EMBEDDED SYSTEM DESIGN

Sunil. MP
Hari Krishna Moorthy

BOOKS ARCADE

Embedded System Design

Embedded System Design

Sunil. MP
Hari Krishna Moorthy

BOOKS ARCADE
KRISHNA NAGAR, DELHI

Embedded System Design

Sunil. MP
Hari Krishna Moorthy

BOOKS ARCADE

Regd. Office:
F-10/24, East Krishna Nagar, Near Vijay Chowk, Delhi-110051
Ph. No: +91-11-79669196, +91-9899073222
E-mail: info@booksarcade.co.in, booksarcade.pub@gmail.com
Website: www.booksarcade.co.in

Year of Publication 2023

International Standard Book Number-13: 978-81-19199-89-1

CONTENTS

CHAPTER 1

EMBEDDED SYSTEM

Sunil. MP, Assistant Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-mp.sunil@jainuniversity.ac.in

Operating systems provide users and programmes an environment in which to run programmes and access services. The following functions are provided by one set of operating system services:

Assisting the User

A User Interface (UI) is a feature that almost all operating systems include. Varies between Graphical User Interface (GUI) and command-line (CLI) (GUI), Batch Program execution: The system has to be able to load a programme into memory, execute it, and then stop it either properly or abnormally (indicating error).A particular collection of operating-system services offers the user-beneficial features:

Modification of the File System

Programs must be able to read, write, create, remove, search, list file information, and control permissions for files and directories. On the same computer or between computers connected by a network, processes may share information. Message passing or shared memory may be used for communication (packets moved by the OS).Error detection: OS must be alert to potential errors at all times –. May happen in the user software, I/O devices, CPU, and memory hardware. To guarantee accurate and consistent computing, OS should take the necessary response for each sort of mistake. The user's and programmer's skills to effectively utilise the system may be considerably improved by debugging facilities. Resource allocation: When many users or jobs are active at once, resources must be assigned to each of them. Many resources, including CPU time, main memory, file storage, and I/O hardware –.

Accounting: Keeping track of which users utilise what resources on the computer and how much. Protection and security - Information owners who store data in a multiuser or networked computer system may wish to govern how that data is used, and concurrent operations shouldn't conflict with one another. Protection entails making sure that all system resources have regulated access. User authentication is necessary for system security from outsiders, which also includes protecting external I/O devices from unauthorized access.

The Services of the Operating System

1. Implemented sometimes in the kernel, occasionally by a systems programme
2. Multiple flavors are sometimes used - shells
3. Mostly retrieves a user command and performs it
4. Sometimes built-in commands, other times merely programme names if the latter, changing the shell to add new functionality is not necessary.
5. Desktop metaphor interface that is simple to use

Typically, icons for the mouse, keyboard, and monitor denote files, applications, activities, etc.Different mouse clicks on interface elements result in different actions providing information, choices, performing a function, opening a directory, etc.

1. Developed at the Xerox PARC.
2. Nowadays, many systems provide both CLI and GUI interfaces.
3. The GUI of Microsoft Windows is a CLI "command" shell.
4. The "Aqua" GUI interface of Apple Mac OS X has a UNIX kernel beneath and offers many shells.
5. The CLI on UNIX and Linux has alternative GUI interfaces (CDE, KD).

Because an operating system has a complicated structure, we want a well-defined framework to help us adapt it to our particular needs. A simpler way to construct an operating system is in sections, much as we split down a large issue into smaller, simpler sub problems. Additionally, each division is a part of the operating system , , . An operating system structure is the method of linking and incorporating various operating system components into the kernel. Operating systems are implemented using a variety of structures, as will be stated further down.

Simple Design

Because it lacks definition and has the simplest operating system structure, it can only be used for very tiny and restricted systems. Since the interfaces and functional levels are clearly separated in this structure, programmes are able to access I/O routines, which may result in illegal access to I/O routines. This organisational structure is used by the MS-DOS operating system. There are many levels that make up the MS-DOS operating system, and each has its own set of features.

These are the layers:

1. System Program Application Program
2. Device drivers for MS-DOS
3. ROM BIOS driver software

The MS-DOS operating system benefits from layering since each level may be developed individually and can interact with other levels as required.If the system is built in layers, it will be simpler to design, manage, and upgrade. Because of this, simple structure may be used to build constrained systems that are less complicated. When a user application fails, the operating system as a whole crashes. Because MS-DOS systems have a low degree of abstraction, applications and I/O procedures are accessible to end users, giving them the potential for unwanted access.

Memory and Storage Management

Controlling and managing a computer's primary memory is known as memory management. It makes ensuring that memory space allocation and management are done correctly so that the operating system (OS), programmes, and other processes that are currently executing have access to the memory they need to function. Memory management is a component of this process that takes into consideration the memory device's capacity restrictions, deallocating memory space when it is no longer required, or expanding that space using virtual memory. Memory

management is to maximize memory access so that the CPU can quickly obtain the data and instructions required to carry out the different activities.

Computer Memory organization

The three tiers of memory management are hardware, operating system, and program/application. To maximize memory availability and effectiveness, the management tools at each level cooperate. Hardware-level memory management. The physical parts that store data, such as the random access memory (RAM) chips and CPU memory caches, are the focus of memory management at the hardware level (L1, L2 and L3). The memory management unit (MMU), which regulates the processor's memory and caching activities, is in charge of the majority of management that takes place at the physical level. The MMU's ability to convert the logical addresses utilised by active processes into physical addresses on memory devices is one of its most crucial functions. Although it may be used as a separate integrated circuit, the MMU is commonly incorporated within the CPU.The OS's handling of memory. At the OS level, memory management entails assigning and redistributing constantly particular memory blocks to individual programmes in response to shifting CPU resource requirements. The OS continually switches between memory and storage (hard disc or SSD) devices to facilitate the allocation process, keeping track of each memory location and its allocation state.

The methods and technologies used to manage the resources, programmes, and hardware that go into a data storage system are collectively referred to as "storage management." In addition to assuring the availability, resilience, and security of the data that underpins an increasingly digital world, it requires finding the best possible balance between prices, performance, and capacity. In this post, we'll go through the key procedures and tools required to provide your company with huge, quick, and dependable data.

Processes for Common Storage Management

The precise tools and methods used by various businesses to manage data storage may vary, but the following are some typical storage management procedures:

The practice of allocating storage space to computers, servers, and other equipment is known as provisioning. Virtualization is the process of separating the operating environments for software and hardware using virtual machines (VMs).Containerization: A kind of virtualization where applications and operating systems are separated using fully packaged, portable computing environments. Data compression is a method for increasing storage capacity by reducing retrieval times, filling memory gaps, and freeing up disc space.Moving data from one storage place to another is known as data migration.

Data replication: For redundancy, resilience, and dependability, the same data is stored in several storage locations using techniques like snapshots and mirroring.Catastrophe recovery is the process of returning IT operations to normal after a disaster, using tools, planning, rules, and procedures.Automation: Automation includes all tools and approaches related to automating data storage management procedures, ranging from simple scripts to DevOps.The OS also controls how much memory will be allotted to which processes at what time. An OS may use swapping as part of this function to support additional processes. The OS briefly moves a process from main memory into secondary storage as part of the swapping memory management technique so that other processes may use the memory. The original process will then be switched back into

memory at the proper moment by the addition, the OS is in charge of managing processes when the computer's physical memory is full. When that occurs, the operating system (OS) resorts to virtual memory, a form of fictitious memory allotted from a storage disc configured to act as the computer's primary memory. The OS may automatically assign virtual memory to a process as it would physical memory if memory demand exceeds the physical memory's capacity. However, since secondary storage is significantly slower than a computer's main memory, the usage of virtual memory may have an influence on the speed of an application.

Operating-System Design and Implementation

A computer is really just a worthless piece of metal without its software. A computer's software allows it to play music and movies, send e-mail, search the Internet, and do a variety of other useful tasks actions to pay for itself. The two main categories of computer software are system programmes, which control how the computer functions, and application programmes, which carry out the tasks the user requests. The operating system, whose responsibility it is to manage all computer resources and serve as a foundation for application programme development, is the most basic system software. In this book, operating systems are the subject. The operating system known as MINIX 3 is specifically used as a model to show design concepts and the practicalities of putting a design into practise.

One or more processors, some main memory, discs, printers, a keyboard, a display, network interfaces, and other input/output devices make up a contemporary computer system. Overall, a complicated system. It is quite difficult to create systems that keep track of all these elements and utilise them appropriately, much alone optimally. It seems doubtful that many applications could be built at all if every programmer had to be concerned with how disc drives operate and the many issues that might arise while reading a disc block.

Long ago, it became plainly evident that a method for protecting programmers from the complexity of the hardware was required. Putting a layer of software on top of the raw hardware, controlling every aspect of the system, and providing the user with an interface or virtual. The hardware is located at the bottom and is often made up of two or more levels (or layers). Physical components including integrated circuit chips, wiring, power supply, cathode ray tubes, and similar components are found at the lowest level of a computer hierarchy.

The electrical engineer is responsible for how things are built and function:

1. Banking
2. Airline
3. Running system
4. Web
5. Compilers Editors
6. Applications software
7. Hardware
8. System
9. Command
10. Machine translation
11. Microarchitecture
12. Physical apparatus

The physical devices are then organised into functional units at the microarchitecture level. This level often includes some CPU (Central Processing Unit) internal registers and a data channel with an arithmetic logic unit. One or two operands are pulled from the registers and merged in the arithmetic logic unit during each clock cycle (for instance, via addition or Boolean AND). One or more registers are used to store the outcome. On some devices, software known as the microprogram regulates how the data flow functions. On other devices, hardware circuits are directly in charge of it.

It is a top-down perspective to think of the operating system as essentially giving its consumers an easy-to-use interface. An alternate, bottom-up perspective asserts that the operating system's purpose is to control every component of a complicated system. Processors, memory, timers, drives, mice, network connections, printers, and a vast range of other components make up modern computers. According to a different perspective, the operating system's role is to ensure the fair and regulated distribution of the processors, memory, and I/O devices among the numerous applications that are vying for them.

Imagine what would happen if three computer programmes attempted to print their results to the same printer at the same time. The printout's first few lines may come from programme, then its next few lines from programme, then its following few lines from programme, and so on. Chaos would be the outcome. By buffering all output going for the printer on the disc, the operating system can control the potential mayhem. When one programme is done, the operating system may transfer its output from the disc file where it was saved to the printer, while the other programme can continue to generate additional output while being unaware that the output is not really going to the printer.

Multiple users increase the necessity for controlling and safeguarding memory, I/O devices, and other resources since they may otherwise interfere with one another on a computer (or network). Additionally, users often need to exchange information (files, databases, etc.) in addition to hardware. According to this theory of the operating system, the main duties of the system are to grant resource requests, account for utilization, and resolve disputes arising from competing demands made by various applications and users.

Managing resources involves multiplexing (sharing) them in both time and space. When a resource is time multiplexed, several users or programmes utilise it alternately. The resource is first used by one of them, then by another, and so on. For instance, when a single CPU is available and many applications compete for its usage, the operating system initially assigns the CPU to the first programme. Once that programme has had a chance to run for a while, however, another programme is given the opportunity to utilise the CPU. The operating system is responsible for determining how the resource is time multiplexed, including who goes next and for how long. The sharing of a printer is another example of time multiplexing. A choice must be made about which print job will be produced next when many are in the queue to print on a single printer.

Space multiplexing is the other kind of multiplexing. Each consumer receives a portion of the resource rather than having to wait their turn. For instance, main memory is often shared by a number of active applications so that each one may operate simultaneously (for example, in order to take turns using the CPU).

If there is adequate memory to accommodate numerous applications, it is more effective to run several of them simultaneously than giving one of the access to the whole memory, particularly if it only requires a tiny portion of the total. Naturally, this causes concerns with justice, protection, and other issues, and the operating system must address them. The (hard) disc is another resource that uses space multiplexing. A single disc may store data from several users at once in numerous systems. A common operating system resource management job is allocating disc space and monitoring who is utilizing which disc blocks.

Operating System History

In spite of its vast size and issues, OS/360 and other third-generation operating systems created by other computer manufacturers actually managed to satisfactorily serve the majority of their consumers. They also helped to promote a number of crucial methods that were not included in second-generation operating systems. One of them, multiprogramming, was perhaps the most significant. On the 7094, the CPU just sat idle while the present work was put on hold while it waited for a tape or other I/O operation to finish. I/O is uncommon in highly CPU-bound scientific computations, therefore this lost time is not important. Something had to be done to stop the (expensive) CPU from being idle for such a long period since with commercial data processing, the I/O wait time may often reach 80 or 90 percent of the entire time.

The solution that emerged included breaking up the memory into numerous sections and assigning a separate task to each section, as indicated. A task may be utilising the CPU while another was waiting for I/O to finish. The CPU could be kept active almost constantly if enough tasks could be held in main memory at once. The 360 and other third-generation systems were equipped with this technology, making it possible to run numerous processes securely in memory at once. This needs specialised hardware to shield each programme from spying and mischief by the others.

Third-generation operating systems also had the capacity to read tasks from cards onto the disc as soon as they were brought into the computer room, which was a significant feature. The operating system could then load a new task from the disc into the now-empty partition and execute it whenever an existing job terminated. Spooling, which stands for Simultaneous Peripheral Operation on Line, is a method that was also utilised for output. The 1401s were no longer required with spooling, and carrying tapes significantly decreased.Third-generation operating systems were remained mostly batch systems, despite being well suited for large-scale commercial data processing operations and complex scientific computations. Many programmers yearned for the first-generation period, when they could swiftly debug their programmes since they had the computer to themselves for a few hours. With third-generation systems, it may take hours for a task to be submitted and for the output to be returned. As a result, a single missing comma could ruin a compilation and cost the programmer a whole day's work.

Due to this need for speedy responses, timesharing a kind of multiprogramming in which each user has an online terminal was made possible. If 20 individuals are signed in and 17 of them are chatting, thinking, or sipping coffee, the CPU may be distributed in turns to the three tasks that need to be served in a timesharing system. Since short commands like "compile a five-page procedure" are more frequently used when debugging software than longer ones like "sort a million-record file," the computer can offer quick, interactive service to many users while also potentially working on large batch tasks in the background when the CPU is otherwise idle. On a specially modified 7094, M.I.T. researchers created CTSS (Compatible Time Sharing System),

the first substantial timesharing system .However, timesharing did not fully take off until the third generation, when the required hardware for protection became widely available.

Following the CTSS system's success, MIT, Bell Labs, and General Electric (at the time, a significant computer manufacturer) made the decision to start working on the creation of a "computer utility," a device that would handle hundreds of concurrent timesharing users. Their model was the way energy is distributed; when you need electricity, all you have to do is put a plug into the wall, and, within reason, and you'll have all the power you need. The MULTICS system's creators had an enormous computer in mind that would provide computing capacity for everyone in the Boston region. Similar to how the concept of supersonic trans-Atlantic submarine trains is today, the thought that computers considerably more powerful than their GE-645 mainframe will be marketed for less than $1,000 by the millions barely 30 years later was pure science fantasy.

Success for MULTICS was uneven. Although it had substantially higher I/O capacity, it was built to serve hundreds of users on a system that was only somewhat more powerful than an Intel 80386-based PC. Given that individuals in those days understood how to develop short, effective programs a talent that has since been lost this is not nearly as absurd as it may seem. There are several reasons why MULTICS did not conquer the globe, not the least of which is that it was built in PL/I and that, when it eventually did, the PL/I compiler was years late and hardly functional. MULTICS was also, like Charles Babbage's analytical engine in the nineteenth century, a very ambitious project for its time.

Many ground-breaking concepts were brought into computer literature by MULTICS, but it proved to be far more difficult to transform the book into a viable product and a financial success than anybody had anticipated. Bell Labs pulled out of the project, and General Electric completely gave up on the computer industry. However, M.I.T. persevered, and MULTICS ultimately began to function. In the end, Honeywell, the firm that acquired GE's computer division, offered it as a commercial product, and roughly 80 large corporations and colleges all over the globe installed it. Users of MULTICS were fervently devoted despite their very tiny numbers. For instance, MULTICS systems from GM, Ford, and the US National Security Agency weren't terminated until the late 1990s. In October 2000, the Canadian Department of National Defence stopped operating its final MULTICS system. We shall use the words "procedure," "subroutine," and "function" interchangeably in this book despite the fact that it has not been a commercial success.

Kernel's Architecture

Each job is a nearly separate programme, as we saw when we examined the multi-tasking paradigm in earlier sections. Even if tasks in an embedded programme are somewhat independent from one another, this does not imply that they are unaware of one another. Even though some tasks will truly be isolated from one another, it is very common for tasks to need to communicate and synchronise with one another. This is a crucial aspect of the functionality an RTOS offers. The most we can provide in this post is an overview of the frequently used facilities since the actual range of choices provided by various RTOSes may vary very substantially, as May some terminology.

Task-owned facilities are characteristics an RTOS bestows on tasks that provide input (communication) facilities. Signals will be the example we examine in greater detail.Kernel

objects are RTOS-provided services that act as independent synchronisation or communication tools. Event flags, mailboxes, queues/pipes, semaphores, and mutexes are a few examples. RTOSs that support message passing enable the generation of message objects that may be passed from one task to another or to a number of others. This is essential to the kernel's architecture and explains why a product like this is referred to as a "message passing RTOS."

For each application, a different set of facilities will be best. Additionally, there is considerable overlap in their capabilities, so giving scalability some attention is necessary. For instance, it can be more effective to implement the mailbox using a single-entry queue if an application requires several queues but just a single mailbox. This object will be somewhat suboptimal, but because there won't be any mailbox handling code in the application, scalability will result in a smaller RTOS memory footprint.

Common Variables or Memory Locations

Simple methods for inter-task communication include having variables or memory spaces that are available to all involved activities. Despite how basic it is, there are some uses for this strategy. Access has to be restricted. If the variable is only a single byte, writing or reading to it will likely be a "atomic" (i.e., uninterruptible) action; however, if the processor permits additional operations on memory bytes, attention must be taken since these activities may be interruptible and lead to a timing issue. Simply turning off interruptions for a brief period of time may be used to lock or unlock a system.

Of course, locking is still necessary if you are utilising a memory region. If the memory architecture allows for atomic access to the initial byte, using it as a locking flag is an option. One process inserts data into the memory space, sets a flag, and then waits for the flag to be cleared. The second job receives the data, waits for the flag to be set, and then clears the flag. It is not advisable to use interrupt disable as a lock since transferring the whole data buffer could take some time. Similar to how many inter-processor communication capabilities are implemented in multicore systems, this kind of shared memory use follows a similar design philosophy. The inter-processor shared memory interface may sometimes include a hardware lock and/or an interrupt.

Signals

The simplest inter-task communication method available in traditional RTOSes is probably signals. They are made up of a group of bit flags 8, 16, or 32, depending on the implementation—each of which is connected to a particular job.Any activity that uses an OR kind of operation may set one or more signal flags. The signals can only be read by the job that owns them. In general, reading is harmful; thus, the flags are also removed.When any signal flags are set, a particular function designated by the signal owning task is automatically called in various systems where signals are implemented. This eliminates the need for the job to independently monitor the flags. This resembles an interrupt service procedure in certain ways. The implementation of signals in Nucleus SE will be covered in greater detail in a subsequent post.

Groups for Event Flags

Similar to signals, event flag groups are a bit-oriented inter-task communication capability. They may also be implemented in 8, 16, or 32 bit groupings. They are autonomous kernel objects as

opposed to signals, and they do not "belong" to any one job.Using the OR and AND procedures, any job may set and remove event flags. Similar to this, any job may query event flags by using the same sort of action. A task may be stopped until a certain combination of event flags has been set in various RTOSes thanks to the ability to perform a blocking API call on an event flag combination. When querying event flags, there can additionally be a "consume" option available that would delete all read flags.

Semaphores Semaphores offer a flagging mechanism that is often used to regulate access to a resource. They are independent kernel objects. Binary semaphores (which only have two states) and counting semaphores are the two main categories (that have an arbitrary number of states). Some processors are capable of carrying out (atomic) instructions that make it simple to create binary semaphores.

Counting semaphores with a limit of one are another way to describe binary semaphores.To get access to a resource, any process may try to acquire a semaphore. The acquire will be successful if the semaphore value is larger than 0, which decreases the semaphore value. To receive a semaphore, it is often feasible to perform a blocking call; this allows a job to be paused until the semaphore is released by another process. A semaphore may be released by any job, increasing its value.In a subsequent article, which details how semaphores are used in Nucleus SE, there is additional information regarding semaphores.

Queues

Queues are autonomous kernel objects that provide tasks a way to send messages to one another. Compared to mailboxes, they are a bit more adaptable and sophisticated. The size of the message will often be fixed and word/pointer oriented, depending on the implementation. Any job may send to a queue, and it may do so repeatedly up to the point when the queue is full, at which point any efforts to send will fail. When the queue is formed or the system is setup, the depth is typically user-specified. When sending to a queue, it is often feasible to do so blockingly in many RTOSes.

If the queue is full, a job may be put on hold until the queue is read by another process. A queue may be read from by any job. First in, first out dictates the sequence in which messages are read (FIFO). A job will encounter an error if it attempts to read from an empty queue. Many RTOSes allow blocking calls to be made in order to read from a queue, which allows a job to be halted in the event that the queue is empty until another process sends a message to it. The capability of "jamming," or sending a message ahead of the queue, is likely supported by an RTOS. A "broadcast" capability is also supported by several RTOSes. This makes it possible to send a message to every job that has been halted while reading a queue. An RTOS may also allow for the sending and receiving of messages with various lengths, which offers more flexibility but also adds some overhead. Another kernel object type called "pipes" is one that many RTOSes support. A pipe handles byte-oriented data but otherwise functions similarly to a queue.

Although the fundamental workings of queues are not relevant here, it should be noted that they need more memory and runtime than mailboxes. This is mostly due to the need of maintaining two pointers, one for the head and one for the tail of the queue. Future articles that cover the implementation of queues and pipelines in Nucleus SE will have additional details regarding these concepts.

Mailboxes

Independent kernel objects called mailboxes provide tasks a way to convey messages. The implementation will determine the message size, which is often fixed. Standard message sizes range from one to four pointer-sized objects. A mailbox is often used to send a reference to some more complicated data. Some kernels use mailboxes, which simply store the data in a normal variable and let the kernel control access to it. Exchanges are another term for mailboxes, however it's less prevalent nowadays. A mailbox may receive any job; once it is filled, it is empty. The task will get an error answer if it then attempts to transmit to a mailbox that is already full. A job may be halted in various RTOSes so that another task may read the mailbox. This is achievable by using a blocking call to transmit to a mailbox. A mailbox may be read from by any job, which makes it once again empty. A job will encounter an error if it attempts to read from an empty mailbox. A job may be put on hold in many RTOSes until the mailbox is filled by another task. This is accomplished by making a blocking call to read from the mailbox.

A "broadcast" capability is supported by several RTOSes. This makes it possible to send a message to all jobs that are now halted while reading a certain mailbox.Some RTOSes completely lack functionality for mailboxes. Instead, it is advised to utilise a single-entry queue. Although functionally comparable, this has more runtime and memory overhead.In a subsequent article, which details how mailboxes are implemented in Nucleus SE, there is additional information regarding mailboxes.

Mutexes

Mutexes, also known as mutual exclusion semaphores, are independent kernel objects that function quite similarly to standard binary semaphores. They integrate the idea of transitory ownership and are a little more complicated (of the resource, access to which is being controlled). When a task acquires a mutex, that task is the only one that can release it again since the mutex (and therefore the resource) are owned momentarily by the task.Not all RTOSes have mutexes, but it is simple to customise a standard binary semaphore in its place. Writing a "mutex get" function that acquires the semaphore and records the task identification would be required. The calling task's identification would then be checked by a complimentary "mutex release" function, which would only release the semaphore if it matched the stored value and raise an error otherwise.

Semaphores

Semaphores are a programming tool for coordinating or synchronising operations when several processes compete for the same operating system resources, notably on UNIX systems. A semaphore is a value that each process may check and then modify that is located at a specific location in the operating system's (or kernel's) storage. Depending on the value discovered, the process will either be able to access the resource or discover that it is already in use and will need to wait a while before attempting again. Semaphores may contain extra values in addition to being binary (0 or 1). Usually, a semaphore-using process checks the value and modifies it if it needs the resource so that future semaphore users will know to wait.Semaphores are often used for two tasks: sharing access to files and a shared memory space. One method for interprocess communication is semaphores (IPC). There are a number of interfaces or "functions" available in the C programming language for controlling semaphores.

Semaphore is just a shared, non-negative variable that is used by several threads. A semaphore is a signalling device, and another thread may signal a thread that is awaiting a semaphore. It employs two atomic actions for process synchronisation:

1) Wait,
2) Signal.

In accordance with how it is configured, a semaphore either permits or prohibits access to the resource. Semaphore's distinctive features.The following are characteristics of a semaphore:

1. It is a device that may be used to synchronise tasks.
2. A basic synchronisation mechanism.
3. A non-negative integer value will always be stored in a semaphore.
4. Test operations and interrupts, which ought to be carried out using file descriptors, may be used to implement semaphore.
5. Semaphores come in two main types, which are as follows:
6. Summarizing semaphores
7. Semaphores in binary.

Semaphores that utilise a count this kind of semaphore employs a count to enable a job to be obtained or released repeatedly. The counting semaphore should be generated in the unavailable state if the starting count is 0.In recent years, both business and academics have given security in embedded systems an increasing amount of attention. Embedded systems are being used in a variety of application fields, from data collecting in dangerous locations to controlling safety-critical devices. Due to their integrated design, these devices are naturally susceptible to several operating issues and deliberate assaults. Increased remote exploit opportunities are made possible by network connection. Security solutions are being created in response to this need for robustness, attack defence, and recovery capabilities.We provide an overview of embedded system security in this post. We go through how certain aspects of embedded systems might result in a number of possible weaknesses. We also provide a succinct overview of embedded system assaults and related defences.

CHAPTER 2

TRAITS AND WEAKNESSES OF EMBEDDED SYSTEMS

Hari Krishna Moorthy, Associate Professor,
Department of Electronics and Communication Engineering,Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-harikrishna.moorthy@jainuniversity.ac.in

Numerous embedded system features that are intrinsic have a direct bearing on security-related concerns. We go through some of its consequences for embedded system vulnerabilities.When traditional workstations or servers are insufficiently useful, expensive, power-hungry, bulky, or heavy for a certain application, embedded systems are utilised. Embedded system specialisation often has one or more of the following drawbacks:

Due to its limited processing capability, an embedded system is often unable to execute programmes that protect against assaults on more traditional computer systems (such as virus scanners and intrusion detection systems.One of the main limitations of embedded systems is the limited amount of accessible power. These systems often use batteries, and higher power consumption shortens system lifespan (or increases maintenance frequency). Because of this, embedded systems can only devote a small amount of power resources to maintaining system security.

When embedded systems are used in environments outside the direct control of the owner or operator, they often experience physical exposure (e.g., public location, customer premise). As a result, embedded systems are predisposed to being subject to intrusions that take advantage of the attacker's physical proximity.Inaccessible deployment of embedded systems necessitates remoteness and unattended operation (e.g., harsh environment, remote field location). This restriction suggests that it is difficult and requires automation to release updates and patches as is done with traditional workstations. These automated systems provide potential points of attack. Network connection for embedded systems, whether via wireless or wired access, is becoming more widespread. For remote control, data collection, and upgrades, such access is required. Vulnerabilities may be remotely exploited from anywhere if the embedded system is linked to the Internet. These traits result in a distinct set of vulnerabilities in embedded systems that must be taken into account.

