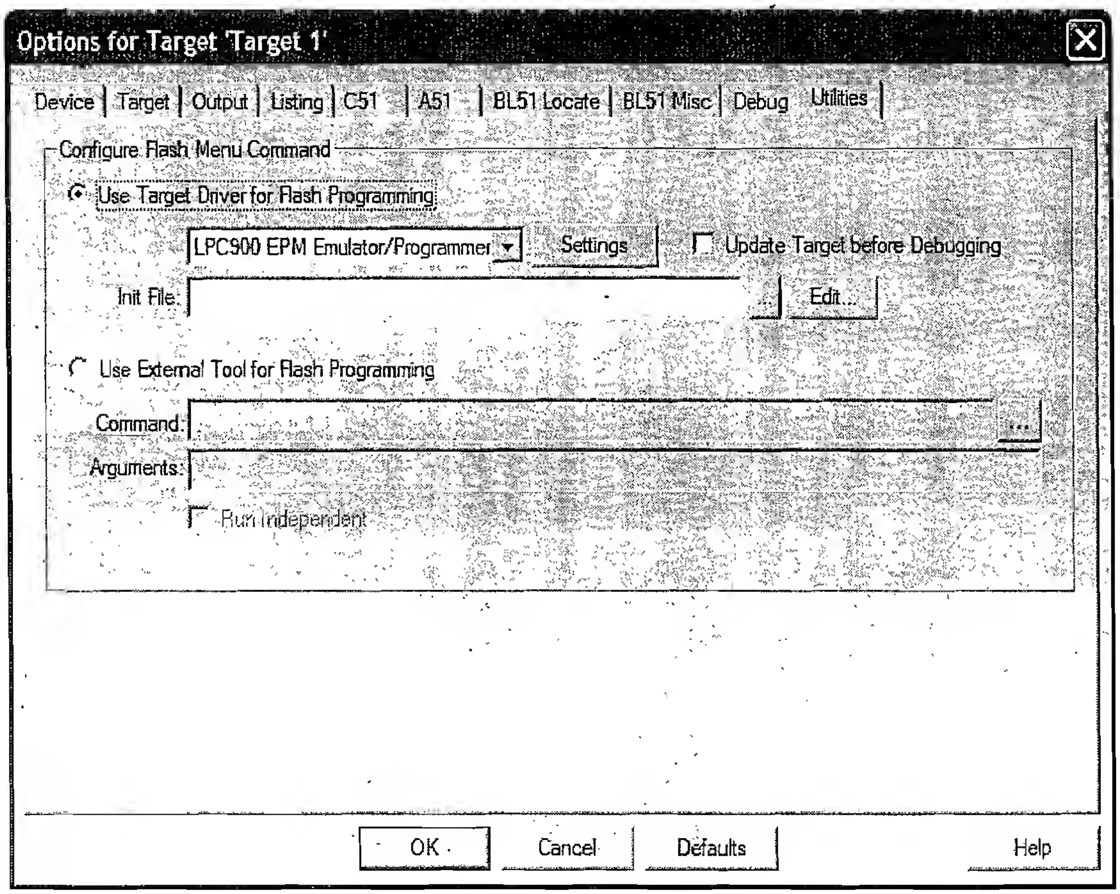
OK   Cancel   Defaults   Help

Fig. 13.13 **Target hardware debug serial link configuration**

https://hemanthrajhemu.github.io

the target board using the configured serial connection and the firmware execution occurs in real time. The firmware is single stepped (Executing instruction-by-instruction) within the target processor and the monitor program running on the target device reflects the various register and memory contents in the IDE using the serial interface.

The '*Utilities*' tab is used for configuring the flash memory programming of the target processor/controllers from the Keil IDE (Fig. 13.14). You can use either Keil IDE supported programming drivers or a third party tool for programming the target system's FLASH memory. For making use of Keil IDE provided flash memory programming drivers, select the option '*Use Target Driver for Flash Programming*' and choose a driver from the drop-down list. To use third party programming tools, select the option '*Use External Tool for Flash Programming*' and specify the third party tool to be used by giving the path in the '*Command*' column and specify the arguments (if any) in the '*Arguments*' tab to invoke the third party application.



Configure Flash Menu Command

● Use Target Driver for Flash Programming

LPC900 EPM Emulator/Programmer

□ Update Target before Debugging

Init File:

○ Use External Tool for Flash Programming

Command:

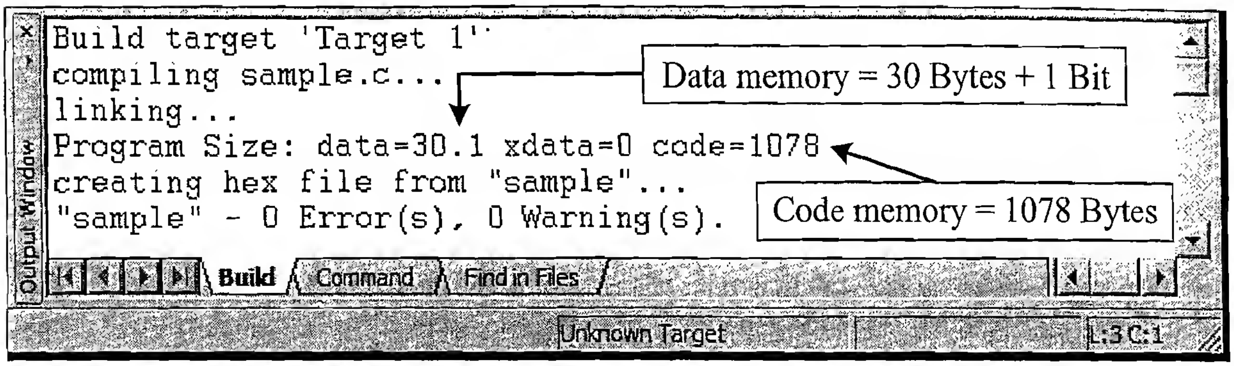
Arguments:

□ Run Independent

Fig. 13.14 **Target Flash Memory Programming configuration**

With this we are done with the writing of our first simple Embedded C program and configuring the target controller for running it. The next step is the conversion of the firmware written in Embedded C to machine language corresponding to the target processor/controller. This step is called cross-compilation. Go to '*Project*' tab in the menu and select '*Rebuild all target files*'. This cross-compiles all the files within the target group (for modular programs there may be multiple source files) and link the object codes created by each file to generate the final binary. The output of cross-compilation for the "*Hello World*" application is given in Fig. 13.15.

https://hemanthrajhemu.github.io



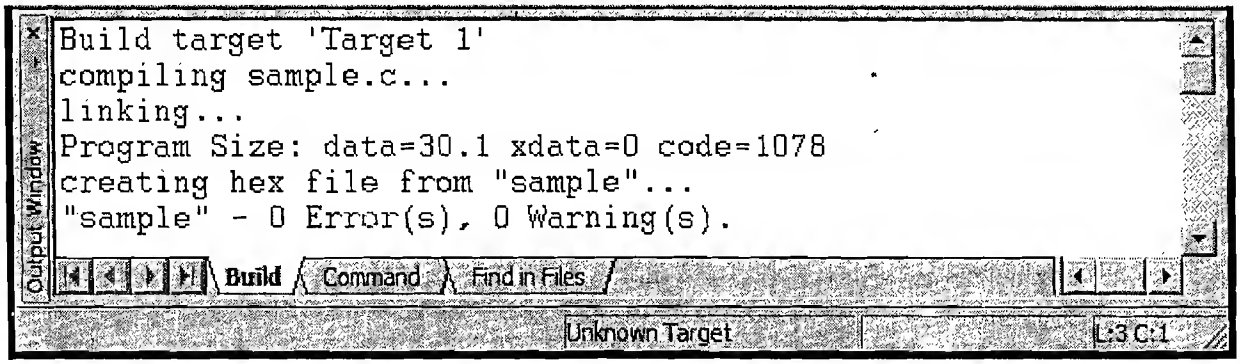
```
Build target 'Target 1'
compiling sample.c...
linking...
Program Size: data=30.1 xdata=0 code=1078
creating hex file from "sample"...
"sample" - 0 Error(s), 0 Warning(s).
```

Data memory = 30 Bytes + 1 Bit

Code memory = 1078 Bytes

Fig. 13.15 **Conversion of the Embedded C program to *8051* Machine code**

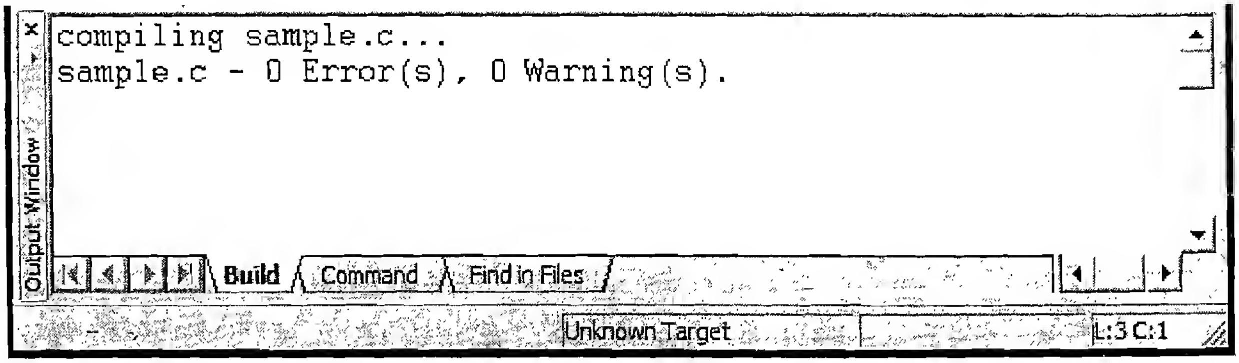
You can see the cross-compilation step & linking in the o/p window along with cross-compilation error history. Now perform a '*Build Target*' operation. This links all the object files created (in a multi-file system where each source files are cross-compiled separately) (Fig. 13.16).



```
Build target 'Target 1'
compiling sample.c...
linking...
Program Size: data=30.1 xdata=0 code=1078
creating hex file from "sample"...
"sample" - 0 Error(s), 0 Warning(s).
```

Fig. 13.16 **Linking of all object Files**

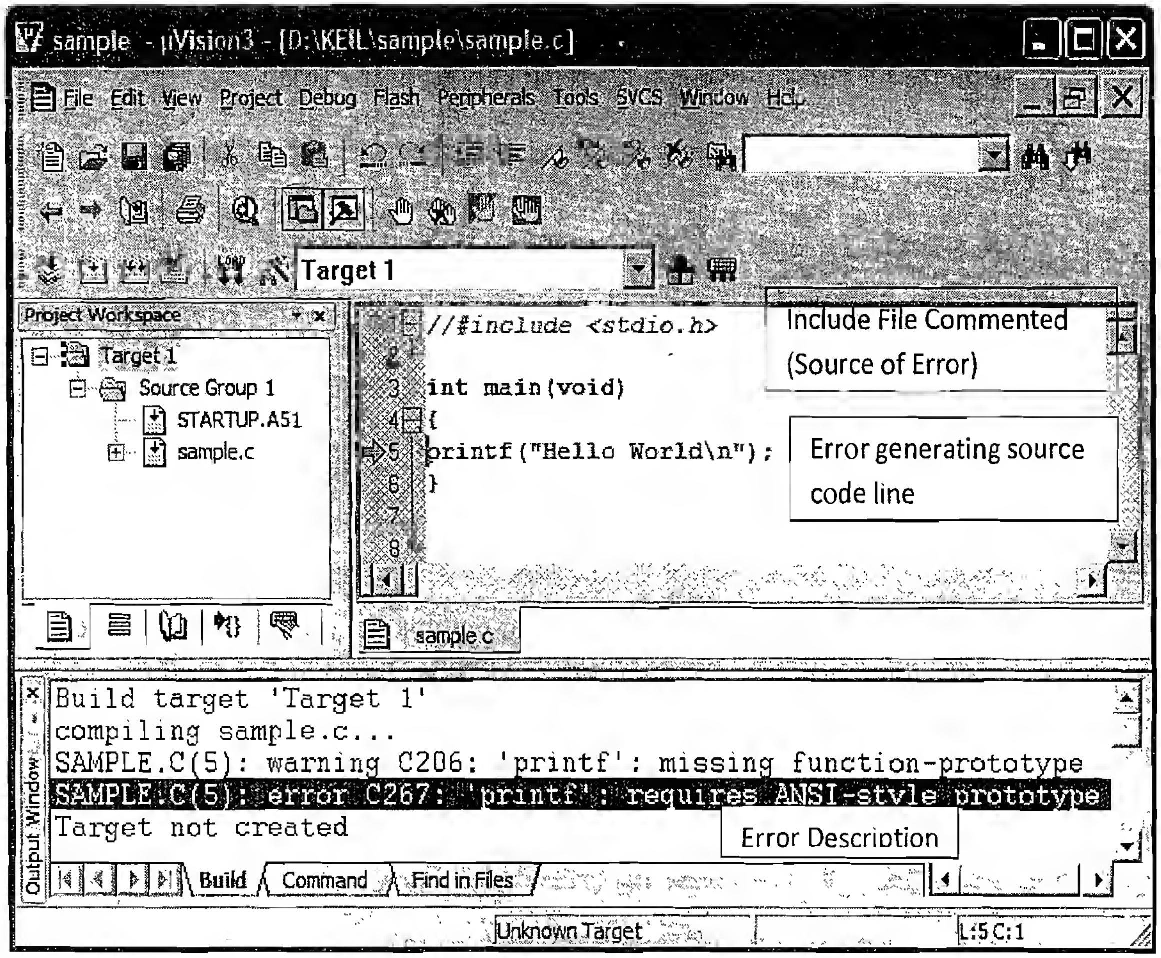
In a multi source file project (source group containing multiple .c files) each file can be separately cross-compiled by selecting the '*Translate current file*' option. This is known as Selective Compilation. Remember this generⁿtes the object file for the current file only and it needs to be combined with object files generated for other files, by using '*Build Target*' option for creating the final executable (Fig. 13.17). Selective compilation is very helpful in a multifile project where a modified file only needs to be re-compiled and it saves the time in re-compiling all the files present in the target group.



```
compiling sample.c...
sample.c - 0 Error(s), 0 Warning(s).
```

https://hemanthrajhemu.github.io

See the difference between the three; selective compilation cross-compiles a selected source file and creates a re-locatable object file corresponding to it, whereas '*Build Target*' performs the linking of different re-locatable object files and generates an absolute object file and finally creates a binary file. '*Rebuild all target file*' creates re-locatable object files by cross-compiling all source files and links all object files to generate the absolute object file and finally generates the binary file. If there is any error in the compilation, it is displayed on the output window along with the line number of code where the error is occurred. So it is easy to trace the code to find out the error part in the code. The error is listed out in the output window along with line number and error description. On clicking the error description at the output window, the line of code generated the error is highlighted with a bold arrow. One example of error code is given below. In the "Hello World" example, the include file is commented and the code while compiling will generate the error indicating *printf* function is not defined. Have a look at the same (Fig. 13.18).



```
//#include <stdio.h>

int main(void)
{
printf("Hello World\n");
}
```

Include File Commented (Source of Error)

Error generating source code line

```
Build target 'Target 1'
compiling sample.c...
SAMPLE.C(5): warning C206: 'printf': missing function-prototype
SAMPLE.C(5): error C267: 'printf': requires ANSI-style prototype
Target not created
```

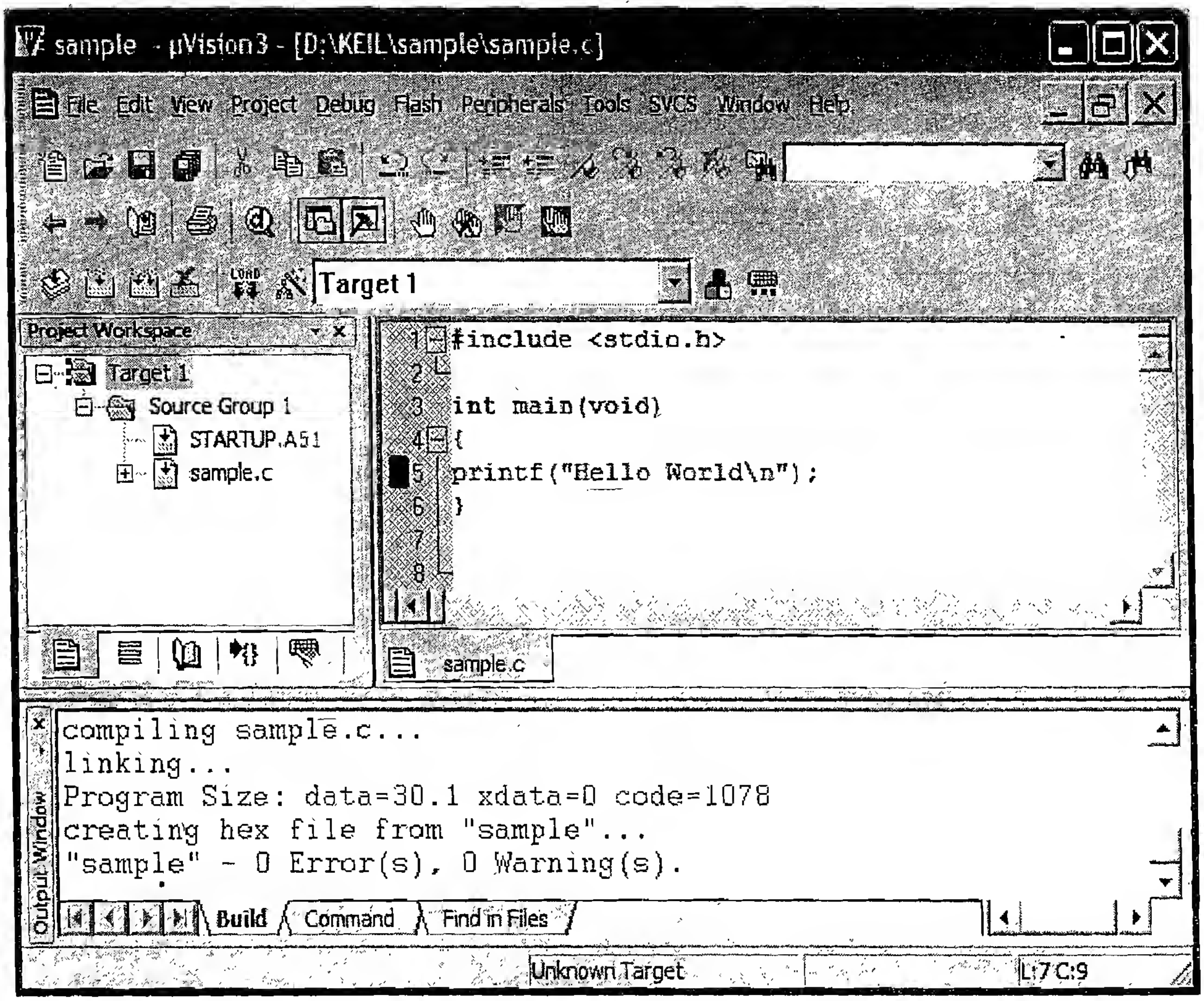
Error Description

Fig. 13.18 **Compilation Error Example**

13.1.1.1 Debugging with Keil µVision3 IDE Debugging firmware is the process of monitoring the program flow, various registers and memory contents while the firmware is executed. You can debug the firmware in two methods. The first method is by using breakpoints and simulator (a software tool which simulates the functionalities of the target processor). The second method is hardware level debugging.

https://hemanthrajhemu.github.io

For simulator-based debugging, select the option '*Use Simulator*' from the '*Debug*' tab of the '*Options for Target*' as illustrated in a previous section on debug. Insert a '*Breakpoint*' in the code line of the source program where debugging needs to be started. Breakpoints can be inserted by right clicking on the desired source code line and by selecting the '*Insert/Remove Breakpoint*' option. It can also be inserted by using the '*Debug*' tab present in the menu of the IDE (Not the debug tab of Options for Target Pop-up dialog). It toggles the breakpoint (if the breakpoint is already inserted, it is removed and if not a breakpoint is added). The breakpoint breaks the firmware execution at the selected point and further execution requires user interaction. After a break, the code can be executed by single stepping or by a complete run again. For the above 'Hello World' example, we are going to debug the firmware at the source code line *printf* and a breakpoint is inserted as shown in Fig. 13.19.



```
#include <stdio.h>

int main(void)
{
printf("Hello World\n");
}
```