Security Flaws

Embedded systems are susceptible to a variety of abuses that may be aimed to steal sensitive data, consume the system's power source, disable the system, or hijack the system for a different use. Examples of embedded system vulnerabilities include. Energy drainage (exhaustion attack): Because embedded systems have limited battery power, they are susceptible to assaults that exhaust this resource. Increasing the computational load, decreasing the number of sleep cycles, or using more sensors and peripheral devices are all ways to drain more energy.Physical intrusion (tampering): Embedded systems are vulnerable to assaults that require physical access to the system because of their close proximity to a possible attacker. Attacks using power analysis or system bus spying are two examples.

Attacks and Retaliation

The goals of the attacks or the methods used to launch them can be used to categorise security threats to embedded systems. The attack's goals can include preventing privacy, subverting integrity, or reducing availability, as was shown above. Attacks can be launched using physical, logical, or side channel-based methods. Common privacy attacks target confidentiality, access control, and authenticity. On the other hand, logical attacks can be either software-based or cryptographic.Eavesdropping, reverse engineering, and micro probing are a few examples of physical assaults. If a malicious system compromise is attempted by someone with manufacturing knowledge, the resources available for reverse engineering significantly increase. Once the bare silicon circuitry has been exposed by acid or other chemical means, integrated circuits may be susceptible to micro probing or analysis under an electron microscope. When confidential information is sent through electronic media like email or instant messaging, eavesdropping occurs when unintended recipients listen in on conversations.Side channel attacks include attacks on fault injection, power analysis including Simple Power Analysis (SPA) and Differential Power Analysis (DPA), timing analysis, and electromagnetic analysis. Side-channel attacks work by observing system characteristics as the system executes cryptographic operations.

Cyber Attacks

Examples of software attacks that now make up the bulk of all software assaults include code injection attacks. The network may be used to remotely introduce the harmful code.When performing security attacks, such as getting into a system by guessing the password, cryptographic assaults take use of flaws in the information included in the cryptographic protocol provides a concise outline of typical cryptographic and protocol weaknesses. The use of secure proof-carrying code and run-time monitors that identify security policy breaches are two solutions that have been suggested in the literature to thwart cryptographic assaults.The majority of current security threats destroy an application program's code integrity. To gain control over a program's execution flow, they contain instructions that can change dynamically. Code injection attacks are ones that compromise the integrity of software. Code injection attacks, also known as security vulnerabilities, frequently take advantage of regular implementation errors in application programmers. The quantity of software code is always correlated with an increase in malicious attacks. Stack-based buffer overflows, heap-based buffer overflows, the use of the double-free vulnerability, integer problems, and the use of format string vulnerabilities are some of the assaults.

Attacks on Side Channels

Side channel attacks are renowned for their simplicity of use and effectiveness in silently stealing confidential information from the target device. While the chip is processing secure transactions, adversaries can monitor side channels like power consumption, processing time, and electromagnetic (EM) emissions. The adversary runs a cryptographic algorithm while monitoring the side channels and providing various input values to the system (e.g., encryption using a secret key). The internal calculations are then connected with these observed exterior manifestations. The secret keys used for encryption and/or decryption may be effectively identified at either the transmitter or the recipient via side channel attacks.

The most often used chip attribute to derive secret keys through side channel attacks is power consumption. In 1999, developed the first power analysis attack, which allowed for the successful discovery of secret encryption programme keys via the observation of chip power dissipation. Microprocessor chips are incorporated into devices like Smart Cards, PDAs, and Mobile Phones, executing safe transactions utilizing secret keys.

Defending Against Computer Assaults

In the literature, a number of defenses against code injection attacks that take use of widespread implementation flaws have been suggested. Based on the system component where the suggested countermeasure is implemented and the countermeasure methodologies, these may be split into nine types. The nine groupings addressed here are as follows:

1. An overview of embedded systems security
2. Countermeasures based on architecture
3. Secure dialects
4. Three. Static code debuggers
5. Interactive code editors
6. Techniques for detecting anomalies

Damage control or sandboxing strategies

1. Support for compilers
2. Support for libraries
3. Countermeasures based on the operating system

There are several hardware/architecture aided countermeasures that seek to safeguard these addresses since return addresses of functions are the most often targeted target of buffer overflows. In, several of these strategies are detailed. Making ensuring code integrity is maintained at runtime is another method for preventing code injection attacks. In order to guarantee programme code integrity at runtime and hence avoid code injection attacks, the authors of have developed a microarchitectures method makes a suggestion for an embedded monitoring system to verify proper software execution.

Some of the implementation problems highlighted here may be avoided by using safe languages like Java and ML. However, as more and more low and high level applications are implemented by regular programmers using C and C++, there is a need for secure implementation of these languages. Safe versions of C and C++ use methods like memory management restrictions to guard against implementation mistakes.

Software analysis tools called static code analyzers examine it without actually running any programmes created with it. The majority of the time, the analysis is done on the source code, but it may also be done on particular types of object code. These tools' analysis quality varies from those that merely take into account the behaviour of simple statements and declarations to those that use the whole source code of a programme in their analysis. The data gathered by these analyzers may be used in a variety of ways, from formal techniques to formally establish programme characteristics to the detection of coding flaws.In dynamic code analysis, test runs are executed to find vulnerabilities after the source code is instrumented at build time. Even though dynamic code analysis is more accurate than static code analysis (more information about

the execution is accessible at runtime compared to compile-time), dynamic code checking may miss certain problems since they may not be on the execution path while being examined.

A profile of all permitted application activity is compared against the program's actual behaviour in behavior-based anomaly identification. A flag for a possible security attack will be raised for any divergence from the profile. This model is a positive security model since it determines that everything else is harmful and only looks for previously recognised favourable behaviours. Attacks on application code, including undiscovered and novel attacks, may be found via behaviour anomaly detection. The majority of the time, system call execution is watched, and if it deviates from one of the previously obtained patterns, it is noted as an anomaly. When the predetermined threshold for the number of anomalies is achieved, the anomaly may be reported to the system, and further action, such as ending the programme or rejecting a system call, can be performed.

On the down side, a significant proportion of false positives might result from behaviour anomaly detection. For instance, if an application is changed after a behaviour profile is developed, behavior-based anomaly detection may mistakenly classify access to the changed application as a possible attack.

According to the concept of least privilege, sandboxing is a common technique for creating constrained execution environments that may be used to execute untrusted applications. A sandbox restricts or scales down the degree of system access that its programmes have. Systems researchers have been interested in sandboxes for a very long time. In his 1971 study, Butler Lampson provided a conceptual model that highlighted the characteristics of a number of current protection and access-control enforcement techniques.

Attack on the Stack based Structure

The ability of programmes created using language requirements to operate on hardware is greatly aided by compilers. The compiler is the easiest area to include a range of fixes and defences without altering the programming languages used to create susceptible applications. Researchers have suggested stack-frame protection measures after noting that buffer overflows, which are the most common kind of security attack, are brought on by stack-based buffers. A defence against stack-based buffer overflow attacks is stack-frame protection, which often includes protecting the stack-return frame's address as well as other important data like frame pointers. Protecting programme references in the code is another often suggested counter measure. Given that all code injection attacks need code pointers to be modified to refer to the injected code, this countermeasure was developed. Since writing data that exceeds the capacity of the buffers leads to buffer overflows, it is possible to verify the bounds of the buffers as the data is written to stop buffer overflow attacks. This section also covers the solutions put out as compiler support for bounds checking.

By suggesting new string manipulation routines that are less prone to exploitations or entirely invulnerable, safe library functions make an effort to stop vulnerabilities. The authors of offer an alternative to the current functions that assume strings are always NULL terminated when it comes to handling strings. In addition to the strings themselves, the new suggested functions additionally take a size argument. Operating system-based solutions have offered ways to stop the execution of such injected code based on the fact that most attackers want to run their own code. The majority of current operating systems divide the process memory into a minimum of

two sections, code and data. Making the data segment non-executable and the code segment read-only will make it more difficult for an attacker to inject and execute code into a running application.

Protection against Side Channel Assaults

Several defences exist against side channel assaults. Six categories have been created from these:

1. Masking
2. Window technique
3. Insertion of phoney instructions
4. Modification of code or algorithms
5. Juggling
6. Other techniques

Noise may be introduced during code execution to conceal it and perplex an enemy. In, examples of masking methods are provided. Boxes of substitution .The execution of embedded systems security—an overview (SBOXes), which are often employed in cryptology, may also be concealed. In, a few examples of SBOX masking methods are provided. To counter side channel attacks based on power analysis, public key cryptosystems may use a window technique. By splitting the exponent into windows of different sizes and doing the exponentiation in iterations per window by selecting the window at random, a modular exponentiation may be carried out using the window technique.

You may insert fake instructions to create unpredictable delays. When trying to connect the source implementation with the power profile, this perplexes the attacker. According to, random delay-related countermeasures (i.e., fake instructions used to induce unpredictable delays in an execution) should be carried out thoroughly; otherwise, they risk being undone and rearranged, leading to a successful assault.

In, a number of fake instruction techniques are provided.Because of conditional branching in the encryption, public key cryptosystems like RSA and ECC have been extensively attacked using Simple Power Analysis (SPA). Such software vulnerabilities may be avoided by changing the implementation or substituting a superior new algorithm that does the same goal demonstrate the essential code change strategies to avoid power analysis.

Complementary events may be incorporated into the software code in order to counteract the consequences of the real calculations provide examples of such code balancing strategies. Since power is consumed/dissipated dependent on the switching activity in gates, balancing at the gate level is undoubtedly the best way to avoid power analysis. In order to balance hardware, two gates are often connected in parallel such that one gate complements the other while switching. In, several hardware balancing methods are provided.Signal suppression circuits are among the other strategies that may be utilised to lower the Signal-to-Noise Ratio (SNR) and stop the opponent from discriminating the power profile. In, examples of suppression circuits are shown. Software level current balancing methods include changing the source code and adding nops to maintain a constant current.

According to a non-deterministic processor architecture put out by the processor will randomly choose which independent instructions to execute out of sequence. This violates the standard attack rule that forbids the adversary from comparing different runs for power analysis by

deleting the correlation between several executions of the same application. The non-deterministic processor architecture is suggested in a number of different enhanced iterations.Another countermeasure recommended to avoid power analysis is to randomise the clock signalfor the secure processor to perplex the opponent. This makes it impossible for the opponent to examine the clock signals and find certain important instruction executions in the power profile provide further instances of managing the clock signal to avoid power analysis.Designing unique instructions with difficult-to-analyse power signatures or with data-independent power usage may also help avoid power analysis .In, a number of instances of writing extendable instructions are provided. It is also possible to modify these extendable instruction architectures to thwart power analysis assaults.

System on Chip (SOP)

One of the main from the National Academies is that. A deliberate and ongoing commitment to simplicity, including simplicity of vital operations and simplicity in system interactions, is one way to ensuring reliability at a reasonable cost. True skill is often identifiable by this dedication. We see cognitive complexity as the opposite of simplicity. An innovative technique is to attempt to examine an embedded system design from the perspective of cognitive complexity and derive design recommendations from this study. The initial attempt, like with any novel strategy, is likely to be immature and call for additional development.

Reasoning about the behaviour of a System on Chip (SoC) with a billion transistors switching at a frequency of 1 GHz at the level of each transistor's activity is difficult. Therefore, in order to design for simplicity, we must create objects whose important qualities can be studied by simple models at various levels of abstraction.

The main design problem is creating a software/hardware artefact (an embedded computer system that fulfils the required functions within the established restrictions and whose pertinent attributes can be adequately represented by models at various abstraction levels.The conception of the proposed system's desired high-level behaviour is the first step in top-down computer system design. For instance, while designing a car's computer-controlled braking system, we begin with a high-level behavioral specification that links the inputs and outputs in the value and time domains: Within one millisecond of depressing the brake pedal, the computer should begin to brake the vehicle.

A high-level behavioural model of the planned computer system may be used to define a high-level specification. This model is improved and altered at various stages throughout a top-down model-based design process until an executable representation (again, a model) that can be run on the chosen distributed hardware target platform is produced. We may examine more pertinent characteristics of our growing artefact using such a sophisticated behavioural model by developing suitable analytic models from the bottom up. We may create a dependability model or a power/energy model, for instance.

The mechanical form of our artefact or its cost, for example, may also be handled by further models, where each model focuses on the aspects of the item that are relevant for the particular purpose. The fact that each of these models is focused on the same item links them all together. A high-level definition of the system's structure and of the solution process may be used to create some of these models.

The Fundamentals of Modeling

Because our cognitive powers are very restricted, the only way we can comprehend the world around us is by creating simplified models of the qualities that are important and interesting to us and ignoring (abstracting from) information that we deem unnecessary for the given goal. In this sense, a model is a purposeful reduction of reality with the aim of illuminating a specific reality-relevant characteristic. As an example, in celestial mechanics, the variety of the whole globe is reduced to a mass point so that the interactions with other mass points (heavenly bodies) may be examined.

Simplicity arises when a new abstraction level (a new model) is developed that conceptualises the qualities pertinent for the given goal and ignores the rest. This simplicity, made possible by appropriate abstractions, leads to fresh understandings that are the foundation of the natural laws. Laws of nature can never be demonstrated to be entirely accurate, as argues, because of the inherent flaws in the abstraction process.

A hierarchy of models known as the abstraction ladder, results from the recursive application of the abstraction and refinement concepts.

More broad ideas are generated via abstraction, while more specific concepts are formed by refinement, starting with basic-level concepts that are developed early in the development of the human mind and are crucial for comprehending a topic. Think of the lower-level notion of an armchair as being refined from the basic-level concept of a chair, which is generalised in the more abstract concept of furniture.Entities may interact at any level of abstraction and create a scenario of emergent complexity up until a few of these scenario's attributes are captured in a new conceptualization (model) at a higher degree of abstraction, which causes a sudden simplification.

When interactions between things produce global features that are absent at the level of the entities, a new structure or behaviour may develop. These characteristics, which result from the interaction of the entities, are referred to as emergent characteristics. For instance, the distinctive qualities of a diamond, such as brightness and hardness, which are brought about by the coherent alignment of the carbon atoms, vary significantly from those of graphite (which consists of the same atoms). We may disregard the diamond's fundamental structure and composition in favour of seeing it as a novel notion with its own qualities. The complex interactions between the components that produce a new whole with new distinguishing characteristics are what give rise to simplicity.

Control Timer

A control timer is designed to offer precise timing-based controls for processing, manufacturing, and home appliances in settings where time measurement is essential for efficient functioning without sporadic human input. It is clear that for appropriate events should be planned and completed on time. There are certain machines made to change or carry out a given duty at precise times for convenience. Using a "Timer" is a nice example. Its job is to turn each event ON or OFF individually. In daily life, it is quite helpful. A straightforward example is the employment of clocks to warn individuals at a certain moment by setting off an alarm. Military applications include the employment of certain sophisticated electronic timers. Timer has shown to be crucial in heaters, microwave ovens, computer systems, lab research, and other items used

in civilian life away from the military. Additionally, without a timer, a number of industrial procedures would not function properly. In this work, a programmable control timer that can turn an output load ON or OFF and count down from a specified time to zero is developed and built. It can be programmed to function with a variety of time inputs, however for this prototype, the duration cannot be more than 999 seconds (16.65 minutes).

In this design, a 4060B CMOS integrated circuit is used to handle the duty of operating a 32768Hz crystal. The inbuilt dividers of the 4060B integrated circuit split the input frequency (32768Hz). 2Hz is the result of the division. To supply the required 1Hz, an extra flip-flop is installed (or 1 second period pulse). Before being fed into a down counter unit, this leading pulse first goes via control logic. The Control Logic assigns a certain preset number to each of the three BCD down counters that make up the Counter Unit. Flip-flops or locks make up the Control Unit. To give specialized controls, external switches are attached to the flip-flops' individual inputs. The counter unit is also programmed to count down from the control unit. A zero detector alerts the control unit to the status when a zero value is achieved, which causes the output load to be turned ON or OFF. Through a relay circuit external to the timer, the electric power supply to the output load is changed. As a result, switching performance is improved, and load capacity is increased.

A crystal CMOS oscillator is the leading design's additional feature. A 555 timer is unable to operate precisely. When using a crystal type oscillator, precision functioning is highly guaranteed. The oscillator in the design runs on a 32768Hz crystal, and the integrated circuit in question has internal frequency dividers to help generate a 1Hz frequency for clocking the counters. Any impact of frequency instability is minimised by the timing approach. The design is based on easily accessible, reasonably priced electrical components. As a result, the total cost of building is reasonable.The distinguishing feature of the programmable control timer is its capacity to turn ON and OFF an electrical load that is linked to it after a certain period of time.

Clarity Oscillator

A 4060B integrated circuit with minimal external electronics components serves as the crystal oscillator in Figure (Configuration of the 4060B as a Crystal Oscillator). The device's frequency stability is ok.The 14-stage counter/divider logic that is already incorporated into it just splits the primary frequency into smaller numbers. The 4060B uses this method to produce 10 frequencies simultaneously.All of the outputs have been buffered for high load capacity. The gadget needs its pin 12 to be grounded in order to function. The following formula determines how often each of the 10 outputs occurs:

The logic diagram for the logic control unit, which is the main component of the system architecture. The device's functions include controlling the counters, further dividing the crystal oscillator's frequency outputs, and operating the output's load. This device performs latch logic operations. Four latches are stored in the logic control unit. The locks of the 4013B 1 and 2 are divided by 2. The crystal oscillator's 2Hz pulse is divided by 4013B 1 to produce a 1Hz pulse. This signal that is produced is intended for counting. Evidently, it has a period of one second.

The Logic Control

This timing gives the counter unit a timing of one second. The crystal oscillator's 8Hz pulse is divided in half by 4013B 2. The counter is pre-set using the resultant 4Hz pulse. It’s interesting

to note that the two latches serve as controls. For such control, their reset terminals are helpful. The control entails logically setting a certain reset input to HIGH. The latch is disabled as a consequence. Device control is performed using 4013B 3 and 4. 4013B 3 is concerned with output control. When the counter reaches its lowest level, "000," it reacts.

A computer hardware system with software embedded in it is an example of an embedded system. An embedded system may be a standalone unit or a component of a larger system. A system built on a microcontroller or microprocessor and intended to carry out a certain function is called an embedded system. A fire alarm, for instance, is an embedded device that only detects smoke. An embedded system consists of three parts. It has hardware. It contains software for applications. Its Real Time Operating System (RTOS) manages the application software and offers a framework to allow the processor operate a task in accordance with scheduling by adhering to a plan to limit latencies. The RTOS specifies how the system operates. It establishes the rules for the application program's execution. It's possible that a tiny embedded system lacks an RTOS. As a result, a dependable, real-time control system based on microcontrollers and software may be described as an embedded system.

CHAPTER 3

CHARACTERISTICS OF AN EMBEDDED SYSTEM

Asha. KS, Associate Professor,
Department of Electronics and Communication Engineering,Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-ks.asha@jainuniversity.ac.in

A single-functioned embedded system typically carries out a certain task and repeats it many times. An example might be: A pager always works as a pager.Design metrics are tightly limited in all computer systems, but they might be more so in embedded systems. Design metrics are a way to gauge an implementation's cost, size, power, and other characteristics. It has to be small enough to fit on a single chip, quick enough to analyze data in real time, and power-efficient enough to prolong battery life.Real-time and responsive many embedded systems must constantly respond to changes in their surroundings and calculate specific outcomes without any delay in real time. Take a cruise control system in a vehicle as an example; it continuously analyzes and responds to speed and braking sensors. It must repeatedly calculate accelerations and decelerations within a certain amount of time; if the calculation is delayed, the automobile may not be controlled.It must be microprocessor-based, either via a microprocessor or a microcontroller.

Memory: As its software is often embedded in ROM, it must have a memory. The PC doesn't need any more memory.It must have peripherals that are connected in order to link input and output devices.

Software is utilized in HW-SW systems to provide additional functionality and flexibility. Figure 3.1 illustrates the utilization of hardware for security and performance.

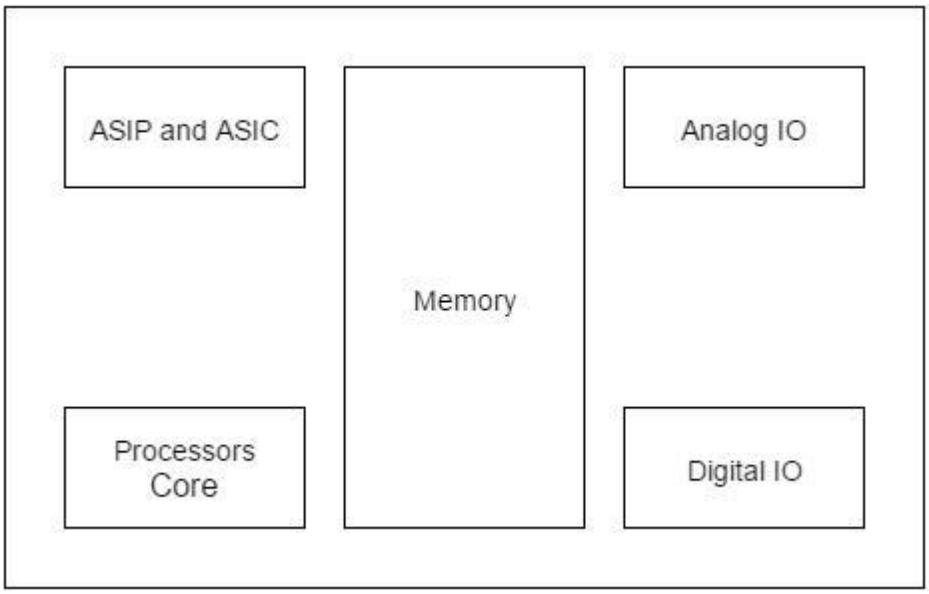

Figure 3.1: Illustrates the component of memory.

Advantages

1. Easily Customizable
2. Low power consumption
3. Lowcost
4. Enhanced performance

Disadvantages

1. High development effort
2. Larger time to market

Basic Structure of an Embedded System

The following illustration shows the basic structure of an embedded system in Figure 3.2.

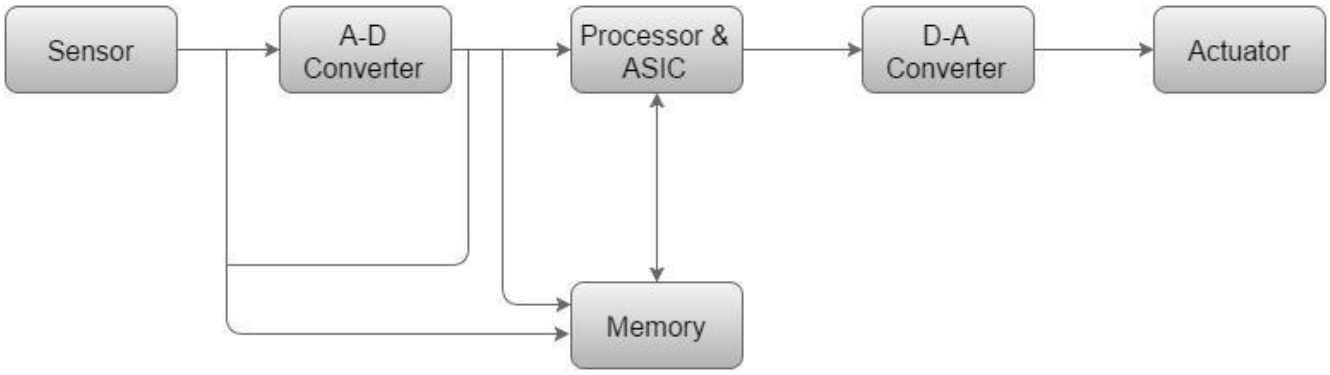

Figure 3.2: Illustration shows the basics tructure of an embedded system

Sensors take physical measurements and transform them into electrical signals that may be interpreted by observers or by other electronic devices like A2D converters. The amount that was measured is stored in memory by a sensor. An analog-to-digital converter (A-D converter) transforms the analog signal that the sensor sends into a digital signal.

Processors and ASICs: Processors transform data into output measurements and memory storage. A digital-to-analog converter (D-A converter) transforms the digital data provided to the processor into analog data.

Actuator: An actuator saves the permitted output after comparing the D-A Converter's output to the real (anticipated) output stored in it. The brain of an embedded system is the processor. It is the fundamental component that receives inputs and processes data to generate an output. A designer of embedded systems must be knowledgeable about both microprocessors and microcontrollers.

System Processors

A processor comprises two crucial components.

Execution Unit for the Program Flow Control Unit (CU) (EU)

A fetch unit is part of the CU and is used to retrieve instructions from memory. The European Union has circuits that put the rules governing data transfer operation and data translation from one form to another into practice. The Arithmetic and Logical Unit (ALU) and circuits that carry out program control tasks like interrupting or jumping to another set of instructions are both included in the EU.A processor performs the fetch cycles and carries out the instructions in the order in which they were retrieved from memory.

Various Processor Types

The hardware of an embedded system consists of components such as a user interface, input/output interfaces, a display, memory, etc.Power supply, CPU, memory, timers, serial communication ports, and system application specific circuits are often found in embedded systems. Figure 3.3.

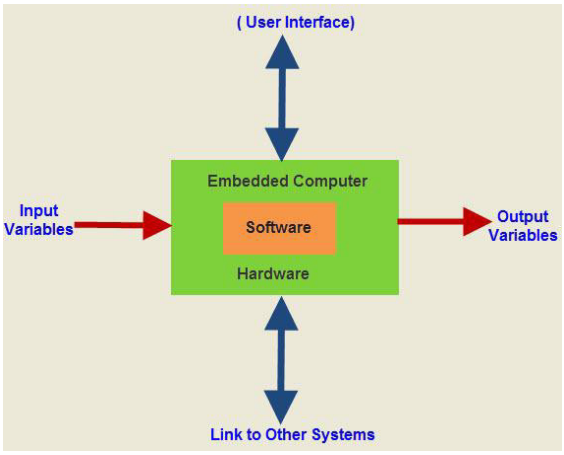

Figure 3.3: Illustrates the component of embedded computer.

Types of Embedded Systems

Embedded systems can be classified into different types based on performance, functional requirements and performance of the microcontroller in Figure 3.4.

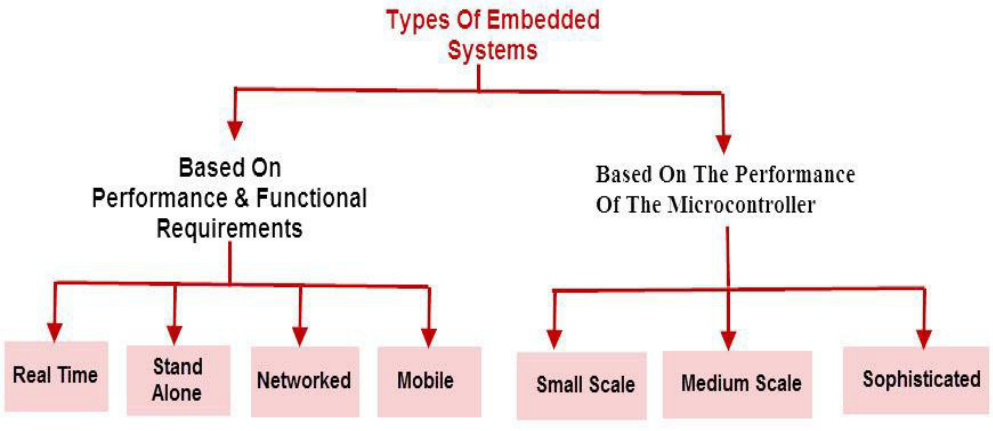

Figure 3.4: Illustrates the types of embedded system.

Types of Embedded Systems

Based on their functional and performance needs, embedded systems are divided into four groups: individual embedded systems Instantaneous embedded system:

3. Embedded systems that are networked
4. Embedded systems on the move

Embedded Systems may be divided into three categories depending on the performance of the microcontroller, including:

1. Compact embedded systems
2. Systems installed at a medium size
3. Highly developed embedded systems

Embedded Stand-Alone Systems

Standalone embedded systems function on their own; they don't need a host system like a computer. The linked device receives the processed, calculated, and converted data from the input ports, either analog or digital, and either drives, controls, or displays the connected devices with the resultant data. Mp3 players, digital cameras, video gaming consoles, microwave ovens, and temperature measuring devices are a few examples of stand-alone embedded systems.