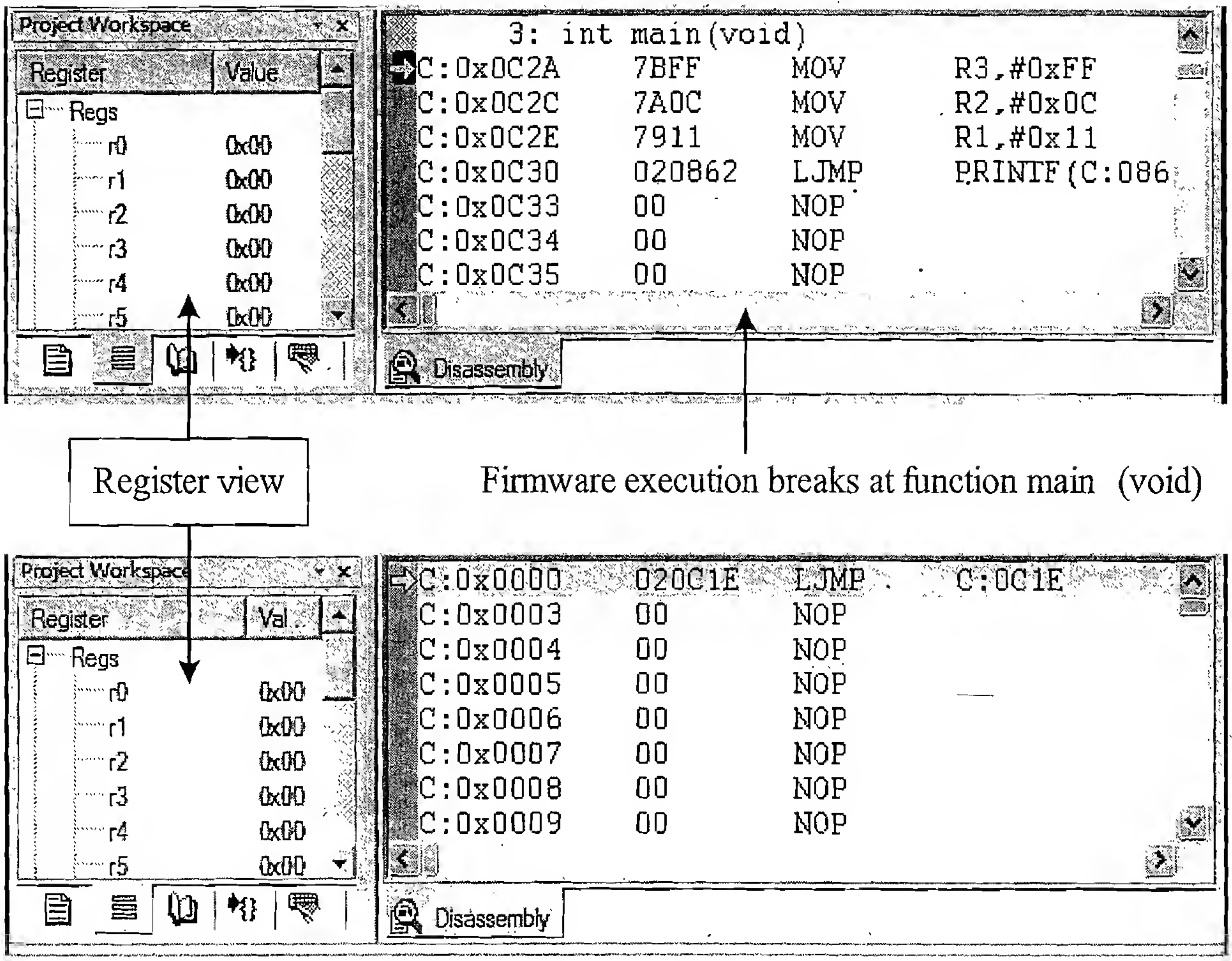
```
compiling sample.c...
linking...
Program Size: data=30.1 xdata=0 code=1078
creating hex file from "sample"...
"sample" - 0 Error(s), 0 Warning(s).
```

Fig. 13.19 **Breakpoint insertion and debugging**

The '*breakpoint*' is distinguishable with a special visible mark. All debug related actions are grouped within the '*Debug*' tab of the IDE menu. Each debug instruction has a hot key (e.g. Ctrl+F5 for start and stop of debugging) associated with it. Identify the hotkey and use it or use mouse to activate the debug related action each time from the debug menu. To start debugging, select the option '*Start/Stop Debug Session (Ctrl+F5)*' from the '*Debug*' menu. If you set the option '*Go till main()*' on the '*Debug*' tab of

https://hemanthrajhemu.github.io

'*Options for Target*', the application runs till the code memory where the main() function starts. If this option is not selected, the execution breaks at the Reset vector (0x0000) itself. Both of these options are shown below. Note that while debugging, the project window tab automatically selects the '*Regs*' tab and shows the various register contents in this window when the program is at the break stage. You can switch the project window in between the '*Files*' section and '*Regs*' section to find out the changes happening to various registers on executing each line of code (Fig. 13.20).



Project Workspace

Register	Value
Regs	
r0	0x00
r1	0x00
r2	0x00
r3	0x00
r4	0x00
r5	0x00

```
     3: int main(void)
C:0x0C2A    7BFF      MOV      R3,#0xFF
C:0x0C2C    7A0C      MOV      R2,#0x0C
C:0x0C2E    7911      MOV      R1,#0x11
C:0x0C30    020862    LJMP     PRINTF(C:086
C:0x0C33    00        NOP
C:0x0C34    00        NOP
C:0x0C35    00        NOP
```

Disassembly

Register view

Firmware execution breaks at function main (void)

Project Workspace

Register	Val.
Regs	
r0	0x00
r1	0x00
r2	0x00
r3	0x00
r4	0x00
r5	0x00

```
C:0x0000    020C1E    LJMP     C:0C1E
C:0x0003    00        NOP
C:0x0004    00        NOP
C:0x0005    00        NOP
C:0x0006    00        NOP
C:0x0007    00        NOP
C:0x0008    00        NOP
C:0x0009    00        NOP
```

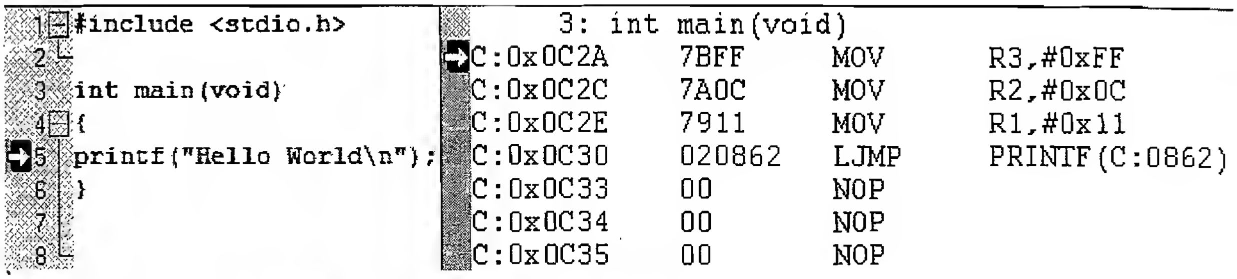
Disassembly

Firmware execution breaks at Reset vector (0x0000)

Fig. 13.20 **Firmware execution Break options**

If you observe these two figures you can see that code memory execution starts at location 0x0000 and the firmware corresponding to the function main is located at 0x0C2A and it is not the first executable code. Before executing *main ()* some other code placed at location 0x0C1E (jump from the reset vector to location 0x0C1E) is executed. The code at this memory location is the code memory generated for the startup file. Startup file code is always executed before entering the function *main*. It should be noted that the code memory location mentioned here for function *main* is not always fixed. It varies depending on the changes made to the startup file. From this breakpoint you can go to the breakpoint you set in the code memory either by single stepping ('*Step*' command (*F11*)) or by a run to the next breakpoint ('*Go*' command (*F5*)).

https://hemanthrajhemu.github.io

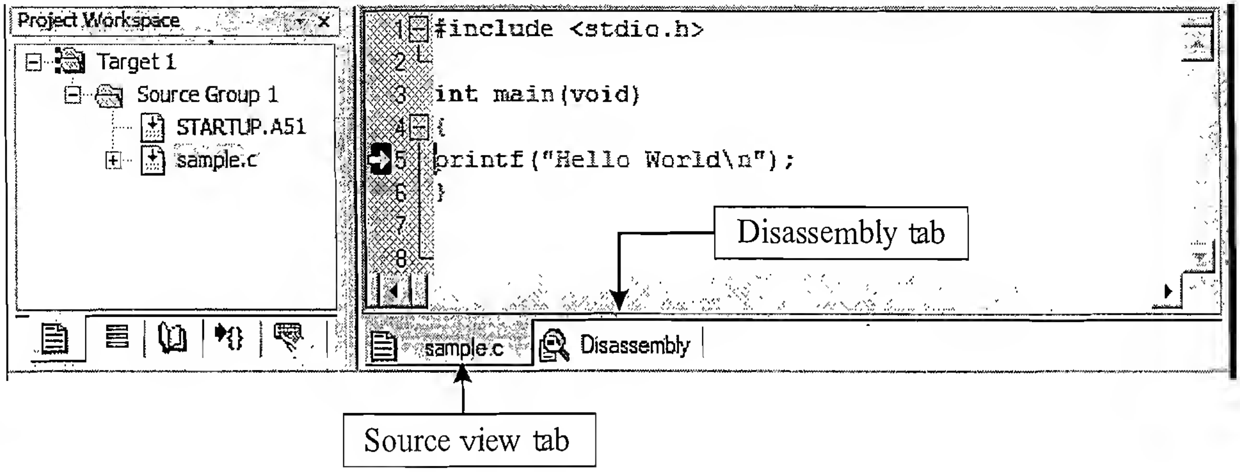


```
#include <stdio.h>

int main(void)
{
printf("Hello World\n");
}
```

```
        3: int main(void)
C:0x0C2A    7BFF     MOV     R3,#0xFF
C:0x0C2C    7A0C     MOV     R2,#0x0C
C:0x0C2E    7911     MOV     R1,#0x11
C:0x0C30    020862   LJMP    PRINTF(C:0862)
C:0x0C33    00       NOP
C:0x0C34    00       NOP
C:0x0C35    00       NOP
```

Fig. 13.21 **Source Code and corresponding Disassembly View**

By default, the text editor window in debug mode contains two tabs, namely Source code view tab and Disassembly tab. The Source code and corresponding Disassembly view is shown in Fig. 13.21. While debugging the firmware you can switch the view between Assembly code and original source code lines by selecting the corresponding tab. This switches the view between the original source code and the corresponding Assembly code for it. The Disassembly view disappears temporarily if you click the '*Disassembly Window*' option under the '*View*' tab of the IDE menu. To enable Disassembly View click again on the '*Disassembly Window*' option under the '*View*' tab of the IDE menu (Fig. 13.22).



```
#include <stdio.h>

int main(void)
{
printf("Hello World\n");
}
```

Disassembly tab

Source view tab

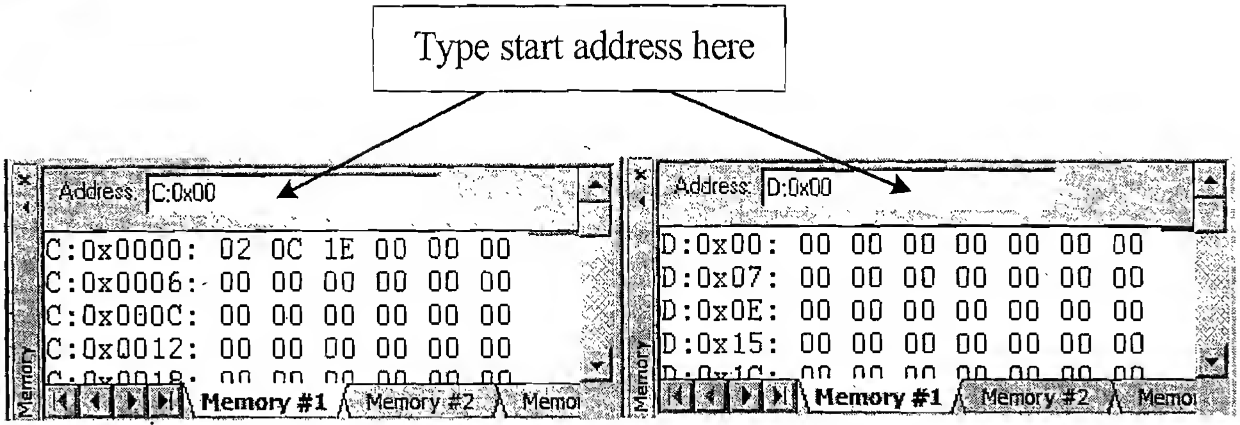
Fig. 13.22 **Workbook Mode for Source – Disassembly code View**

During debugging the content of various CPU and general purpose registers are displayed under the '*Regs*' tab of the project Window. Apart from this you can inspect the data memory and code memory by invoking the '*Memory Window*' under the '*View*' tab of the IDE menu (Fig. 13.23).

For viewing the Data memory, use the Prefix 'D:' before the address and type it at the Address edit box and press enter (For example, for viewing the data memory starting from 0x00, type D:0x00 at the Address box and press enter). You can edit the content of the desired data memory by right clicking on the current contents pointed by the address or just by a double click on the current content. Code memory can also be inspected and modified in a similar way to the Data memory. The only difference is instead of D: use 'C:' (D stands for Data and C stands for Code) (Fig. 13.24).

Similar to other Desktop application development IDEs, this also provides option for viewing local variables and call stack details. Invoke '*Watch & Call Stack Window*' from the '*View*' tab of the IDE menu (Fig. 13.25).

https://hemanthrajhemu.github.io



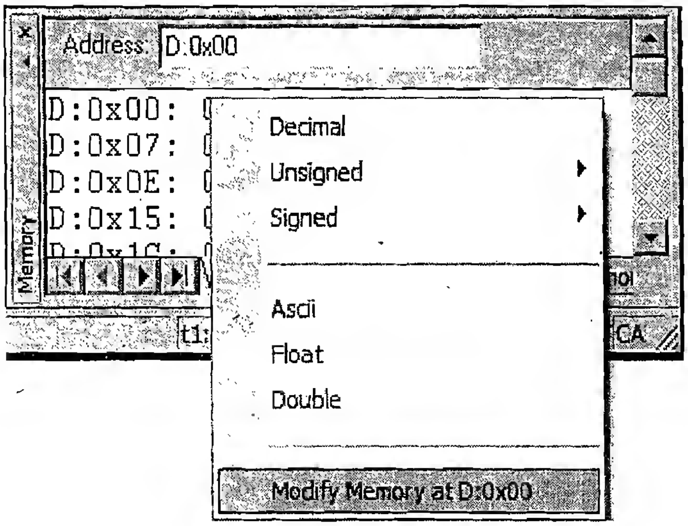
Address: C:0x00

```
C:0x0000: 02 0C 1E 00 00 00
C:0x0006: 00 00 00 00 00 00
C:0x000C: 00 00 00 00 00 00
C:0x0012: 00 00 00 00 00 00
C:0x0018: 00 00 00 00 00 00
```

Address: D:0x00

```
D:0x00: 00 00 00 00 00 00 00
D:0x07: 00 00 00 00 00 00 00
D:0x0E: 00 00 00 00 00 00 00
D:0x15: 00 00 00 00 00 00 00
D:0x1C: 00 00 00 00 00 00 00
```

Fig. 13.23 **Memory Window for memory inspection in firmware debugging**



Address: D:0x00

```
D:0x00:
D:0x07:
D:0x0E:
D:0x15:
D:0x1C:
```

Fig. 13.24 **Memory modification while debugging**

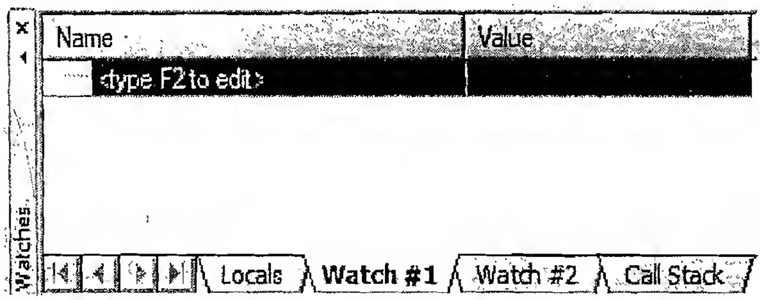
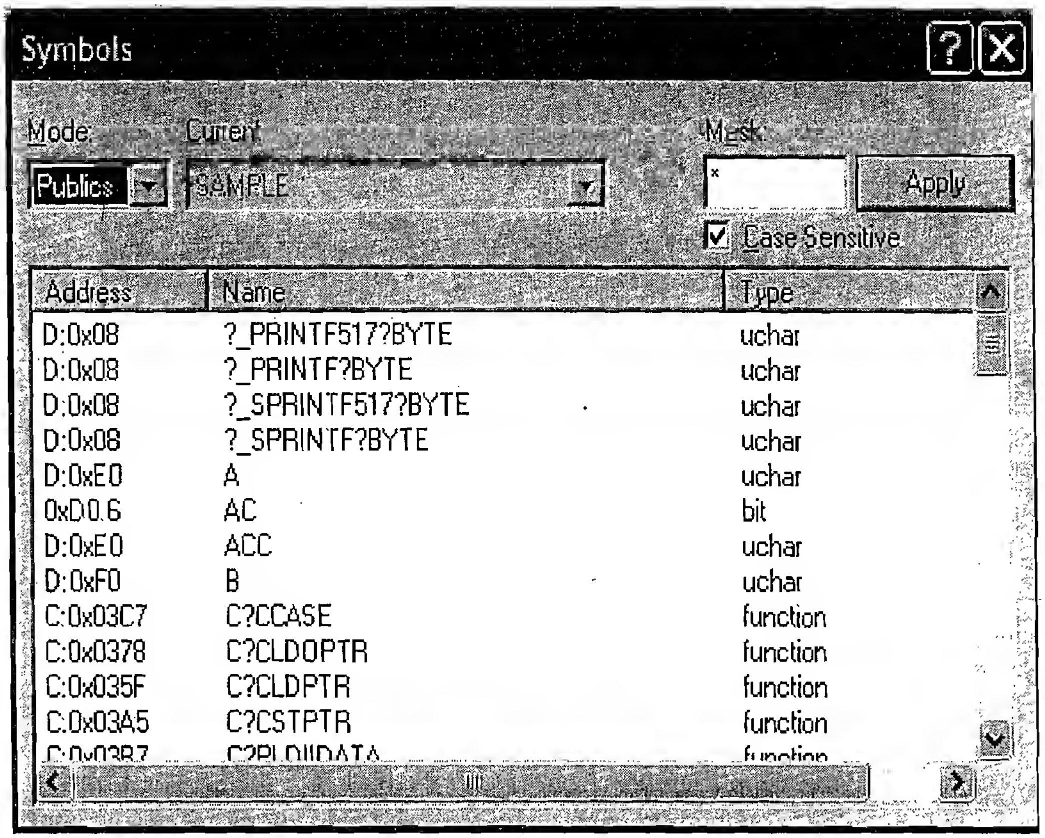


Fig. 13.25 **Watch and Call Stack Window for Debugging**

The local variables can be viewed in the '*Locals*' section of '*Watch and call Stack Window*' whereas users can add other variables to the '*Watch*' window to get their values. '*Call Stack*' gives the '*Callee-Caller*' details for subroutine calls. Invoking '*Symbol Window*' under the '*View*' tab of IDE menu while debugging, displays the list of '*Publics*', '*Locals*' and '*Lines*' used by the application, in the disassembly

https://hemanthrajhemu.github.io

window (Assembly code). Selecting '*Publics*' shows the different variables and functions present in the Assembly code along with their 'Address', 'Name' and 'Type'. Address can either be Data memory address or Code memory address. 'Type' indicates whether it is a variable or a function. The different variables supported are 'unsigned char (*uchar*)', 'signed char (*char*)', *bit*, unsigned integer (*uint*), signed int (*int*), etc. Figure 13.26 shows the Symbols Window displaying the various symbols used in Assembly for our sample application.