Systems Embedded in Real Time

A system that provides the needed output at a certain time is referred to as a real-time embedded system.These embedded system varieties adhere to job completion deadlines. There are two categories for real-time embedded systems: soft real-time systems and hard real-time systems.

Embedded Networked Systems

To access the resources, many embedded system types rely on a network. LAN, WAN, or the internet are all examples of linked networks. There are both wired and wireless connectivity options.

The use of this kind of embedded system is the one that is expanding the quickest. A system known as an embedded web server connects all embedded devices to a web server so that they may all be viewed and managed using a web browser. An illustration for LAN a home security system with a networked embedded system has all of its sensors linked and using the TCP/IP protocol.

Embedded Mobile Systems

Portable embedded devices including cell phones, mobiles, digital cameras, mp3 players, personal digital assistants, etc. employ mobile embedded systems. These devices' primary drawbacks are their limited memory and other resource availability.

Systems Embedded at a Little Scale

These embedded systems use only one 8-bit or 16-bit microprocessor, which might potentially be powered by a battery. The primary programming tools for creating embedded software for tiny embedded systems include an editor, an assembler, a cross-assembler, and an integrated development environment (IDE).

Systems Embedded at a Medium Scale

These embedded systems are created using RISCs, DSPs, or a single 16- or 32-bit microcontroller. These embedded devices contain complicated hardware and software. The primary programming languages for creating embedded software for medium-sized embedded systems include C, C++, JAVA, Visual C++, RTOS, debugger, and source code engineering tool, simulator, and IDE.

Intelligent Embedded Systems

These kinds of embedded systems need ASIPs, IPs, PLAs, scalable or adjustable processors due to their extreme hardware and software complexity. They are used for innovative applications that need both hardware and software. Components that must come together to form the final system and co-design.

Embedded System Applications

Figure 3.5 illustrates the employment of embedded systems in a variety of fields, including digital consumer electronics, smart cards, missiles, satellites, and telecommunications.

Figure 3.5: Illustrates the different application of embedded system.

Embedded System Initialization

A developer can build and execute a Hello World! Application on a non-embedded machine in a matter of minutes. Yet, the work is not that simple for an embedded coder. Days might pass before you notice a fruitful outcome. For a developer who is just starting out in embedded system development, this procedure might be challenging. Many novices find it difficult to understand how to boot the target system, whether it is a third-party evaluation board or a bespoke design. In fact, it is intimidating to open a programmer's reference manual for the target board, pore over tables of memory addresses and registers, or examine the interconnection diagrams of hardware components, wondering what it all means, what to do with the information (some of which makes little sense), and how to relate the information to running an image on the target system. With the help of Figure 3.6, we debunk the booting and initialization procedure of embedded systems in this chapter in an effort to allay confusion.

Figure 3.6: illustrates the serial communication

Serial Communication

Serial Transfer

Serial communication is the process of delivering and receiving data one bit at a time in computer science and telecommunications (Figure 3.7). It seems as if you are shooting machine gun shots at a target... They are fired one at a time ;)

Parallel Communication

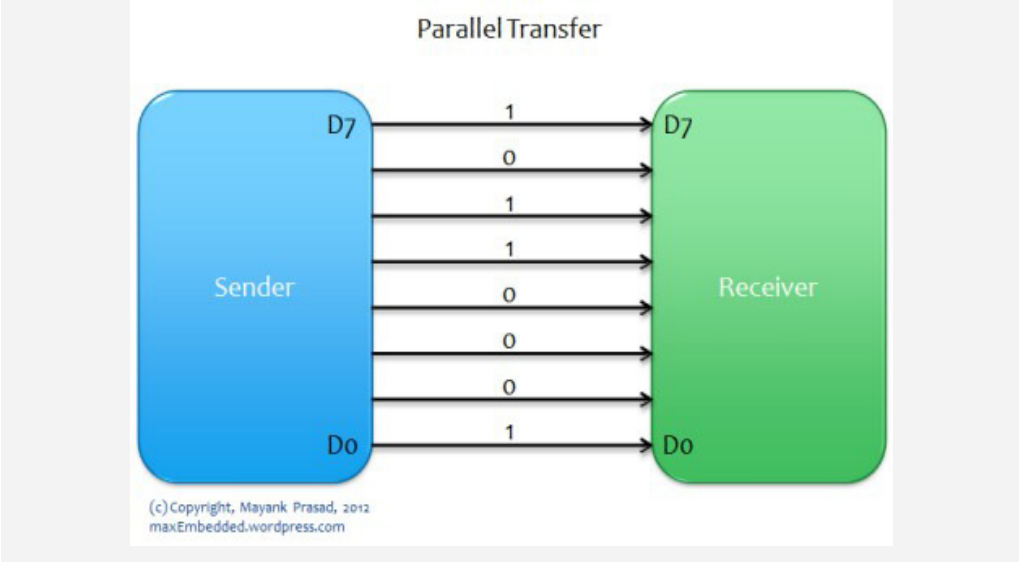

Figure 3.7: illustrates the parallel communication

Parallel Transfer

As seen in Figure 3.8, parallel communication involves transmitting and receiving many data bits at once across parallel channels. It resembles shooting at a target with a shotgun, when many shots are fired from the same gun at once ;)

Communication in Parallel vs. Serial

Let's now quickly review Table 3.1 to see how the two sorts of communications vary from one another.

Table 3.1: illustrates the differences between the two types of communications.

Serial Communication	Parallel Communication
1.One data bit is transceiver rata time	1.Multipledatabitsaretransceiveratatime
2.Slower	2.Faster
3.Lessnumberofcablesrequiredtotransmitdata	3.Highernumberofcables required

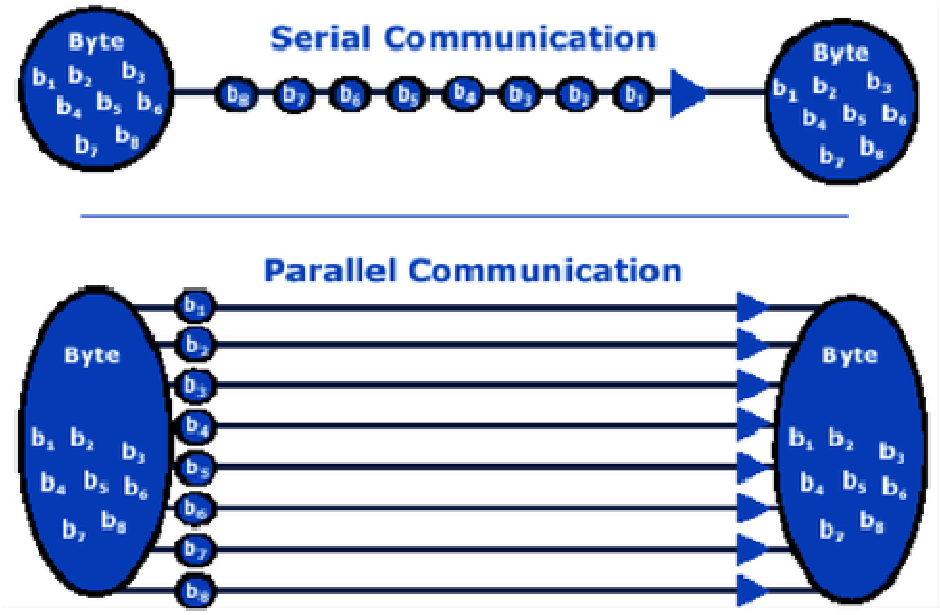

Figure 3.8: illustrates the serial input and parallel output.

Input/output Devices

The Address Bus

Remember from our talk of microprocessors earlier that each CPU has a number of pins that function as an address bus. In most cases, RAM chips are the kind of memory that is read from or written to using the address bus. Nevertheless, the address bus is often used by contemporary microprocessors for more than merely reading and writing to memory. The CPU may go from using the address bus to access Memory to utilizing the address bus to communicate with other semi-intelligent processors that are also linked to the address bus by flipping a specific pin. We are stated to be utilizing I/O port addressing in this situation rather than regular memory addresses. I find it a little strange when a port is referred to as a register since a register typically refers to an internal CPU register. The semi-intelligent device chips are only turned on when they see that the address bus has a memory value pointing to that particular chip and the special I/O

pin is asserted. Most input and output from devices like serial ports, parallel ports, floppy drives, hard drives, and other controllers happens in this manner. All RAM chips are momentarily deactivated and read or written from the external I/O chips after the CPU has set the correct address on the address bus and asserted the special I/O pin. Actually, the data is sent in bytes through a separate set of pins known as the data bus.

Information Bus

The CPU's data bus is nothing more than a set of pins used to transfer data into or out of the processor chip. All memory and I/O devices are linked to the data bus, but only one chip may actually be connected to the data bus at any one time depending on the state of the address bus and other control pins on the CPU. The size of the data bus might range from 4, 8, 16, 32, or even 64 bits, depending on the specific processor being utilized. The CPU can read and write more bits of data in a single operation with a bigger data bus. This method enables quicker I/O operations for certain devices when used with PCI-based cards on PC-compatibles. The device attached to the other end of the data bus, however, can only allow 4 or 8 bit transfers at a time, thus utilizing more bits in certain circumstances is a waste of time. It is crucial in this situation to set the unneeded bits to zero via a bit masking operation in order to make them invisible.

Requests to Interrupt

When external devices need the CPU's attention, they utilize a different pin in addition to the data bus, address bus, and special I/O pins that the processor uses for communication. An IRQ Line, or interrupt request line, is what this is known as. For instance, the keyboard controller device typically alerts the main processor that a key is accessible whenever you touch a key on the keyboard by asserting the interrupt line. As it may sometimes be called hundreds or even thousands of times per second, the interrupt handler needs to be compact and well built. An interrupt handler typically does the bare minimum of work required to maintain the device before leaving. The processor then resumes carrying out the paused operation as if nothing occurred. All CPUs typically have two distinct kinds of interrupt lines. The first kind is the sort of interruptions termed mask ables that we have been talking about up to this point. In this context, the term "maskable" refers to the software's ability to selectively activate or disable interrupts. Non-maskable interrupts are the other kind of interrupt. This kind of interruption cannot ever be disabled by software. It is most often employed to carry out the DRAM refresh on memory chips, which MUST take place on a regular basis in order to maintain the viability of memory contents.

There are other ways for the CPU to interface with external devices than Memory Mapped I/O I/O Port addressing. Memory mapped I/O is an additional frequently used method. In this instance, the CPU just accesses a memory address without asserting the I/O pin or addressing a data port. A little amount of RAM or ROM on the external device may be present, which the CPU will only access when necessary.

Memory Access Direct

The direct memory access capability is one method that has been utilized for years to expedite data transfer from main memory to the memory of an external device (DMA). DMA transfers are carried out by the external device's CPU entirely independent of the primary processor. This clearly requires cooperation from the processors. The primary processor is free to do other tasks

while the DMA transfer is in progress, but should wait until the transfer is over before attempting to alter the data in the buffer that is being transferred. When the transmission has begun, the primary processor may focus on other activities. Periodically, the external processor will take control of the address and data lines and perform the DMA transfer. The external device typically raises an interrupt request to inform the main CPU when the transfer is finished. The fundamental benefit of DMA is that each and every byte of data does not need to be transferred into a register before being saved to a memory location by the main processor. Another benefit is that the CPU may focus on other activities while the DMA transfer is taking place. It seems that this causes a speed boost overall.

Communication that is Synchronous, Asynchronous, and Iso-synchronous

Each fundamental piece of data, such as a bit, is sent synchronously in response to a clock communication signal, or, to put it another way, the data is transferred at a predetermined pace. A clock signal is thus required for this data transport mechanism. With asynchronous data transfer systems, there is no predetermined data rate of transmission and the data might be transferred at irregular intervals. In these systems, special bits, such as Start and Stop bits, are reserved to identify the beginning and ending of data transmission, and they also have an error checking mechanism.

Between the other two modes of data transmission, isochronous data transfer is somewhere in the middle. It uses a Synchronous transmission technology to convey Asynchronous data. Each data source in such systems is only granted a certain amount of time to deliver its data. The data source is free to send data at any intervals within that set period of time. If it contains data that can be processed in less time than allocated, it will simply squander the additional time by being idle. Otherwise, it transmits the remaining data in its subsequent turn if it has data that takes longer time to transmit than is allotted. Because of the severe time requirements, these systems lack an error check mechanism that would allow for re-transmission of the data in the event of a mistake. The three types of transmission—synchronous, asynchronous, and iso-synchronous—are not three of a kind; rather, they are two independent pairings, with the possibility that the asynchronous transmission that varies from synchronous transmission may not also vary from iso-synchronous transmission. Both pairings are, of course, about time.

CHAPTER 4

SERIAL COMMUNICATION PROTOCOLS

K. Gopala Krishna, Associate Professor,
Department of Electronics and Communication Engineering,Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-k.gopalakrishna@jainuniversity.ac.in

Throughout the last several decades, a number of communication protocols have been created based on serial communication. Serial Peripheral Interface, or SPI It is a communication system with three wires. There is a wire for clock pulses and one for each of the directions from master to slave. An extra SS (Slave Select) line is included and is often used to transmit and receive data between different ICs.Inter-Integrated Circuit (I2C): This more sophisticated version of USART is pronounced eye-two-see or eye-square-see. The maximum transmission speed is an astounding 400KHz. The I2C bus includes two wires: a clock wire and a bidirectional data line, which is why it is often (but not always—there are specific circumstances) referred to as a two-wire interface (TWI). It is a very recent and ground-breaking technology created by Philips.

Apple created the high-speed buses known as FireWire, which can transmit music and video. Depending on the port in Figure 4.1, which may be either a 4-pin, 6-pin, or 8-pin one in Figure 4.2, the bus has a number of wires.

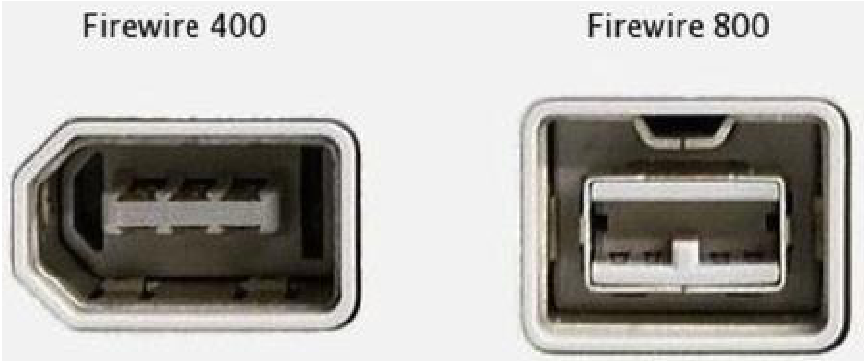

Figure 4.1: illustrates the bus contains a number of wires depending upon the port.

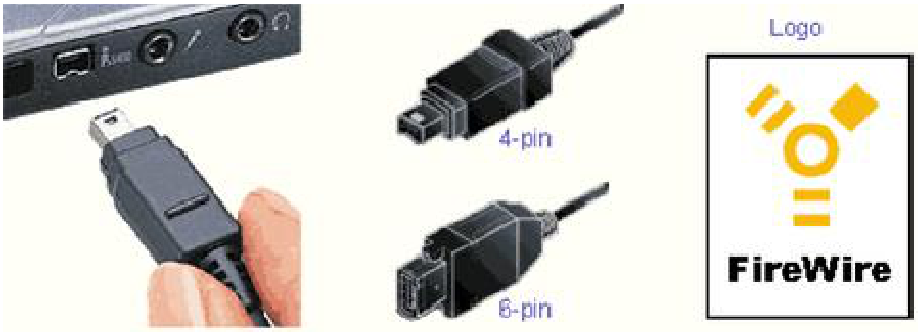

Figure 4.2: illustrates the several component of fire wire.

Ethernet: The bus, which has 8 lines, or 4 Tx/Rx pairs, is mostly used in LAN connections. The most well-known of these is the universal serial bus (USB) is used for almost all connections. VCC, Ground, Data+, and Data- are the bus's 4 lines.

USB Plugs

Regulation 232: A DB9 connection is commonly used to connect an RS-232 device. It includes nine pins, five of which are used for input, three for output, and one for ground. Several older Computers still include this so-called "Serial" port. The RS232 and USART of AVR microcontrollers will be the key topics of discussion in our next blogs.

CPU Design for PIC Microcontrollers

PIC Microcontroller: The smallest microcontrollers in the world, PIC (Programmable Interface Controllers) microcontrollers may be configured to do a wide variety of activities. Several electronic products, including phones, embedded systems, computer control structures, alarm systems, etc., include these microcontrollers.

While there are many different kinds of microcontrollers, the best may be found in the GENIE line of programmable microcontrollers. Software called circuit-wizard is used to simulate and program these microcontrollers. Every PIC microcontroller has a stack and a few registers, with the stack serving as return addresses storage and the registers acting as random access memory (RAM). RAM, flash memory, Timers/Counters, EEPROM, I/O Ports, USART, CCP (Capture/Compare/PWM module), SSP, Comparator, ADC (analog to digital converter), PSP (parallel slave port), LCD, and ICSP are the primary characteristics of PIC microcontrollers (in circuit serial programming) According to its internal design, the 8-bit PIC microcontroller may be divided into four categories, including Base Line PIC, Mid Range PIC, Improved Mid-Range PIC, and PIC18 (Figure 4.3).

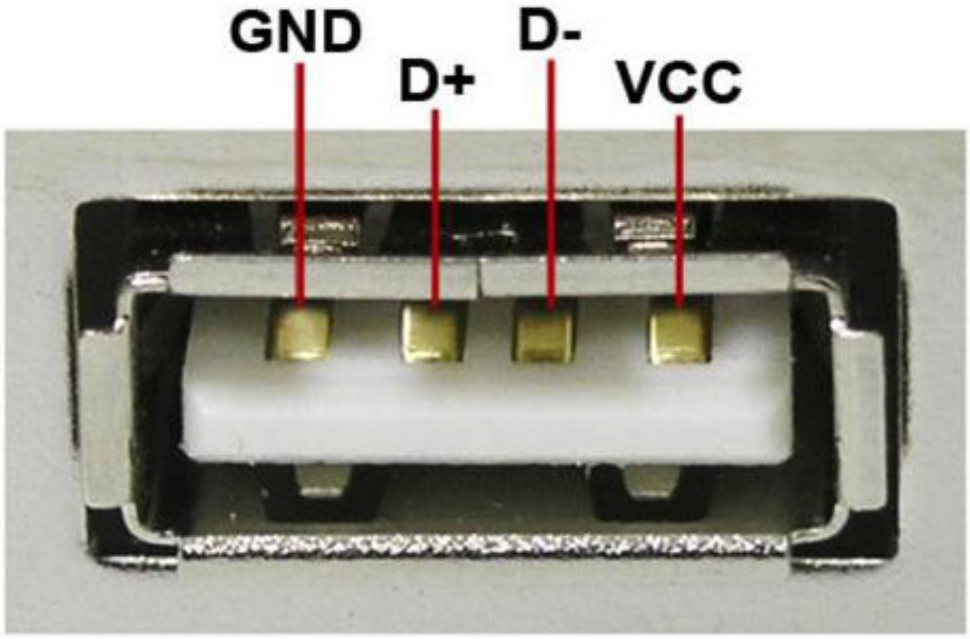

Figure 4.3: illustrates the Enhanced Mid-Range PIC and PIC18.

Architecture of PIC Microcontroller

The PIC microcontroller architecture comprises of CPU, I/Oports, memory organization in Figure 4.4,A/D converter, timers/counters, interrupts, serial communication, oscillator and CCP module which are discussed in detailed below.

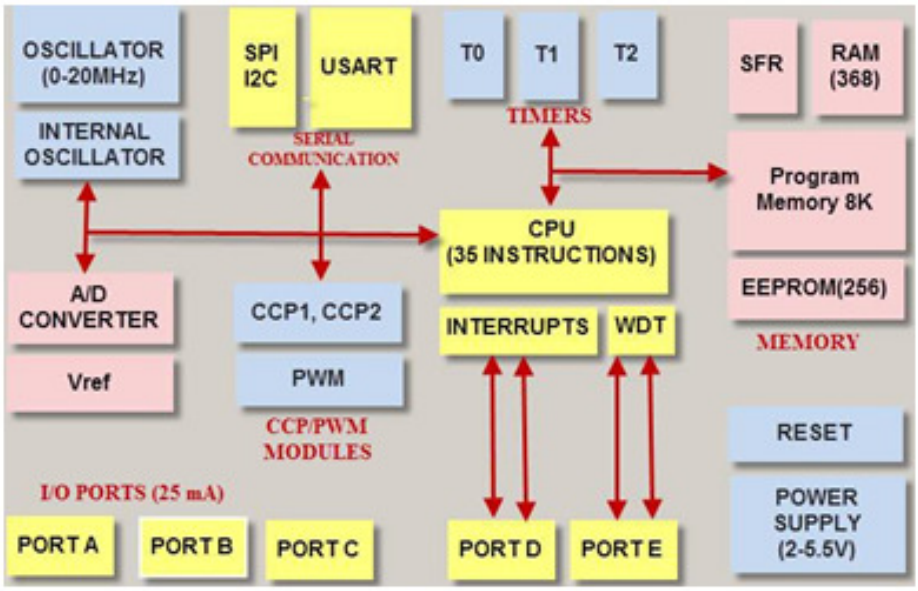

Figure 4.4: illustrates the PIC microcontroller architecture comprises of CPU.

CPU (Central Processing Unit)

The PIC microcontroller CPU is similar to other microcontroller CPUs in that it includes the ALU, CU, MU, and accumulator, among other components. The primary functions of an arithmetic logic unit are arithmetic operations and making logical judgments. After processing, the instructions are stored in memory. A control unit attached to the CPU is used to operate internal and external peripherals, while an accumulator is used to store results and power subsequent processes.

Memory Management

The RAM (Random Access Memory), ROM (Read Only Memory), and STACK components of the memory module make up the PIC microcontroller architecture.

Access Memory Device (RAM)

Data is briefly stored in RAM's registers, which is an unstable memory. Each of the two banks in the RAM memory has a certain number of registers. Special Function Registers (SFR) and General Purpose Registers are the two kinds of RAM registers (GPR).

Registered for All Purposes (GPR)

As their name suggests, these records are solely used for general purposes. For instance, the PIC microcontroller may be used to multiply two values. Typically, we multiply numbers in registers and store the results in other registers. Hence, these registers have no particular purpose; the CPU may readily retrieve the data in the registers.

Registers for Special Functions

As implied by the term SFR, these registers are exclusively used for special reasons. These registers can only be utilized for the tasks that have been allocated to them; otherwise, they will behave as expected. These registers are used, for instance, to display the operation or status of the program if the STATUS register cannot be utilized to store the data. The SFR's function cannot be changed by the user since it was predetermined by the store at the time of production figure 4.5.

Figure 4.5: illustrates the function is given by the retailer at the time of manufacturing.

Memory Organization

Read only Memory (ROM)

A reliable memory that is used to store data permanently is read-only memory. In the PIC microcontroller architecture, the architectural ROM holds the instructions or program, and the microcontroller behaves in accordance with the program.

The ROM is also known as program memory. It is where users create programs for microcontrollers, store them permanently, and then wait for the CPU to execute them. The performance of the microcontrollers is influenced by the CPU's execution of the instruction.

Programmable Electrically Erasable Just Read Memory (EEPROM)

The microcontroller can only be used once with a program written in a standard ROM; it cannot be used again. Yet, we may repeatedly program the ROM in EEPROMs.

Swift Memory

Moreover, flash memory has programmable read-only memory (PROM), allowing us to read, write, and delete programs many times. Typically, this kind of ROM is used by the PIC microcontroller.

Stack

The PIC microcontroller must first execute the interrupt and the active process address when an interrupt occurs. After that, the execution is saved on the stack. The microcontroller calls the

process using the address saved in the stack after finishing the interrupt execution and starts the process.

Ports of I/O

Five ports, including Port A, Port B, Port C, Port D, and Port E, make up the PIC16 series. Depending on the state of the TRISA (Tradoc Intelligence Support Activity) register, Port A, a 16-bit port, may be utilized as an input or output port. An 8-bit port called Port B may be utilized for both input and output. Port C is an 8-bit, and the TRISC register's state determines whether it is being used for input or output. A slave port for connecting to the microprocessor BUS, Port D is an 8-bit port.

The control signals for the analog to digital converter are another use for Port E, a 3-bit port. Data is sent and received between peripherals using the BUS BUS protocol. It is divided into two categories, such as address and data bus.

Data Bus: It is solely used to send or receive data.

The memory address in Figure 4.6 is sent from the peripherals to the CPU via the address bus. I/O pins are used to connect to external peripherals, and the serial communication protocols USART and UART are both utilized to connect to devices like GSM, GPS, Bluetooth, IR, and others.

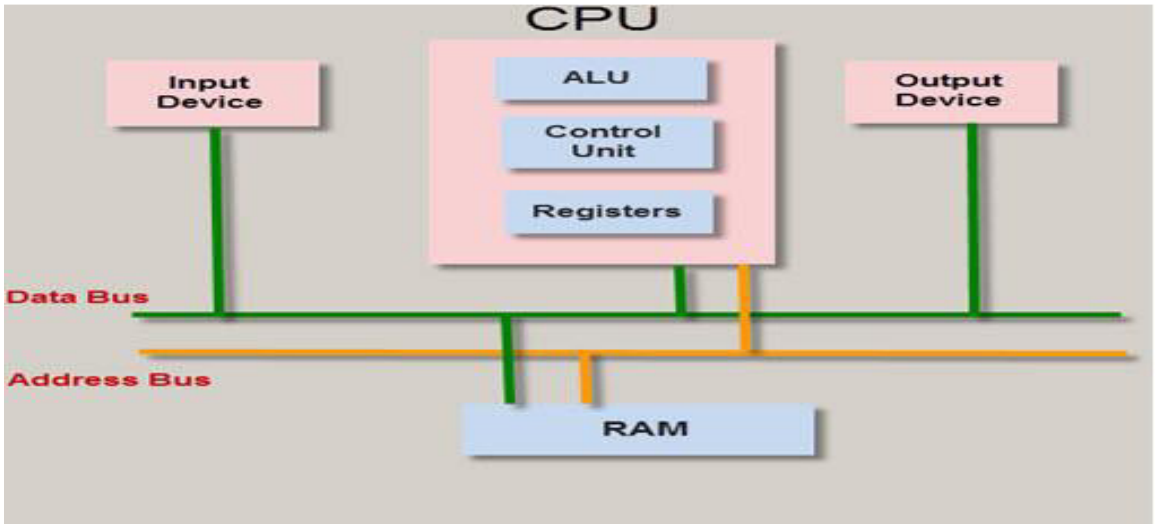

Figure 4.6: illustrates the Address bus is used to transmit the memory address.

A/D Converters

This analog to digital converter's primary goal is to translate analog voltage values into digital voltage values. The PIC microcontroller's A/D module has 5 inputs for devices with 28 pins and 8 inputs for devices with 40 pins.

The ADCON0 and ADCON1 special registers in Figure 4.7 regulate how the analog to digital converter operates.

The converter's lower bits are saved in register ADRESL, while its higher bits are kept in register ADRESH. It needs a 5V analog reference voltage for this operation.

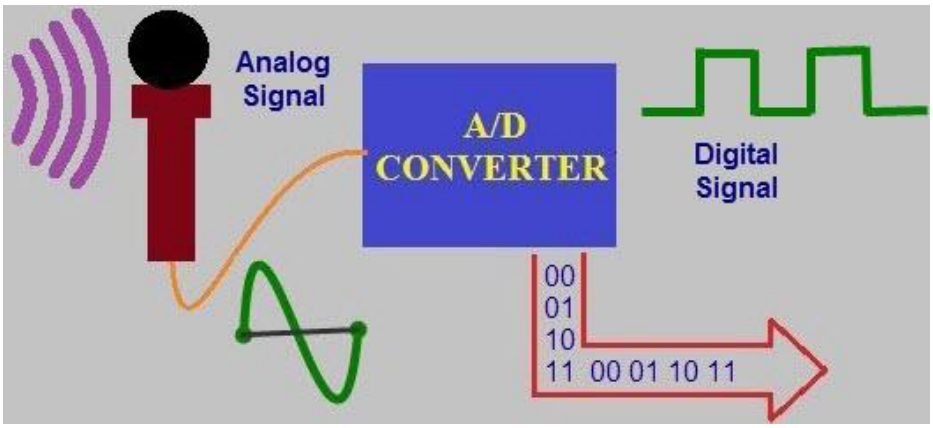

Figure 4.7: Illustrates the A/DCONVERTER

Timers/Counters

The PIC microcontroller contains four timer/counters, one of which is an 8-bit timer. The other three timers may operate in either 8-bit or 16-bit mode. Timers are used to produce accurate activities, such as setting up particular time intervals between two tasks.

Interrupts

The 20 internal interrupts and three external interrupt sources on the PIC microcontroller are connected to various peripherals like the ADC, USART, Timers, and so on.

Communicating in Series

USART: The term USART refers to a serial communication device that supports two different protocols and stands for Universal Synchronous and Asynchronous Receiver and Transmitter. With regard to clock pulses, it is used to send and receive data bit by bit across a single wire. TXD and RXD are the only two pins on the PIC microcontroller. These pins are used for serial data transmission and reception.

SPI Protocol: Serial Peripheral Interface is what the acronym SPI refers to. The PIC microcontroller and additional peripherals, including SD cards, sensors, and shift registers, communicate via this protocol. Two devices may communicate across three wires in SPI using a PIC microcontroller and a shared clock source. SPI protocol offers a higher data rate than USART.

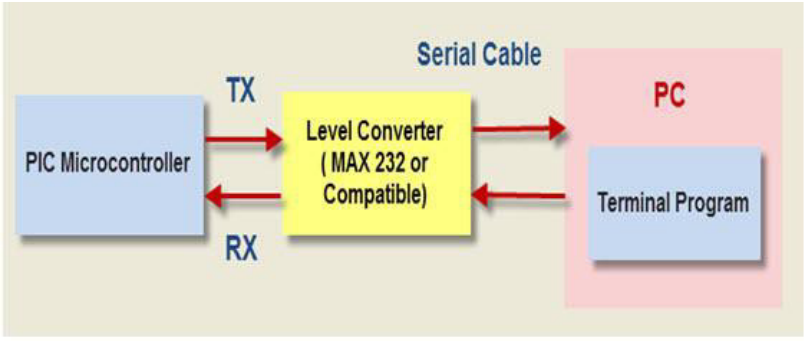

Figure 4.8: Illustrates the pic microcontroller with input and output.

Inter Integrated Circuit is the abbreviation for the I2C serial protocol, which is used to link low-speed devices such EEPROMs, microcontrollers, A/D converters, etc. Figure 4.8 shows how the PIC microcontroller supports I2C communication between two devices that may function as both master and slave devices.

Serial Communication

Oscillators

Timing is generated using oscillators. External oscillators like RC oscillators or crystal oscillators are used in pic microcontrollers between the two oscillator pins, which is where the crystal oscillator is linked. Each pin that controls the oscillator's operating mode is coupled to a capacitor whose value. The three modes are crystal, high-speed, and low-power. The resistor and capacitor values of RC oscillators define the clock frequency, which ranges from 30 KHz to 4 MHz.

Module CCP

The CCP module is an acronym for capture/compare/PWM and operates in three different modes, including capture, compare, and PWM. Capture Mode: When the CCP pin goes high, the Timer1 value is captured in capture mode, which records the time at which a signal arrives. In comparison mode, an analog comparator is used. An output is produced when the timer1 value reaches a certain reference value.PWM Mode: PWM mode offers configurable duty cycle and pulse width modulated output with a 10-bit resolution.

Uses of the PIC Microcontroller

The PIC microcontroller projects may be used for a variety of purposes, including video games, audio accessories, and peripherals. The project that follows shows how the PIC microcontroller works so that you may understand it better.

A street light that illuminates when a moving vehicle is detected:

The major goal of this project is to save energy by turning off the trailing lights and turning on a block of street lights ahead of moving cars that are detected on highways. This project uses embedded C or assembly language to program a PIC microcontroller. The power supply converts, filters, rectifies, and regulates the AC mains supply to provide power to the whole circuit. All lights will switch off when there are no cars on the road in order to save electricity. On the road, IR sensors are positioned to detect vehicle movement. When there are cars on the road, the IR sensor detects the movement of the car right away and promptly sends orders to the PIC microcontroller to turn on/off the Lights. As a vehicle approaches the sensor, a number of LEDs will light on, and when the vehicle moves away from the sensor, the intensity of the LEDs will decrease and they will turn off.

PIC Microcontroller Benefits:

PIC microcontrollers are reliable, and their proportion of defective units is quite low. Due to the usage of RISC architecture, the PIC microcontroller performs exceptionally quickly. Power consumption is quite low compared to other microcontrollers, and programming is also very simple. It is simple to interface an analog device without additional hardware.

PIC Microcontroller drawbacks: The use of RISC architecture results in a lengthy program (35 instructions). There is just one accumulator, and program memory is not used.

Function of Reset

One of the most cutting-edge features accessible on all contemporary microcontrollers is the reset function. The PIC16F8xx family has many different reset types. The several types of reset options available on the PIC 16F877.The picture below displays a condensed block diagram of the on-chip Reset circuit. His paper includes several examples of assembly language x86 applications. Programming in x86 assembly language is a difficult subject because. There are several assemblers available, including MASM, NASM, gas, as86, TASM, a86, Terse, and more. All use assembly languages that are very different. You must code differently for Linux, OS/X, Windows, and other platforms. ELF, COFF, Win32, OMF, a.out for Linux, a.out for FreeBSD, rdf, IEEE-695, as86, and many more object file formats are among them. You will often be calling functions that are found in other libraries or the operating system, so you will need to be familiar with certain technical aspects of how libraries are linked. Not all linkers operate in the same manner. There are several variations between the 32-bit and 64-bit operating modes used by contemporary x86 CPUs.

We'll include examples created in Linux and Win32 for NASM, MASM, and gas. Even a section on DOS assembly language applications will be included for historical context. These notes are not meant to teach you assembly language or to replace the documentation that comes with the CPU and the assemblers. Its only purpose is to demonstrate how to link and assemble programs using various assemblers and linkers.

Put together Linkers

In Figure 4.9, each assembly language file is assembled into a "object file" and linked with other object files to create an executable. Programmers often utilize libraries for things like I/O and arithmetic. A "static library" is essentially nothing more than a collection of (presumably related) object files.

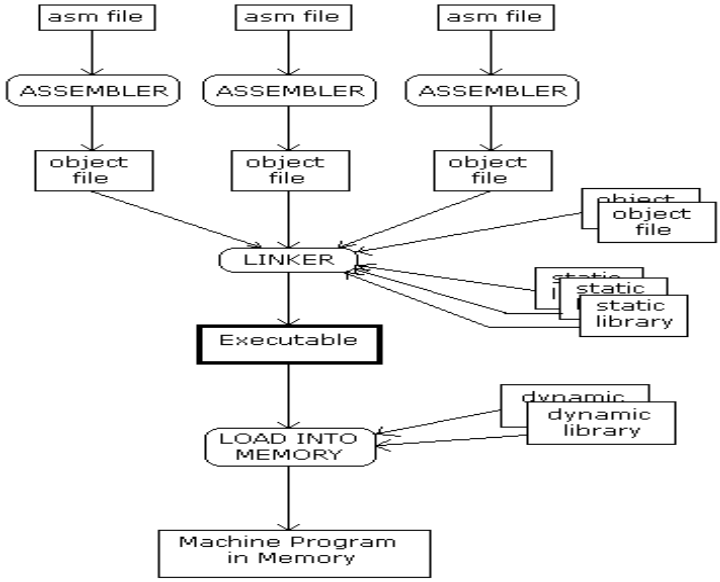

Figure 4.9: illustrates the flow diagram of assembler linkers.

Assemblers you Should know About Include

Microsoft Assembler, or MASM. While Microsoft's linker can convert them to win32 format, it produces OMF files. It supports an enormous, cumbersome assembly language. The addressing of memory is not obvious. Programming is unpleasant due of the commands needed to build up a program. The GNU assembler, or GAS. Many people dislike this because it utilizes the somewhat unattractive AT&T-style syntax; however, you can adjust it so that it uses and understands the Intel-style. It was intended to be a component of the GNU compiler collection's back end (gcc)."Net wide Assembler," or NASM the finest parts of it are that it is tiny, free, and that it can produce zillions of various kinds of object files. In many ways, the language is far more reasonable than MASM. Several object file types are available. Some that you need to be aware of are:

1. OMF: used in DOS but with 32-bit Windows extensions. Old.
2. AOUT: used in early versions of Linux and BSD
3. "Common object file format" (COFF)
4. Microsoft's version of COFF, Win32, isn't precisely the same! swaps out OMF.
5. Win64: The Win64 format from Microsoft.
6. Used in 32-bit Linux nowadays and other places are ELF and ELF32.
7. ELF64: Applied to 64-bit Linux and other platforms
8. macho32: 32-bit NeXTstep, OpenStep, Rhapsody, Darwin, and OS X
9. macho64: 64-bit NeXTstep, OpenStep, Rhapsody, Darwin, and OS X

Oscillator Choice Function

The PIC16F8xx series primarily supports PIC16F87XA devices as well as other oscillator types. Moreover, it contains a Watchdog Timer that can only be turned off by configuration settings. For more dependability (Configurations compared to typical microcontrollers/processors), it operates off of its own RC oscillator. The user may choose between the various oscillator modes with ease. Two configuration bits (foscillator1 and foscillator0) may be programmed to choose one of the fundamental four modes.

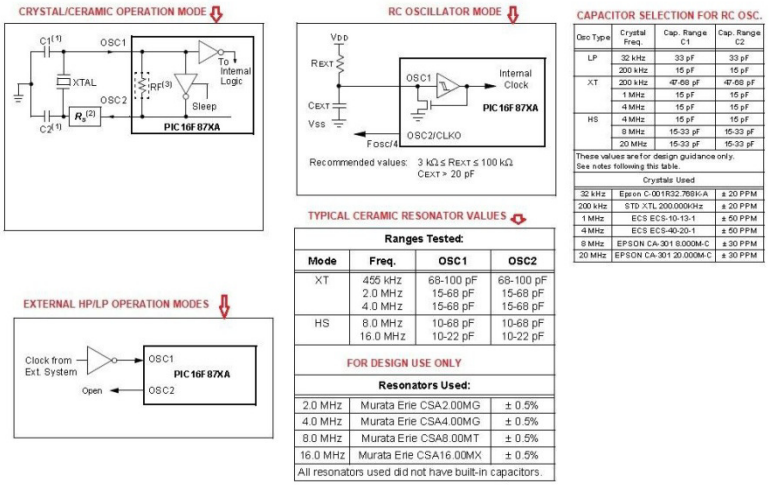

Figure 4.10: Illustrates the Oscillator Selection Function.

Figure 4.10 below shows the fundamental oscillator modes and typical values for various oscillators.

1. LPLow-PowerCrystal
2. XTCrystal/Resonator
3. HSHigh-SpeedCrystal/Resonator
4. RC resistor/capacitoroscillator

CHAPTER 5

PIC PROGRAMMING

Shweta Gupta, Associate Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-shwetagupta832000@gmail.com

The signal known as an interrupt is delivered to the microcontroller to indicate an occurrence that has to be attended to right away. This signal instructs the microcontroller to momentarily halt running the current program in order to run a specific code. It indicates that the microcontroller will be informed when the external device completes the duty placed on it and is free to access, receive, and utilize the information. Waiting for the phone to ring and interruptions are the same.

Microcontrollers include Interruption Sources

The microcontroller may be asked to temporarily halt running the current software for a variety of reasons: by using external hardware devices, such as pushing a particular key on a keyboard, which causes the microcontroller to receive an interrupt and read the information from the pushed key. The microcontroller has the ability to send interrupts to itself while the program is running in order to notify a programming problem. For instance, dividing by 0 will result in an interruption. The microcontrollers in the multi-processor system may interact with one another by sending interrupts to each other. For instance, they may transmit signals to one another to split up the job.

PIC Microcontrollers' Interrupt Types

With the PIC microcontroller, there are two interrupt kinds that might result in a break.

Software Interrupt: A software interrupt is a request sent to the processor to halt the execution of a program while it is being performed by a microcontroller, or we could say that it is created by internal peripherals of the microcontroller.

Hardware Interrupt: These interrupts are transmitted to specific microcontroller pins by external hardware devices.

Microcontroller PIC18F452 External Interrupt

Interrupt from the External Side in PIC18F452: Microcontrollers are sometimes linked to external devices. The microcontroller has to be aware of this circumstance in order to get the information from the external device if it needs to communicate it. Digital thermometers are one kind of external gadget.

It takes temperature readings and sends the findings to the microcontroller after each measurement. The goal of this article is to clarify how the microcontroller determines where to get the needed data from an external source.

Various Interruptions

There are two ways for the microcontroller and the outside device to communicate:

Polling

The external gadgets in this strategy are not independent. We establish how often the microcontroller must communicate with the external device. At the precise time period, the microcontroller contacts that device and obtains the necessary data. A polling approach is similar to checking your phone to see if you have any calls every few seconds. This method's primary flaw is the microcontroller's time wastage. It must wait and ascertain whether or not the fresh information has come.

Register Configuration for External Interrupt

The PIC18F452 has the following registers for interrupt operation, and at least one register may be used to regulate interrupt operation:

RCON (Reset Control Register) (Reset Control Register)

Interrupt Control Registers (INTCON, INTCON2, INTCON3) PIR1, PIR2 (Peripheral Interrupt Request Registers). PEI 1, PEI 2 (Peripheral Interrupt Enable Registers).

Reset Control Register: RCON Register

IPEN bit to activate the interrupt priority scheme; 1 = enable the interrupt priority level Additional bits were utilized to specify the reason for the reset. PD (Power on Detection flag), RI (Reset Instruction flag), TO (Watchdog Time Out flag), POR (Power on Reset status), and BOR (Brown out Reset status bit).

Three interrupt control registers, INTCON Register The read-write registers INTCON1, INTCON2, and INTCON3 include a variety of enable and flag bits.

When an Interrupt Situation Occurs, Interrupt Flag Bits are Set

Include bits for port B pin change, TMR0 overflow interrupt, and external interrupt enable, priority, and flag, Peripheral Interrupt Enable Register (PIE Register), Depending on how many peripheral interrupt sources there are, there might be more than one register (PIE1, PIE2). Include the distinct bits needed to activate or disable Use of peripheral interruptions.

PIR Register: Peripheral Interrupt Flag Register the number of peripheral interrupt sources may need more than one register (PIR1, PIR2) bits that indicate the interrupt that occurs (flags) When the interrupt occurred, corresponding bits were set.

Setting for EXTERNAL INTERRUPT Registers

The only use of the INTCON registers is to configure the external PIC interrupts. We shall go into more depth here since the PIC18F452's external interrupts are also covered in this article.

PERIODICAL INTERRUPT

The Peripheral (Internal) Interrupts are configured using the PIE (Peripheral Interrupt Enable) and PIR (Peripheral Interrupt Request) registers.

Global Interrupt Enable (GIE)

- To enable all interrupts on the PIC18F452, set this bit high.
- 1: Turn on all interruptions
- 0 = Turn off all interruptions
- Peripheral Interrupt Enable (PEIE)
- This bit is set high to enable all of the microcontroller's internal and peripheral interrupts.
- 1 = Turn on every peripheral interrupt
- 0 = Turn off each and every peripheral interrupt
- TMR0 Overflow Interrupt Enable: T0IE
- To enable the External Interrupt 0, set this bit high.
- Toggle TMR0 overflow interrupt with 1
- 0 = Turn off the TMR0 overflow interruption

External Interrupt Enable: INTE

- In order to allow external interrupts, this bit is set high.
- 1 = Turns on the external INT interrupt
- 0 = Turns off the external INT interrupt.
- RB Interrupt Enable: RBIE
- To enable the RB Port Change interrupt pin, set this bit high.
- Enables the RB port change interrupt when set to 1.
- The RB port change interrupt is disabled at 0
- TMR0 Overflow Interrupt Flag: T0IF
- TMR0 register overflow indicated by 1 (it must be cleared in software)
- 0 indicates that the TMR0 register is empty.
- INT External Interrupt Flag (INTTF)
- 1 indicates an INT external interrupt (it must be cleared in software)
- 0 indicates that there was no INT external interrupt.
- RB Port Change Interrupt Flag (RB-PF)
- 1 indicates that at least one RB7:RB4 pin has changed states (must be cleared in software)
- 0: No RB7:RB4 pins have undergone status changes.

TIMERS

The PIC16F887 microcontroller's timers may be succinctly explained in a single statement.

Three entirely separate timers/counters with the designations TMR0, TMR1, and TMR2 are present. But it's not quite that easy.

Clock TMR0

There are several practical uses for the timer TMR0. It is used in very few programs in some capacity.

Writing programs or subroutines for creating pulses of nearly any length, measuring time, or counting external pulses (events) is highly straightforward and simple to do with this technology.

The 8-bit timer/counter TMR0 module has the following features:

1. 8-bit timer/counter, 8-bit presale, programmable internal or external clock source, interrupt on overflow, and 8-bit presale (shared with watchdog timer)
2. External clock edge selection using programming.

Figure 5.1 below represents the timer TMR0 schematic with all bits which determine its operation. These bits are stored in the OPTION_REG Register.

OPTION_REG Register

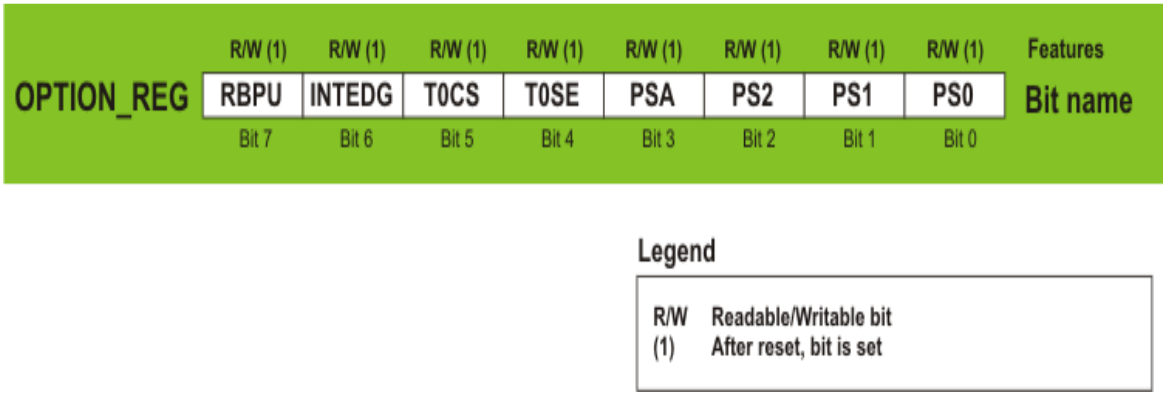

Figure 5.1: Timer TMR0 schematic with all bits which determine its operation

PORTB - RBPU Pull-up enable bit 0 allows PORTB pins to be linked to pull-up resistors, whereas pull-up enable bit 1 disables PORTB pull-up resistors.

Interrupt Edge (INTEDG) Choose bits 1 and 0 to interrupt on the rising and falling edges of the INT pin respectively (1-0).

TMR0 - T0CS Clock Choose bit 1 for the TMR0 timer/counter input receiving pulses through the RA4 pin, and choose bit 0 for the internal cycle clock (Fosc/4).

TMR0 Source Edge - T0SE Choose bit 1 to increment on the TMR0 pin's transition from high to low and 0 to increment on the transition from low to high.

The WDT is allocated a prescaler, and the TMR0 timer/counter is assigned a prescaler, according to the PSA (Prescaler Assignment) bit 1 and bit 0.

Prescaler Rate: PS2, PS1, and PS0 Choose bit:

These bits are combined to change the prescaler rate. The same set of bits results in a different presale rate for the timer/counter and watch-dog timer.

In Figures 5.2 and 5.3 below, the PSA bit's function is shown:

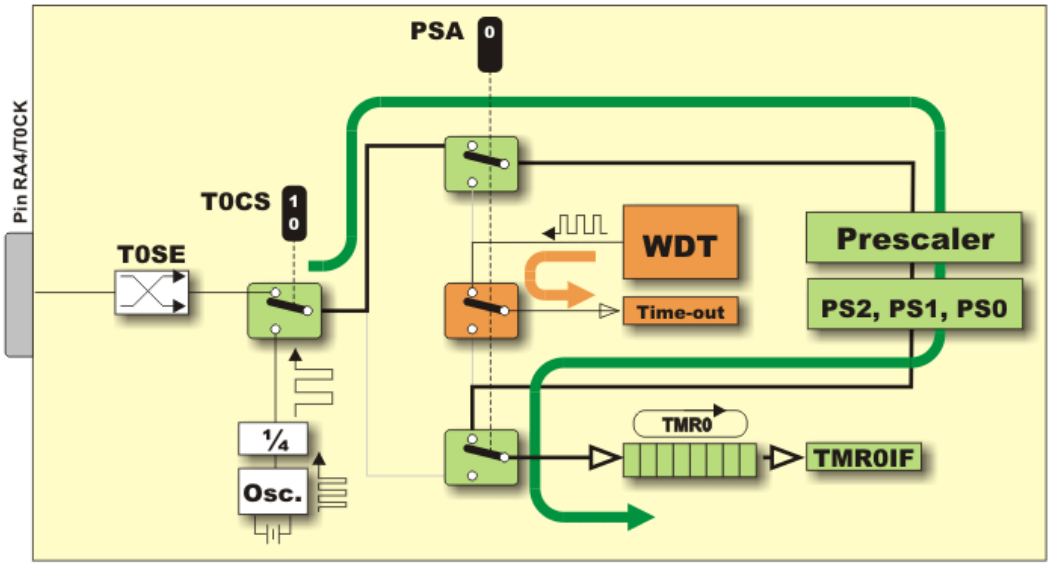

Figure 5.2: illustrates the functionofthePSAbit0.

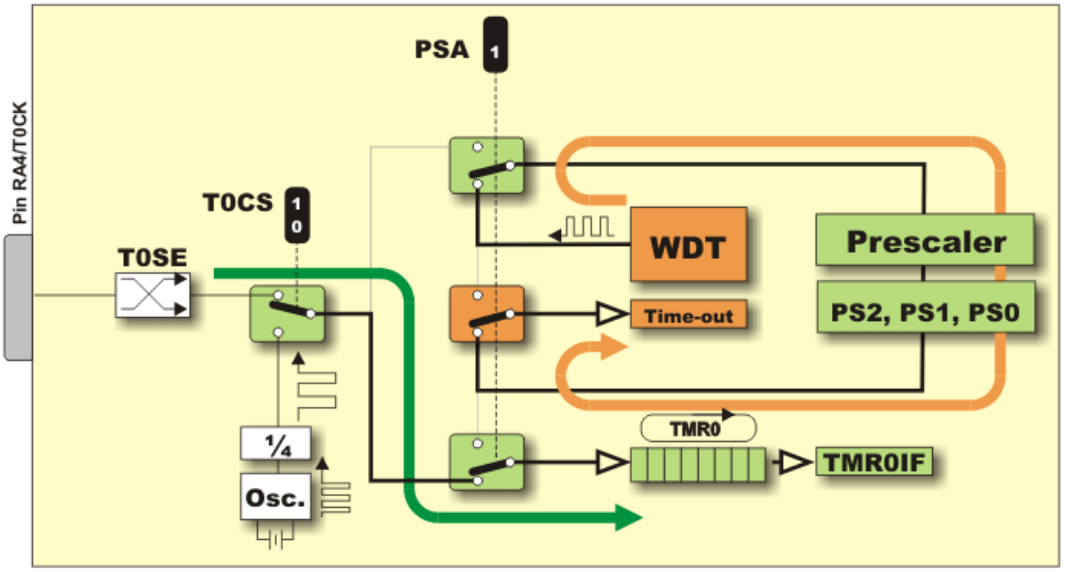

Figure 5.3. Thefunction ofthe PSAbit1

As can be seen, the prescaler's assignment to the watch-dog timer or timer/counter depends on the logic state of the PSA bit.

It is also important to note that:

Any write to the TMR0 register after the prescaler has been assigned to the timer/counter will cause the prescaler to be cleared; A CLRWDT instruction will clear the prescaler and WDT when the watch-dog timer is assigned to the prescaler; The pulse counting will begin after a two-instruction cycle delay when writing to the TMR0 register, which is utilized as a timer. Hence, the value written to the TMR0 register has to be adjusted; the oscillator is disabled when the microcontroller is set up in sleep mode. Since there are no pulses to count, overflow cannot

happen. The TMR0 overflow interrupt is unable to awaken the CPU from Sleep state due to this. A minimum pulse duration or gap between two pulses when used as an external clock counter without a prescaler must be 2 Tosc + 20 nS.The smallest pulse width or time interval between two pulses when used as an external clock counter with prescaler is 10nS; The user does not have access to the 8-bit prescaler register, hence it cannot be read or written to directly; To prevent reset while switching the prescaler assignment from TMR0 to the watch-dog timer, the next instruction sequence must be carried out:

Timer TMR1 The 16-bit timer/counter of the Timer TMR1 module consists of two registers (TMR1L and TMR1H). It can count up to 65.535 pulses in a cycle, or before the counting in Figure 5.4 begins from zero.

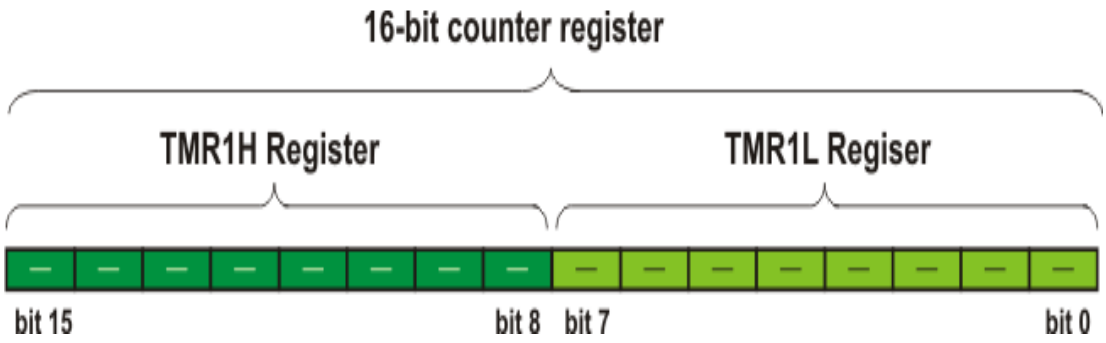

Figure 5.4: Illustrates the 16 bit counter register.

TimerTMR1

Similar to the timer TMR0, these registers can be read orwrittento at any moment. In case an overflow occurs, an interrupt is generated. The timer TMR1 module may operate in one of two basic modes- as a timer ora counter. However,unlike the timer TMR0, each of thesemodules has additional functions.Parts of the T1CON register are in control of theoperation of the timer TMR1 in Figure 5.5.