Mode: Publics

Current: SAMPLE

Mask: *

☑ Case Sensitive

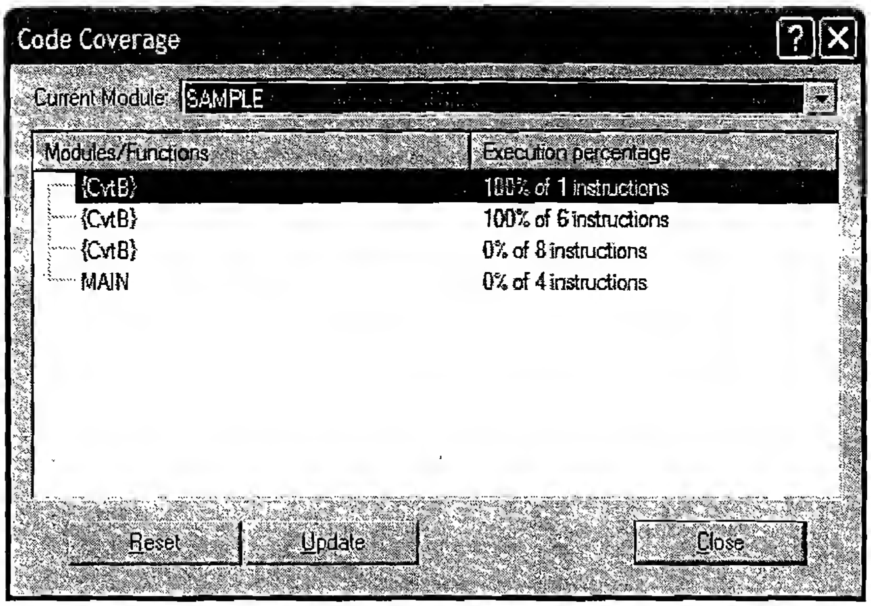
Address	Name	Type
D:0x08	?_PRINTF517?BYTE	uchar
D:0x08	?_PRINTF?BYTE	uchar
D:0x08	?_SPRINTF517?BYTE	uchar
D:0x08	?_SPRINTF?BYTE	uchar
D:0xE0	A	uchar
0xD0.6	AC	bit
D:0xE0	ACC	uchar
D:0xF0	B	uchar
C:0x03C7	C?CCASE	function
C:0x0378	C?CLDOPTR	function
C:0x035F	C?CLDPTR	function
C:0x03A5	C?CSTPTR	function
C:0x03B7	C?PLDIIDATA	function

Fig. 13.26 **Symbols Window showing the various symbols used in Assembly**

Activating the '*Code Coverage*' option under the '*View*' tab of the IDE menu when the execution is at halted stage, gives the details of instructions in each function/module and how many of them are executed till execution break. Figure 13.27 illustrates the same.

13.1.1.2 Simulating Peripherals and Interrupts with Keil µVision3 IDE Embedded systems are designed to interact with real world and the actions performed by them may depend on the inputs from various sensors connected to the processor/controller of the embedded system. By a mere software simulation of the firmware, we can only inspect the memory, register contents, etc. of the processor and cannot infer anything on the real-time performance of the system since the inputs provided by the sensors are real-time and dynamic. To a certain extent, we can simulate these inputs using the simulator support provided by the IDE. Keil provides extensive support for simulating various peripherals like; ports, memory mapped I/O, etc. and Interrupts (External, Timer and Serial interrupts). The main limitation of simulation is that we can simulate it with only known values (in a real application the simulated value may not be the real input) and also it is difficult to predict the real-time behaviour of the system

https://hemanthrajhemu.github.io



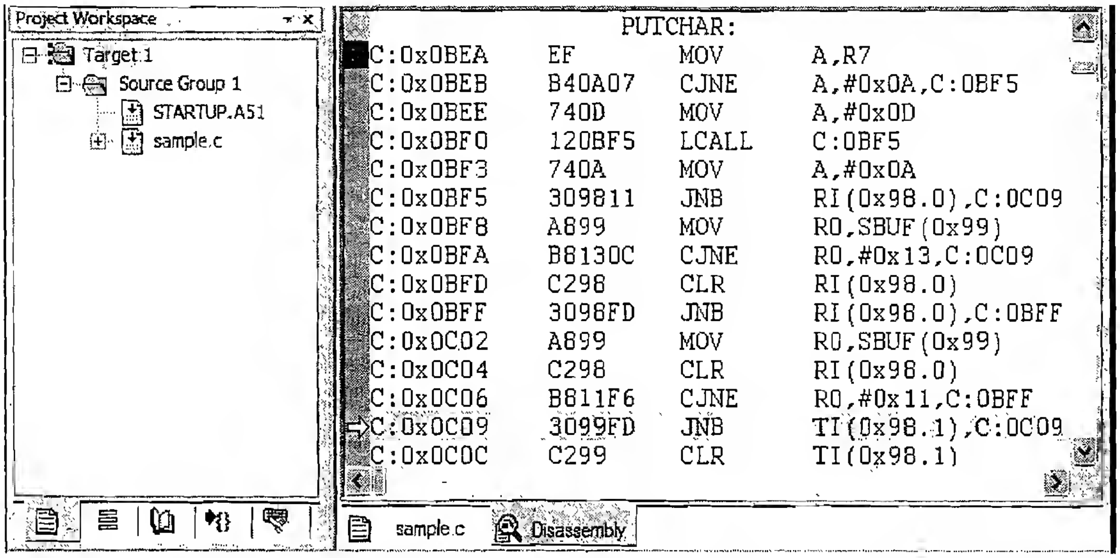
Code Coverage

Current Module: SAMPLE

Modules/Functions	Execution percentage
{CvtB}	100% of 1 instructions
{CvtB}	100% of 6 instructions
{CvtB}	0% of 8 instructions
MAIN	0% of 4 instructions

Fig. 13.27 **Code Coverage Details at the execution break**

based on these simulations. In our "*Hello World*" application, if we debug the application by placing a breakpoint at the '*printf*' function, in the source code window you can observe that the '*printf*' function is not getting completed on single stepping and it gives you the feeling that the firmware is hang up somewhere. If you are single stepping (using *F11*) the firmware from the beginning, in the disassembly window you can see that for the code corresponding to *printf*, the application is looping inside another function called 'putchar', which checks a bit TI (Transmit Interrupt) and loops till it is asserted high (Fig. 13.28). As we mentioned earlier, the *printf* function is outputting the data to the Serial port. As long as the Transmit Interrupt is not asserted, the firmware control is looped there and it generates an application hang-up behaviour. If you are not able to view the point where the firmware loops in the Disassembly window, invoke '*Stop Running*' from '*Debug*' tab of IDE menu. The firmware execution will be stopped and in the disassembly window you can see the exact point where the application was looping.

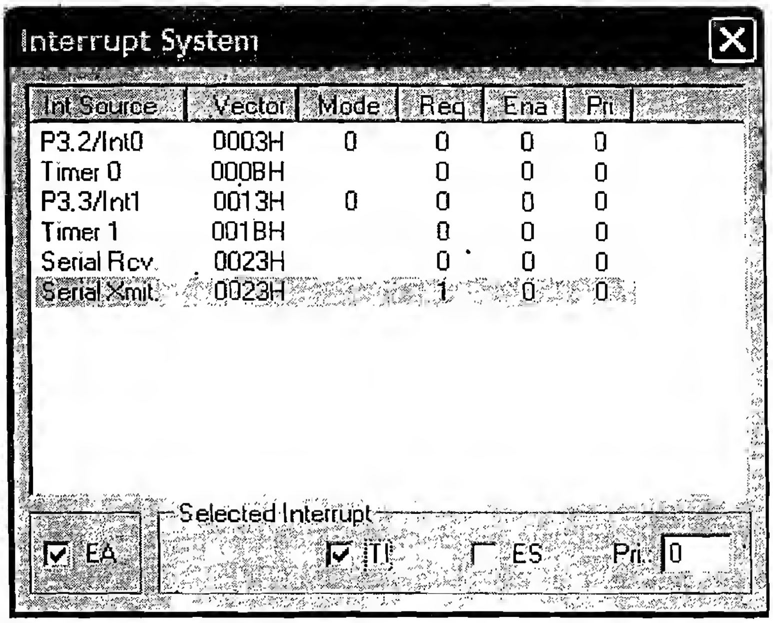


```
                    PUTCHAR:
C:0x0BEA    EF      MOV     A,R7
C:0x0BEB    B40A07  CJNE    A,#0x0A,C:0BF5
C:0x0BEE    740D    MOV     A,#0x0D
C:0x0BF0    120BF5  LCALL   C:0BF5
C:0x0BF3    740A    MOV     A,#0x0A
C:0x0BF5    309811  JNB     RI(0x98.0),C:0C09
C:0x0BF8    A899    MOV     R0,SBUF(0x99)
C:0x0BFA    B8130C  CJNE    R0,#0x13,C:0C09
C:0x0BFD    C298    CLR     RI(0x98.0)
C:0x0BFF    3098FD  JNB     RI(0x98.0),C:0BFF
C:0x0C02    A899    MOV     R0,SBUF(0x99)
C:0x0C04    C298    CLR     RI(0x98.0)
C:0x0C06    B811F6  CJNE    R0,#0x11,C:0BFF
C:0x0C09    3099FD  JNB     TI(0x98.1),C:0C09
C:0x0C0C    C299    CLR     TI(0x98.1)
```

Fig. 13.28 **Firmware Execution waiting for Transmit Interrupt (TI)**

https://hemanthrajhemu.github.io

We can simulate the Transmit Interrupt '*TI*' using the simulator support provided by Keil. If you have already stopped the execution, continue the code execution by invoking '*Go*' option from '*Debug*' tab or by pressing '*F5*' key. Now select '*Interrupt*' option from the '*Peripherals*' tab of the IDE menu. The interrupt simulation window will pop-up. Select the '*Serial Xmit*' interrupt and enable the Global Interrupt Enable (*EA*) as well as Transmit Interrupt (*TI*) (Fig. 13.29).



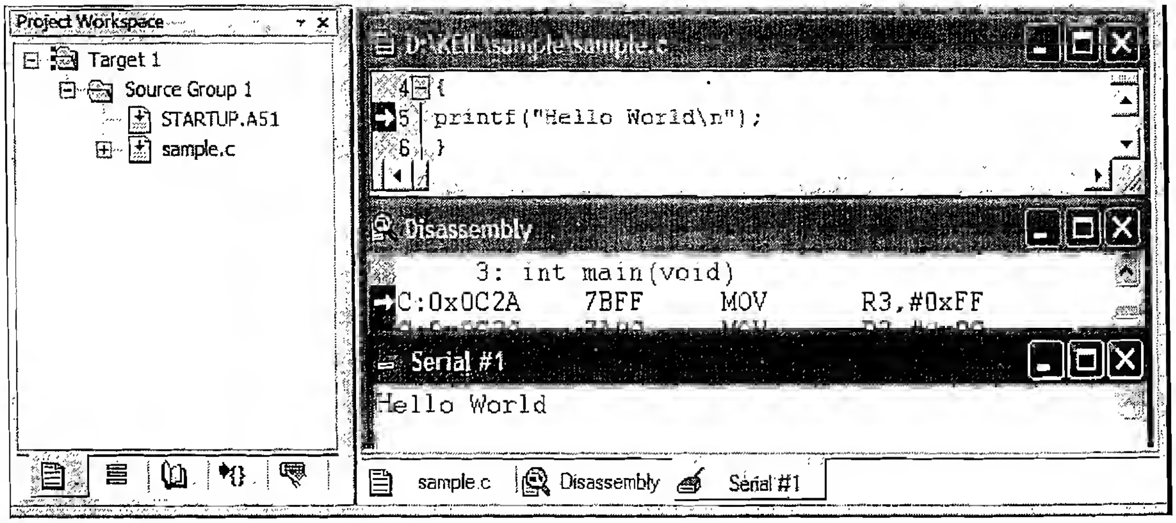
Int Source	Vector	Mode	Req	Ena	Pri
P3.2/Int0	0003H	0	0	0	0
Timer 0	000BH		0	0	0
P3.3/Int1	0013H	0	0	0	0
Timer 1	001BH		0	0	0
Serial Rcv.	0023H		0	0	0
Serial Xmit.	0023H		1	0	0

☑ EA

Selected Interrupt: ☑ TI □ ES Pri: 0

Fig. 13.29 **Serial Transmit Interrupt Simulation**

You can simulate any other interrupt listed in the interrupt system in a similar fashion. On enabling TI, firmware execution comes out of the infinite loop and dumps the string "Hello World" to the serial port. The output of 'Hello World' application is shown in Fig. 13.30. You can capture whatever data going through the serial port while simulation. For this invoke '*Serial Window #1*' from the '*View*' tab of the IDE menu.



```
{
printf("Hello World\n");
}
```

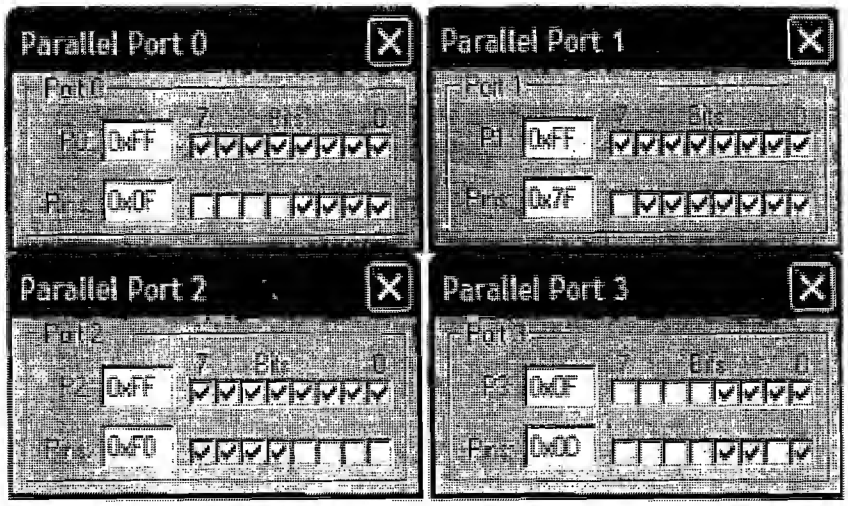
```
        3: int main(void)
C:0x0C2A    7BFF      MOV      R3,#0xFF
```

```
Hello World
```

Fig. 13.30 **Output of 'Hello World' Application at Serial Port**

https://hemanthrajhemu.github.io

To simulate the Ports and their status, select '*I/O-Ports*' from the '*Peripherals*' tab of IDE menu (Fig. 13.31).



Parallel Port 0
Port 0
P0: 0xFF Bits 7 ☑☑☑☑☑☑☑☑ 0
Pins: 0x0F □□□□☑☑☑☑

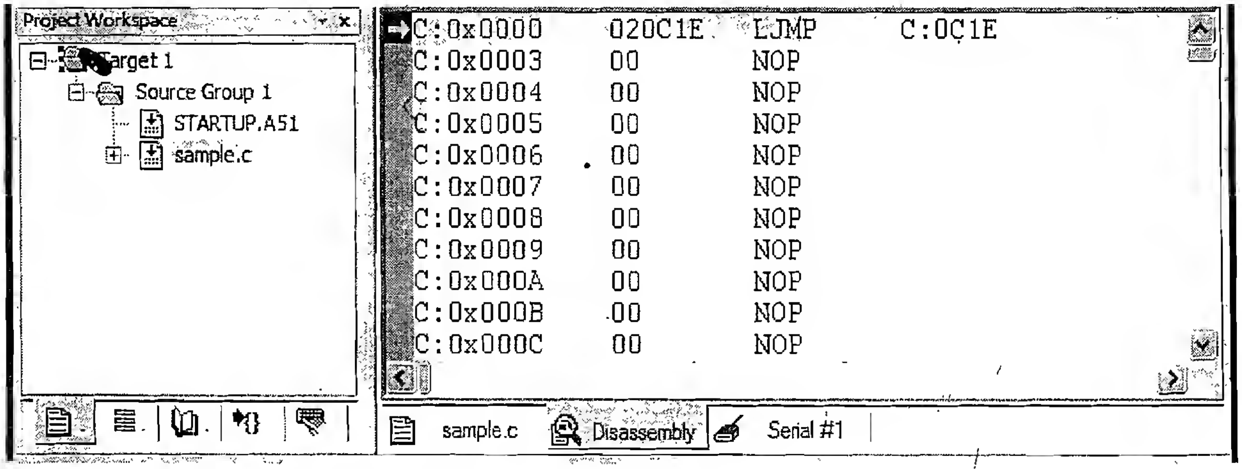
Parallel Port 1
Port 1
P1: 0xFF Bits 7 ☑☑☑☑☑☑☑☑ 0
Pins: 0x7F □☑☑☑☑☑☑☑

Parallel Port 2
Port 2
P2: 0xFF Bits 7 ☑☑☑☑☑☑☑☑ 0
Pins: 0xF0 ☑☑☑☑□□□□

Parallel Port 3
Port 3
P3: 0x0F Bits 7 □□□□☑☑☑☑ 0
Pins: 0x0D □□□□☑☑□☑

Fig. 13.31 **Simulation of Ports and Port Pins**

The first value (E.g. P0: 0xFF) simulates the contents of Port 0 Special Function Register. In order to make the port pins as input pins the port pin's corresponding SFR bit should be set as 1. With the SFR bit set to 1, the state of the corresponding port pin can be changed to either logic 1 or logic 0. To output logic 0 to a port pin, clear the corresponding SFR bit and for outputting logic 1, set the corresponding port bit in the SFR bit. In the above example, if Port 0 is viewed as an output port, the port will be outputting all 1s. If it is treated as an input port, the value inputted will be 0x0F (which is the state of the port pins). Serial Port and Timers can also be simulated by selecting the respective commands from the '*Peripherals*' tab of the IDE menu. It is left for the readers for experimenting. The '*Reset CPU*' option available under the '*Peripherals*' tab is used for resetting the firmware execution. Resetting the CPU while debugging brings the program execution to the reset vector (0x0000).

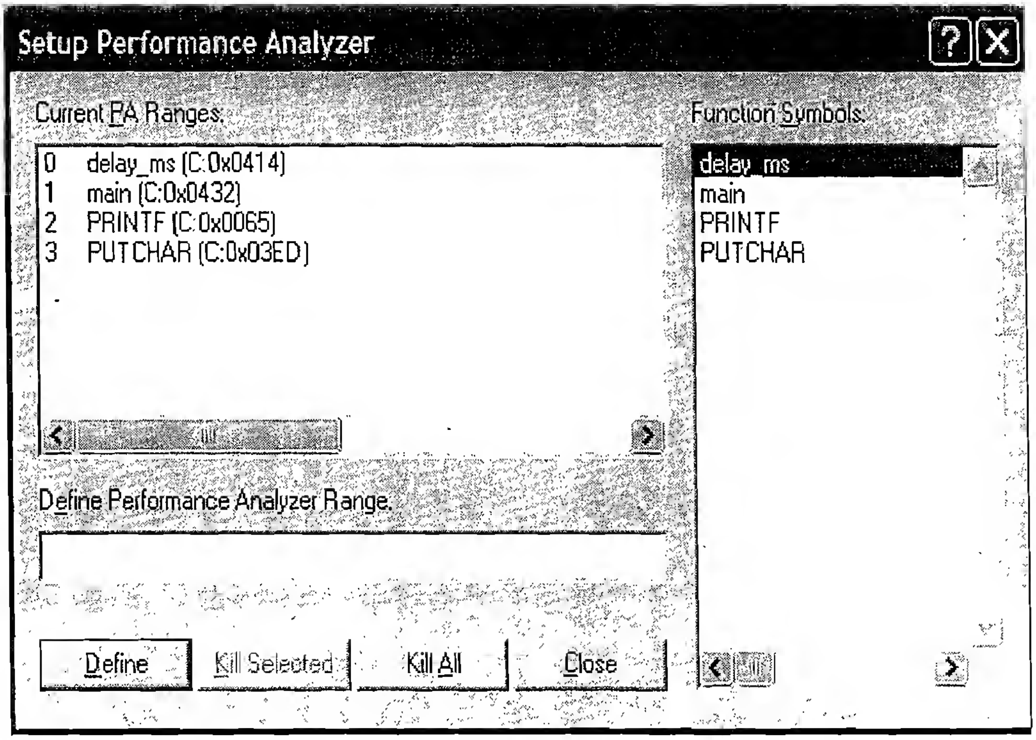


```
C:0x0000    020C1E    LJMP     C:0C1E
C:0x0003    00        NOP
C:0x0004    00        NOP
C:0x0005    00        NOP
C:0x0006    00        NOP
C:0x0007    00        NOP
C:0x0008    00        NOP
C:0x0009    00        NOP
C:0x000A    00        NOP
C:0x000B    00        NOP
C:0x000C    00        NOP
```

Fig. 13.32 **Resetting CPU while Debug in progress**

https://hemanthrajhemu.github.io

Generating precise delays in software for super loop based embedded applications are often quite difficult and there are no standard functions available for generating precise delays. If the '*C*' program is running on DOS platform, there is a standard function '*delay()*' available which generates delay with multiples of millisecond. There are no such standard functions available for generating delays in non-operating system based embedded programming. The only way of generating delays is writing a loop and set its parameters according to the clock frequency used. Often we need to do a trial and error method to finetune the parameters depending on the crystal used. The '*Performance analyser*' support by the Keil IDE helps in calculating the time consumed by a function. To activate this, select '*Performance Analyser...*' from the '*Debug*' tab of the IDE menu while debugging is on, before entering the function whose execution time needs to be calculated. A Performance analyser set up for selected function is shown in Fig. 13.33.



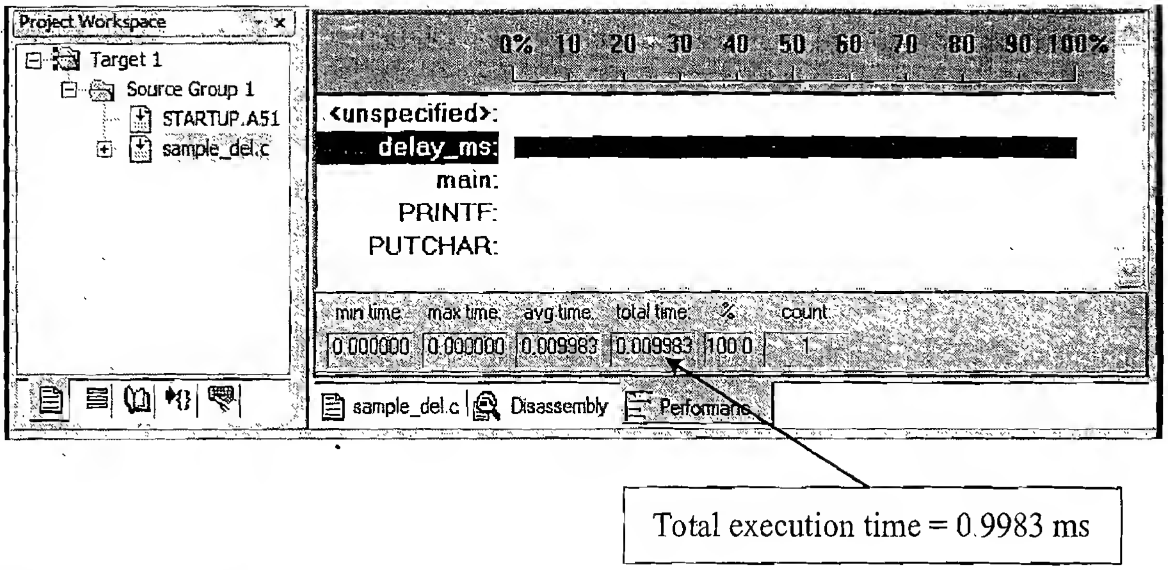
```
0  delay_ms (C:0x0414)
1  main (C:0x0432)
2  PRINTF (C:0x0065)
3  PUTCHAR (C:0x03ED)
```

```
delay_ms
main
PRINTF
PUTCHAR
```

Fig. 13.33 **Performance Analyser setup for selected function**

The various functions available on the current module is listed on the right side of the '*Setup Performance analyser*' window under the '*Function symbols:*' section. Double clicking on the function of interest displays the same on the '*Define Performance Analyser Range:*' Click the 'Define' button to add it to the analyser. Invoke the '*Performance analyser window*' from the '*View*' tab of the IDE menu. A new tab with name '*Performance...*' will be added to the text editor window. Now execute the function whose execution time is to be analysed by single step (*F11*) or by giving a step Over (*F10*) command. Select the tab '*Performance...*' and click on the function name for which the analysis is performed. The time taken for executing the function is displayed at the '*total time*' display window. This gives a rough estimate on the execution time of the function (Fig. 13.34). Note that in addition to this time, the time taken for calling this function also needs to be taken into account. Moreover, all these calculations are based on the target clock frequency set on the target options. We will discuss the same for a new function for generating milliseconds delay added to our program 'Hello World' application. The main

https://hemanthrajhemu.github.io



min time:	max time:	avg time:	total time:	%	count
0.000000	0.000000	0.009983	0.009983	100.0	1

Total execution time = 0.9983 ms

Fig. 13.34 **Execution time from Performance Analyzer**

function calls the delay routine *delay_ms()* before executing the *printf* function. To verify that this function is taking 1 millisecond time to finish execution with parameter 1, add the function to the performance analyser as mentioned above. Invoke the performance analyser and execute the function. Switch view to '*Performance...*' tab and observe the execution time. The complete source code for including delay is given below.

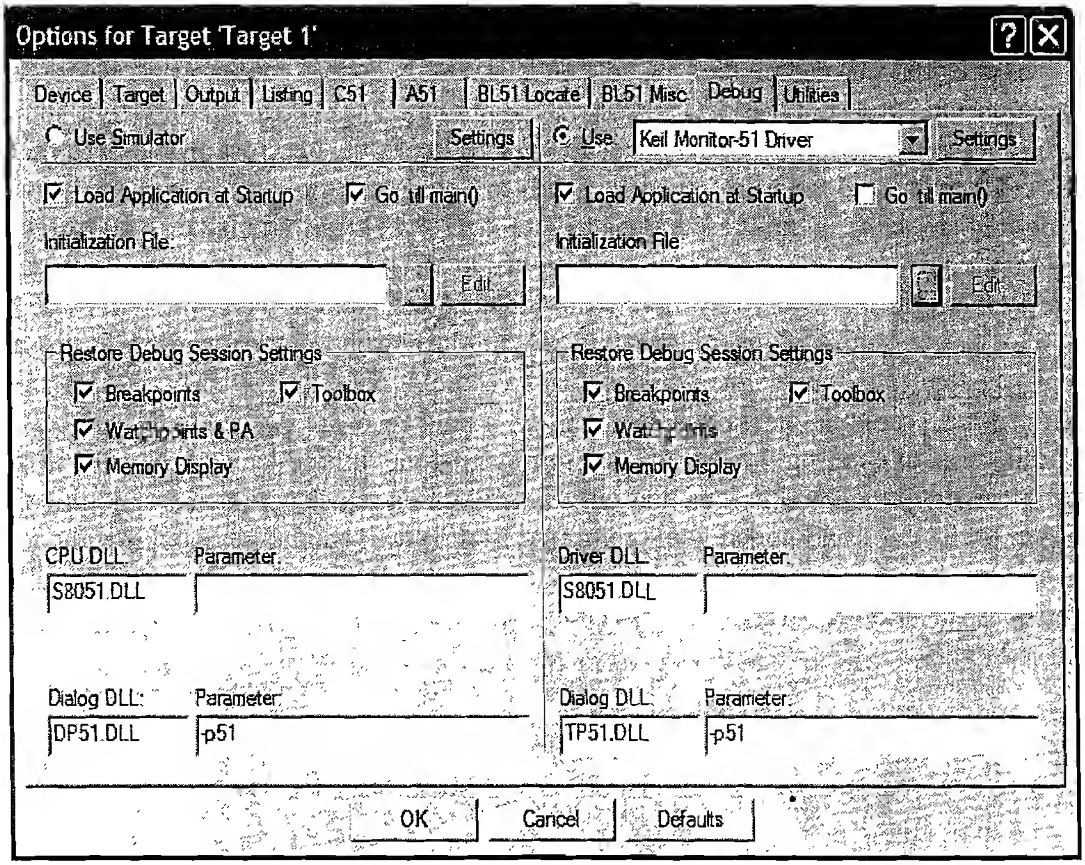
```
#include <stdio.h>
//Function Prototype for milliseconds delay
void delay_ms (int);

int main(void)
{
//Calling 1 milli second delay
delay_ms (1);
printf("Hello World\n");
}
//####################################################################
//Function for generating milli seconds delay
//Assumes clock frequency=24MHz

void delay_ms (int n)
{
int j, k;
for (j=0; j<n; j++)
{
for (k=0; k<247; k++);
}
}
```

It should be noted that generating highly precise delay is very difficult and there may be a tolerance of +/– fraction of milliseconds in the delay. Also the delay is purely dependent on the system clock

https://hemanthrajhemu.github.io

frequency. In a real world scenario, the stability of the clock is expressed in terms of +/– some parts per million (ppm) of the fundamental frequency. Any change in the target clock frequency needs the re-tuning of the code to bring it back to the desired tolerance level. Precise time delay can be generated using the hardware timer unit of the microcontroller.

13.1.1.3 Target Level Firmware Debugging with Keil µVision3 IDE Simulation based debugging technique lacks real-time behaviour and various input conditions needs to be simulated using the IDE support for debugging the firmware. Target level hardware debugging involves debugging the firmware which is embedded in the target board. This technique is also known as In Circuit Debugging/ Emulation (ICD/E). For performing target level debugging with Keil, the target board should be connected to the development machine through a serial interface. The IDE debug settings should be modified to activate target level debugging. Select the '*Debug*' option from '*Options for Target*' and change '*Use Simulator*' to '*Use:*' selected debug support from a list of available target debug supports (Fig. 13.35).



Options for Target 'Target 1'

Device | Target | Output | Listing | C51 | A51 | BL51 Locate | BL51 Misc | Debug | Utilities

○ Use Simulator — Settings

☑ Load Application at Startup ☑ Go till main()

Initialization File:

Restore Debug Session Settings
☑ Breakpoints ☑ Toolbox
☑ Watchpoints & PA
☑ Memory Display

CPU DLL: S8051.DLL Parameter:

Dialog DLL: DP51.DLL Parameter: -p51

● Use: Keil Monitor-51 Driver — Settings

☑ Load Application at Startup ☐ Go till main()

Initialization File:

Restore Debug Session Settings
☑ Breakpoints ☑ Toolbox
☑ Watchpoints
☑ Memory Display

Driver DLL: S8051.DLL Parameter:

Dialog DLL: TP51.DLL Parameter: -p51

Fig. 13.35 **Configuring IDE Debug settings for target level debugging**

Select the target supported debug option from the drivers available on the drop-down list and configure the serial link properties using the 'Settings' tab. Keil supports two types of target level debugging, namely '*Monitor program based debugging*' and '*Emulator based debugging*'. For Monitor program based debugging, the target processor should have Keil IDE supported monitor program installed and the target hardware should be configured to run the monitor program. Debug instructions from the IDE communicates with the monitor program running on the target processor and it returns the debug information to the IDE using the configured serial link. For Emulator based debugging, Keil IDE acts as an

https://hemanthrajhemu.github.io

interface between a Keil supported third party Emulator hardware and the target board. The Serial link should be connected to the emulator hardware and other end of the emulator hardware should be interfaced to the target hardware using any ICE supported interfaces like JTAG, BDM or pin-to-pin socket (For JTAG/BDM interface, target processor/controller should have this support at processor/controller side. If not, pin-to-pin socket is used).

13.1.1.4 Writing ISR in Embedded C using the µVision3 IDE The C51 cross-compiler from Keil supports the implementation of Interrupt Service Routines (ISRs) in Embedded C. The general form of writing an Interrupt Service Routine in C for C51 cross-compiler is illustrated below.

```
void ISR_Name (void) interrupt INTR_NO using REG_BANK
{
//Body of ISR
}
```

The attribute *interrupt* informs the cross compiler that the function with given name (Here *ISR_Name*) is an Interrupt Service Routine. The attribute *INTR_NO* indicates the interrupt number for an interrupt. It is essential for placing the ISR generated assembly code at the Interrupt Vector for the corresponding interrupt. Keil supports 32 ISRs for interrupt numbers 0 to 31. The interrupt number 0 corresponds to External Interrupt 0, 1 corresponds to Timer 0 Interrupt, 2 corresponds to External Interrupt 1, 3 corresponds to Timer 1 Interrupt, 4 corresponds to Serial Interrupt and so on. The keyword *using* specifies the Register bank mentioned in the attribute *REG_BANK* for the registers R0 to R7 inside the ISR. It is an optional attribute and if it is not mentioned in the ISR implementation, the current register bank in use by the controller is used by the ISR for the registers R0 to R7. The value of *REG_BANK* can vary from 0 to 3, representing register banks 0 to 3. With the attribute *interrupt*, the C51 cross compiler automatically generates the interrupt vector and entry and exit code for the specified interrupt routine. The contents of the Accumulator, B register , DPH, DPL, and PSW are Pushed to the stack as and when required, at the time of entering the ISR and are retrieved at the time of leaving the ISR. If the register bank is not specified in the ISR implementation, all the working registers which are modified inside the ISR is pushed to the stack and poped while returning from ISR. It should be noted that the ISR neither takes any parameter nor return any. The following piece of code illustrates the implementation of the Serial Interrupt ISR for 8051 using C51 cross compiler.

```
#include <reg51.h>        //Keil C51 Header file
//ISR for handling Serial Interrupts. Interrupt number = 4
//Interrupt Vector at 0023H. Use Register Bank 1

         void Serial_ISR (void) interrupt 4 using 1
         {
                  if (TI)    //Check transmit Interrupt (TI) Flag
                  {
                  //Transmit Interrupt Handler
                  }
                  else
                  {
                  //Receive Interrupt Handler
                  }
         }
```

https://hemanthrajhemu.github.io

13.1.1.5 Assembly Based Firmware Development with μVision3 IDE Apart from Embedded C based development, Keil IDE also supports Assembly language based firmware development. The steps involved in creating a project for writing assembly instructions are same as that of Embedded C based development. The only difference is that there is no need to include the standard startup file at the time of creating a project. Select the option '*No*' to the query '*Copy standard 8051 startup Code to Project Folder and add File to Project?*' while creating a new project (Refer to Section 13.1.1 for details on creating a new project). Since we are not using the IDE supplied startup code, the actions performed by startup code (mainly initialisation of stack pointer), should be written by the programmer before the start of the main program in the assembly file. Create a new file and start writing Assembly instructions in the file. Save the file with extension '*.src*' and add the file to the 'Source Group' of the target as illustrated in embedded C based development (Section 13.1.1). A program written in Assembly Language corresponding to the "Hello World" program written in Embedded C is illustrated below.

```
;#####################################################################
;                        Start of Main Program
;#####################################################################
ORG    0000H
               JMP MAIN
ORG    0003H   ; External Interrupt 0 Handler
               RETI
ORG    000BH   ; Timer0 Interrupt Handler
               RETI
ORG    0013H   ; External Interrupt 1 Handler
               RETI
ORG    001BH   ; Timer1 Interrupt Handler
               RETI
ORG    0023H   ; Serial Interrupt Handler
               RETI
ORG    0100H
MAIN:          MOV SP, #50H     ; Initialise stack pointer at 50H
;#####################################################################
               ; 8051 Serial Routine Parameter settings
               MOV SCON, #50h   ; Configure Serial Communication
                                ; Register.
               MOV TMOD, #21h   ; Baud Generation Settings
               MOV TH1, #0FDH   ; Re-Load Value for TIMER 1 in Auto
                                ; Re-Load
               MOV TL1, #0FDH   ; 9600 baud
               MOV A, PCON
               ANL A, #7FH
               MOV PCON, A
               SETB TR1         ; Start Timer1
;#####################################################################
OUT_TEXT:      CALL    TEXT_OUT
               JMP     OUT_TEXT
;#####################################################################
; Text outputting Routine
; Same as printf () in Embedded C
```

https://hemanthrajhemu.github.io

```
TEXT_OUT:        MOV DPTR, #HELLO_WORLD
SERIAL_OUT:      CLR TI              ; Clear Transmit Interrupt
                 CLR A
                 MOVC A,@A+DPTR
                 CJNE A, #'\', FOLLOW
                 MOV SBUF, #0AH      ; Send new line character
                 JNB TI, $
                 RET
FOLLOW:MOV       SBUF, A
                 JNB TI, $           ; Wait till Transmit Interrupt before
                                     ; sending next byte
                 INC DPTR
                 JMP SERIAL_OUT
;######################################################################
; Store the string "Hello World" in program memory
;######################################################################
HELLO_WORLD:

DB      'Hello World\'
        END                          ; End of Assembly
```

Compile this source code using the same compile options illustrated for Embedded C based development. Here also you can have multiple source (*.src*) files. Add all source files to the 'Source Group' and compile the program. The Assembly program is written to store the string "Hello World" in the code memory and it is retrieved from the code memory for sending to the serial port. The string can also be stored in the data memory and retrieve it from there for sending to the serial port. In that case the storage memory for the string "*Hello World*" (12 bytes with string termination character '\') in the code memory will be released and the same will occupy in the data memory. Figure 13.36 shows the assembling of source code written in assembly language.

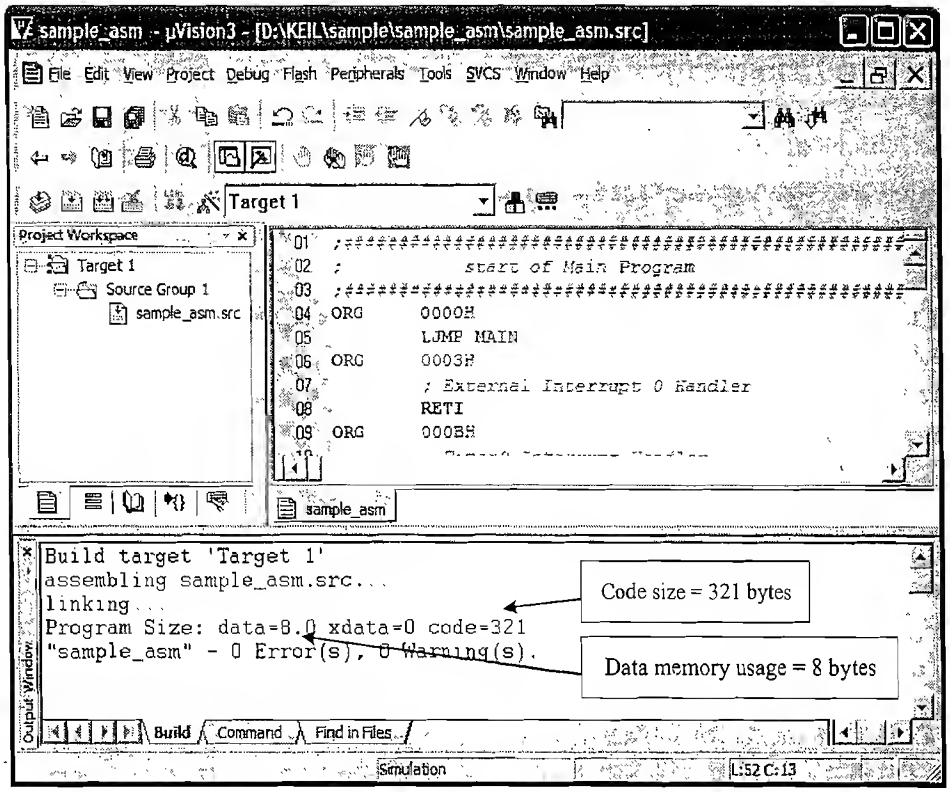
If you observe the output window while compiling, you can see that the source code is '*assembled*' instead of '*compiling*'. Why this strange thing happens? - The answer is conversion of program written in assembly language to object code is carried out by the utility '*Assembler*'. Assembler is same as compiler in functioning but the input is different. You can debug this application using the simulator in a similar way as that of debugging the 'C' source code. On debugging the assembly code you can view the output on the '*Serial Window*' and it is exactly the same as the output produced by the "*Hello World*" Application written in Embedded C.

Now let's have a comparison between the application written in Embedded C and Assembly Language for outputting the string "Hello World" to the Serial port. Read carefully.