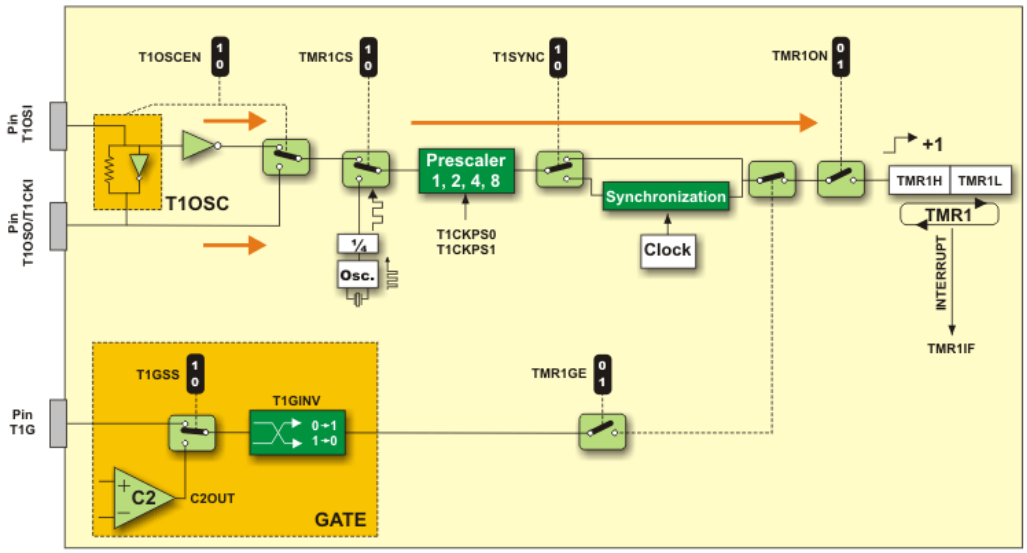

Figure 5.5: illustrates the TimerTMR1Overview.

TimerTMR1 Prescaler

The prescaler for timer TMR1 is entirely independent and supports 1, 2, 4, or 8 divisions of the clock input. The prescaler cannot be read or written to directly. Nevertheless, when writing to the TMR1H or TMR1L register, the prescaler counter is immediately emptied.

The RC0/T1OSO and RC1/T1OSI pins of the Timer TMR1 Oscillator are used to record pulses arriving from peripheral devices, but they also serve another purpose. These are set up as the input (pin RC1) and output (pin RC0), respectively, of the extra LP quartz oscillator, as can be seen in Figure 5.6 (low power).

This extra circuit is particularly intended for use with quartz crystals that have a frequency of 32,768 KHz or lower (up to 200 KHz). These crystals are utilized in quartz timepieces because it is simple to divide this frequency into pulses that last one second.

This oscillator can function even while in sleep mode since it is not dependent on internal clocking. Setting the T1OSCEN control bit of the T1CON register enables it. For appropriate oscillator startup, the user needs input a software time delay (a few milliseconds).

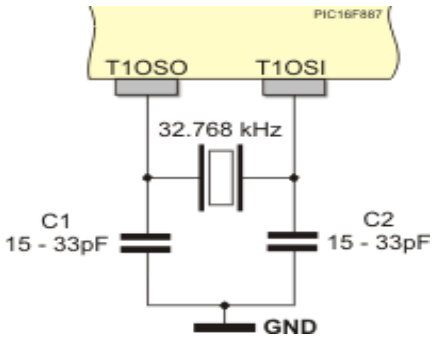

Figure 5.6: illustrates a softwaretimedelay(afewmilliseconds)toensureproperoscillatorstart-up.

The capacitor settings that work best with a quartz oscillator. These numbers do not need to be precise. The general rule is, however, that the stability increases with capacitor capacity, which also lengthens the period required for oscillator stability.

TMR1 Gate Timer

Software may be used to set the T1G pin or the output of comparator C2 as the timer 1 gate source. This gate enables the timer to directly clock analog events using the comparator C2 output or logical events using the T1G pin's logic state. Please see Figure 5.7. Enabling this gate and counting the pulses that flow through it is sufficient to determine a signal's length in timer mode on TMR1.

The TMR1CS bit must be cleared in order to choose this mode. Following this, the 16-bit register will be increased on every pulse originating from the internal oscillator. It will be increased every microsecond if the 4MHz quartz crystal is in use. As it counts internal clock pulses in this mode, the timer is unaffected by the T1SYNC bit. As these pulses are used throughout the electronics, synchronization is not necessary.

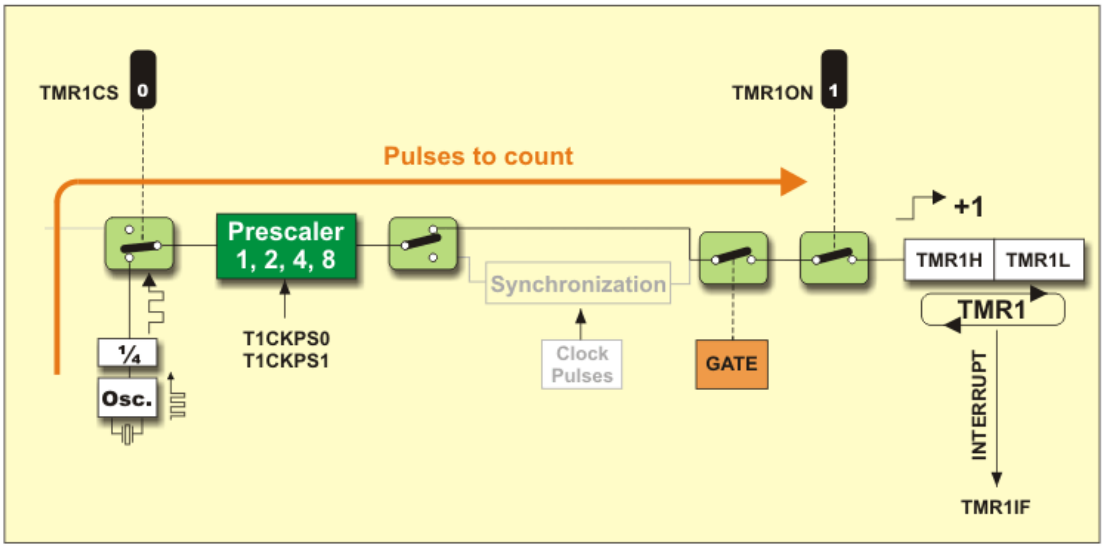

Figure 5.7: Illustrates the TMR1intimer mode

Since the clock oscillator on the microcontroller is not operating when it is in sleep mode, an interrupt from a timer register overflow is not possible.

TMR1 Oscillator Timer

In the Sleep mode, the microcontroller's power consumption is at its lowest. The oscillator must be stopped. By adding an SLEEP command to the program, it is simple to set the timer to this mode. When the microcontroller has to be awakened, a difficulty arises since only an interrupt can do it. Due to the microcontroller's tendency to "sleep," an interrupt has to be initiated by outside circuitry. If it becomes essential for Figure 5.8 "wake up" to happen at regular intervals, everything may become exceedingly difficult.

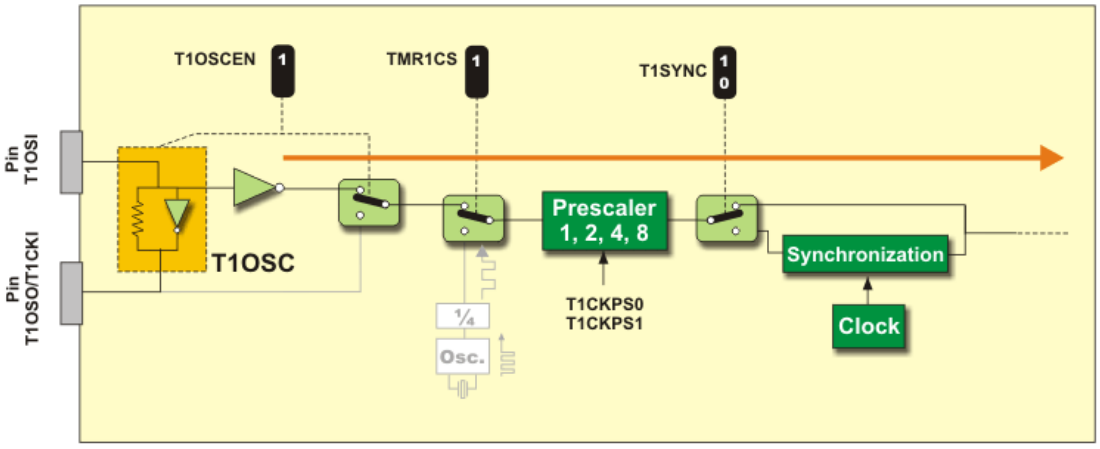

Figure 5.8: Illustrates the TimerTMR1Oscillator

The PIC16F887 microcontroller has a totally separate Low Power quartz oscillator that may run in sleep mode to address this issue. Simply said, what was formerly a separate circuit is now a part of the microcontroller and is connected to timer TMR1. By setting the T1OSCEN bit of the T1CON register, the oscillator is turned on. The timer TMR1 then utilizes pulse sequences from

that oscillator, which is determined by the TMR1CS bit of the same register. By setting the T1SYNC bit to zero, the quartz oscillator's signal is brought into phase with the microcontroller clock. The timer is unable to function in sleep mode in the situation.

In counter mode, TMR1

By setting the TMR1CS bit, timer TMR1 becomes a counter and begins to run. This indicates that on the rising edge of the external clock input T1CKI, the timer TMR1 is increased. On their route to the TMR1 register, the external clock inputs will be synced if control bit T1SYNC of the T1CON register is cleared. In other terms, the timer TMR1 is referred to as a synchronous counter and is synced to the microcontroller system clock.Although if clock pulses are seen on the input pins while the microcontroller is functioning in this manner and in sleep mode, the TMR1H and TMR1L timer registers are not incremented. Simply expressed, there are no clock inputs available for synchronization since the microcontroller system clock is not operational in this mode. Nevertheless, as Figure 5.9 just contains a simple frequency divider, the circuit will still function even if there are clock pulses on the pins.

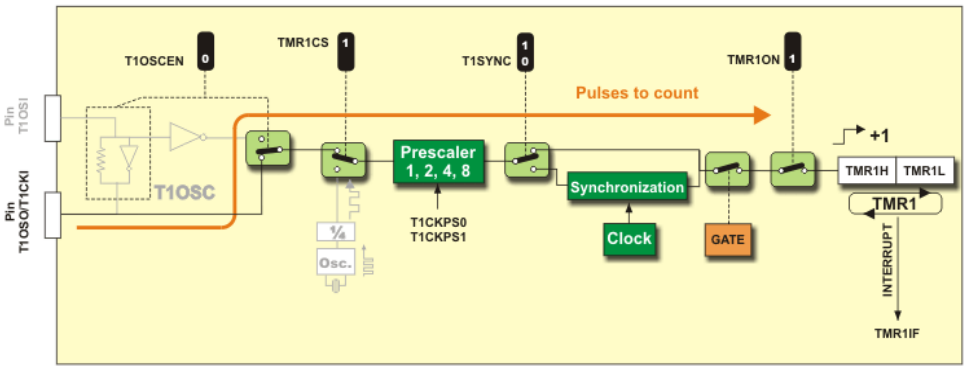

Figure 5.9: Thepresale willcontinuetoruniftherearеclockpulses onthe pins sinceitisjustasimplefrequencydivider.

CounterMode

This counterregisters alogicone (1)oninputpins.Itisimportanttounderstandthatatleastone fallingedgemustbe registeredpriortothefirstincrementonrisingedge.Refertofigureonthe left.Thearrows inFigure5.10denotecounterincrements.

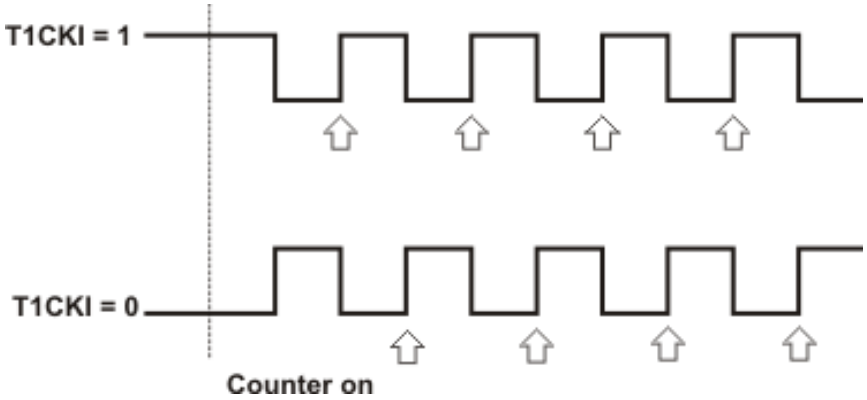

Figure 5.10: illustrates the waveform of counter increments

T1CONRegister

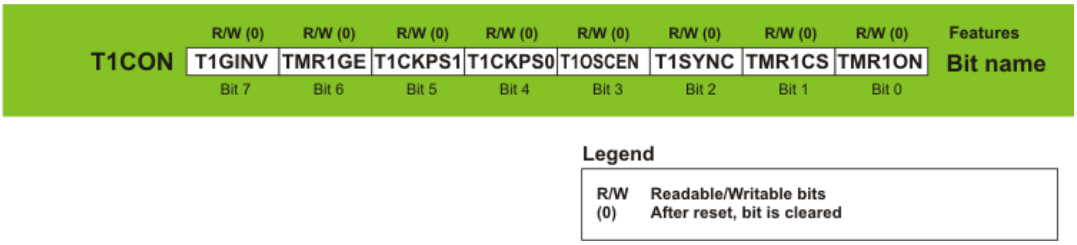

Figure 5.11: Illustrates the timer to mea sure time whilst the gate is high or low.

On the T1G pin gate or the comparator C2 output (C2OUT) gate, the T1GINV - Timer1 Gate Invert bit functions as a logic state inverter. In Figure 5.11, it allows the timer to measure the exact time whether the gate is open or closed.When the pin T1G or bit C2OUT gate is high, timer 1 counts as one (1), and when it is low, timer 1 counts as one (0).

Timer1 Gate (TMR1GE) the enable bit controls whether or not the comparator C2 output (C2OUT) gate on pin T1G will be active. Only when the timer TMR1 is active (bit TMR1ON = 1) does this bit take effect. If not, this piece is not considered. Timer 1 gate must be inactive for timer TMR1 to be on, and the 0 Gate has no effect on the timer TMR1.

LP Oscillator Enable Control bit: T1OSCEN

An oscillator with a low power need and a frequency of 32.768 kHz is enabled for the timer TMR1 clock if it is set to 1, and it is disabled if it is set to 0.The LP oscillator input or T1CKI pin input may be synchronized with the internal clock of the microcontroller using the T1SYNC - Timer1 External Clock Input Synchronization Control bit.

This bit is disregarded while counting pulses from the local clock source (bit TMR1CS = 0).1 indicates that external clock input should not be synchronized, whereas 0 indicates that it should be.

Timer TMR1 Clock Source Select bit (TMR1CS)

One counts the pulses on the T1CKI pin (on the rising edge of 0–1), while zero counts the pulses of the microcontroller's internal clock.TMR1ON - Timer1 On bit 1 enables the timer TMR1, whereas bit 0 disables it.The following activities must be carried out in order to correctly operate the timer TMR1:

The prescaler cannot be disabled, hence its rate must be changed using bits T1CKPS1 and T1CKPS0 of the register T1CON. The TMR1CS bit of the same register should be used to choose the mode (TMR1CS: 0 indicates a quartz oscillator as the clock source, 1 indicates an external clock source).

The TMR1 timer is activated and the TMR1H and TMR1L registers are increased on each clock input by setting the T1OSCEN bit of the same register. Clearing this bit causes the counting to halt, and clearing or writing the counter registers clears the prescaler.The flag TMR1IF is set by filling both timer registers, and counting begins at 0.As seen in Figure 5.12, the 8-bit timer TMR2 functions in a very precise manner.

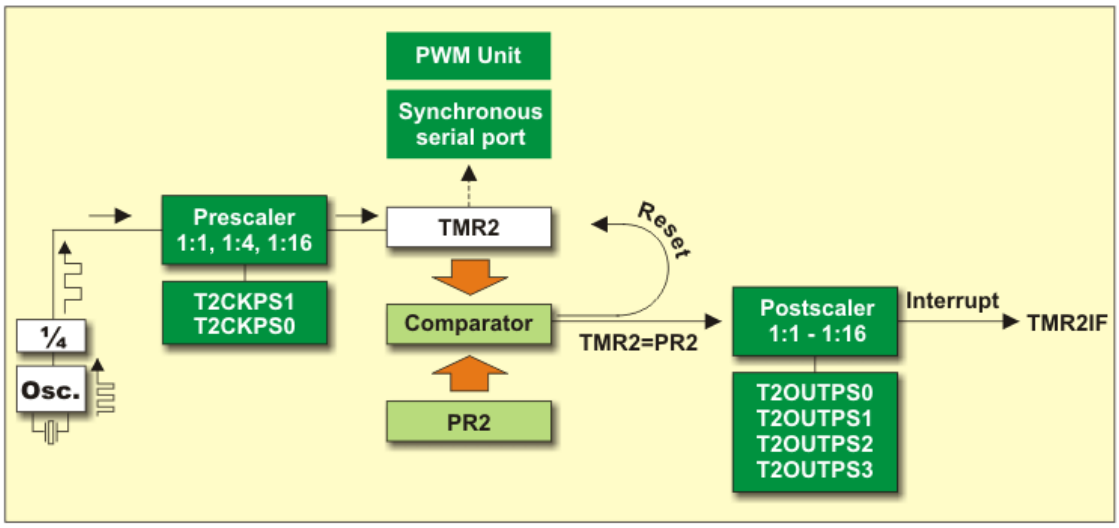

Figure 5.12: TimerTMR2moduleis an8-bittimerwhichoperates

The prescaler, whose rate may be altered by fusing the T2CKPS1 and T2CKPS0 bits, receives the pulses from the quartz oscillator first. The TMR2 register is then incremented beginning at 00h using the prescaler's output. TMR2 and PR2 values are continuously compared, and the TMR2 register is increased until it equals the PR2 value. The TMR2 register is automatically reset to 00h after a match. If it is enabled, the timer TMR2 Postscaler is incremented and its output is utilized to produce an interrupt. Both the TMR2 and PR2 registers allow for complete reading and writing. Clearing the TMR2ON bit makes counting stoppable and helps save power. As an extra option, the TMR2 reset time may also be utilized to calculate the baud rate for synchronous serial transmission. Many bits of the T2CON register regulate the timer TMR2.

T2CONRegister

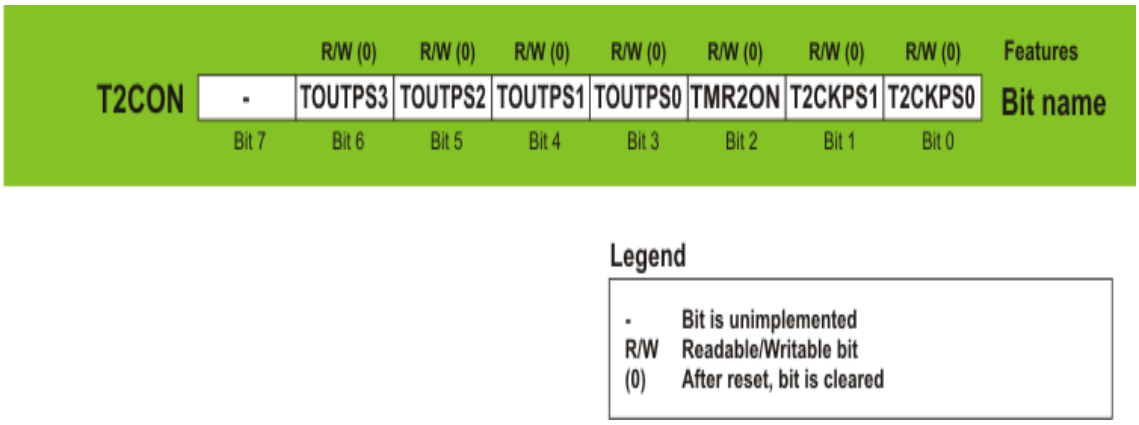

Figure 5.13: illustrates bitturnsthetimerTMR2.

Timer2: TMR2ON the timer TMR2 is turned on by bit 5.13.T2's timer is either on or off when it reads 1 or 0.The following particular information about the TMR2 timer's registers is important to understand when utilizing it: Prescaler and postscaler are cleared by writing to the TMR2 register, prescaler and postscaler are cleared by writing to the T2CON register, and the PR2 register initially has the value FFh. Prescaler and postscaler are both cleared on any reset.

LCD with PIC Microcontroller Interfacing

Sixteen characters the LCD module, which is a very simple LCD, is often used in electronics projects and goods. It has 2 rows, each of which can show 16 characters. The presentation of each character uses a 58 or 510 dot matrix. A microcontroller can simply be interfaced with it. In this tutorial, we'll demonstrate how to use the Hi-Tech C Compiler and a PIC Microcontroller to output data to an LCD. As Hi-Tech C lacks built-in LCD libraries, controlling an LCD requires hardware expertise. Figure 5.14 illustrates how commonly used LCD displays employ HD44780 compatible controllers.

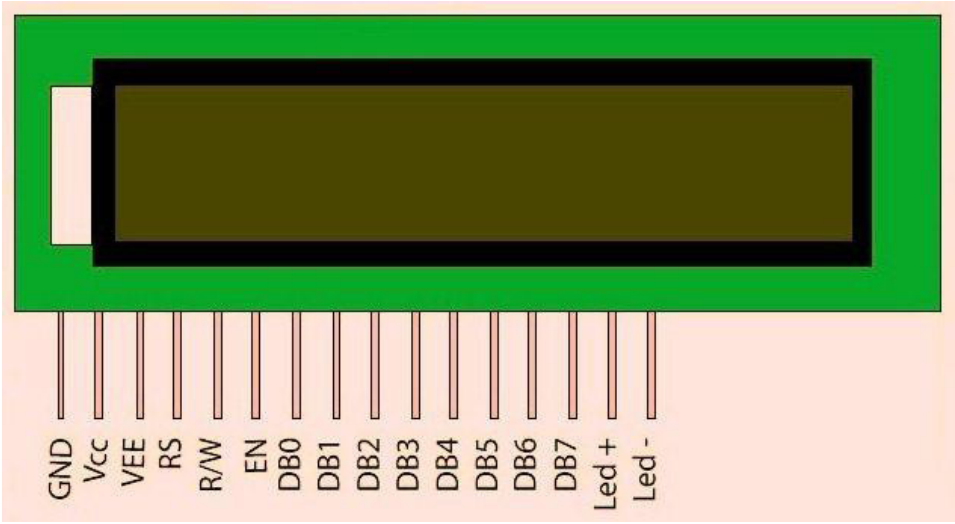

Figure 5.14: illustrates the 16×2LCDPinDiagram

This is a 162 Character LCD display's pin diagram. It also has two inputs for power—Vcc and GND—as do all gadgets. The display's contrast is determined by the voltage at the VEE. To change contrast, a 10K potentiometer with fixed ends linked to Vcc and GND and a variable end attached to VEE may be utilized. To activate this LCD module, a microcontroller must supply two pieces of information: data and commands. Command controls the other LCD actions, such as the location to be shown, while Data specifies the ASCII value (8 bits) of the character to be displayed. The same data lines used for sending data and commands are multiplexed utilizing the LCD's RS (Register Select) input. When it is HIGH, the LCD interprets the signal as data to be shown; when it is LOW, the LCD interprets the signal as a command. Data Strobe is provided utilizing the LCD's E (Enable) input. The LCD interprets HIGH for the E (Enable) as legitimate data or a command. Data is either written to or read from the LCD depending on the input signal R/W (Read or Write). Normal circumstances only need writing, thus the circuits below have it connected to GROUND. The way that data or instructions are sent to the LCD might vary depending on whether the interface between the LCD and the microcontroller is 8 bits or 4 bits. In the 8 bit mode, the data lines DB0 through DB7 are used to deliver 8 bit data and instructions, while the LCD's E input is used to provide data strobe. Yet just 4 data lines are used in 4 bit mode. This 8-bit system divides data and instructions into two separate chunks of four bits each, which are supplied consecutively over data lines DB4 through DB7 with a separate data strobe through the E input. In order to save microcontroller pins, the concept of 4 bit communication is established. It's possible that you believe 4 bit mode will be slower than 8 bit. Yet there isn't much of a difference in speed. The minimal speed difference between both modes is not

significant since LCDs are slow-moving gadgets. Just keep in mind that the microcontroller is running at a fast speed, about MHz, while we are looking at the LCD with our eyes.

CHAPTER 6

FUNCTIONS IN IDENTIFICATION

Ryan Dias, Assistant Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-ryan.dias@jainuniversity.ac.in

Lcd8_Init()&Lcd4_Init(): These functions will initialize the LCD Module connected to the correspondingly, the 8-bit and 4-bit modes' specified pins. Using the PIC Microcontroller's ADC with MPLAB XC8. This article will teach you how to utilize the MPLAB XC8 compiler with a PIC Microcontroller's ADC module. We'll utilize the widely accessible PIC 16F877A micro-controller for our example. All natural physical quantities, such as temperature, humidity, pressure, and force, have analog counterparts. These analog values must be converted to digital before being processed by a digital computer or microcontroller. Analog to digital converters are used for this. A device known as an Analog to Digital Converter, or ADC, transforms continuous analog quantities (in this case, voltage) into equivalent discrete digital values. With its 8 ADC inputs, the PIC 16F877A microcontroller can transform analog inputs into matching 10 bit digital numbers. Take ADC Lower Reference as 0V and Upper Reference as 5V for the sake of explanation.

Vref- = 0V

n = 10 bits, Vref+ = 5V.

Resolution = 5/1023 = 0.004887V (Vref+ - Vref-)/(2n - 1)

The least voltage needed to shift a bit is thus 0.00487V, or the ADC resolution. View the illustrations below. The PIC 16F877A's ADC module includes four registers.

1. A/D Result High Register: ADRESH
2. A/D Result Low Register, or ADRESL

Figure 6.1's ADC Block Diagram shows ADCON0, the A/D Control Register 0 and ADCON1, the A/D Control Register 1.

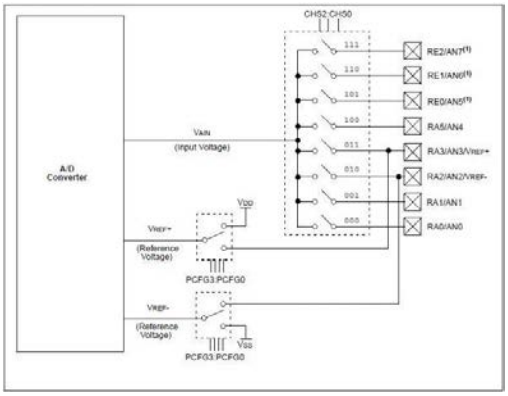

Figure 6.1: The working of ADC channel selection and reference voltage selection.

You can quickly comprehend how the ADC channel selection and reference voltage selection operate from this block diagram. To achieve the precision specified in the datasheet, the A/D Acquisition Time Holding capacitor (CHOLD) must be charged to the input voltage. So, in order to charge the capacitor, we must supply a delay longer than the minimum acquisition time needed. The datasheet's minimum time is 19.72 seconds. For further information, please see the datasheet.

The choice of A/D clock:

The A/D conversion clock must be chosen to guarantee the lowest TAD. TAD, or conversion time per bit, is 1.6 seconds. Please see Table 6.1 in the datasheet for further information.

Table 6.1 shows how the AD clock works in simple terms.

AD Clock Source (TAD)		Maximum Device Frequency
Operation	ADCS2:ADCS1:ADCS0	
2 Tosc	000	1.25 MHz
4 Tosc	100	2.5 MHz
8 Tosc	001	5 MHz
16 Tosc	101	10 MHz
32 Tosc	010	20 MHz
64 Tosc	110	20 MHz
RC(1, 2)	x11	(Note 1)

Note 1: The RC source has a typical TAD time of 4 µs but can vary between 2-6 µs.
2: When the device frequencies are greater than 1 MHz, the RC A/D conversion clock source is only recommended for Sleep operation.

ADC Clock Selection Table–PIC 16F877A

If you have previously read through our initial tutorials, PIC Microcontroller MPLAB XC8 Tutorials, you will have no trouble understanding the circuit. The program has VDD (5V) and VSS (GND) specified as the reference voltages for A/D conversion (see the code below). A potentiometer is used to provide an analog input to Channel 0, allowing us to adjust the input voltage from 0 to 5V. The analog input will be converted via A/D to a 10 bit digital value (0–1023). Ten LEDs linked to the microcontroller's PORTB and PORTC show this digital value.

Picfile Format in Hex

By providing all the information needed to program a PIC microcontroller in a single Hex file, Microchip has defeated Atmel AVR. Code, EEPROM data, User bytes (User ID), and configuration words—most importantly—all fall under this category. Because all the necessary information is in one file, it is significantly simpler to move the project from development to production or between engineers. The AVR has no analogous system, thus explaining fuse settings for it is never easy. Typical CONFIG directives for the Kanda PIC18F include CONFIG WDT=OFF and deactivate the watchdog timer.

SET DEBUG = ON; ENABLE DEBUG MODE; CONFIG MCLRE = ON; MCLEAR Pin on

For further information, see CONFIG LVP = OFF in your compiler documentation. The names of the configuration bits differ from PIC device to PIC device; for further information, refer to the "Special Features of CPU section" in the PIC device datasheets.

The most recent version of MPLAB X handles Configuration Bytes in C files differently. The CONFIG and CONFIG directives still function with MPLAB X assembler files, but the C compiler needs a different structure. It necessitates the usage of the phrase #pragma config WDTE = ON. Using a built-in Configuration Bytes Tool is the simplest approach to produce the necessary #pragma directives. A list of the potential configuration bytes accessible on the device that is set in the project appears in a new window. For PIC18F chips, the available Configuration bytes vary, but the process is the same. Input your desired settings and press the "Generate Source Code to Output" button. This generates the code that you can either put in a separate C file or use #include directive in your main source file, or you can cut and paste it.

Format of Hex File for Pic16f Devices

Due to the PIC16F devices' lower size and comparable file format to the PIC18F file, extended addressing is not used. For information on the most recent PIC16F1xxx chips, check the dedicated section below.

Code: The first line of a.hex file is always code. Several compilers and assemblers have varied layouts, varying things like the amount of data per line and whether or not blank lines are used. Since the PIC16F devices only support 14-bit instructions, code is saved as Words with the lowest-ordered byte. As a result, an empty space shows up as FF3F. Nevertheless, addressing is done in bytes.

EEPROM Data: If the device contains EEPROM, the data is located in the HEX file at addresses 0x4200 and above. Word format is used for storage, however only the bottom byte of each word carries data; the upper byte is always zero and is ignored.

Word for configuration: For the majority of PIC16F, there is just one 14-bit configuration word, which is saved at address 0x400E. The high byte is saved first. A few gadgets have up to three words. User Information: Stored at 0x4000, up to 8 bytes. Close of File: All Intel Hex files include the following End Of File marker: 00000001FF.

Tool for Programming

A computer program used by software developers to design, debug, maintain, or otherwise support other programs and applications is known as a programming tool or software development tool. The phrase often refers to relatively straightforward programs that may be used in combination to complete a job, much as how many hand tools could be used to mend a real thing. One sign of a talented software engineer is their capacity for effective tool utilization.

1. Tools for binary compatibility analysis
2. Bug registries: Systems for tracking issues, including bug tracking systems, are compared.
3. Tools for building: Build automation, Software for build automation
4. Contact graph

Coverage of codes: Coverage of codes

1. Software coverage instruments.
2. List of resources for code reviews

Websites for exchanging code: GitHub, Sourceforge, Freshmeat, and Krugle. See also search engines for code.Tools for compilation and linking Microsoft Visual Studio, CodeWarrior, Xcode, GNU toolchain, and gcc.

Debuggers: List of debuggers (debugger). Also see debugging.

Disassemblers: tools used for reverse engineering. Producing documentation Comparison of asciidoc, help2man, and Plain Old Documentation as documentation generators

Formal techniques: Specification, development, and verification methods using mathematics Interface creators for the GUI

Generators of library interfaces: Tools for SWIG Integration, Programming languages (like C and C++) that provide manual memory management and, therefore, the potential for memory leaks and other issues, typically utilize memory debuggers. They help to maximize memory utilization effectiveness. Examples include dmalloc, Valgrind, Electric Fence, and Insure++.

Parser generators: Software for creating parsers List of performance analysis tools for performance analysis or profiling. Comparison of revision control software and a list of revision control programs. PHP, Awk, Perl, Python, REXX, Ruby, Shell, and Tcl are some scripting languages.

1. Using grep and find
2. Code of origin Clones/Duplications Duplicate code detection#Tools
3. Editor for source code
4. Comparison of text editors and a list of text editors
5. Formatting for source code: indent
6. Tools for creating source code: automated coding Implementations
7. Analysis of static code: lint, List of static code analysis tools
8. List of unit testing frameworks for testing inside units.

IDE

Integrated development environments mix together the functions of several tools. For instance, they make it simpler to do certain activities like looking for material exclusively in files associated with a particular project.

IDEs might be used, for instance, to create enterprise-level software. This comparison of integrated development environments includes various features of IDEs for certain programming languages.

Controlling PIC

Electronic display devices known as multiplexed displays do not drive the full display at once. Instead, the display's sub-units—typically rows or columns for a dot matrix display, individual characters for a character-oriented display, and occasionally individual display elements—are multiplexed, or driven one at a time.

However, the electronics and the persistence of vision work together to give the impression that the entire display is active at all times.

A multiplexed display offers various benefits over a non-multiplexed display, including the requirement for fewer wires—often much fewer wires—the use of simpler driving circuits, which may decrease cost, and lower power usage.

Two general categories may be used to categorize multiplexed displays:

1. Character-centric visuals
2. Screens that are pixel-focused

Character-centric visuals

The majority of character-oriented displays, including seven-, fourteen-, and sixteen-segment displays, show a whole character at once. When the "row" and "column" lines of the two-dimensional diode matrix are at the proper electrical potential, the different portions of each letter will glow. In order to keep the various "row" and "column" lines of the matrix electrically segregated from one another, the light-emitting device often takes the shape of a light-emitting diode (LED). The junction of the row and column is not at all conductive in liquid crystal displays. The plates of several separate triode vacuum tubes that share the same vacuum enclosure are what are lighted in the VCR display example above. Just one digit is lighted at a time due to the triode grid arrangements. All of the comparable plates across all of the numbers are linked together in parallel, for instance, all of the lower-left plates across all of the digits. The microcontroller powering the display turns on each digit individually by applying a positive voltage to its grid and then to the corresponding plates. The grid of that digit conducts electrons, which impact the plates with a positive potential. The display would have needed 49 wires just for the digits, plus additional wires for all of the other lit indications, if it had been constructed with each section independently linked. Just seven "digit selector" lines and seven "segment selector" lines are required thanks to multiplexing the display. The extra indications (in our example, "VCR," "Hi-Fi," "STEREO," "SAP," etc.) are organized to seem as if they were extra segments of one or more existing digits and are scanned using the same multiplexing technique as the actual digits.

The majority of character-oriented displays drive each of the necessary digit segments concurrently. Just one section is driven by a select few character-oriented displays at once. One example of this was the display on the Hewlett-Packard HP-35. The calculator made use of the fact that extremely short pulses of light are viewed as brighter than longer pulses of light with the same time-integral of intensity. This effect is caused by the functioning of pulsed LEDs.Similar in layout to a multiplexed display, a keyboard matrix circuit offers many of the same benefits. Some individuals choose to "share" the wires between a multiplexed display and a keyboard matrix in order to further minimize the amount of cables.

Screens that are Pixel-focused

In contrast, individual pixels in dot matrix displays are found where the "row" and "column" lines of the matrix converge, and each pixel is individually controllable. Here, the wire savings are much more noticeable. Non-multiplexed control would need 2,359,296 cables for a common 1024x768 (XGA) computer screen. That many cables would be totally unworkable. Nevertheless, just 1792 wires are required when the pixels are organized into a multiplexed matrix, which is an entirely feasible condition. A single pixel or a whole row or column of pixels

may be driven concurrently by pixel-oriented displays. Each pixel on active-matrix liquid crystal displays has a storage element so that it can maintain its proper state even when it is not being powered.

Electric Washing Machine Controller

It's 2007. My beloved Aurora T-5502 washing machine became stuck during the spin cycle and spun incessantly. When I realized what had happened, it was already too late and the equipment had sustained significant damage. Cause? Timer that is electromechanical. The contacts couldn't rotate as they were intended to since it was old and worn out. The old timer had to be replaced since it couldn't be fixed. The timer resembled an antique clock in which various functions are activated when the hands move over certain contacts along the route. The device offers setup, two brief programs, and four separate complete programs:

Washing Machine Internals

Three sections make up the system:

1. The display screen, Board of directors the washing machine's interface
2. The command console
3. It is the user's interface. Contains:
4. LCD display (16x2)
5. 3-button row (Left, Right, Enter)
6. Power led in red (to show the power stage is working)
7. A led yellow (show established link to pc, if connected)

Piezoelectric Sound Source (to generate tones)

The LCD is driven by himself (microcontroller). The HD44780 was the one I used, and the datasheet has the protocol. I avoided the hassle by using the library. Unfortunately, the driver's high EMI sensitivity led to a lot of problems. When purchasing inexpensive buttons, extra care must be used since the metallic parts bounce at an extremely microscopic level when you push the button. The bounce often causes the electric current to be interrupted up to many times for a few microseconds before closing the circuit fully as intended. The microcontroller may mistakenly believe that the button has been pushed more once since it is quick enough. There are two choices: equipment filters (low pass), Sort it with software. The sounder, which can produce a variety of tones, is directly linked to the microcontroller.

Board of Directors

It serves as the centerpiece of the design. It houses the microcontroller and establishes connections to the accessories. Eagle CAD was used to create the design. It's available on Github. To conserve money, the PCB is composed of low-cost Pertinax (FR-2). There are no high frequencies to provide a better quality. The power source comes first. 9VAC 1A (a 220v/9v transformer with fuse) is the input voltage. The motherboard has a diode bridge that provides 12VDC for the relays, servo, and voltage regulator (LM7805) that provides 5VDC power for the LCD and micro. The motherboard is linked to peripherals through housing connections, which makes taking the motherboard out fairly simple.

The PIC 16F877PI at 4 MHz is the microcontroller in use. There was no reason to run more often. In addition, the tone generator library for the piezo sounder performed better at this

frequency. There has an ICSP socket for programming it, enabling immediate firmware upgrades without removing the micro. Using opt couplers, the power stage has optical isolation (HPCL-817). The firmware was created using C code and the PCM compiler from CCS, one of the best PIC compilers available at the time. The firmware requires some ugly tactics to put them in since it takes around 85% of the ROM.

CHAPTER 7

INTERFACE WITH THE WASHING MACHINE

Vinay Kumar S B, Assistant Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-sb.vinaykumar@jainuniversity.ac.in

It consists in a 'panel' of relays, acting as switches to control the electric flow to the different components of the machine: The panel diagram in Figure 7.1:

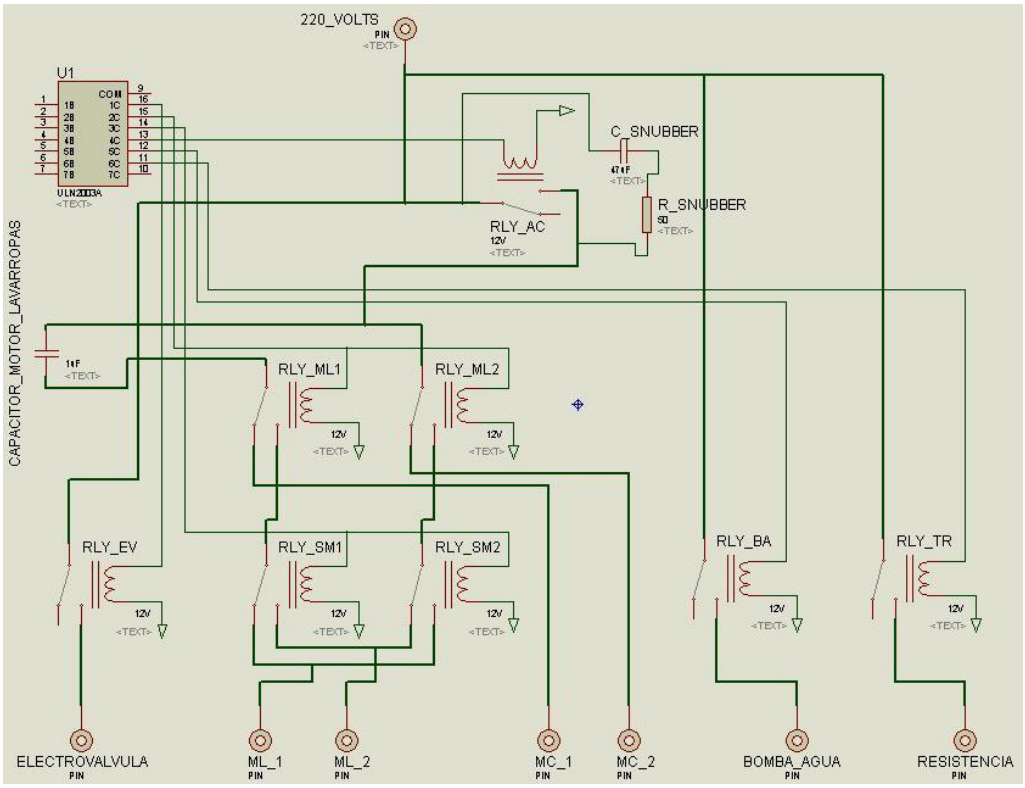

Figure 7.1: Illustrates the electric flow to the different components of the machine.

The actual panel:

An electrically regulated water intake is an electrovalve (RLY EV). When an electrical current was introduced, it allowed water to flow. The dispenser system receives the water immediately. Using resistance to heat the water, the heater (RLY TR) Drain pump (RLY BA): to drain the wash drum's water. The drum is rotated by the motor (RLY AC, RLY ML1, RLY ML2, RLY SM1, RLY SM2). One motor with two separate coils powers my gadget. One for the spin cycle and one for slow speed.

The connections to the heater, pump, and electro valve were all quite simple. I'll thus skip the justification. I had to explore to figure out how to make the motor interface function. The findings are as follows: Let's give the connecting pins numbers (from left to right). Upper row 1,

2, 3, and bottom row 4, 5, and 6. The relays RLY MLx switch between the high speed coil (for spinning) MC x and the low speed coil (for typical washing) ML x.

The RLY SM relays regulate rotation whenever the low speed coil is engaged (see panel diagram). They have no impact while the high speed coil is operating. The motor is switched on and off by the relay RLY AC. The EMI of the commutation is decreased by the snubber, which is terrible for heavy inductive loads like the motor. I utilized an array of Darlington transistors to regulate the relays (ULN2003AN). When driving inductive loads, each Darlington comes with a freewheeling diode to safeguard the transistors from coil auto induction (Figure 7.2).

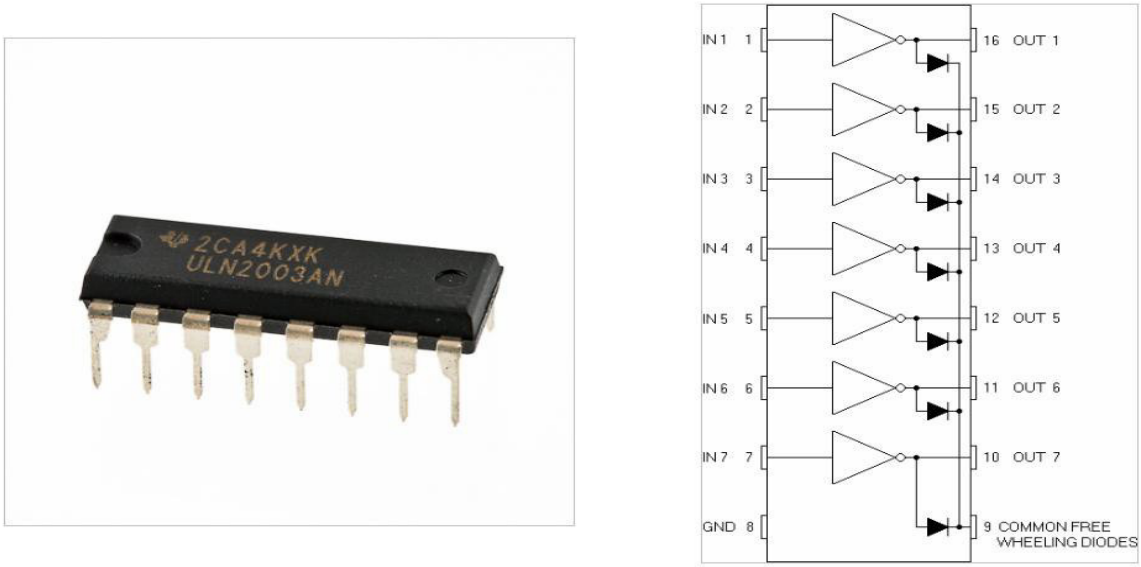

Figure 7.2: Illustrates the circuit diagram of wheeling diodes.

Sensors

By detecting the air pressure that the water exerts in a certain air chamber, a pressure switch may be used to determine when the water has reached a certain level. It is used as a standard switch. Door switches are really main switches that are only turned off by doors. The machine turns on when the door shuts. I continue to use it as intended. There is no lock mechanism within the door switch. To detect water temperature and operate the heater, the temperature sensor was initially a bimetalic thermostat. I used the thermostat's original metallic casing to house the LM35 temperature sensor, replacing the old one entirely. Super glue was used to attach the LM35 to the housing.

Dispensing Device

There is just one water entrance on this device, and a rotating arm directs the water flow into the various compartments. The electromechanical timer was initially in charge of controlling the arm. I used a cheap model aircraft servo to do the same task (I took the idea from Pablo Canello)

A small bit of bricolage inventiveness was used to connect the servo to the rotatory arm (green). Three lines make up the servo: VCC, GND, and signal. The PIC will provide a PWM signal at 50Hz to regulate the position of the servo. The linear translation of a pulse variation between 1 ms and about 2.5 ms is 0 degrees and 180 degrees, respectively. The servo will use all of its

effort to move to the desired position as long as the PWM signal is available. The servo maintains its previous position and does not attempt to counteract externally imposed forces if the signal is stopped. The servo will always be placed in the appropriate compartment before filling water.With the use of a calibration method, the servo position corresponding to each dispenser compartment will be recorded in order to operate the rotatory arm accurately (a once time operation). Each location will be saved in the PIC EEPROM as a number between 0 and 255 (one byte).

Operation

By way of simplification, choosing the washing settings is the first step in the process. The following tasks cycle repeatedly after it: On the dispenser system, choose the appropriate compartment. Until the pressure switch shuts, load the water Wash in alternate directions and drain the water if the spin option wasn't omitted while choosing the parameters, the operation ends with the spinning cycle. In embedded control applications, interacting with sensors and actuators. From basic scientific and analytical measurements to consumer electronics, the use of sensors in embedded electronic devices is widespread. However, each type of sensor—whether it be a temperature sensor, humidity sensor, strain gauge, or RTD—presents its own set of challenges when interacting with an embedded controller. Contemporary analytical and laboratory equipment generates a ton of data and has a clear communication demand. They employ remarkable technology that is implemented with ever-more-complex circuits, yet people want user-friendly front panel controls with straightforward graphical user interfaces.

integrated controllers from Mosaic Industries and their wide-ranging growth These requirements are met by I/O modules known as Wildcards, which offer computing power for the instrument's application software, a GUI for local operation, mix-and-match I/O for sensing and control, serial communications for interface to other serial-controlled instruments and sensors, Ethernet and TCP/IP for connection to something like a local area network (LAN), and web service for communication via remote web browsers. A smooth and transparent interface between sensors and actuators is provided by this group of hardware and software components. For connecting to sensors and actuators, Mosaic's embedded controllers encourage a high degree of software/hardware integration. All Mosaic sensor interface controllers come with pre-coded device drivers that provide programmers using either the C or Forth programming languages complete high level access to their functionalities. Data collection, pulse width modulation, motor control, PID control, sensor calibration, frequency measurement, data analysis, data recording, analog control, and communications are all made easier by pre-coded I/O drivers. You can easily complete your application thanks to extensive documentation and precoded example applications. Using a PIC microcontroller and PWM to control the speed of a DC motor

You may believe that a DC motor's speed may be adjusted by connecting a variable resistor in series with it. "Resistor is not a suitable option for regulating the speed of a DC Motor" is unjustified for three different reasons. The fundamental issue is that a resistor cannot perform this function since the motor has a variable electrical load. When compared to while it is operating, starting up requires more electricity. As a mechanical load is given to the motor shaft, it also draws greater current. Excess energy is released as heat by the resistor. Thus, it is bad for a battery-powered gadget. Motors, as we all know, demand greater current, hence resistors with larger power ratings are needed to dissipate more energy.

A PIC Microcontroller's built-in CCP module makes it simple to produce PWM. Capture/Compare/PWM is also known as CCP. There are a variety of PIC Microcontrollers that come with CCP modules. The majority of them have several CCP modules. Our work is made extremely easy by the built-in library functions for PWM in MikroC Pro for PIC Microcontroller. L293D Motor Driver is used in this example project to link a DC motor with a PIC microcontroller. There are two push-button switches available to adjust the motor's speed. Here, a 12V DC motor is being used, and the average DC value that is given to the motor may be changed by adjusting the PWM's duty ratio. The typical DC voltage for a duty cycle of 0% is 0 volts, 25% is 3 volts, 50% is 6 volts, 75% is 9 volts, and 100% is 12 volts.

System for Real-Time Operating

The Macintosh user and development communities gain a lot from OS X. These advantages include better networking functionality, greater support for industry standards, improved reliability and performance, and an object-based system programming interface. Apple substantially redesigned the Mac OS core operating system when it developed OS X. The kernel is what OS X is built upon. Figure 7.3 shows how OS X is structured.

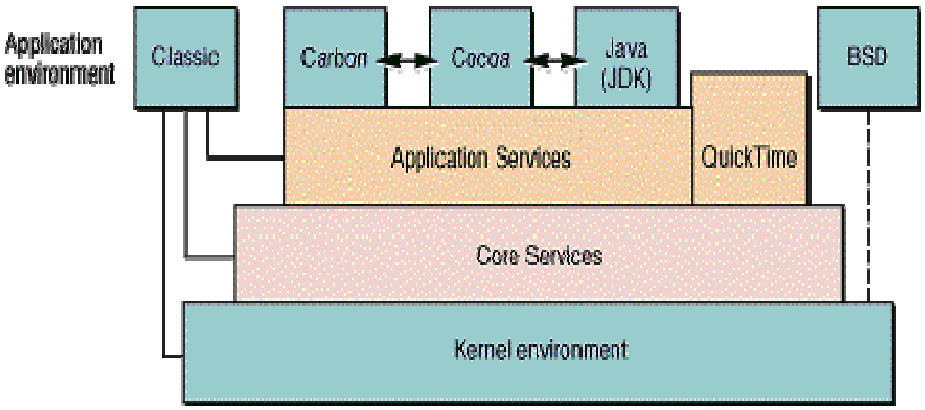

Figure 7.3: Illustrates the OSX architecture

There are several improvements made to OS X by the kernel. They include object-oriented APIs, preemption, memory protection, increased networking capabilities, greater performance, and compatibility for both Macintosh (Extended and Standard) and non-Macintosh (UFS, ISO 9660, and so on) file systems. Preemption and memory protection are two of these elements that contribute to a more stable environment. Applications work together to share CPU time in Mac OS 9. Similar to this, all apps share the computer's memory. Cooperative multitasking is possible in Mac OS 9. If even one program doesn't collaborate, the responsiveness of all procedures is hampered. On the other hand, time-sensitive, predictable behavior must be guaranteed for real-time applications like multimedia. OS X is a preemptive multitasking environment, in comparison. The OS X kernel enforces collaboration by arranging programs to share time (preemption). This enables real-time functionality in applications that need it.

Processes do not often share memory on OS X. The kernel, which manages access to these address spaces, instead gives each process its own address space. This control makes sure that no program may accidentally access or change the memory of another application (protection). The OS X virtual memory architecture allows each program access to a 4 GB address space, therefore size is not a concern. All apps are referred to be running in user space when taken as a whole,

although this does not mean that they share memory. The collective address spaces of all user-level apps are referred to as user space. Kernel space refers to the address space used by the kernel itself. No program running on OS X is able to change the system software's memory directly (the kernel). There is still a chance for communication (and potentially memory sharing) between apps, even if user processes do not share memory by default as in Mac OS 9. For instance, the kernel provides a wide range of primitives to enable certain information exchange across processes. POSIX shared memory, frameworks, and shared libraries are examples of these primitives. Another strategy is provided by Mach messaging, which transfers memory from one process to another. Nevertheless, unlike Mac OS 9, memory sharing cannot take place without the programmer's express permission.

Darwin

A project that is open source is the OS X kernel. Along with other essential components of OS X, the kernel is referred to as Darwin. A full operating system called Darwin is built on many of the same technologies as OS X. Nevertheless, Apple's exclusive graphics and application layers, including as Quartz, QuickTime, Cocoa, Carbon, and OpenGL, are not included in Darwin. The link between Darwin and OS X. These systems have a same kernel, but OS X includes QuickTime, Core Services, and Application Services in addition to the Classic, Carbon, Cocoa, and Java (JDK) application environments. The BSD command-line application environment is present in both Darwin and OS X; however, since OS X does not require its usage, the user cannot access it unless they specifically want to do so.

BSD, Mach 3.0, and Apple technologies form the foundation of Darwin technology. The best part of Darwin technology is that it is open source, giving developers complete access to the source code. Third-party OS X developers effectively have access to the Darwin core system software development team. Moreover, programmers may copy (or alter) code from Apple's core operating system and utilize it in their own products by observing how the company operates. For further information, see the Apple Public Source License (APSL). Since OS X and Darwin are both based on the same software, low-level applications may be developed once and operate on both platforms with few, if any, modifications. The software's interaction with the application environment is most likely the sole distinction.

Darwin is based on established science from several sources. This technology borrows heavily on FreeBSD, a 4.4BSD variant that provides cutting-edge networking, performance, security, and compatibility capabilities. Other system software components, such as Mach, are based on NeXT technology as well as prior Apple MkLinux and OS X Server technologies. A large portion of the code is cross-platform. The source code for the whole core operating system is accessible. The primary technologies were selected for a number of factors. For dealing with memory management, interprocess (and interprocessor) communication (IPC), and other low-level operating-system operations, Mach offers a clear set of abstractions. This offers a helpful layer of insulation between the operating system and the underlying hardware in the quickly evolving hardware environment of today.

BSD is a sophisticated, meticulously developed operating system with a wide range of features. In actuality, a significant amount of BSD code may be found in the majority of commercial UNIX and UNIX-like operating systems today. A group of industry-standard APIs are also made available by BSD. Apple has created and constructed new technologies like the I/O Kit and Network Kernel Extensions (NKEs) to benefit from cutting-edge features like those offered by

an object-oriented programming approach. These cutting-edge innovations are combined with tried-and-true industry standards in OS X to provide a dependable, adaptable, and expandable operating system.

Architecture

As shown in Figure 7.4, the foundation layer of Darwin and OS X is made up of many architectural elements. These elements make up the kernel environment when combined.