Q1. How many lines of code you wrote for the program "Hello World"?
Embedded C: Only a single line of Code excluding the framework of main
Assembly: More than 25 lines

Q2. How many registers of the target processor you are familiar with for writing the "Hello World" program?
Embedded C: None
Assembly: Almost all

https://hemanthrajhemu.github.io


sample_asm - µVision3 - [D:\KEIL\sample\sample_asm\sample_asm.src]
File Edit View Project Debug Flash Peripherals Tools SVCS Window Help
Target 1
Project Workspace
Target 1
Source Group 1
sample_asm.src
sample_asm
Output Window
Build
Command
Find in Files
Simulation
L:52 C:13

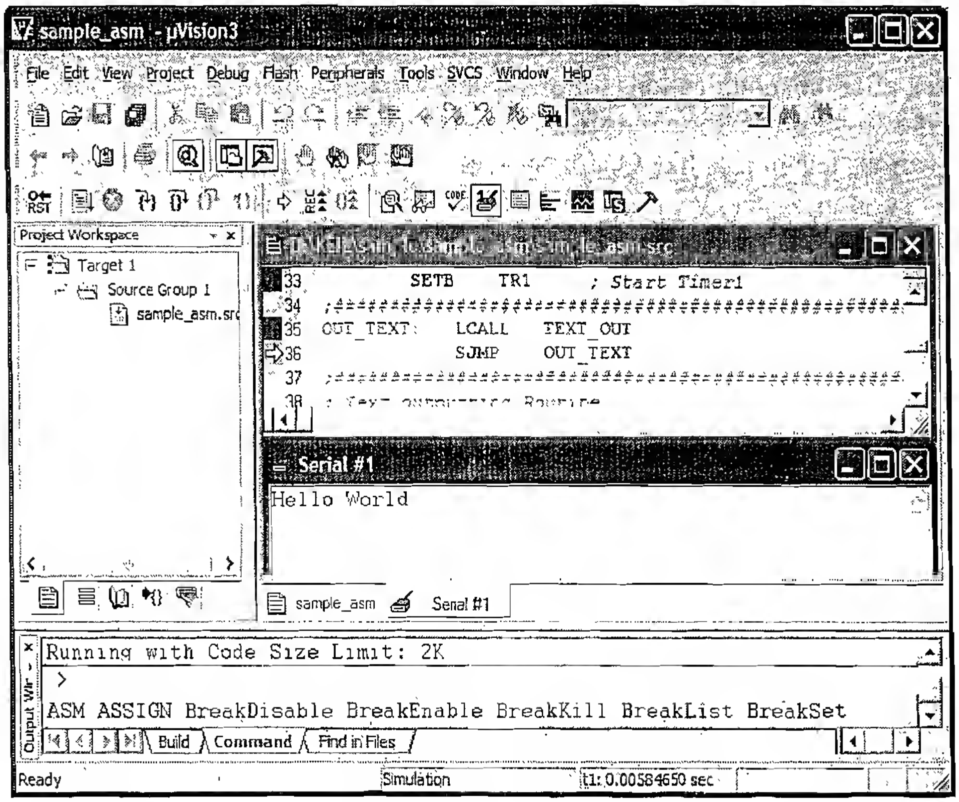
```
;#################################################################
;                   start of Main Program
;#################################################################
ORG       0000H
          LJMP MAIN
ORG       0003H
          ; External Interrupt 0 Handler
          RETI
ORG       000BH
```

```
Build target 'Target 1'
assembling sample_asm.src...
linking...
Program Size: data=8.0 xdata=0 code=321
"sample_asm" - 0 Error(s), 0 Warning(s).
```

Code size = 321 bytes

Data memory usage = 8 bytes

Fig. 13.36 **Assembling of source code written in Assembly Language**



```
              SETB     TR1          ; Start Timer1
;#################################################################
OUT_TEXT:     LCALL    TEXT_OUT
              SJMP     OUT_TEXT
;#################################################################
```

```
Hello World
```

```
Running with Code Size Limit: 2K
>
ASM ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet
```

Fig. 13.37 **Debugging the Source code written in Assembly Language for outputting the text "*Hello World*" to serial port**

https://hemanthrajhemu.github.io

Q3. How many Assembly Instructions of the target processor you are familiar with for writing the "Hello World" program?

Embedded C: None

Assembly: Almost all

Q4. What is the code size for the "Hello World" program written in?

Embedded C: 1078 bytes

Assembly: 321 bytes

Q5. What is the data memory size for the "Hello World" Program written in?

Embedded C: 30 bytes + 1 bit

Assembly: 8 bytes Minimum and 21 Bytes Maximum (If the "Hello World" string is stored in Data memory)

Summary

1. Compared to Assembly programming, Embedded C requires lesser number of lines of code to implement a task
2. Embedded C programmers require less or little knowledge on the internals of the target processor whereas Assembly programmers require thorough knowledge of the internals of the processor
3. Embedded C programmers need not be aware of the instruction set of the target processor whereas Assembly language programmers should be well versed in the same.
4. The Code memory usage by programs written in Assembly is optimal compared to the one written in Embedded C
5. The Data memory usage by programs written in Assembly is optimal compared to the one written in Embedded C

13.1.1.6 Flash Memory Programming with Keil μVision3 IDE Once the firmware starts functioning properly, after the modifications following debug and simulation, the next step is embedding the firmware into the target processor/controller. As mentioned in a previous chapter, firmware can be embedded using various techniques like Out of System Programming, In System Programming (ISP), etc. Keil IDE provides an In System Programming (ISP) support which is selected at the time of configuring the '*Options for Target*'. It can be either Keil provided flash programming driver or any third party tool which can be invoked from the IDE. The flash programming makes use of the serial connection established between the Host PC and the target board. The target processor should contain a bootloader or a monitor program which can understand the flash memory programming related commands sent from the IDE. Cross-compile all the source files within the module, link them and generate the binary code using '*Rebuild all target files*' option. Download the generated binary file to the target processor's/controller's memory by invoking '*Flash Download*' option from the '*Project*' tab of the IDE menu. The target board should be powered and interfaced with the host PC where the IDE is running and the connection should be configured properly.

13.1.1.7 Summary of usage of Keil μVision3 IDE So far we discussed about IDE and how IDE is helpful in Embedded application development and debugging. Our discussion was focusing only on *8051* family 8bit microcontroller firmware application development using Keil μVision3 IDE. Though the discussion was specific to Keil IDE, I believe it was capable of giving you the basic fundas of IDEs and how the IDE is used in embedded system development. The hot key usage, GUI details and functions offered by different IDEs are totally different. Illustrating all IDEs is out of scope of this book and also there is no common IDE supporting multiple family devices that can be used for illustrating the same. Users are requested to go through the respective manuals of the IDE they are using for firmware development. Also, IDEs undergo rapid changes by adding new features, functions, etc. and whatever

https://hemanthrajhemu.github.io

we discussed here need not be the same after three months or six months or one year, in terms of features, UI, hot keys, menu options, etc. When I started the work of this book, the IDE offered from Keil was Keil µVision2 and at the time of finishing this work, the latest IDE available from Keil is Keil µVision3. It incorporated lots of new features and changes compared to the old IDE. Users are requested to visit the web site www.keil.com to get the latest updates on thc new versions of the IDE. You can directly download the trial version of the IDE from the above site.

13.1.2 An Overview of IDEs for Embedded System Development

The table shown below gives an overview of the Integrated Development Environments (IDEs) used for developing embedded systems for various processors/Real-time operating systems.

Processor Family/ RTOS	IDE Name	Supplier/Remarks
8051	Keil Micro Vision	Keil Software. An ARM Company. www.keil.com
8051	RIDE	Raisonance http://www.raisonance.com/products/info/RIDE.php
8051	SC51	SPJ Embedded Technologies http://www.spjsystems.com/sc51.htm
8051	IAR Embedded Workbench FOR 8051	IAR Systems http://www.iar.com/website1/1.0.1.0/244/1/index.php
PIC	MPLAB	Microchip Technology Inc http://www.microchip.com/
PIC	IAR Embedded Workbench for PIC	IAR Systems http://www.iar.com/website1/1.0.1.0/214/1/index.php
AVR	AVR Studio AVR32 Studio	Atmel Corporation http://www.atmel.com/dyn/Products/tools_card.asp?tool_id=2725
ARM	RealView MDK	Keil Software. An ARM Company http://www.keil.com/arm/
ARM	IAR Embedded Workbench for ARM	IAR Systems http://www.iar.com/website1/1.0.1.0/68/1/index.php
ARM	CodeWarrior	Freescale Semiconductor http://www.freescale.com/
ARM	Sourcery G++	CodeSourcery http://www.codesourcery.com/gnu_toolchains/sgpp It is a complete software development environment based on the GNU Tool chain. Sourcery G++ includes the GNU C and C++ compilers, the Eclipse IDE. Supports ARM®, ColdFire®, fido™, MIPS®, Power Architecture™, Stellaris®, x86
Infineon Family	DAvE	Infineon Technologies. For the 8, 16 and 32 bit Infineon processors/controllers. http://www.infineon.com/

https://hemanthrajhemu.github.io

MIPS	Eclipse Ganymede	Macraigor Systems. Eclipse based GNU tool suite. http://www.macraigor.com/Eclipse/index.htm
PowerPC	Eclipse Ganymede	Macraigor Systems. Eclipse based GNU tool suite. http://www.macraigor.com/Eclipse/index.htm
ST Family	ST Visual Develop	IDE for ST Microelectronics processor/controller products. http://www.st.com/mcu/contentid-44.html
Blackfin Family DSP	VisualDSP	Analog Devices www.analog.com
TI Family of DSP	Code Composer Studio	Texas Instruments. http://www.ti.com/
Windows CE RTOS	Platform Builder	Microsoft. www.microsoft.com Available as bundled package with Visual Studio IDE
VxWorks RTOS	Wind River Workbench	Wind River Systems. www.windriver.com
QNX RTOS	QNX Momentics	QNX Software Systems www.qnx.com
3rd Party Tools	MULTI IDE	Green Hills Software Supports a variety of family of processors. Visit www.ghs.com for more details

And.... The list continues. There are thousands of IDEs available in the market as either commercial or non-commercial and as either Open source tools or proprietary tools. Listing all of them is out of the scope of this book. The intention is to just make the readers familiar with some of the popular IDEs for some commonly used processors/controllers and RTOSs for embedded development.

13.2 TYPES OF FILES GENERATED ON CROSS-COMPILATION

Cross-compilation is the process of converting a source code written in high level language (like 'Embedded C') to a target processor/controller understandable machine code (e.g. ARM processor or *8051* microcontroller specific machine code). The conversion of the code is done by software running on a processor/controller (e.g. x86 processor based PC) which is different from the target processor. The software performing this operation is referred as the 'Cross-compiler'. In a single word cross-compilation is the process of cross platform software/firmware development. Cross assembling is similar to cross-compiling; the only difference is that the code written in a target processor/controller specific Assembly code is converted into its corresponding machine code. The application converting Assembly instruction to target processor/controller specific machine code is known as cross-assembler. Cross-compilation/cross-assembling is carried out in different steps and the process generates various types of intermediate files. Almost all compilers provide the option to select whatever intermediate files needs to be retained after cross-compilation. The various files generated during the cross-compilation/cross-assembling process are:

List File (.lst), Hex File (.hex), Pre-processor Output file, Map File (File extension linker dependent), Object File (.obj)

https://hemanthrajhemu.github.io

13.2.1 List File (*.LST File*)

Listing file is generated during the cross-compilation process and it contains an abundance of information about the cross compilation process, like cross compiler details, formatted source text **('C' code)**, assembly code generated from the source file, symbol tables, errors and warnings detected during the cross-compilation process. The type of information contained in the list file is cross-compiler specific. As an example let's consider the cross-compilation process of the file *sample.c* given as the first illustrative embedded C program under Keil µVision3 IDE discussion. The '*list file*' generated contains the following sections.

Page Header A header on each page of the listing file which indicates the compiler version number, source file name, date, time, and page number.

```
C51 COMPILER V8.02 SAMPLE                    05/23/2006 11:29:58 PAGE 1
```

Command Line Represents the entire command line that was used for invoking the compiler.

```
C51 COMPILER V8.02, COMPILATION OF MODULE SAMPLE
OBJECT MODULE PLACED IN sample.OBJ
COMPILER INVOKED BY: C:\Keil\C51\BIN\C51.EXE sample.c BROWSE DEBUG OBJECTEXTEND
CODE LISTINCLUDE SYMBOLS
```

Source Code The source code listing outputs the line number as well as the source code on that line. Special cross compiler directives can be used to include or exclude the conditional codes (code in #if blocks) in the source code listings. Apart from the source code lines, the list file will include the comments in the source file and depending on the list file generation settings the entire contents of all include files may also be included. Special cross compiler directives can be used to include the entire contents of the include file in the list file.

```
line level    source
 1        //Sample.c for printing Hello World!
 2        //Written by xyz
 3        #include <stdio.h>
 1    =1  /*------------------------------------------------------------------------
 2    =1  STDIO.H
 3    =1
 4    =1  Prototypes for standard I/O functions.
 5    =1  Copyright © 1988–2002 Keil Elektronik GmbH and Keil Software, Inc.
 6    =1  All rights reserved.
 7    =1  ------------------------------------------------------------------------*/
 8    =1
 9    =1  #ifndef __STDIO_H__
10    =1  #define __STDIO_H__
11    =1
12    =1  #ifndef EOF
13    =1   #define EOF -1
14    =1  #endif
15    =1
16    =1  #ifndef NULL
```

https://hemanthrajhemu.github.io

```
17    =1  #define NULL ((void *) 0)
18    =1  #endif
19    =1
20    =1  #ifndef _SIZE_T
21    =1   #define _SIZE_T
22    =1   typedef unsigned int size_t;
23    =1  #endif
24    =1
25    =1  #pragma SAVE
26    =1  #pragma REGPARMS
27    =1  extern char _getkey (void);
28    =1  extern char getchar (void);
29    =1  extern char ungetchar (char);
30    =1  extern char putchar (char);
31    =1  extern int printf  (const char *, ...);
32    =1  extern int sprintf  (char *, const char *, ...);
33    =1  extern int vprintf  (const char *, char *);
34    =1  extern int vsprintf (char *, const char *, char *);
35    =1  extern char *gets (char *, int n);
36    =1  extern int scanf (const char *, ...);
37    =1  extern int sscanf (char *, const char *, ...);
38    =1  extern int puts (const char *);
39    =1
40—   =1  #pragma RESTORE
41    =1
42    =1  #endif
43    =1
4
5         void main()
6         {
7   1      printf("Hello World!\n");
8   1      }
9
```

Assembly Listing Assembly listing contains the assembly code generated by the cross compiler for the 'C' source code. Assembly code generated can be excluded from the list file by using special compiler directives.

```
ASSEMBLY LISTING OF GENERATED OBJECT CODE
            ; FUNCTION main (BEGIN)
                                ; SOURCE LINE # 5
                                ; SOURCE LINE # 6
                                ; SOURCE LINE # 7
0000 7BFF                      MOV     R3, #0FFH
0002 7A00       R              MOV     R2, #HIGH ?SC_0
```

https://hemanthrajhemu.github.io

```
0004 7900      R        MOV    R1, #LOW ?SC_0
0006 020000    E        LJMP   _printf

            ; FUNCTION main (END)
```

Symbol Listing The symbol listing contains symbolic information about the various symbols present in the cross compiled source file. Symbol listing contains the sections symbol name (NAME) symbol classification (CLASS (Special Function Register (SFR), structure, typedef, static, public, auto, extern, etc.)), memory space (MSPACE (code memory or data memory)), data type (TYPE (int, char, Procedure call, etc.)), offset ((OFFSET from code memory start address)) and size in bytes (SIZE). Symbol listing in list file output can be turned on or off by cross-compiler directives.

```
NAME        CLASS        MSPACE     TYPE      OFFSET     SIZE
====        =====        ======     ====      ======     ====
size_t .....TYPEDEF      -----      U_INT      -----     2
main .......PUBLIC       CODE       PROC      0000H      -----
_printf.....EXTERN       CODE       PROC      -----      -----
```

Module Information The module information provides the size of initialised and un-initialised memory areas defined by the source file

```
MODULE INFORMATION:         STATIC     OVERLAYABLE
  CODE SIZE          =      9           ----
  CONSTANT SIZE      =      14          ----
  XDATA SIZE         =      ----        ----
  PDATA SIZE         =      ----        ----
  DATA SIZE          =      ----        ----
  IDATA SIZE         =      ----        ----
  BIT SIZE           =      ----        ----
END OF MODULE INFORMATION.
```

Warnings and Errors Warnings and Errors section of list file records the errors encountered or any statement that may create issues in application (warnings), during cross-compilation. The warning levels can be configured before cross compilation. You can ignore certain warnings, e.g. a local variable is declared within a function and it is not used anywhere in the program. Certain warnings require prompt attention.

```
C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

List file is a very useful tool for application debugging in case of any cross compilation issues.

13.2.2 Preprocessor Output File

The preprocessor output file generated during cross-compilation contains the preprocessor output for the preprocessor instructions used in the source file. Preprocessor output file is used for verifying the operation of macros and conditional preprocessor directives. The preprocessor output file is a valid C source file. File extension of preprocessor output file is cross compiler dependent.

https://hemanthrajhemu.github.io

13.2.3 Object File (*.OBJ* File)

Cross-compiling/assembling each source module (written in C/Assembly) converts the various Embedded C/Assembly instructions and other directives present in the module to an object (*.OBJ*) file. The format (internal representation) of the *.OBJ* file is cross compiler dependent. OMF51 or OMF2 are the two objects file formats supported by C51 cross compiler. The object file is a specially formatted file with data records for symbolic information, object code, debugging information, library references, etc. The list of some of the details stored in an object file is given below.

1. Reserved memory for global variables.
2. Public symbol (variable and function) names.
3. External symbol (variable and function) references.
4. Library files with which to link.
5. Debugging information to help synchronise source lines with object code.

The object code present in the object file are not absolute, meaning, the code is not allocated fixed memory location in code memory. It is the responsibility of the linker/locater to assign an absolute memory location to the object code. During cross-compilation process, the cross compiler sets the address of references to external variables and functions as 0. The external references are resolved by the linker during the linking process. Hence it is obvious that the code generated by the cross-compiler is not executable without linking it for resolving external references.

13.2.4 Map File (*.MAP*)

As mentioned above, the cross-compiler converts each source code module into a re-locatable object (OBJ) file. Cross-compiling each source code module generates its own list file. In a project with multiple source files, the cross-compilation of each module generates a corresponding object file. The object files so created are re-locatable codes, meaning their location in the code memory is not fixed. It is the responsibility of a *linker* to link all these object files. The *locater* is responsible for locating absolute address to each module in the code memory. Linking and locating of re-locatable object files will also generate a list file called '*linker list file*' or '*map file*'. Map file contains information about the link/locate process and is composed of a number of sections. The different sections listed in a map file are cross compiler dependent. The information generally held by map files is listed below. It is not necessary that the map files generated by all *linkers/locaters* should contain all these information. Some may contain less information compared to this or others may contain more information than given in this. It all depends on the *linker/locater*.

Page Header A header on each page of the linker listing (MAP) file which indicates the linker version number, date, time, and page number.

e.g.

```
BL51 LINKER/LOCATER V3.62    02/29/2004  09:59:51  PAGE 1
```

Command Line Represents the entire command line that was used for invoking the linker.

e.g.

```
BL51 BANKED LINKER/LOCATER V6.00, INVOKED BY:
C:\KEIL\C51\BIN\BL51.EXE STARTUP.obj, sample.obj TO sample
```

https://hemanthrajhemu.github.io

CPU Details Details about the target CPU and memory model (internal data memory, external data memory, paged data memory, etc.) come under this category.

e.g. `MEMORY MODEL: SMALL`

Input Modules This section includes the names of all object modules, and library files and modules that are included in the linking process. This section can be checked for ensuring all the required modules are lined in the linking process

e.g.

```
INPUT MODULES INCLUDED:
STARTUP.obj (?C_STARTUP)
  sample.obj (SAMPLE)
  C:\KEIL\C51\LIB\C51S.LIB (PRINTF)
  C:\KEIL\C51\LIB\C51S.LIB (?C?CLDPTR)
  C:\KEIL\C51\LIB\C51S.LIB (?C?CLDOPTR)
  C:\KEIL\C51\LIB\C51S.LIB (?C?CSTPTR)
  C:\KEIL\C51\LIB\C51S.LIB (?C?PLDIIDATA)
  C:\KEIL\C51\LIB\C51S.LIB (?C?CCASE)
  C:\KEIL\C51\LIB\C51S.LIB (PUTCHAR)
```

Memory Map Memory map lists the starting address, length, relocation type and name of each segment in the program.

e.g.