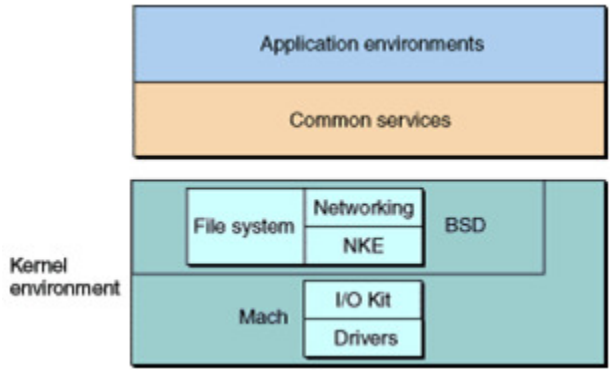

Figure 7.4 OSX kernel architecture

Remember that OS X uses the word "kernel" a little differently than you may anticipate. According to conventional operating-system nomenclature, a kernel is a tiny piece of software that merely offers the basic infrastructure required to enable additional operating-system functions. — taken from The Design and Implementation of the BSD 4.4 Operating System, 1996 study by McKusick, Bostic, Karels, and Quarterman.

Similar to this, in conventional Mach-based operating systems, the kernel only makes reference to the Mach microkernel and disregards further low-level code, which is necessary for very little of what Mach performs. Yet with OS X, the Mach kernel is just a small part of the kernel environment.

The Mach kernel, BSD, the I/O Kit, file systems, and networking components are all included in the OS X kernel environment. They are often referred to as the kernel as a whole. The following sections provide a short description of each of these elements. Go to the relevant component chapters or the sources indicated in the bibliography for further information. While there are three fundamental parts of OS X (Mach, BSD, and the I/O Kit), there are usually up to three APIs for certain critical operations.

Generally speaking, the API you choose should be compatible with the area of the kernel where it is being utilized, which is in turn determined by what your code is trying to do. The next sections of this chapter provide an overview of the functions that are offered by Mach, BSD, and the I/O Kit.OS, or operating system, and the process.

We are able to swap the process' status between various jobs thanks to these two abstractions. Although the OS offers the mechanism for switching execution across the processes, the process clearly specifies the state of a program that is currently running between programs.

Fundamentals of Operating Systems

Bridges connecting system resources and user applications and tasks. The OS controls how system resources are used and makes them accessible to user applications and tasks. A computer system is made up of several working and storage memory subsystems.

A kernel

The OS's main component, the kernel, is in charge of controlling the system and facilitating communication between the hardware and system services. Between system resources and user applications, the kernel serves as an abstraction layer. A group of system libraries for services are present in the kernel. The kernel of general purpose operating systems comprises a variety of management-related services.

A computer environment known as a real-time operating system (RTOS) responds to input within a certain amount of time. Real-time deadlines may be so brief that system response seems to occur instantly. Yet, "slow real-time" output with a longer but set time limit has also been referred to as real time computing. Imagine yourself in a computer game to understand the difference between real-time and regular operating systems. Every move you make in the game is comparable to a program that is active there. Since you may anticipate a precise "lag time"—the interval between your request for action and the computer's apparent execution of it—a game with a real-time operating system for its environment might seem like an extension of your body. Yet, a typical operating system could seem disconnected because of the inconsistent lag time. Real-time applications and the operating environment in which they run must put deadline actualization first in order to achieve time dependability. When response time and visual effects compete, this could lead to missing frames or worse visual quality in the game example.

Actual Kernel

The kernel is the brain of a real-time OS, and the brain of any OS, for that matter. An operating system's kernel serves as the system's brain, handling all of the OS's tasks:

1. Booting Task Scheduling
2. Library of Common Functions

The kernel will commonly start an embedded system, establish the ports, and set up the global data items. The scheduler will then be launched, along with any required hardware timers. Following then, the scheduler will begin executing the child tasks and the kernel will essentially run out of memory (apart from any library functions, if any). Libraries of common functions: Large function libraries are seldom kept in memory, if at all, in an embedded system. If functions are provided, they ought to be both tiny and significant.

Fundamental Kernel Services

The kernel, a crucial component of an operating system, provides processor-based application software the most fundamental functions. An "abstraction layer" is provided by the "kernel" of a real-time operating system (RTOS), which shields application software from the hardware specifics of the processor (or collection of processors) that it will use.

Scheduling

Typically, a task exists in one of three states:

Running (using the CPU to execute); Ready (ready to be executed); blocked (awaiting an event, such as I/O).

Since typically only one job may operate at a time per CPU, the majority of processes are stopped or ready most of the time. Depending on how many jobs the system needs to complete and the sort of scheduler it employs, the number of items in the ready queue might vary significantly. A job must give up its time on the CPU to other tasks in simpler non-preemptive but still multitasking systems, which may result in the ready queue having more tasks overall in the ready to be performed state (resource starvation). Pre-emption is often prohibited and, in certain situations, all interruptions are blocked during the important area of the scheduler, which is typically where the ready list data structure in the scheduler is meant to reduce. Nevertheless, the maximum number of jobs that may be on the ready list also affects the choice of data structure. A doubly linked list of ready jobs is probably ideal if there are never more than a few things on the list. The ready list should be ordered by priority if it often only contains a small number of jobs but sometimes includes more. This eliminates the need to repeatedly go through the list in order to discover the job with the greatest priority to execute. The next step in inserting a task is to go through the ready list until you either reach the bottom of the list or a task with a lower priority than the job you are inserting.

Throughout this search, caution must be taken not to obstruct pre-emption. Longer crucial portions should be broken up into manageable chunks. A high priority job may be inserted and performed before a low priority task is inserted if an interrupt makes a high priority task available while a low priority task is being inserted. The time it takes to queue a new ready work and change the status of the highest priority task back to running is known as the crucial reaction time, also known as the fly back time. A well-built RTOS will require between 3 and 20 instructions to prepare a new job for execution, and between 5 and 30 instructions to restore the highest-priority ready task. Real-time jobs share computational resources with several non-real-time activities in more complex systems, and the ready list may be as large as necessary. A scheduler ready list built as a linked list wouldn't work in these systems.

Types of Kernel:

Monolithic and Micro kernels may be distinguished based on the design of the kernel. All kernel modules execute in the same memory area under a single kernel thread in a monolithic kernel architecture where all kernel services run in the kernel space. The disadvantage of a monolithic kernel is that any problem or malfunction in any kernel module, as seen in Figure 7.5, results in the whole kernel program crashing. Like LINUX, SOLARIS, and MS-DOS.

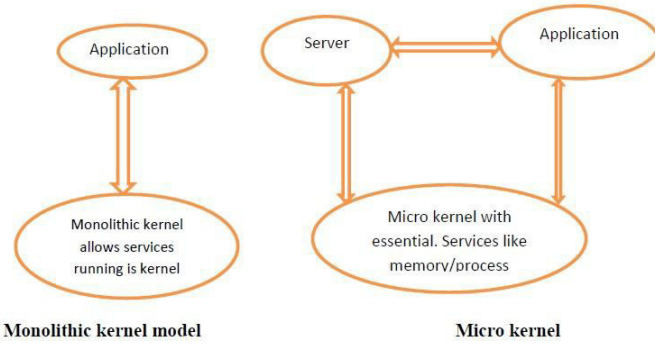

Figure 7.5: illustrates the application of microkernel.

Microkernel: design in corporate only the essential set of OS services into the kernel. The rest of the OS services are implanted in programs known as “servers” which runs in user space. The essential services of the Microkernel.

CHAPTER 8

TYPES OF OPERATING SYSTEM

Mamatha G N, Assistant Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-gn.mamatha@jainuniversity.ac.in

Operating systems (OS) are divided into two categories based on the kind of kernel and kernel services.

1. Universal Operating System
2. Operating System in Real Time
3. Universal Operating System

General purpose OS refers to the operating systems that are used in general computer systems. Such operating systems have more universal kernels that include all the services needed to run generic applications. GPOS have non-deterministic behavior, such as Windows XP and MS-DOS on desktop computers.

Operating System in Real Time

The word "RTOS" doesn't have a single, accepted meaning. An OS designed to support real-time application demands is known as an RTOS.

Processing time requirements are measured in tenths of seconds, and it must be able to handle the data as it comes in without buffering delay. For the task &task/process termination/deletion, hard RTOS has less jitter than soft RTOS. Task control blocks, or TCBs, are used to store data related to tasks. The TCB includes the following group of details.

1. Job ID: Task reference number
2. Job State: The task's condition right now
3. Task Type: Describes the nature of the task, which may be a background task, a soft real-time activity, or both.
4. Effort Priority: For example, if the task's priority is 1, set it to 1.
5. Context Pointer for a Task: context pointer, context preserving pointer
6. Pointers to the job's code memory, data memory, and stack memory are included in the task memory pointer.
7. Task System Resource Pointers: A pointer to a task's system resource Pointers for tasks: Pointers to other TCBs
8. Additional parameters Additional pertinent task-related factors
9. Depending on the task management, various kernels have varying TCB settings.
10. Implementation
11. Creates a TCB for a job upon task creation
12. After the job is ended or removed, remove or delete the TCB.
13. Gets the task's status by reading the TCB.
14. Updates the TCB with new settings for the essential needs
15. Change the TCB such that the task priority changes on demand.

Scheduling tasks and processes: Allocating the CPU to different tasks and processes. Task scheduling is handled by a kernel program named scheduler. The scheduler, which executes the effective & optimum scheduling of tasks to offer predictable behavior, is nothing more than an algorithm performed all to. Task/Process Synchronization is concerned with the concurrent use of a shared resource by numerous tasks and the communication between those tasks.

Error/Exception Handling: Takes care of registering and managing errors that occur during job execution. Examples include not enough memory, timeouts, deadlocks, missing deadlines, bus errors, divide by zero, and executing unexpected instructions. Exceptions and errors may occur at two different service levels. Core Level Service At the task level, RTOS uses "Block Based Memory" allocation methods rather than GPOS' customary dynamic memory allocation method. The RTOS kernel allocates blocks for tasks based on their requirement and employs blocks of fixed size dynamic memory. A few RTOS kernels employ virtual memory ideas to reduce the cost associated with trash collection.

Handler for Interrupts: Handles a variety of interruptions. Interrupts help systems develop embedded systems with real-time behavior. The processor receives interruptions to let them know that a connected task or an external device needs their urgent attention. Both synchronous and asynchronous interruptions are possible. Asynchronous interruptions are those that happen concurrently with a job that is already running. Typically, system interruptions are classified as memory segmentation errors or Ex Divide by zero synchronous errors. A synchronous interruptions are those that happen at any moment during the execution of any activity and are out of sync with other processes that are already in progress.

Task, Process & Threads

Task: A reference to a necessary action. The tasks in question might be those that managers assign or those that just one member of the processing family needs to assign. We also have a schedule with an order of importance for completing these activities. An OS job is defined as the running application and any associated data that the System maintains. The OS refers the task as "JOB" as well.

Process: A process is a program that runs as a component of the program; it is also referred to as an instance of the running program since numerous instances of the same program may run at once. Processes need a variety of system resources, including the CPU to execute them, memory to store the associated code, I/O devices for in-function communication, etc.

States in Progress and State Transition

A process's memory structure and related threads Run: As soon as a task begins running on the process, it enters this state.

Tasks that are prepared to run but can't because the processor is occupied by another job are in the ready state.

When a task runs a synchronization primitive to wait for an event, such as a wait primitive on a semaphore, it enters the blocked state and is added to a queue connected to the semaphore (Figure 8.1).

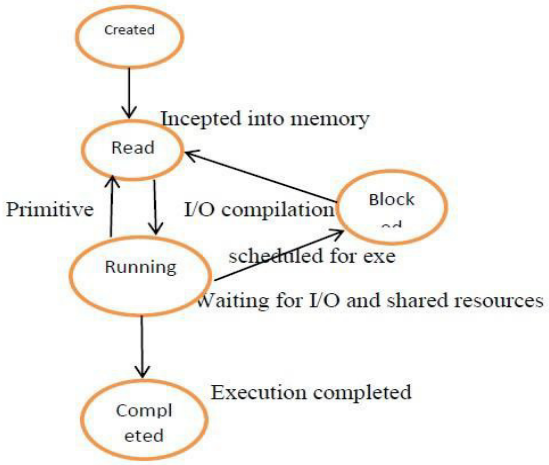

Figure 8.1: illustrates the Process States & State Transition.

A periodic task that has been created enters this stage when it computes its execution and must wait until the start of the following period. Process administration. Process management involves setting up a process's memory space, loading the process's code into that memory space, and allocating system resources. Establishing a PCB for the process and for process detection/termination.

Scheduling tasks:

The execution switching between the several tasks develops as a result of multitasking. The same technique should be in place to determine which process or job should be run at a certain moment by displaying the CPU among the many takes. The phrase "task/process scheduling" refers to the decision of which activity or process will be carried out at a certain moment. Task management based on multitasking for deciding which work should be completed when, scheduling policies are taken from the rules. Scheduling is the process of selecting the sequence in which processes will be carried out. Processes are chosen for promotion from the ready state to the running state in accordance with a scheduling policy. As a process changes states, the scheduling decision for that process may occur. Running state to Ready state Waiting or blocked from the running state Blocked/wait state to Ready state Figure 8.2 shows the completed (executed) state.

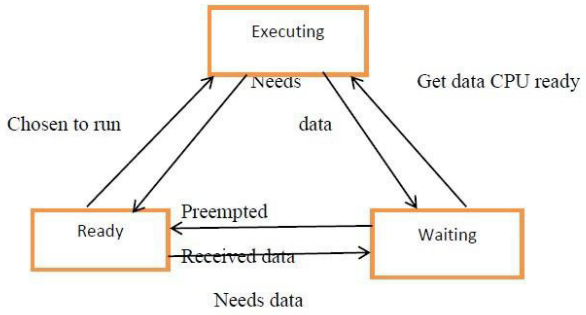

Figure 8.2: illustrates the process scheduling decision.

Scheduling for Scenario 2 may either be preemptive or non-preemptive. When an important process changes from the blocked/wait state to the ready state after finishing its I/O. If the scheduling policy is priority-based pre-emptive, the scheduler chooses it for execution.

Semaphores

In its most basic form, a semaphore is a protected integer variable that controls access to shared resources in a multi-processing context. Binary and counting semaphores are the two types of semaphores that are most often used. Whereas binary semaphores, as the name suggests, represent two potential states, counting semaphores represent numerous resources (generally 0 or 1; locked or unlocked). The late Edsger Dijkstra was the creator of semaphores. Semaphores may be seen as a depiction of finite resources, such as the amount of seats at a restaurant. The semaphore would be initialized to 50 if a restaurant can hold 50 patrons but no one is present. The number of seats available in the restaurant decreases as customers arrive, which in turn causes the semaphore to become less effective. The semaphore will be at zero when the restaurant's capacity is met, making it impossible for anybody else to enter. Instead, eager diners must wait until someone has finished using the resource, or in this case, finished eating. The semaphore is advanced when a user departs, making the resource accessible once again.

The only two operations that may access a semaphore are wait() and signal (). When a process needs access to a resource, wait() is called. This is comparable to a consumer who is late attempting to grab a table. He may take that resource and a seat at the table if there is an available spot and the semaphore is higher than zero. If the semaphore is at zero and there is no open table, the process must wait till one does. After a process has completed utilizing a resource or the customer has finished his meal, signal() is invoked. This counting semaphore is implemented as follows (where the value may be more than:

In the past, signal() was referred to as V (for Dutch "Verhogen" meaning to increase) and wait() was referred to as P (for Dutch "Proberen" meaning to attempt). Instead, P and V are referred to as "acquire" and "release," respectively, in the standard Java library.

While P or V are running, no other process is allowed to access the semaphore. The implementation of this uses atomic hardware and software. An atomic action may be thought of as performing as a single unit since it is indivisible. A binary semaphore, which can only have the values of 0 or 1, is employed if there is only one count of a resource. They're often used as mutex locks. Here is a binary semaphore implementation of mutual exclusion: in order to reach the crucial area, the system must first acquire the binary semaphore, which will then grant it mutual exclusion until the system signals that it is finished.

As an example, consider a semaphore (s) and two processes (P1 and P2) that both desire to access their critical sections simultaneously. P1 initial calls hold (s). P1 reaches its crucial phase as s is decremented to zero. P2 calls wait(s) when P1 is in its critical section, but because s has a value of 0, it must wait until P1 completes its critical section and executes signal (s). When P1 calls signal, s is increased by 1, allowing P2 to continue running in its vital part (after decrementing the semaphore again). One process at a time can only be in its crucial region for mutual exclusion to work.

Mutex

A mutual exclusion object (mutex) is a program object used in computer programming that enables many program threads to share a resource, such file access, but not concurrently. A mutex with a distinctive name is produced when a program is launched. Following this point, each thread that requires the resource must lock the mutex to prevent it from being used by another thread. After the data is no longer required or the procedure is complete, the mutex is set to unlock.

Message Slacks

A message queue offers an asynchronous communications protocol, meaning that a system may process messages without waiting for an instant answer. The finest asynchronous communications example is arguably email. Can the sender of an email go on with other tasks after sending it without immediately receiving a reply from the recipient? The producer and the consumer are separated by this kind of communication management. The message queue does not need simultaneous interaction from the message producer and consumer.

Decentralization Facilitates Accessibility

How much one component of a system depends on another component is referred to as decoupling. Decoupling is the process of separating them to increase the self-sufficiency of their functioning. When two or more systems can communicate with one another without being linked, the systems are said to be decoupled. The systems may continue to operate independently and without ever coming into contact. Decoupling is often a symptom of a well-structured computer system. Typically, it is simpler to expand, maintain, and debug. Figure 8.3 illustrates how further messages may be added to the queue and processed after the system has recovered if one process in a decoupled system fails while processing messages from the queue. A producer posts messages to a queue, which may also be used to postpone processing. The receivers are turned on and begin processing the messages in the queue at the designated time. Once the message is processed, queued messages may be held, forwarded, and redelivered.

Figure 8.3: illustrates the message be redelivered until the message is processed.

It is advantageous to decouple distinct components of your program and just use messages for their asynchronous intercommunication rather than creating one big application. Your application's various components may then develop independently, be developed in various languages, and/or be maintained by several development teams. The processes in your application will remain distinct and autonomous from one another thanks to a message queue. The first process won't ever need to start another process, notify another process, or follow the other processes' workflow. Just adding the message to the processing queue would allow it to go on. The other processes are likewise capable of managing their tasks on their own. When they are ready to handle the messages, they may remove them from the queue. This kind of message processing results in a system that is simple to grow and maintain.

A Simple Use Case is Message Queuing

Suppose that your online service gets several requests per second, that no request may be allowed to be missed, and that each request must be handled by a laborious procedure. Instead of being locked by the processing of previously received requests, imagine that your web service must always be highly accessible and prepared to handle new requests. A queue should ideally be placed in this scenario between the web service and the processing service. The "start processing" message may be placed on a queue by the web service so that other processes can take and process messages as they come in. The two processes will be independent of one another and need not wait for one another. The processing system will be able to handle all of your requests even if you get a lot of them quickly. If there are a lot of requests, the queue will continue to hold them. Next, assume that you need to scale up your system since your company and workload are expanding. The only thing left to do is to increase the number of employees and receivers in order to clear the lineups more quickly.

Connecting Embedded Systems

Embedded systems have had a remarkable metamorphosis over the last two decades, going from being built from discrete components on printed circuit boards—which they still are—to being built from IP components that are "dropped" onto silicon in systems on a chip. Systems on a chip provide the opportunity to integrate sophisticated features and to satisfy the performance demands of demanding applications like DSP, network, and multimedia processors. The development of distributed embedded systems, often known as networked embedded systems because of the significance of the network infrastructure and communication protocol, is another stage of this evolution that has already begun. A networked embedded system is made up of a number of spatially and functionally dispersed embedded nodes that are connected via wireline and/or wireless communication infrastructure and protocols. These embedded nodes interact with the environment (via sensor and actuator elements) and with one another, and possibly with a master node that performs some control and coordination functions, to coordinate computing and communication in order to accomplish a particular goal (s). The networked embedded systems are used for monitoring and controlling, environment monitoring, and, in the future, control as well, in a wide range of application domains, including those in the automobile, railway, airplane, office building, and industrial fields. Networked embedded systems have emerged for a variety of reasons, mostly affected by their application areas. Among of the most significant ones are the advantages of employing dispersed systems and the evolutionary need to replace point-to-

point wire connections in these systems with a single bus. The integration of intelligence into field equipment like sensors and actuators has become possible because to advancements in embedded system design, the availability of tools, and declining manufacturing costs of semiconductor devices and systems. These devices' controllers often provide communication, data and signal processing, and signal conversion on-chip. Field area networks, which are sometimes referred to as field area networks, are a prevalent trend for networking field devices around specialized networks due to the enhanced functionality, processing, and communication capabilities of controllers. In general, the field area networks, or field buses as they are more commonly known, are the networks that connect field devices like sensors and actuators with field controllers (for example, programmable logic controllers (PLCs) in industrial automation or electronic control units (ECUs) in automotive applications), as well as man-machine interfaces, like dashboard displays in cars. In general, there are several advantages to employing such specialized networks, such as higher system performance, greater flexibility made possible by the integration of embedded hardware and software, and simplicity of system installation, upgrading, and maintenance. In particular, they enable the replacement of mechanical, hydraulic, and pneumatic systems with mechatronic systems in the automotive and aerospace industries, for example, where mechanical or hydraulic components are normally restricted to the end-effectors.

Field area networks (FANs) on the other hand, tend to have low data rates, small size of data packets, and typically require real-time capabilities which mandate determinism of data transfer, in contrast to local area networks (LANs), due to the nature of communication requirements imposed by applications. Nonetheless, field area networks now often have data speeds over Mbit/s, which are characteristic of LANs. Many communication technologies, such as twisted pair cables, fiber-optic channels, power line communication, radio frequency channels, infrared connections, etc. are often supported by specialized networks. In general, networks can be divided into three main groups based on the physical media they use: wireline-based networks using twisted pair cables, fiber-optic channels (in hazardous environments like chemical and petrochemical plants), and power lines (in building automation); wireless networks supporting radio frequency channels and infrared connections; and hybrid networks, which extend wireline by wireless links.

The adoption of wireless technology provides a variety of incentives in a number of application areas, notwithstanding the dominance of wireline-based field area networks. For example, wireless device (sensor/actuator) networks in industrial automation may allow mobile operation needed for mobile robots, monitoring and control of equipment in dangerous and challenging-to-access environments, etc. Peer-to-peer communication between stations in a wireless sensor/actuator network is possible, as well as communication with the base station. A hybrid wireless-wireline system results from the base station's transmitter being connected to a field area network cable. The wireless sensor networks fall under a different category and are primarily intended to be used for monitoring.

Several functional and nonfunctional criteria for the functioning of networked embedded systems are imposed by the range of application fields. The majority of them, like systems used for control, must work in a reactive manner. As a result, real-time operation is necessary, in which systems must react within a certain time frame that is dictated by the dynamics of the process

being controlled. In general, a reaction may be aperiodic if it results from unplanned occurrences like an out-of-bounds state of a physical parameter or any other kind of abnormal situations, or periodic if it controls a specific physical quantity by regulating designated end-effector(s). Hard real-time systems, on the other hand, demand deterministic responses to prevent changes in the system dynamics that could potentially have a negative impact on the process under control, which could result in financial losses or harm to human operators. Soft real-time systems, in general, can tolerate a delay in response. Examples of systems that impose strict real-time requirements on their operation include Steer-by-Wire in automotive applications and Fly-by-Wire in aviation control, to name a few. The use of proper scheduling algorithms, which are typically implemented in application domain-specific real-time operating systems or frequently specifically created "bare-bone" real-time executives, is required to provide a predictable response. A high degree of dependability is necessary for the networked embedded systems used in safety-critical applications like Fly-by-Wire and Steer-by-Wire to guarantee that a system failure does not result in a situation where human life, property, or the environment are in risk. The reliability problem is crucial for the deployment of technology; this chapter discusses potential solutions in the context of automotive applications. Building automation control systems, for example, rarely require hard real-time communication; the timing requirements are much more lax compared to applications that mandate hard real-time operation, such as the majority of industrial automation controls or safety-critical automotive control applications.

Building automation systems usually incorporate all seven levels of the ISO/OSI reference model and feature a hierarchical network topology. For example, end-to-end control and routing functions are not often required for field area networks used in industrial automation. As a result, in such networks, just the physical layer, data link layer, which implicitly includes the medium access control layer, and application layer—which also includes the user layer—are commonly employed. The variety of requirements imposed by various application areas (such as soft/hard real-time, safety-critical, network topology, etc.) called for the use of various solutions and protocols with various operating principles. As a consequence, several networks have been created for various application sectors. Networked embedded system design techniques lie under the broad umbrella of system-level design. They consist of three components: timing analysis of the whole system, network architecture design, and node design (which is fully treated in Part I of the book). Many tasks are involved in the design of the network architecture. One of them is choosing a communication protocol and medium that are acceptable. To assure predictable access to the medium, a safety-critical application will use a protocol based on Time Division Multiple Access (TDMA) medium access control. Power line wires in an existing building or specialized twisted pair wires in a new structure may be used as the communication channel for an application in building automation and control. The application domain has a significant impact on the network architecture. Automated industrial systems often use the bus topology. Building networks may have a topology that is complicated and has several logical domains. The distribution of priorities in priority busses or slots in TDMA-based protocols to communication nodes is one aspect of configuring the communication protocol, among others. The purpose of the timing study is to determine the real timings for the selected architecture.

This includes task execution time metrics like worst-case execution time (WCET), best-case execution time (BCET), and average execution time, as well as response times for tasks from invocation to completion, end-to-end delays, and jitter, or variations in task execution times, for example. Finally, the whole system must be schedule-able to ensure that all distributed tasks

communicating via the network will be completed by their due dates under all operating circumstances to which the system is predicted to be exposed. Let's take a basic control loop as an example. It consists of a sensor node with a single application job devoted to sensing, an actuator node processing data obtained from the sensing node, and an actuator receiving the control value through a dedicated connection. The total amount of time required for data processing (WCET) and transmission (worst-case reaction time) must be less than or equal to the whole amount of time permitted by the dynamically controlled process. For safety-critical and hard real-time systems, a contention for the medium access may arise if other nodes connected to the shared communication network form similar control loops. This can be resolved by adopting a fixed transmission schedule, as in the case of the time-triggered TDMA-based protocols, for example. If the worst-case response time for all those composite jobs creating control loops is less than or equal to the deadline, the study of schedulability.

CHAPTER 9

AUTOMOTIVE NETWORKED EMBEDDED SYSTEMS

Sunil. MP, Assistant Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-mp.sunil@jainuniversity.ac.in

Automotive-specific communication protocols are used to connect ECUs in order to control one or more vehicle functions, such as electronic engine control, antilock brakes, active suspension, and telematics, to name a few. Trends for networking have also emerged in the field of automotive electronic systems. Many functional areas for the implementation of automobile networked embedded systems have been highlighted in Ref. These include the control of the engine and transmission in the power train, the suspension, steering, and braking in the chassis, the control of the wipers, lights, doors, windows, seats, and mirrors in the body, the integration of wireless communications, vehicle monitoring systems, and vehicle location systems in the telematics domain, as well as the control of multimedia and human-machine interfaces. The various domains place various performance, safety, or Quality of Services restrictions on the networked embedded systems (QoS). The power train and chassis domains, for example, will need real-time control; normally, bounded latency and fault-tolerant services are required. The automobile industry is interested in mechatronic solutions, also known as X-by-Wire, that attempt to replace mechanical, hydraulic, and pneumatic systems with electrical and electronic systems for a variety of reasons. The primary drivers seem to be cost-effectiveness, enhanced component dependability, and higher functionality that may be gained by combining integrated hardware with software. Systems like throttle-by-wire, brake-by-wire, or steering-by-wire are illustrative instances of those. Nevertheless, it seems that for safety concerns, several safety-critical systems, like Steer-by-Wire and Brake-by-Wire, will be supported by conventional mechanical/hydraulic back-ups. One of the primary criteria, as well as limitations on the deployment of this form of systems, is the reliability of X-by-Wire systems.