```
TYPE   BASE    LENGTH_ RELOCATION   SEGMENT NAME
       -------------------------------------------------------
* * * * * * ****** D A T A   M E M O R Y  * * * * * * ********
REG      0000H    0008H        ABSOLUTE     "REG BANK 0"
DATA   0008H    0014H          UNIT         _DATA_GROUP_
           001CH    0004H               *** GAP ***
BIT        0020H.0  0001H.1  UNIT        _BIT_GROUP_
        0021H.1  0000H.7              *** GAP ***
IDATA  0022H    0001H          UNIT        ?STACK
* * * * * * ****** C O D E   M E M O R Y  * * * * * * ********
CODE   0000H   0003H   ABSOLUTE
       0003H   07FDH                 *** GAP ***
CODE   0800H   035CH   UNIT      ?PR?PRINTF?PRINTF
CODE   0B5CH   008EH   UNIT      ?C?LIB_CODE
CODE   0BEAH   0027H   UNIT      ?PR?PUTCHAR?PUTCHAR
CODE   0C11H   000EH   UNIT      ?CO?SAMPLE
CODE   0C1FH   000CH   UNIT      ?C_C51STARTUP
CODE   0C2BH   0009H   UNIT      ?PR?MAIN?SAMPLE
```

Symbol Table It contains the value, type and name for all symbols from the different input modules

https://hemanthrajhemu.github.io

e.g.

```
SYMBOL TABLE OF MODULE:  sample (?C_STARTUP)

 VALUE       TYPE          NAME
 ----------------------------------------------------------
 -------     MODULE     ?C_STARTUP
 C:0C1FH     SEGMENT    ?C_C51STARTUP
 I:0022H     SEGMENT    ?STACK
 C:0000H     PUBLIC     ?C_STARTUP
 D:00E0H     SYMBOL     ACC
 D:00F0H     SYMBOL     B
 D:0083H     SYMBOL     DPH
 D:0082H     SYMBOL     DPL
```

Inter Module Cross Reference The cross reference listing includes the section name, memory type and the name of the modules in which it is defined and all modules in which it is accessed.

e.g.

```
NAME..................USAGE        MODULE NAMES
----------------------------------------------------------------
?C?CCASE ............CODE;         ?C?CCASE  PRINTF
?C?CLDOPTR ........CODE;           ?C?CLDOPTR  PRINTF
?C?CLDPTR.............CODE;        ?C?CLDPTR  PRINTF
?C?CSTPTR..............CODE;       ?C?CSTPTR  PRINTF
?C?PLDIIDATA.......CODE;           ?C?PLDIIDATA  PRINTF
```

Program Size Program size information contain the size of various memory areas as well as constant and code space for the entire application

e.g. `Program Size: data=30.1 xdata=0 code=1079`

Warnings and Errors Errors and warnings generated while linking a program are written to this section. It is very useful in debugging link errors.

e.g. `LINK/LOCATE RUN COMPLETE.  0 WARNING(S),  0 ERROR(S)`

> **NB:** The file extension for MAP files generated by different *linkers/locaters* need not be the same. It varies across *linker/locater* in use. For example the map file generated for BL51 Linker/locater is with extension *.M51*

13.2.5 HEX File (*.HEX*)

Hex file is the binary executable file created from the source code. The absolute object file created by the linker/locater is converted into processor understandable binary code. The utility used for converting an object file to a hex file is known as *Object to Hex file converter*. Hex files embed the machine code in a particular format. The format of Hex file varies across the family of processors/controllers. *Intel HEX* and *Motorola HEX* are the two commonly used hex file formats in embedded applications. Intel HEX file is an ASCII text file in which the HEX data is represented in ASCII format in lines. The lines in an

https://hemanthrajhemu.github.io

Intel HEX file are corresponding to a HEX Record. Each record is made up of hexadecimal numbers that represent machine-language code and/or constant data. Individual records are terminated with a carriage return and a linefeed. *Intel HEX* file is used for transferring the program and data to a ROM or EPROM which is used as code memory storage.

13.2.5.1 Intel HEX File Format As mentioned, *Intel HEX* file is composed of a number of HEX records. Each record is made up of five fields arranged in the following format:

:llaaaattdd...cc

Each group of letters corresponds to a different field, and each letter represents a single hexadecimal digit. Each field is composed of at least two hexadecimal digits (which make up a byte) as described below:

Field	Description
:	The colon indicating the start of every Intel HEX record
ll:	Record length field representing the number of data bytes (dd) in the record
aaaa:	Address field representing the starting address for subsequent data in the record
tt:	Field indicating the HEX record type. According to its value it can be of the following types: 00: Data Record 01: End of File Record 02: 8086 Segment Address Record 04: Extended Linear Address record
dd:	Data field that represents one byte of data. A record can have number of data bytes. The number of data bytes in the record must match to the number specified by the 'll' field
cc:	Checksum field representing the checksum of the record. Checksum is calculated by adding the values of all hexadecimal digit pairs in the record and taking modulo 256. Resultant o/p is 2's complemented to get the checksum.

An extract from the Intel hex file generated for "Hello World" application example is given below.

```
:03000000020C1FD0
:0C0C1F00787FE4F6D8FD758121020C2BD3
:0E0C110048656C6C6F20576F726C64210A008E
:090C2B007BFF7A0C7911020862CA
:10080000E517240BF8E60517227808300702780B65
:10081000E475F001120BB4020B5C2000EB7F2ED2CA
:10082000008018EF540F2490D43440D4FF30040BD0
:10083000EF24BFB41A0050032461FFE518600215CD
:1008400018051BE51B7002051A30070D7808E475C2
:100BFA00B8130CC2983098FDA899C298B811F6306B
:070C0A0099FDC299F5992242
:00000001FF
```

Let's analyse the first record

:	l	l	a	a	a	a	t	t	d	d	d	d	d	d	c	c
:	0	3	0	0	0	0	0	0	0	2	0	C	1	F	D	0

https://hemanthrajhemu.github.io

: field indicates the start of a new record. 03 (*ll*) gives the number of data bytes in the record. For this record,'*ll*' is 03 and the number of data bytes in the corresponding record is 03. The start address (*aaaa*) of data in the record is 0000H. The record type byte (*tt*) for this record is 00 and it indicates that this record is a data record. The data for the above record is 02, 0C and 1F. They are supposed to place at three consecutive memory locations in the EEPROM with starting address 0000H. The arrangement is given below.

Memory Address in Hex	Data in Hex
0000H	02
0001H	0C
0002H	1F

If you are familiar with 8051 Machine code you can easily identify that 02 is the machine code for the instruction LJMP and the next two bytes represent the 16bit address of the location to which the jump is intended. Obviously the instruction is *LJMP 0C1F*. The last two digits (*cc*) of the record holds the checksum of the values present in the record. The checksum is calculated by adding all the bytes in the record and then taking modulo 256 of the result. The resultant is 2's complemented and represented as checksum in the record field. Above example it is 0xD0. Intel hex files end with an end of file record indicating the end of records in the hex file. Let's examine the end of record structure.

:	l	l	a	a	a	a	t	t	c	c
:	0	0	0	0	0	0	0	1	F	F

End of record also starts with the start of record symbol ':'. Since End of record does not contain any data bytes, field '*ll*' will be 00. The field '*aaaa*' is not significant since the number of data bytes are zero. Field '*tt*' will hold the value 01 to indicate that this record is an End of record. Field '*cc*' holds the checksum of all the bytes present in the record and it is calculated as 2's complement of Modulo 256 of (0 + 0 + 0 + 1) = 0xFF

13.2.5.2 Motorola HEX File Format Similar to the Intel HEX file, Motorola HEX file is also an ASCII text file where the HEX data is represented in ASCII format in lines. The lines in *Motorola HEX* file represent a HEX Record. Each record is made up of hexadecimal numbers that represent machine-language code and/or constant data. The general form of Motorola Hex record is given below.

SOR	RT	Length	Start Address	Data/Code	Checksum

In other words it can be represented as **Stllaaaadddd...cc**
The fields of the record are explained below.

Field	Description
SOR	Stands for Start of record. The ASCII Character 'S' is used as the Start of Record. Every record begins with the character 'S'
RT	Stands for Record type. The character 't' represents the type of record in the general format. There are different meanings for the record depending on the value of 't' 0: Header. Indicates the beginning of Hex File 1: Data Record with 16bit start address 2: Data record with 24bit start address 9: End of File Record

https://hemanthrajhemu.github.io

Length (*ll*):	Stands for the count of the character pairs in the record, excluding the type and record length (Count includes the number of data/code bytes, data bytes representing start address and character pair representing the checksum). Two ASCII characters '*ll*' represent the length field. Each '*l*' in the representation can take values 0 to 9 and A to F.
Start Address (*aaaa*):	Address field representing the starting address for subsequent data in the record.
Code/Data (dd):	Data field that represents one byte of data. A record can have number of data bytes. The number of data bytes in the record must match to the number specified by (*ll* – *no. of character pairs for start address* –1)
Checksum (cc):	Checksum field representing the checksum of the record. Checksum is calculated by adding the values of all hexadecimal digit pairs in the record and taking modulo 256. Resultant o/p is 1's complemented to get the checksum.

Typical example of a Motorola Hex File format is given below.

```
S011000064656D6F5F68637331322E616273E5
S11311000002000800082629001853812341001812
S9030000FC
```

You can see that each Record starts with the ASCII character 'S'. For the first record, the value for field '*t*' is 0 and it implies that this record is the first record in the hex file (Header Record). The second record is a data record. The field '*t*' for second record is 1. Number of character pairs held by second record is 0x13 (19 in decimal). Out of this two character pairs are used for holding the start address and one character pair for holding the checksum. Rest 16 bytes are the data bytes. The start address for placing the data bytes is 0x1100. The data bytes that are going to be placed in 16 consecutive memory locations starting from address 0x1100 are 0x00, 0x02, 0x00, 0x08, 0x00, 0x08, 0x26, 0x29, 0x00, 0x18, 0x53, 0x81, 0x23, 0x41, 0x00 and 0x18. The last two digits (here 0x12) represent the checksum of the record. Checksum is calculated as the least significant byte of the one's complement of the sum of the values represented by the pairs of characters making up the record length, address, and data fields. The third record represents the End of File record. The value for field '*t*' for this record is 9 and it is an indicative of End of Hex file. Number of character pairs held by this record is 03; two for address and one for the checksum. The address is insignificant here since the record does not contain any values to dump into memory. Only one End of File Record is allowed per file and it must be the last line of the file.

13.3 DISASSEMBLER/DECOMPILER

Disassembler is a utility program which converts machine codes into target processor specific Assembly codes/instructions. The process of converting machine codes into Assembly code is known as 'Disassembling'. In operation, disassembling is complementary to assembling/cross-assembling. Decompiler is the utility program for translating machine codes into corresponding high level language instructions. Decompiler performs the reverse operation of compiler/cross-compiler. The disassemblers/decompilers for different family of processors/controllers are different. Disassemblers/Decompilers are deployed in reverse engineering. Reverse engineering is the process of revealing the technology behind the working of a product. Reverse engineering in Embedded Product development is employed to find out the secret behind the working of popular proprietary products. Disassemblers/decompilers help the reverse-engineering process by translating the embedded firmware into Assembly/high level language instructions.

https://hemanthrajhemu.github.io

Disassemblers/Decompilers are powerful tools for analysing the presence of malicious codes (virus information) in an executable image. Disassemblers/Decompilers are available as either freeware tools readily available for free download from internet or as commercial tools. It is not possible for a disassembler/decompiler to generate an exact replica of the original assembly code/high level source code in terms of the symbolic constants and comments used. However disassemblers/decompilers generate a source code which is somewhat matching to the original source code from which the binary code is generated.

13.4 SIMULATORS, EMULATORS AND DEBUGGING

Simulators and emulators are two important tools used in embedded system development. Both the terms sound alike and are little confusing. Simulator is a software tool used for simulating the various conditions for checking the functionality of the application firmware. The Integrated Development Environment (IDE) itself will be providing simulator support and they help in debugging the firmware for checking its required functionality. In certain scenarios, simulator refers to a soft model (GUI model) of the embedded product. For example, if the product under development is a handheld device, to test the functionalities of the various menu and user interfaces, a soft form model of the product with all UI as given in the end product can be developed in software. Soft phone is an example for such a simulator. Emulator is hardware device which emulates the functionalities of the target device and allows real time debugging of the embedded firmware in a hardware environment.

13.4.1 Simulators

In a previous section of this chapter, describing the Integrated Development Environment, we discussed about simulators for embedded firmware debugging. Simulators simulate the target hardware and the firmware execution can be inspected using simulators. The features of simulator based debugging are listed below.

1. Purely software based
2. Doesn't require a real target system
3. Very primitive (Lack of featured I/O support. Everything is a simulated one)
4. Lack of Real-time behaviour

13.4.1.1 Advantages of Simulator Based Debugging Simulator based debugging techniques are simple and straightforward. The major advantages of simulator based firmware debugging techniques are explained below.

No Need for Original Target Board Simulator based debugging technique is purely software oriented. IDE's software support simulates the CPU of the target board. User only needs to know about the memory map of various devices within the target board and the firmware should be written on the basis of it. Since the real hardware is not required, firmware development can start well in advance immediately after the device interface and memory maps are finalised. This saves development time.

Simulate I/O Peripherals Simulator provides the option to simulate various I/O peripherals. Using simulator's I/O support you can edit the values for I/O registers and can be used as the input/output value in the firmware execution. Hence it eliminates the need for connecting I/O devices for debugging the firmware.

https://hemanthrajhemu.github.io

Simulates Abnormal Conditions With simulator's simulation support you can input any desired value for any parameter during debugging the firmware and can observe the control flow of firmware. It really helps the developer in simulating abnormal operational environment for firmware and helps the firmware developer to study the behaviour of the firmware under abnormal input conditions.

13.4.1.2 Limitations of Simulator based Debugging Though simulation based firmware debugging technique is very helpful in embedded applications, they possess certain limitations and we cannot fully rely upon the simulator-based firmware debugging. Some of the limitations of simulator-based debugging are explained below.

Deviation from Real Behaviour Simulation-based firmware debugging is always carried out in a development environment where the developer may not be able to debug the firmware under all possible combinations of input. Under certain operating conditions we may get some particular result and it need not be the same when the firmware runs in a production environment.

Lack of real timeliness The major limitation of simulator based debugging is that it is not real-time in behaviour. The debugging is developer driven and it is no way capable of creating a real time behaviour. Moreover in a real application the I/O condition may be varying or unpredictable. Simulation goes for simulating those conditions for known values.

13.4.2 Emulators and Debuggers

What is debugging and why debugging is required? Debugging in embedded application is the process of diagnosing the firmware execution, monitoring the target processor's registers and memory while the firmware is running and checking the signals from various buses of the embedded hardware. Debugging process in embedded application is broadly classified into two, namely; hardware debugging and firmware debugging. Hardware debugging deals with the monitoring of various bus signals and checking the status lines of the target hardware. The various tools used for hardware debugging will be explaining in detail in a later section of this chapter. Firmware debugging deals with examining the firmware execution, execution flow, changes to various CPU registers and status registers on execution of the firmware to ensure that the firmware is running as per the design. This section deals with the debugging of firmware.

Why is debugging required? Well the counter question why you go for diagnosis when you are ill answers this query. Firmware debugging is performed to figure out the bug or the error in the firmware which creates the unexpected behaviour. Firmware is analogous to the human body in the sense it is widespread and/or modular. Any abnormalities in any area of the body may lead to sickness. How is the region causing illness identified correctly when you are sick? If we look back to the 1900s, where no sophisticated diagnostic techniques were available, only a skilled doctor was capable of identifying the root cause of illness, that too with his solid experience. Now, with latest technologies, the scenario is totally changed. Sophisticated diagnostic techniques provide offline diagnosis like Computerized Tomography (CT), MRI and ultrasound scans and online diagnosis like micro camera based imaging techniques. With the intrusion of a micro camera into the body, the doctors can view the internals of the body in real time.

During the early days of embedded system development, there were no debug tools available and the only way was "*Burn the code in an EEPROM and pray for its proper functioning*". If the firmware does not crash, the product works fine. If the product crashes, the developer is unlucky and he needs to sit back and rework on the firmware till the product functions in the expected way. Most of the time

https://hemanthrajhemu.github.io

the developer had to seek the help of an expert to figure out the exact problem creator. As technology has achieved a new dimension from the early days of embedded system development, various types of debugging techniques are available today. The following section describes the improvements over firmware debugging starting from the most primitive type of debugging to the most sophisticated On Chip Debugging (OCD).

13.4.2.1 Incremental EEPROM Burning Technique This is the most primitive type of firmware debugging technique where the code is separated into different functional code units. Instead of burning the entire code into the EEPROM chip at once, the code is burned in incremental order, where the code corresponding to all functionalities are separately coded, cross-compiled and burned into the chip one by one. The code will incorporate some indication support like lighting up an "LED (every embedded product contains at least one LED). If not, you should include provision for at least one LED in the target board at the hardware design time such that it can be used for debugging purpose)" or activate a "BUZZER (In a system with BUZZER support)" if the code is functioning in the expected way. If the first functionality is found working perfectly on the target board with the corresponding code burned into the EEPROM, go for burning the code corresponding to the next functionality and check whether it is working. Repeat this process till all functionalities are covered. Please ensure that before entering into one level up, the previous level has delivered a correct result. If the code corresponding to any functionality is found not giving the expected result, fix it by modifying the code and then only go for adding the next functionality for burning into the EEPROM. After you found all functionalities working properly, combine the entire source for all functionalities together, re-compile and burn the code for the total system functioning.

Obviously it is a time-consuming process. But remember it is a onetime process and once you test the firmware in an incremental model you can go for mass production. In incremental firmware burning technique we are not doing any debugging but observing the status of firmware execution as a debug method. The very common mistake committed by firmware developers in developing non-operating system-based embedded application is burning the entire code altogether and fed up with debugging the code. Please don't adopt this approach. Even though you need to spend some additional time on incremental burning approach, you will never lose in the process and will never mess up with debugging the code. You will be able to figure out at least 'on which point of firmware execution the issue is arising'–"A stitch in time saves nine". Incremental firmware burning technique is widely adopted in small, simple system developments and in product development where time is not a big constraint (e.g. R&D projects). It is also very useful in product development environments where no other debug tools are available.

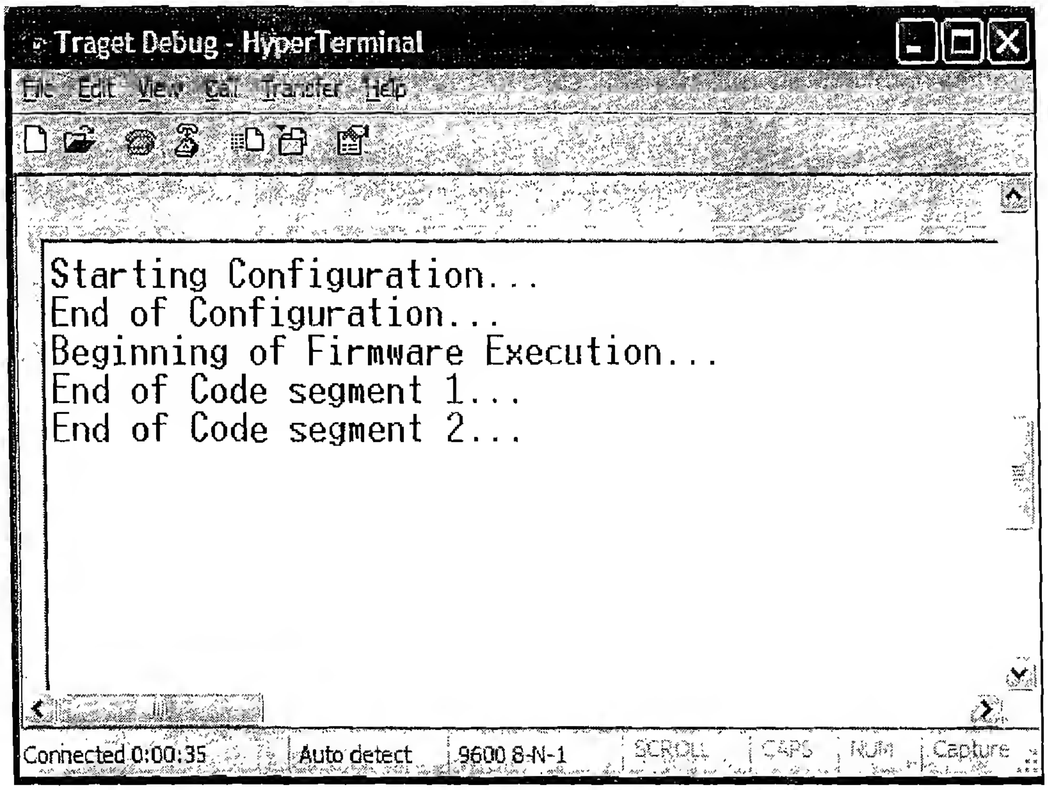
13.4.2.2 Inline Breakpoint Based Firmware Debugging Inline breakpoint based debugging is another primitive method of firmware debugging. Within the firmware where you want to ensure that firmware execution is reaching up to a specified point, insert an inline debug code immediately after the point. The debug code is a *printf()* function which prints a string given as per the firmware. You can insert debug codes (*printf()*) commands at each point where you want to ensure the firmware execution is covering that point. Cross-compile thc source code with the debug codes embedded within it. Burn the corresponding hex file into the EEPROM. You can view the *printf()* generated data on the '*HyperTerminal*—A communication facility available with the Windows OS coming under the *Communications* section of *Start Menu*' of the Development PC. Configure the serial communication settings of the '*HyperTerminal*' connection to the same as that of the serial communication settings configured in the firmware (Say Baudrate = 9600; Parity = None; Stop Bit = 1; Flow Control = None); Connect the

https://hemanthrajhemu.github.io

target board's serial port (COM) to the development PC's COM Port using an RS232 Cable. Power up the target board. Depending on the execution flow of firmware and the inline debug codes inserted in the firmware, you can view the debug information on the '*HyperTerminal*'. Typical usage of inline debug codes and the debug info retrieved on the *HyperTerminal* is illustrated below.

```
//First Inline Debug Code
printf ("Starting Configuration…\n");
Configurations……
……………………
//Inline Debug code ensuring execution of Configuration section
printf ("End of Configuration…\n");
printf ("Beginning of Firmware Execution…\n");
Code segment1….
…………………
//Inline Debug code ensuring execution of Code Segment 1
printf ("End of Code segment 1…\n");
Code segment2….
…………………
//Inline Debug code ensuring execution of Code Segment 2
printf ("End of Code segment 2…\n");
```

If the firmware is error free and the execution occurs properly, you will get all the debug messages on the *HyperTerminal*. Based on this debug info you can check the firmware for errors (Fig. 13.38).


Traget Debug - HyperTerminal
File Edit View Call Transfer Help
Connected 0:00:35
Auto detect
9600 8-N-1
SCROLL
CAPS
NUM
Capture

```
Starting Configuration...
End of Configuration...
Beginning of Firmware Execution...
End of Code segment 1...
End of Code segment 2...
```

Fig. 13.38 **Inline Breakpoint Based Firmware Debugging with HyperTerminal**

https://hemanthrajhemu.github.io

13.4.2.3 Monitor Program Based Firmware Debugging Monitor program based firmware debugging is the first adopted invasive method for firmware debugging (Fig. 13.39). In this approach a monitor program which acts as a supervisor is developed. The monitor program controls the downloading of user code into the code memory, inspects and modifies register/memory locations; allows single stepping of source code, etc. The monitor program implements the debug functions as per a pre-defined command set from the debug application interface. The monitor program always listens to the serial port of the target device and according to the command received from the serial interface it performs command specific actions like firmware downloading, memory inspection/modification, firmware single stepping and sends the debug information (various register and memory contents) back to the main debug program running on the development PC, etc. The first step in any monitor program development is determining a set of commands for performing various operations like firmware downloading, memory/register inspection/modification, single stepping, etc. Once the commands for each operation is fixed, write the code for performing the actions corresponding to these commands. As mentioned earlier, the commands may be received through any of the external interface of the target processor (e.g. RS-232C serial interface/parallel interface/USB, etc.). The monitor program should query this interface to get commands or should handle the command reception if the data reception is implemented through interrupts. On receiving a command, examine it and perform the action corresponding to it. The entire code stuff handling the command reception and corresponding action implementation is known as the "*monitor program*". The most common type of interface used between target board and debug application is RS-232C Serial interface. After the successful completion of the '*monitor program*' development, it is compiled and burned into the FLASH memory or ROM of the target board. The code memory containing the monitor program is known as the '*Monitor ROM*'.

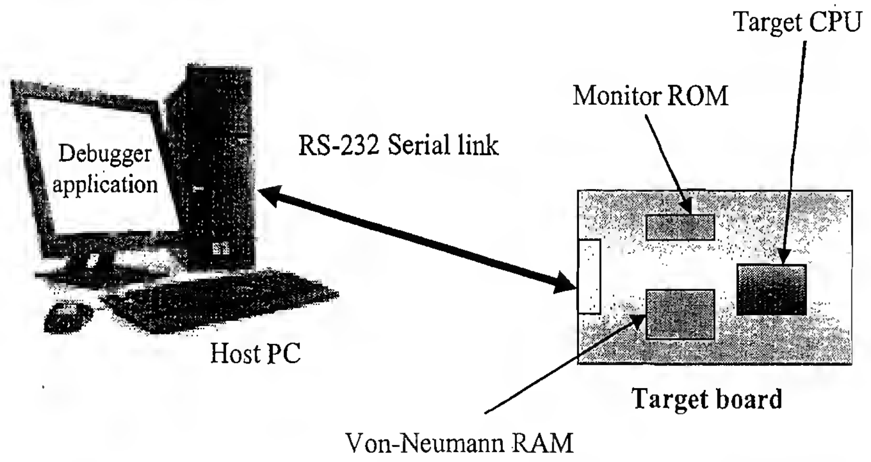


Fig. 13.39 **Monitor Program Based Target Firmware Debug Setup**

The monitor program contains the following set of minimal features.

1. Command set interface to establish communication with the debugging application
2. Firmware download option to code memory
3. Examine and modify processor registers and working memory (RAM)
4. Single step program execution
5. Set breakpoints in firmware execution
6. Send debug information to debug application running on host machine

https://hemanthrajhemu.github.io

The monitor program usually resides at the reset vector (code memory 0000H) of the target processor. The monitor program is commonly employed in development boards and the development board supplier provides the monitor program in the form of a ROM chip. The actual code memory is downloaded into a RAM chip which is interfaced to the processor in the Von-Neumann architecture model. The Von-Neumann architecture model is achieved by ANDing the PSEN\ and RD\ signals of the target processor (In case of *8051*) and connecting the output of AND Gate to the Output Enable (RD\) pin of RAM chip. WR\ signal of the target processor is interfaced to The WR\ signal of the Von Neumann RAM. Monitor ROM size varies in the range of a few kilo bytes (Usually 4K). An address decoder circuit maps the address range allocated to the monitor ROM and activates the Chip Select (CS\) of the ROM if the address is within the range specified for the Monitor ROM. A user program is normally loaded at locations 0x4000 or 0x8000. The address decoder circuit ensures the enabling of the RAM chip (CS\) when the address range is outside that allocated to the ROM monitor. Though there are two memory chips (Monitor ROM Chip and Von-Neumann RAM), the total memory map available for both of them will be 64K for a processor/controller with 16bit address space and the memory decoder units take care of avoiding conflicts in accessing both. While developing user program for monitor ROM-based systems, special care should be taken to offset the user code and handling the interrupt vectors. The target development IDE will help in resolving this. During firmware execution and single stepping, the user code may have to be altered and hence the firmware is always downloaded into a Von-Neumann RAM in monitor ROM-based debugging systems. Monitor ROM-based debugging is suitable only for development work and it is not a good choice for mass produced systems. The major drawbacks of monitor based debugging system are

1. The entire memory map is converted into a Von-Neumann model and it is shared between the monitor ROM, monitor program data memory, monitor program trace buffer, user written firmware and external user memory. For *8051*, the original Harvard architecture supports 64K code memory and 64K external data memory (Total 128K memory map). Going for a monitor based debugging shrinks the total available memory to 64K Von-Neumann memory and it needs to accommodate all kinds of memory requirement (Monitor Code, monitor data, trace buffer memory, User code and External User data memory).
2. The communication link between the debug application running on Development PC and monitor program residing in the target system is achieved through a serial link and usually the controller's On-chip UART is used for establishing this link. Hence one serial port of the target processor becomes dedicated for the monitor application and it cannot be used for any other device interfacing. Wastage of a serial port! It is a serious issue in controllers or processors with single UART.

13.4.2.4 In Circuit Emulator (ICE) Based Firmware Debugging The terms 'Simulator' and 'Emulator' are little bit confusing and sounds similar. Though their basic functionality is the same – "Debug the target firmware", the way in which they achieve this functionality is totally different. As mentioned before, 'Simulator' is a software application that precisely duplicates (mimics) the target CPU and simulates the various features and instructions supported by the target CPU, whereas an 'Emulator' is a self-contained hardware device which emulates the target CPU. The emulator hardware contains necessary emulation logic and it is hooked to the debugging application running on the development PC on one end and connects to the target board through some interface on the other end. In summary, the simulator '*simulates*' the target board CPU and the emulator '*emulates*' the target board CPU.

There is a scope change that has happened to the definition of an emulator. In olden days emulators were defined as special hardware devices used for emulating the functionality of a processor/controller

https://hemanthrajhemu.github.io

and performing various debug operations like halt firmware execution, set breakpoints, get or set internal RAM/CPU register, etc. Nowadays pure software applications which perform the functioning of a hardware emulator is also called as 'Emulators' (though they are 'Simulators' in operation). The emulator application for emulating the operation of a PDA phone for application development is an example of a 'Software Emulator'. A hardware emulator is controlled by a debugger application running on the development PC. The debugger application may be part of the Integrated Development Environment (IDE) or a third party supplied tool. Most of the IDEs incorporate debugger support for some of the emulators commonly available in the market. The emulators for different families of processors/controllers are different. Figure 13.40 illustrates the different subsystems and interfaces of an 'Emulator' device.

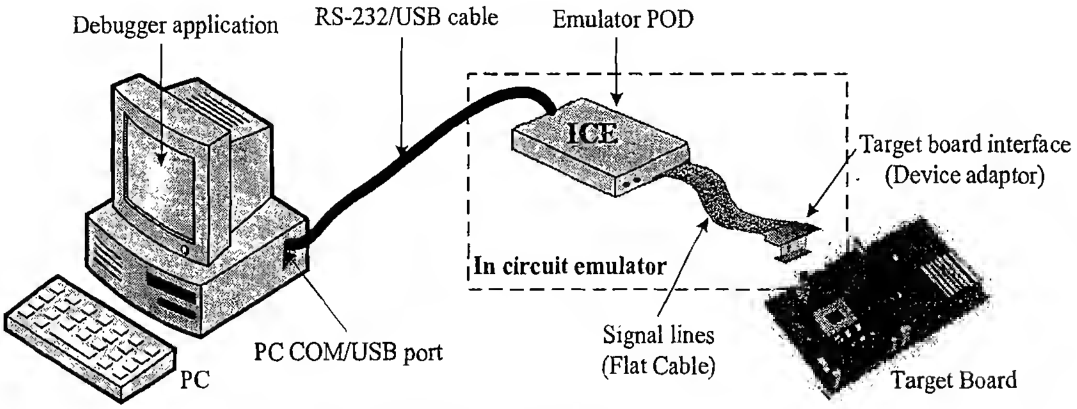


Fig. 13.40 **In Circuit Emulator (ICE) Based Target Debugging**

The Emulator POD forms the heart of any emulator system and it contains the following functional units.

Emulation Device Emulation device is a replica of the target CPU which receives various signals from the target board through a device adaptor connected to the target board and performs the execution of firmware under the control of debug commands from the debug application. The emulation device can be either a standard chip same as the target processor (e.g. AT89C51) or a Programmable Logic Device (PLD) configured to function as the target CPU. If a standard chip is used as the emulation device, the emulation will provide real-time execution behaviour. At the same time the emulator becomes dedicated to that particular device and cannot be re-used for the derivatives of the same chip. PLD-based emulators can easily be re-configured to use with derivatives of the target CPU under consideration. By simply loading the configuration file of the derivative processor/controller, the PLD gets re-configured and it functions as the derivative device. A major drawback of PLD-based emulator is the accuracy of replication of target CPU functionalities. PLD-based emulator logic is easy to implement for simple target CPUs but for complex target CPUs it is quite difficult.

Emulation Memory It is the Random Access Memory (RAM) incorporated in the Emulator device. It acts as a replacement to the target board's EEPROM where the code is supposed to be downloaded after each firmware modification. Hence the original EEPROM memory is emulated by the RAM of emulator. This is known as '*ROM Emulation*'. ROM emulation eliminates the hassles of ROM burning and it offers the benefit of infinite number of reprogrammings (Most of the EEPROM chips available

https://hemanthrajhemu.github.io

in the market supports only 100 to 1000 re-program cycles). Emulation memory also acts as a trace buffer in debugging. Trace buffer is a memory pool holding the instructions executed/registers modified/related data by the processor while debugging. The trace buffer size is emulator dependent and the trace buffer holds the recent trace information when the buffer overflows. The common features of trace buffer memory and trace buffer data viewing are listed below:

- Trace buffer records each bus cycle in frames
- Trace data can be viewed in the debugger application as Assembly/Source code
- Trace buffering can be done on the basis of a Trace trigger (Event)
- Trace buffer can also record signals from target board other than CPU signals (Emulator dependent)
- Trace data is a very useful information in firmware debugging

Emulator Control Logic Emulator control logic is the logic circuits used for implementing complex hardware breakpoints, trace buffer trigger detection, trace buffer control, etc. Emulator control logic circuits are also used for implementing logic analyser functions in advanced emulator devices. The 'Emulator POD' is connected to the target board through a 'Device adaptor' and signal cable.

Device Adaptors Device adaptors act as an interface between the target board and emulator POD. Device adaptors are normally pin-to-pin compatible sockets which can be inserted/plugged into the target board for routing the various signals from the pins assigned for the target processor. The device adaptor is usually connected to the emulator POD using ribbon cables. The adaptor type varies depending on the target processor's chip package. DIP, PLCC, etc. are some commonly used adaptors.

The above-mentioned emulators are almost dedicated ones, meaning they are built for emulating a specific target processor and have little or less support for emulating the derivatives of the target processor for which the emulator is built. This type of emulators usually combines the entire emulation control logic and emulation device (if present) in a single board. They are known as '*Debug Board Modules (DBMs)*'. An alternative method of emulator design supports emulation of a variety of target processors. Here the emulator hardware is partitioned into two, namely, '*Base Terminal*' and '*Probe Card*'. The Base terminal contains all the emulator hardware and emulation control logic except the emulation chip (Target board CPU's replica). The base terminal is connected to the Development PC for establishing communication with the debug application. The emulation chip (Same chip as the target CPU) is mounted on a separate PCB and it is connected to the base terminal through a ribbon cable. The '*Probe Card*' board contains the device adaptor sockets to plug the board into the target development board. The board containing the emulation chip is known as the '*Probe Card*'. For emulating different target CPUs the '*Probe Card*' will be different and the base terminal remains the same. The manufacturer of the emulator supplies '*Probe Card*' for different CPUs. Though these emulators are capable of emulating different CPUs, the cost for '*Probe Cards*' is very high. Communication link between the emulator base unit/ Emulator POD and debug application is established through a Serial/Parallel/USB interface. Debug commands and debug information are sent to and from the emulator using this interface.

13.4.2.5 On Chip Firmware Debugging (OCD) Advances in semiconductor technology has brought out new dimensions to target firmware debugging. Today almost all processors/controllers incorporate built in debug modules called On Chip Debug (OCD) support. Though OCD adds silicon complexity and cost factor, from a developer perspective it is a very good feature supporting fast and efficient firmware debugging. The On Chip Debug facilities integrated to the processor/controller are chip vendor dependent and most of them are proprietary technologies like Background Debug Mode

https://hemanthrajhemu.github.io

(BDM), OnCE, etc. Some vendors add 'on chip software debug support' through JTAG (Joint Test Action Group) port. Processors/controllers with OCD support incorporate a dedicated debug module to the existing architecture. Usually the on-chip debugger provides the means to set simple breakpoints, query the internal state of the chip and single step through code. OCD module implements dedicated registers for controlling debugging. An On Chip Debugger can be enabled by setting the OCD enable bit (The bit name and register holding the bit varies across vendors). Debug related registers are used for debugger control (Enable/disable single stepping, Freeze execution, etc.) and breakpoint address setting. BDM and JTAG are the two commonly used interfaces to communicate between the Debug application running on Development PC and OCD module of target CPU. Some interface logic in the form of hardware will be implemented between the CPU OCD interface and the host PC to capture the debug information from the target CPU and sending it to the debugger application running on the host PC. The interface between the hardware and PC may be Serial/Parallel/USB. The following section will give you a brief introduction about Background Debug Mode (BDM) and JTAG interface used in On Chip Debugging.

Background Debug Mode (BDM) interface is a proprietary On Chip Debug solution from Motorola. BDM defines the communication interface between the chip resident debug core and host PC where the BDM compatible remote debugger is running. BDM makes use of 10 or 26 pin connector to connect to the target board. Serial data in (DSI), Serial data out (DSO) and Serial clock (DSCLK) are the three major signal lines used in BDM. DSI sends debug commands serially to the target processor from the remote debugger application and DSO sends the debug response to the debugger from the processor. Synchronisation of serial transmission is done by the serial clock DSCLK generated by the debugger application. Debugging is controlled by BDM specific debug commands. The debug commands are usually 17-bit wide. 16 bits are used for representing the command and 1 bit for status/control.

Chips with JTAG debug interface contain a built-in JTAG port for communicating with the remote debugger application. JTAG is the acronym for Joint Test Action Group. JTAG is the alternate name for IEEE 1149.1 standard. Like BDM, JTAG is also a serial interface. The signal lines of JTAG protocol are explained below.

Test Data In (TDI): It is used for sending debug commands serially from remote debugger to the target processor.

Test Data Out (TDO): Transmit debug response to the remote debugger from target CPU.

Test Clock (TCK): Synchronises the serial data transfer.

Test Mode Select (TMS): Sets the mode of testing.

Test Reset (TRST): It is an optional signal line used for resetting the target CPU.

The serial data transfer rate for JTAG debugging is chip dependent. It is usually within the range of 10 to 1000 MHz.

13.5 TARGET HARDWARE DEBUGGING

Even though the firmware is bug free and everything is intact in the board, your embedded product need not function as per the expected behaviour in the first attempt for various hardware related reasons like dry soldering of components, missing connections in the PCB due to any un-noticed errors in the PCB layout design, misplaced components, signal corruption due to noise, etc. The only way to sort out these issues and figure out the real problem creator is debugging the target board. Hardware debugging is not similar to firmware debugging. Hardware debugging involves the monitoring of various signals

https://hemanthrajhemu.github.io

of the target board (address/data lines, port pins, etc.), checking the inter. connection among various components, circuit continuity checking, etc. The various hardware debugging tools used in Embedded Product Development are explained below.

13.5.1 Magnifying Glass (Lens)

You might have noticed watch repairer wearing a small magnifying glass while engaged in repairing a watch. They use the magnifying glass to view the minute components inside the watch in an enlarged manner so that they can easily work with them. Similar to a watch repairer, magnifying glass is the primary hardware debugging tool for an embedded hardware debugging professional. A magnifying glass is a powerful visual inspection tool. With a magnifying glass (lens), the surface of the target board can be examined thoroughly for dry soldering of components, missing components, improper placement of components, improper soldering, track (PCB connection) damage, short of tracks, etc. Nowadays high quality magnifying stations are available for visual inspection. The magnifying station incorporates magnifying glasses attached to a stand with CFL tubes for providing proper illumination for inspection. The station usually incorporates multiple magnifying lenses. The main lens acts as a visual inspection tool for the entire hardware board whereas the other small lens within the station is used for magnifying a relatively small area of the board which requires thorough inspection.

13.5.2 Multimeter

I believe the name of the instrument itself is sufficient to give an outline of its usage. A multimeter is used for measuring various electrical quantities like voltage (Both AC and DC), current (DC as well as AC), resistance, capacitance, continuity checking, transistor checking, cathode and anode identification of diode, etc. Any multimeter will work over a specific range for each measurement. A multimeter is the most valuable tool in the toolkit of an embedded hardware developer. It is the primary debugging tool for physical contact based hardware debugging and almost all developers start debugging the hardware with it. In embedded hardware debugging it is mainly used for checking the circuit continuity between different points on the board, measuring the supply voltage, checking the signal value, polarity, etc. Both analog and digital versions of a multimeter are available. The digital version is preferred over analog the one for various reasons like readability, accuracy, etc. Fluke, Rishab, Philips, etc. are the manufacturers of commonly available high quality digital multimeters.

13.5.3 Digital CRO

Cathode Ray Oscilloscope (CRO) is a little more sophisticated tool compared to a multimeter. You might have studied the operation and use of a CRO in your basic electronics lab. Just to refresh your brain, CRO is used for waveform capturing and analysis, measurement of signal strength, etc. By connecting the point under observation on the target board to the Channels of the Oscilloscope, the waveforms can be captured and analysed for expected behaviour. CRO is a very good tool in analysing interference noise in the power supply line and other signal lines. Monitoring the crystal oscillator signal from the target board is a typical example of the usage of CRO for waveform capturing and analysis in target board debugging. CROs are available in both analog and digital versions. Though Digital CROs are costly, featurewise they are best suited for target board debugging applications. Digital CROs are available for high frequency support and they also incorporate modern techniques for recording waveform over a period of time, capturing waves on the basis of a configurable event (trigger) from the target board

https://hemanthrajhemu.github.io

(e.g. High to low transition of a port pin of the target processor). Most of the modern digital CROs contain more than one channel and it is easy to capture and analyse various signals from the target board using multiple channels simultaneously. Various measurements like phase, amplitude, etc. is also possible with CROs. Tektronix, Agilent, Philips, etc. are the manufacturers of high precision good quality digital CROs.

13.5.4 Logic Analyser

A logic analyser is the big brother of digital CRO. Logic analyser is used for capturing digital data (logic 1 and 0) from a digital circuitry whereas CRO is employed in capturing all kinds of waves including logic signals. Another major limitation of CRO is that the total number of logic signals/waveforms that can be captured with a CRO is limited to the number of channels. A logic analyser contains special connectors and clips which can be attached to the target board for capturing digital data. In target board debugging applications, a logic analyser captures the states of various port pins, address bus and data bus of the target processor/controller, etc. Logic analysers give an exact reflection of what happens when a particular line of firmware is running. This is achieved by capturing the address line logic and data line logic of target hardware. Most modern logic analysers contain provisions for storing captured data, selecting a desired region of the captured waveform, zooming selected region of the captured waveform, etc. Tektronix, Agilent, etc. are the giants in the logic analyser market.

13.5.5 Function Generator

Function generator is not a debugging tool. It is an input signal simulator tool. A function generator is capable of producing various periodic waveforms like sine wave, square wave, saw-tooth wave, etc. with different frequencies and amplitude. Sometimes the target board may require some kind of periodic waveform with a particular frequency as input to some part of the board. Thus, in a debugging environment, the function generator serves the purpose of generating and supplying required signals.

13.6 BOUNDARY SCAN

As the complexity of the hardware increase, the number of chips present in the board and the interconnection among them may also increase. The device packages used in the PCB become miniature to reduce the total board space occupied by them and multiple layers may be required to route the interconnections among the chips. With miniature device packages and multiple layers for the PCB it will be very difficult to debug the hardware using magnifying glass, multimeter, etc. to check the interconnection among the various chips. Boundary scan is a technique used for testing the interconnection among the various chips, which support JTAG interface, present in the board. Chips which support boundary scan associate a boundary scan cell with each pin of the device. A JTAG port which contains the five signal lines namely TDI, TDO, TCK, TRST and TMS form the Test Access Port (TAP) for a JTAG supported chip. Each device will have its own TAP. The PCB also contains a TAP for connecting the JTAG signal lines to the external world. A boundary scan path is formed inside the board by interconnecting the devices through JTAG signal lines. The TDI pin of the TAP of the PCB is connected to the TDI pin of the first device. The TDO pin of the first device is connected to the TDI pin of the second device. In this way all devices are interconnected and the TDO pin of the last JTAG device is connected to the TDO pin of the TAP of the PCB. The clock line TCK and the Test Mode Select (TMS) line of the devices are

https://hemanthrajhemu.github.io

connected to the clock line and Test mode select line of the Test Access Port of the PCB respectively. This forms a boundary scan path. Figure 13.41 illustrates the same.

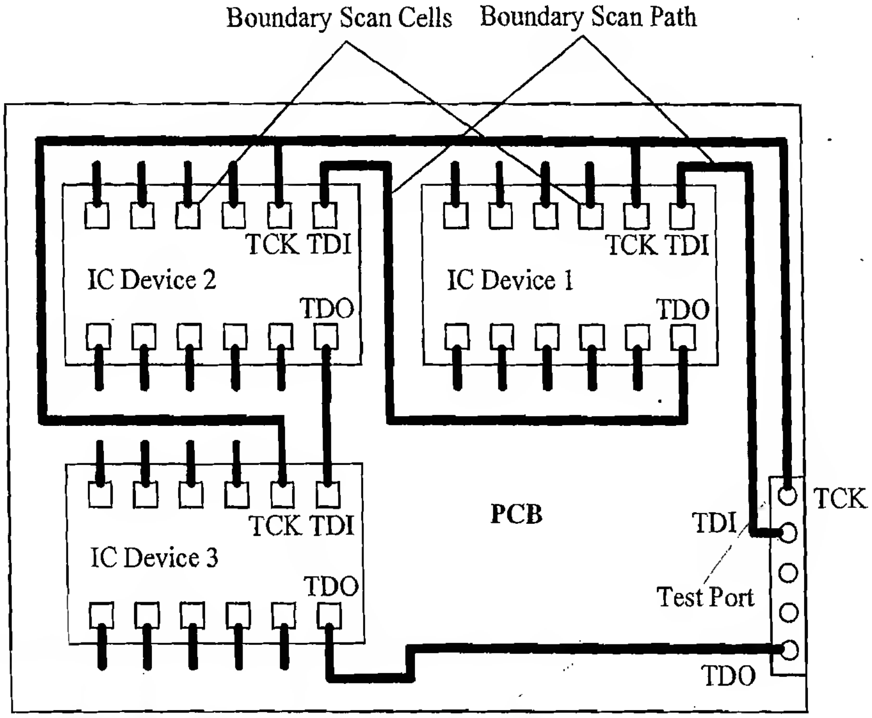


Fig. 13.41 JTAG based boundary scanning for hardware testing

As mentioned earlier, each pin of the device associates a boundary scan cell with it. The boundary scan cell is a multipurpose memory cell. The boundary scan cell associated with the input pins of an IC is known as 'input cells' and the boundary scan cells associated with the output pins of an IC is known as 'output cells'. The boundary scan cells can be used for capturing the input pin signal state and passing it to the internal circuitry, capturing the signals from the internal circuitry and passing it to the output pin, and shifting the data received from the Test Data In pin of the TAP. The boundary scan cells associated with the pins are interconnected and they form a chain from the TDI pin of the device to its TDO pin. The boundary scan cells can be operated in Normal, Capture, Update and Shift modes. In the Normal mode, the input of the boundary scan cell appears directly at its output. In the Capture mode, the boundary scan cell associated with each input pin of the chip captures the signal from the respective pins to the cell and the boundary scan cell associated with each output pin of the chip captures the signal from the internal circuitry. In the Update mode, the boundary scan cell associated with each input pin of the chip passes the already captured data to the internal circuitry and the boundary scan cell associated with each output pin of the chip passes the already captured data to the respective output pin. In the shift mode, data is shifted from TDI pin to TDO pin of the device through the boundary scan cells. ICs supporting boundary scan contain additional boundary scan related registers for facilitating the boundary scan operation. Instruction Register, Bypass Register, Identification Register, etc. are examples of boundary scan related registers. The Instruction Register is used for holding and processing the instruction received over the TAP. The bypass register is used for bypassing the boundary scan path of the device and directly interconnecting the TDI pin of the device to its TDO. It disconnects a device from the boundary scan path. Different instructions are used for testing the interconnections and the functioning of the

https://hemanthrajhemu.github.io

chip. *Extest*, *Bypass*, *Sample* and *Preload*, *Intest*, etc. are examples for instructions for different types of boundary scan tests, whereas the instruction *Runbist* is used for performing a self test on the internal functioning of the chip. The *Runbist* instruction produces a pass/fail result.

Boundary Scan Description Language (*BSDL*) is used for implementing boundary scan tests using JTAG. BSDL is a subset of VHDL and it describes the JTAG implementation in a device. BSDL provides information on how boundary scan is implemented in an integrated chip. The BSDL file (File which describes the boundary scan implementation for a device in *.bsd* format) for a JTAG compliant device is supplied the device manufacturers or it can be downloaded from internet repository. The BSDL file is used as the input to a Boundary Scan Tool for generating boundary scan test cases for a PCB. Automated tools are available for boundary scan test implementation from multiple vendors. The ScanExpress™ Boundary Scan (JTAG) product from Corelis Inc. (www.corelis.com) is a popular tool for boundary scan test implementation.

Summary

- ✓ Integrated Development Environment (IDE) is an integrated environment for developing and debugging the target processor specific embedded firmware. IDE is a software package which bundles a 'Text Editor (Source Code Editor)', 'Cross-compiler (for cross platform development and compiler for same platform development)', 'Linker' and a 'Debugger'
- ✓ Keil µVision3 is a licensed IDE tool from Keil Software (www.keil.com), an ARM company, for *8051* family microcontroller based embedded firmware development
- ✓ *List File* (.lst), *Pre-processor Output file*, *Map File* (File extension linker dependent), *Object File* (.obj), *Hex File* (.hex). etc. are the files generated during the cross-compilation process of a source file
- ✓ Hex file is the binary executable file created from the source code. The absolute object file created by the linker/locater is converted into processor understandable binary code. Object to Hex file converter is the utility program for converting an object file to a hex file
- ✓ *Intel HEX* and *Motorola HEX* are the two commonly used Hex file formats in embedded applications
- ✓ *Disassembler* is a utility program which converts machine codes into target processor specific Assembly codes/instructions. *Disassembling* is the process of converting machine codes into Assembly code
- ✓ *Decompiler* is the utility program for translating machine codes into corresponding high level language instructions
- ✓ *Simulator* is a software application that precisely duplicates (mimics) the target CPU and simulates the various features and instructions supported by the target CPU
- ✓ *Emulator* is a self-contained hardware device which emulates the target CPU. Emulator hardware contains necessary emulation logic and it is hooked to the debugging application running on the development PC on one end and connects to the target board through some interface on the other end
- ✓ Incremental EEPROM Burning technique, Inline breakpoint based Firmware Debugging, Monitor Program based Firmware Debugging, In Circuit Emulator (ICE) based Firmware Debugging and On Chip Firmware Debugging (OCD) are the techniques used for debugging embedded firmware on the target hardware
- ✓ Background Debug Mode (BDM) Interface and JTAG are the two commonly used interfaces for On Chip Firmware Debugging (OCD)
- ✓ Magnifying glass, Multimeter, Cathode Ray Oscilloscope (CRO), Logic Analyser and Function generator are the commonly used hardware debugging tools
- ✓ Boundary scan is a technique for testing the interconnection among the various chips, which support boundary scanning, in a complex board containing too many interconnections and multiple planes for routing
- ✓ *Boundary Scan Description Language* (*BSDL*) is a language similar to VHDL, which describes the boundary

https://hemanthrajhemu.github.io

scan implementation of a device and is used for implementing boundary scan tests using JTAG. BSDL file is a file containing the boundary scan description for a device in boundary scan description language, which is supplied as input to a Boundary scan tool for generating the boundary scan test cases

Keywords

Integrated Development Environment (IDE) : A software package which bundles a 'Text Editor (Source Code Editor)', 'Cross-compiler (for cross platform development and compiler for same platform development)', 'Linker' and a 'Debugger'

Keil μVision3 : A licensed IDE tool from Keil Software (www.keil.com), an ARM company, for 8051 family microcontroller based embedded firmware development

Listing file (.LST File) : File generated during cross compilation of a source code and it contains information about the cross compilation process, like cross compiler details, formatted source text ('C' code), assembly code generated from the source file, symbol tables, errors and warnings detected during the cross-compilation process, etc.

Object File (.OBJ File) : A specially formatted file with data records for symbolic information, object code, debugging information, library references, etc. generated during the cross-compilation of a source file

Map file : File generated during cross-compilation process and it contains information about the link/locate process

Hex file : The binary executable file created from the source code

Intel HEX : HEX file representation format

Motorola HEX : HEX file representation format

Disassembler : Utility program which converts machine codes into target processor specific Assembly codes/instructions.

Decompiler : Utility program for translating machine codes into corresponding high level language instructions

Simulator : Software application that precisely duplicates (mimics) the target CPU and simulates the various features and instructions supported by the target CPU

Monitor Program : Program which acts as a supervisor and controls the downloading of user code into the code memory, inspects and modifies register/memory locations, allows single stepping of source code, etc.

In Circuit Emulator (ICE) : A hardware device for emulating the target CPU for debug purpose

Debug Board Module (DBM) : ICE device which contains the emulation control logic and emulation chip in a single hardware unit and is designed for a particular family of device

Background Debug Mode (BDM) : A proprietary serial interface from Motorola for On Chip Debugging

JTAG : A serial interface for target board diagnostics and debugging

Boundary Scan : A target hardware debug method for checking the interconnections among the various chips of a complex board

Boundary Scan Description Language (BSDL) : A language similar to VHDL, which describes the boundary scan implementation of a JTAG supported device

https://hemanthrajhemu.github.io

Objective Questions

1 Which of the following intermediate file, generated during cross-compilation of an Embedded C file holds the assembly code generated corresponding to the c source code.
(a) List File (b) Preprocessor output file (c) Object file (d) Map file

2 Which of the following detail(s) is(are) kept in an object file generated during the process of cross-compiling an Embedded C file.
(a) Variable and function names (b) Variable and function reference
(c) Reserved memory for global variables (d) All of these
(e) None of these

3 Which of the following intermediate file, generated during the cross-compilation of an Embedded C files holds the information about the link/locate process for the multiple object modules of the project?
(a) List file (b) Preprocessor output file (c) Object file (d) Map file

4 Which of the following file generated during the cross-compilation process of an Embedded C project holds the machine code corresponding to the target processor?
(a) List file (b) Preprocessor output file (c) Object file (d) Map file

5 Examine the following Intel HEX Record
:03000000020C1FD0
This record is ?
(a) a Data Record (b) an End of File Record (c) a Segment Address Record
(d) an Extended Linear Address record

6 Examine the following Intel HEX record
:03000000020C1FD0
What is the number of data bytes in this record?
(a) 0 (b) 3 (c) 2 (d) 20

7 Examine the following Intel HEX record
:03000000020C1FD0
What is the start address of the data bytes in this record?
(a) 0x0000 (b) 0x3000 (c) 0x1FD0 (d) 0x20C1

8 Examine the following Intel HEX Record
:03000000020C1FD0
Which all are the data bytes present in this record?
(a) 03, 00, 00 (b) 02, 0C, 1F (c) 0C, 1F, D0 (d) 00, 00, 20

9 The program that converts machine codes into target processor specific Assembly code is known as
(a) Disassembler (b) Assembler (c) Cross-compiler (d) Decompiler

10 Which of the following is true about a *simulator* used in embedded software debugging?
(a) It is a software tool (b) It requires target hardware for simulation
(c) It doesn't require target hardware for simulation (d) (a) and (b) (e) (a) and (c)

11 Which of the following is an example for on chip firmware debugging?
(a) OnCE (b) BDM (c) All of these

Review Questions

1 Explain the various elements of an embedded system development environment.

2 Explain the role of *Integrated Development Environment* (IDE) for Embedded Software Development.

https://hemanthrajhemu.github.io

3 What are the different files generated during the cross-compilation of an Embedded C file? Explain them in detail.
4 Explain the various details held by a *List file* generated during the process of cross-compiling an Embedded C project.
5 Explain the various details held by a *Map file* generated during the process of cross-compiling an Embedded C project.
6 Explain the various details stored in an *Object file* generated during the cross-compilation of an Embedded C file.
7 Explain the difference between *Intel Hex* and *Motorola Hex* file format.
8 Explain the format of *Hex records* in an *Intel Hex* File.
9 Explain the format of *Hex records* in a *Motorola Hex* File.
10 What is the difference between an *assembler* and a *disassembler*? State their use in Embedded Application development.
11 What is a *decompiler*?
12 What is the difference between a *simulator* and an *emulator*?
13 Explain the advantages and limitations of *simulator* based debugging.
14 What are the different techniques available for embedded firmware debugging? Explain them in detail.
15 What is a *Monitor program*? Explain its role in embedded firmware debugging?
16 What is *ROM emulation*? Explain *In Circuit Emulator* (ICE) based debugging in detail.
17 Explain *On Chip Debugging* (OCD).
18 Explain the different tools used for hardware debugging.
19 Explain the *Boundary Scan* based hardware debugging in detail.

Lab Assignments

1 Write an Embedded C program for building a dancing LED circuit using *8051* as per the requirements given below
   (a) 4 Red LEDs are connected to Port Pins P1.0, P1.2, P1.4 and P1.6 respectively
   (b) 4 Green LEDs are connected to Port Pins P1.1, P1.3, P1.5 and P1.7 respectively
   (c) Turn On and Off the Green and Red LEDs alternatively with a delay of 100 milliseconds
   (d) Use a clock frequency of 12 MHz
   (e) Use Keil µVision 3 IDE and simulate the application for checking the firmware
   (f) Write the application separately for delay generation using timer and software routines
2 Implement the above requirement in Assembly Language
3 Write an Embedded C application for reading the status of 8 Dip switches connected to the Port P1 of the *8051* microcontroller using µVision 3 IDE. Debug the application using the simulator and simulate the state of DIP switches using the Port simulator for P1
4 Implement the above requirement in Assembly language
5 Write an Embedded C program to count the pulses appearing on the C/T0 input pin of *8051* microcontroller for a duration of 100 milliseconds using µVision 3 IDE
6 Implement the above requirement in Assembly language
7 Write an Embedded C program for configuring the INT0 Pin of the *8051* microcontroller as edge triggered and print the message "External 0 Interrupt in Progress" to the serial port of *8051* when the External 0 interrupt occurs. Use Keil µVision 3 IDE. Assume the crystal frequency for the controller as 11.0592 MHz. Configure the serial Port for baudrate 9600, No parity, 1 Start bit, 1 Stop bit and 8 data bits.
8 Implement the above requirement in Assembly language

https://hemanthrajhemu.github.io

9. Write an Interrupt Service Routine in Embedded C to handle the INT0 interrupt as per the requirements given below.
   (a) Use Register Bank 1
   (b) Read port P1 and save it in register R7
   (c) Use Keil µVision 3 IDE and simulate the application

10. Write an embedded 'C' program using Keil µViston 3 IDE to control the stepping of a unipolar 2-phase stepper motor in full step mode with a delay of 5 second between the steps. The stepper motor coils, A is connected to P1.0, B is connected to Port pins P1.1, C is connected to P1.2 and D to P1.3 of 8051 through NPN transistor driving circuits. Assume the clock frequency of operation as 12.00MHz.

11. Write an Embedded C program using Keil µVision 3 IDE to interface 8255 PPI device with *8051* microcontroller. The 8255 is memory interfaced to *8051* in the address map 8000H to FFFFH. Initialise the Port A, Port B and Port C of 8255 as Output ports in Mode0.

12. Write an Embedded C program using Keil µVision 3 IDE to interface ADC *0801* with *8051* microcontroller as per the requirements given below.
   (a) Data lines of ADC is interfaced to Port 1
   (b) The Chip Select of ADC is supplied through Port Pin P3.3, Read signal through P3.0 and Write signal through P3.1 of *8051* microcontroller.
   (c) ADC interrupt line is interfaced to External Interrupt 0 of *8051*.
   (d) The ADC Data is read when an interrupt is asserted. The data is kept in a buffer in the data memory
   (e) The buffer is located at 20H and its size is 15 bytes. When the buffer is full, the buffer address for the next sample is reset to location 20H.

https://hemanthrajhemu.github.io