A safety-critical X-by-Wire system must, in this situation, guarantee that a system failure does not result in a situation in which human life, property, or the environment is threatened and that the failure of a single component does not result in the failure of the whole X-by-Wire system. For X-by-Wire systems, it is necessary that the chance of a safety-critical system failing does not exceed the number of hours/system when utilizing the Safety Integrity Level scale. The SIL level is represented by this number. For safety-critical systems, end-to-end response times must be constrained for X-by-Wire systems in order to adhere to tight real-time limits imposed by system dynamics. A breach of this requirement might have additional negative effects in addition to lowering the performance of the control system. Not every automobile electronic system is vital for safety. For example, the system(s) in place to regulate seats, door locks, interior lighting, etc. Various in-car application domains' demands for performance, safety, and quality of service required the adoption of varied solutions, which in turn led to the emergence of a sizable number of communication protocols for automotive applications. Since they provide predictable access to the medium, time-triggered protocols (TTP) based on TDMA medium access control technology are especially well suited for the safety-critical solutions. TTP/C and FlexRay are

two protocols in this category that, in theory, satisfy the criteria of X-by-Wire applications (FlexRay can support a combination of both time-triggered and event-triggered transmissions). The next conversation will mostly concern TTP/C and Flex Ray. One of the two protocols in the time-triggered architecture (TTA) is the fault-tolerant TTP/C. The other is a fieldbus protocol called TTP/A that is inexpensive. Two replicated communication channels join the nodes to create a cluster in TTA. A network in TTA may connect using either the bus or star interconnection topologies. Each node is linked to two replicated passive buses by bus guardians in the bus configuration. The bus guardians are autonomous devices that block the transmission line to prevent related nodes from sending outside of preset time intervals; a suitable example may be a controller with a broken clock oscillator that tries to communicate continually. The guardians are included into two duplicated central star couplers in the star topology. The guardians must have their own clocks, a distributed clock synchronization system, and a power source. In order to maximize their resistance to spatial proximity defects, they should also be placed far away from the protected node. TTA uses so-called fault-tolerant units (FTUs), which are collections of many stations carrying out the same computing tasks, to handle internal physical failures. Since each node in a TDMA round is (statistically) assigned a transmission slot, the service will not be negatively impacted by a node failure or frame corruption. Moreover, data redundancy makes it possible to determine the right data value by a voting procedure.

The TTP/C medium access control technique uses replicated channels and synchronous TDMA to provide fault-tolerant transmission with known latency and constrained jitter across cluster nodes. The ability to disguise a temporary defect on one of the channels is made possible by the use of duplicated channels and redundant transmission. A -bit CRC checksum protects up to bytes of data in the message frame's payload portion. The exchange of information takes place in rounds in TTP/C. Different stations in a round may be given slots of varying sizes. Nevertheless, in subsequent rounds, slots from the same station are the same size. In each round, every node is required to transmit a message. Fault-tolerant clock synchronization, which provides a global time basis without the requirement for a central time provider, is another characteristic of TTP/C. The message schedule is stored on each node in the cluster. Based on the data, a node calculates the time difference between a proper messages's expected and actual arrival. A fault-tolerant technique that can modify the local clock to keep it in sync with the clocks of other nodes in the cluster averages the discrepancies. TTP/C implements the fault-tolerant clock synchronization method and offers what is known as membership service to let each node in the cluster know what the status of every other node in the cluster is. This service locates nodes with broken connections using a distributed agreement technique. A node with a transmission error is not allowed to rejoin the group until it has a correct protocol state. A clique avoidance algorithm is a key component of TTP/C that helps to identify and prevent the creation of cliques when the fault hypothesis is broken. In general, if the fault hypothesis is broken, the fault-tolerant operation based on FTUs cannot be maintained. When this occurs, TTA employs the never-give-up (NGU) approach. TTP/C launches the application-specific NGU approach in tandem with the application with the intention of allowing operation in a degraded condition.

Since the Technical University of Berlin's Maintainable Architecture for Real-Time Systems project began, the TTA infrastructure and the TTP/A and TTP/C protocols have been around. The work was subsequently completed at the Vienna University of Technology. TTP/C protocols have been tested and have been taken into consideration for implementation for a long time. Unfortunately, there haven't been any real implementations of that protocol including safety-

critical equipment in trucks or commercial vehicles as of yet. In a "proof of concept" demonstration, a vehicle fitted with a "Brake-by-Wire" system based on a time-triggered protocol was shown off by Vienna University of Technology and DaimlerChrysler.

A modified TDMA medium access control system is used by FlexRay, which seems to be the front-runner for future automotive safety-critical control applications, on a single or duplicated channel. A frame's payload section may include up to bytes of data that are secured by a -bit CRC checksum. FlexRay additionally permits duplicate data transmission over the same channel(s) with a delay in between broadcasts to address transitory problems. A network communication period plus a network idle time make up the FlexRay communication cycle. An application cycle might consist of two or more communication cycles. The order of static segment, dynamic segment, and symbol window describes the network communication time. The TDMA MAC protocol is used for the static section. The static segment consists of fixed-duration static slots. The static allocation of slots to a node (communication controller), unlike TTP/C, only applies to one channel. Another node on the opposite channel may occupy the same slot. A node may also have many slots in a static section. Flexible Time Division Multiple Access (FTDMA) MAC protocol, which enables a priority- and demand-driven access pattern, is used by the dynamic segment. Each node is given a set number of so-called mini-slots, which are not have to be consecutive, as part of the dynamic segment. Mini-slots are substantially shorter than static slots and have a set length. A mini-slot must be extended in order to allow the transmission of a frame since its length is insufficient (a mini-slot simply specifies the prospective start time of a transmission in the dynamic segment). As a result, there are fewer mini-slots in the dynamic segment's reminder. Each mini-slot that has nothing to transmit is quiet. Less time is expected to be available for transmission for the nodes assigned mini-slots at the conclusion of the dynamic segment. This in turn imposes a system of priorities. The symbol window is a predetermined period of time used for network administration. A protocol-specific time period during which no traffic is planned on the communication channel is known as the network idle time. It is used by communication controllers for the clock synchronization activity, and is conceptually comparable to that of TTP/C. The idle period and minimal static segment must be present in a communication cycle even if the dynamic segment and idle window are optional. For fault-tolerant clock synchronization, at least four static slots or two degraded static slots must be present. With all of that, FlexRay supports three configurations: pure static, mixed (where the amount of bandwidth assigned to each kind of communication depends on the application), and pure dynamic.

In order to maximize scalability and provide a respectable amount of freedom in the configuration of embedded electronic systems for automotive applications, FlexRay enables a variety of network topologies. Bus, active stars, active cascaded stars, and active stars with bus extension are among the configurations that may be used. The bus guardians are used by FlexRay in a manner similar to TTP/C. Bit rates of up to Mbps on two channels are supported by the current FlexRay communication controllers. A selection of services tailored to the automobile network are also offered via the communication controller's transceiver component. Alarm management and wake-up management are two important services. An ECU additionally gets the alarm symbol from the communication controller in addition to the alarm information received in a frame. This redundancy may be employed to verify important signals, such a directive to light an airbag. If electrical components have a sleep mode to save power consumption, the wake-up service is necessary.

FlexRay is a collaborative project between a group of top automakers and technology companies, with BMW, Bosch, Daimler, Freescale Semiconductor, General Motors, NXP Semiconductors, and VW serving as the consortium's primary members. Time-Triggered Controller Area Networks (TTCAN) use the physical and Data-Link Layer of the Controller Area Network (CAN) protocol and may handle a mix of time- and event-triggered broadcasts. It is doubtful that this protocol, as specified by the standard, would play any part in fault-tolerant communication in automotive applications since it does not provide the requisite dependability services.

According to the Society for Automotive Engineers' categorization, TTP/C and FlexRay protocols are class D networks. Despite the classification's historical roots, it continues to serve as a useful benchmark for categorizing various protocols according to the network's functionalities and the speed at which data is sent. There are four classes in the categorization. Networks having a data rate of less than Kbit/s are classified as Class A. Local Interconnect Network (LIN) and TTP/A are two examples of representative protocols. The body domain functions are mostly implemented using class A networks. Class B networks operate at speeds between and Kbit/s. Protocols like J, low-speed CAN, and Vehicle Area Network (VAN) are examples of representative ones. Kbit/s to Mbit/s is the operating speed range for Class C networks. High-speed CAN and J networks are two examples of this kind of networks. The power train and chassis domains are controlled via networks in this class. High-speed CAN is used to manage the power train and chassis domains, but since it lacks the requisite fault-tolerant services, it is unsuitable for safety-critical applications. Networks having a data rate above Mb/s are included in Class D networks, which has not yet been fully defined. This category includes TTP/C and FlexRay networks, which offer X-by-Wire solutions. Moreover, this class includes Media Oriented System Transport (MOST) and IDB- for multimedia applications. Networked embedded automotive applications are developed collaboratively, which results in a heterogeneity of hardware and software components. The support for reuse of hardware and software components is restricted, even with the inevitable standardization of those components, interfaces, and even whole system architectures. This might make the creation of networked embedded automotive applications time-consuming, costly, and error-prone. This circumstance prompted a number of standardization attempts, the most noteworthy of which being OSEK/VDX and AUTomotive Open System Architecture (AUTOSAR), to build component-based design integration approaches (both discussed in detail in this book).

Networks Embedded Systems in Industrial Automation

Although one can trace the beginnings of field area networks back to the CAMAC network at the end of the s in the nuclear instrumentation domain and the MIL-STD-bus at the beginning of the s in avionics and aerospace applications, it was the industrial automation area that saw the majority of advancements. Due to the difficulty of integrating diverse systems at the time owing to a lack of standards, two significant projects were launched that have had a significant influence on the integration ideas and the design of the protocol stack of field area networks. These programs were the Manufacturing Automation Protocol (MAP) and Technological and Office Protocol (TOP) initiatives. The two initiatives revealed certain flaws in the implementation of the ISO/OSI model's whole seven-layer stack. So, in the field area networks, often just the layers (physical layer), (data link layer, including implicitly the medium access control layer), and (application layer, which covers also user layer) are employed. This is exactly what the international fieldbus standard, IEC, specifies. Functions of layers and are always covered in layer according to IEC ; network and transport layers are not necessary in a single

segment network typical of process and industrial automation (although the situation is different in building automation where the routing functionality and end-to-end control may be required arising from a hierarchical network structure).

Fieldbus technology has evolved significantly over the last two decades, leading to a wide range of solutions that represent the conflicting business interests of their creators and worldwide and national standardization organizations, including IEC, ISO, ISA, CENELEC, and CEN. This is also reflected in IEC, which accepts all national standards and fieldbus systems supported by user organizations. Implementation guidelines were subsequently gathered into Communication Profiles, IEC -. These Communication Profiles identify seven major systems, or Communication Profile Families, including Factory Fieldbus (H, HSE, H), used in process and factory automation; ControlNet and EtherNet/IP, both used in factory automation; PROFIBUS (DP, PA), used in factory and process automation respectively; PROFINET; P-Net (RS, RS), used in factory automation and shipbuilding; WorldFIP; and INTERBUS. The primary application areas are those that are stated. Fieldbus-level communication in factories and plants is increasingly being implemented using Ethernet, the foundational technology for office networks. Other solutions are taking the place of the random and native CSMA/CD arbitration mechanism, allowing for the deterministic behavior necessary in real-time communication to support soft and hard real-time deadlines, time synchronization of activities necessary, for example, to control drives, and for the exchange of small data records typical of monitoring and control actions. Real-Time Ethernet (RTE), a new fieldbus technology that uses Ethernet for the bottom two levels of the OSI model and is being standardized by the IEC/SCC committee, is Ethernet enhanced with real-time extensions. The IEC - standard provided additional profiles for ISO/IEC. (Ethernet) based communication networks in real-time applications. Many solutions that employ one of the three ways to satisfy real-time requirements currently exist.

The cost of ownership and maintenance may be reduced by using standard components such protocol stacks, Ethernet controllers, bridges, etc. The direct support for Internet technologies enables the vertical integration of different industrial enterprise hierarchy levels, including seam-less integration between automation and business logistic levels to communicate jobs and production (process) data; transparent data interfaces for all phases of the plant life cycle; Internet- and Web-enabled remote diagnostics and maintenance; and electronic orders and transactions. In terms of industrial automation, the development and use of networking have made it possible for industrial firms to be integrated both horizontally and vertically.

Many of these systems may be vulnerable to potential security attacks, which could compromise their integrity and result in damage. This is because embedded systems are increasingly being networked and interconnected with LAN, WAN, and the Internet (for example, there is an increase in demand for remote access to process data at the factory floor). The individual system or device being secured, the application domain, the scope of internetworking, and the design of the system all play a significant role in potential security solutions. Office IT operational security needs often include confidentiality to shield data from unauthorized parties, integrity to guard against illegal data tampering, and availability to guarantee data are accessible when required. Contrary to office IT, operational security requirements for automation and process control systems place a greater emphasis on safety to ensure the avoidance of catastrophic consequences for people and the environment, as well as system/plant availability. The automation system and plant must be able to operate safely for extended periods of time, even if they do so in a degraded manner due to a fault brought on by a security attack. Attacks on electronic security might

jeopardize these systems' integrity and put plant, workers, and possibly the public at risk. Financial losses might arise from non-availability. In the automated plant control hierarchy, the security precautions that must be taken at the corporate and control network levels are largely the same as those that apply to general networking. At the field and device level, the situation is different. Fieldbuses often lack inbuilt security safeguards at the field level. Eavesdropping or message tampering would need physical access to the medium since they are commonly found at the locations where access permits are required. The access point (PLC, for example) control is one feasible method to provide a given degree of security. The usage of Ethernet and TCP/IP protocols and services makes the newly developed Ethernet-based fieldbuses more susceptible to assaults. The standard TCP/IP communication security mechanisms are applicable here.

The implementation of efficient security rules, which are often resource-intensive, is significantly hampered at the device and embedded level by the controllers' constrained processing, memory, and communication bandwidth capabilities. The application of common cryptographic protocols, even vendor-specific ones, is so constrained. Operating systems operating on controllers with limited footprints often just implement necessary services and don't provide authentication or access control to safeguard field devices that are vital to the mission and safety. Digest Access Authentication, a security extension to the Hypertext Transfer Protocol (HTTP), may provide an alternate and workable solution in applications limited to HTTP, such as embedded Web servers. A creative interrupt priority allocation and/or selection is required in the event of a denial of service (DoS) attack since the processor is busy with addressing communication interruptions that might jeopardize the (hard) real-time requirements and operational safety. Embedded controllers may become unavailable over time due to interruptions handled outside of standard working settings, which is a severe issue in wireless sensor networks installed on manufacturing floors when the unavailable node's role cannot be performed by other nodes. An embedded controller is often expected to withstand several security assaults on its own, such as the "buffer overflow" attack, which has the ability to crash the system; adequate error and exception handling is needed in this case.

Wireless Sensor Networks

Wireless sensor networks, another kind of networked embedded systems, have lately become a trend in the networking of field equipment. This is especially true for ad hoc and self-organizing networks where the nodes may be embedded in the ecosystem or a battlefield, to name a few, where the "embedded" component is not as obvious as in other applications. While there are few reports on actual implementations and possible applications in the anticipated sectors are still being considered, the industrial industry is deploying wireless sensor/actuator networks. For field equipment like sensors and actuators, using wireless connectivity enables flexible installation and maintenance, mobile operation—which is necessary for mobile robots—and solves cabling-related issues. A wireless communication system must provide high reliability, low and predictable data transfer delays (typically less than ms for real-time applications), support for a large number of sensors and actuators (over in a cell of a few meters radius), and low power consumption, just to name a few, in order to function effectively in the industrial/factory floor environment. The presence of electric motors or other equipment that causes an electric discharge may exacerbate the characteristic for wireless channel degradation artifacts in industrial settings, which results in even higher levels of bit error and packet losses. Designing resilient and loss-tolerant applications and control algorithms, as well as attempting to enhance the channel quality, are two ways to at least partly ease the issue. All of these are active areas of

study and development. The demands placed on self-organizing and ad hoc wireless sensor networks and those made by industrial applications are quite different. Self-containment, absence of predetermined network architecture (arranged by nodes on an as-needed basis), and the capacity for self-healing are some of the key features of self-organizing and ad-hoc wireless sensor networks (network operation not affected if a node goes down). On the other hand, wireless sensor networks used in industrial applications often have a predetermined network architecture, no self-healing capabilities, extended node and network lifespan requirements, and a high cost. The discrete manufacturing or continuous process equipment layout, or system architecture, which dictates where the information is to be gathered, establishes the planned network topology. Whenever a node fails, the functionality of the network and system is impacted; for example, a physical process parameter like temperature or pressure is no longer visible and controlled. Node redundancy is an option here, although a pricey one. The economics of the manufacturing process and the early investments necessitate the nodes to have a lengthy lifespan. Replacement of a node or a node's components can only be financially justifiable within a device or system's planned maintenance or proactive maintenance window. Among of the demands made on the nodes include fault tolerance, physical resilience, and operational robustness. If the node is powered by batteries, the necessary extended lifespan also applies to the batteries. Increased node functionality to offer a certain degree of redundancy on chips or printed circuit board assemblies, which has a cost, is one of the effects of the extended lifespan requirement. The IC packaging and node housing, which guard the node from damaging environmental factors like temperature, fluids, and to name a few, which have the potential to reduce the operational lifetime of the node—or to prevent any electrical discharge from the node in hazardous environments, such as the presence of flammable gases, for example—are other factors to have an impact on the cost.

In industrial applications, wireless sensor network operation commonly imposes real-time constraints and frequently rigid limits on the maximum delay. The majority of "traditional" wireless sensor networks don't seem to have a problem with packet delivery times. For discrete manufacturing, this latency is often measured in tens of milliseconds, for process control in tens of seconds, and for asset management in minutes to hours. High capacity, mains powered wireline is used for data dissemination from the collecting point in wireless sensor networks in industrial applications, which often feature a hybrid wireless-wireline design linked through a gateway device. The star topology with wireless connections between the sensor nodes and the gateway is the most often used network architecture in this application. This method is appropriate for time-bounded communication supporting operations with rapid dynamics due to the one-hop connection with the gateway. Another often used design is the hybrid mesh topology, which has the gateway device at its core and typically mains powered router nodes linked by wireless networks. The sensor nodes communicate with the router nodes in one-hop fashion.

A key need for wireless sensor networks used in industrial applications is the dependability of message transmission. For example, lost communications might negatively alter the dynamics of a process that is being controlled. The redundancy of the transmission is one method for enhancing dependability. Many choices may be taken into consideration depending on the application: Diversity in space, frequency, time, and modulation technique refers to transmission across various transmission channels, on various frequencies, at various times, and using various modulation systems. Low power consumption is of the utmost importance in the wireless sensor

networks used in industrial applications because nodes are frequently situated in inaccessible and/or dangerous locations, and because the cost of unscheduled energy source replacement would be prohibitive and necessitate pausing operational activities. Utilizing low power components—the CPU, for example, operates at a slower clock rate with less on-chip functionality; choosing the proper operational regime—adopting a sleep/wake-up mode of operation, with transmission being activated only if the value of a measured physical quantity is greater than the predetermined bound; selecting the appropriate communication protocol; etc.—are some of the factors that reduce power consumption. The lowest limit for power consumption is ultimately determined by the communication protocol.

Design and Implementation of Middleware for Networked Embedded Systems

Applications supported by networked embedded systems range from temperature monitoring to planning combat strategy. The following characteristics apply to systems in this field:

1. High-density networks
2. Numerous end systems with memory constraints
3. Strict timeliness standards

Online reconfiguration of computation and communication rules and procedures that are adaptive Networked embedded systems represent an active research topic with many unanswered concerns because they upend the presumptions about resource availability and scalability established by traditional methods to distributed computing. For instance, improvements in MEMS hardware technology have made it feasible to bring software closer to actual sensors and actuators in order to more effectively use their capabilities. Nevertheless, new networked embedded systems technologies are required to achieve this prospect. A network of hundreds or even thousands of tiny microcontrollers, each intimately connected to local sensors and actuators, may make up the hardware architecture for such systems.

A variety of design dimensions

The design decisions for the creation of several networked embedded systems are guided by the following four dimensions: Several of these design factors often have a contra-variant connection with one another.

Construction Life Cycle

The development, deployment, and testing of large-scale networked embedded systems is often costly and time-consuming. Reduce development costs and cycle times by allowing independent middleware and target system hardware development and testing. For special-purpose middleware, this separation offers extra design and implementation issues.

For instance, hundreds of nodes must be simulated simultaneously for physical, computational, and communication operations in order to assess how well the distributed ping-scheduling method performs in the real system. Tools like Matlab or Simulink must be linked into the simulation environment in order to model physical processes. Using the real program that will be installed on the target system should be used for computation. Yet, in the simulation environment, the software may be used on a much fewer and/or very different real end systems than it would be in the target system. Similarly, communication will often take place across traditional networks in the simulation environment, such as switched Ethernet, which may not be

an accurate representation of the target system's network. While developing middleware that is appropriate for both the simulation environment and the target system environment, the following concerns must be taken into account:

In the simulation environment, we must make use of as much of the software that will be utilized in the target system as feasible. This enables us to get data about the middleware and application that will be integrated with the target system that are comparatively accurate. For the simulation, we must allow for any setup. Each machine used to conduct the simulation may have a different hardware and software setup, and the types and quantities of target system nodes that are simulated on each computer may vary.

As it cannot be guaranteed that the nodes are synced, simple time scaling will not function. Secondly, because one purpose of the simulation may be to estimate such durations, it is not realistic to demand that all processing and communication times be known in advance. Moreover, even if we could reduce the duration to a "safe" upper limit, the simulation's wall-clock runtime would probably be too long.

Certain simulated nodes may run quicker than others due to the heterogeneous setup of the simulation environment, which might produce causal errors in the simulation. In order to mimic real-time performance on top of general-purpose operating systems and networks and to keep simulations of physical processes in the loop, additional infrastructure is thus required. As an example, consider periodic reconfiguration brought on by online sensor/actuator failures.

Difficulties in Middleware Design and Implementation

A middleware framework that offers common services like remote object method invocation is required to make it easier for nodes to communicate with one another as part of the distributed algorithm. The development of ORB-style middleware for networked embedded devices is driven by two main factors: () remote communication and () location independence.

Remote interaction: Even if a collection of sensor and actuator components may be connected by a defined physical topology, the "logical" grouping of these components may not always correspond to the "physical" grouping.

Geographic independence to the greatest degree feasible, communication components' behavior should be location-independent. In certain circumstances, such as those involving temporal restrictions or explicit connectivity to physical sensors or actuators, true location independence may not be possible. But, if appropriate, the decision of whether an object accesses other objects locally or remotely should be able to be separated from the implementation of object functionality. Hence, a standard programming abstraction for both remote and local access should be supplied by the programming model given to the object developer.

Item Adapter

Each server-side ORB in conventional CORBA may provide many object adapters. An object adaptor that registers servant objects demultiplexes each client request and routes it to the proper servant. A set of rules, such as those for servant threading, retention, and lifetime, may be attached to each object adapter. Each ORB in standard CORBA is capable of supporting many object adapters. With the ability to set each object on a server in accordance with the preferences of the server administrator or even the end user, this enables the implementation of

heterogeneous object policies in a client-server context, which is desired in applications like online banking. Nevertheless, numerous object adapters are not assumed in nORB. Instead, a single object adaptor per ORB is thought to be more convenient and footprint-efficient. As nORB anticipates a low number of objects hosted on embedded nodes, fewer policies—and hence fewer object adapters—are required. Due to the decreased object adapter functionality, a considerable footprint reduction is achieved even if the resultant object adapter does not adhere to the Portable Object Adapter standard. Also, in order to prevent developers from writing repeating code, as is common in many CORBA projects, we have made the object registration procedure simpler. We keep a lookup table of object IDs and references to servant implementation objects in the object adapter. An RW lock is used to synchronize the lookup table. Moreover, by transferring object registration from the object adapter interface to the ORB interface, we have combined it with other middleware setup operations.

CHAPTER 10

REAL-TIME EMBEDDED SYSTEMS

Hari Krishna Moorthy, Associate Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-harikrishna.moorthy@jainuniversity.ac.in

The use of real-time embedded systems has spread widely. Your watches, TVs, PDAs, vehicles, cell phones, and other household electrical equipment all include them. Real-time database applications, industrial process control systems, networked multimedia systems, and bigger, more intricate real-time embedded systems are some examples. When all of the optional amenities are fitted, the Lexus LS-460, which was introduced in September 2006, is said to have more than 100 integrated microprocessors. Moreover, 98% of all microprocessors are thought to be produced as parts for embedded systems. In reality, real-time embedded applications have grown more and more crucial to our everyday lives. This chapter defines hard and soft real-time systems, clarifies the terms embedded systems and real-time systems, and covers the basic traits of real-time embedded systems. To demonstrate a practical embedded system, the car antilock braking system (ABS) is utilized as an example.

System Embedded in Real-Time

A microcomputer system that is embedded in a bigger system and created for one or two specific services is called an embedded system. It is integrated into a whole device, which often includes physical and mechanical components. Examples are the controllers included in our household appliances' electrical systems. Real-time computing restrictions apply to the majority of embedded systems. They are also known as real-time embedded systems as a result. Embedded systems are often devoted to certain tasks as opposed to general-purpose computer systems, which contain a variety of features. For instance, the integrated airbag control system is just in charge of detecting collisions and activating the airbag when required, while the embedded air conditioner controller is only in charge of keeping track of and controlling the room's temperature. A general-purpose computer system and an embedded system differ significantly in another important way: a general-purpose system supports full-scale operating systems, while embedded systems may or may not do so. Many compact embedded systems are designed to do simple tasks, so they don't need operating system assistance. Reactive systems by definition are embedded systems. In essence, they are made to control a physical variable in response to the input signal sent by users or sensors that are attached to the input ports. For instance, a grain-roasting embedded system aims to control the furnace's temperature by varying the quantity of fuel pumped into it. Based on the discrepancy between the target temperature and the actual temperature as measured by temperature sensors, regulation or control is carried out.

Small-scale, medium-scale, and large-scale embedded systems may be categorized according to their level of complexity and performance. Small-scale systems often use low-end 8- or 16-bit microprocessors or microcontrollers to execute straightforward tasks. The primary programming tools for creating embedded software for tiny embedded systems include an editor, an assembler, a cross-assembler, and an integrated development environment (IDE). The mouse and television

remote control are two examples of tiny embedded systems. Usually, batteries are used to power them. Such systems often lack an operating system. Hardware and software complexity are both present in medium-scale systems. They use microprocessors or microcontrollers with 16 or 32 bits. The primary programming languages for creating embedded software for medium-sized embedded systems include C, C++, JAVA, Visual C++, debugger, source-code engineering tool, simulator, and IDE. They often provide help for operating systems. Vending and washing machines are two examples of medium-scale embedded systems. Big or complicated embedded systems, based on 32- or 64-bit microprocessors or microcontrollers, as well as a variety of other high-speed integrated circuits, have significant hardware and software complexity. They are used for cutting-edge applications that need co-design methods for both hardware and software. Military applications, automotive braking systems, and flight-landing gear systems are a few examples of large-scale embedded systems. Both real-time and non-real-time embedded systems are possible. When a non-real-time system performs the intended tasks in response to internal or external triggers with a satisfied level of QoS, we claim that it has been properly planned and constructed (Quality of Service). Calculators and TV remote controls are two examples. Yet, real-time systems are necessary to calculate and transmit accurate results in a certain amount of time. In other words, a real-time system has a deadline for each task, whether it is strict or flexible.

Even if the outcome is accurate, it is pointless if a firm deadline is missed. Think about how autos regulate their airbags. In most situations, airbags in cars are designed to deploy in frontal collisions. In an accident, cars change speed so fast that airbags must deploy promptly to lessen the chance that the occupant would strike the inside. The average deployment and inflation time from the start of the crash is 0.04 seconds, whilst the maximum is 0.1 seconds. Time restrictions may also apply to embedded systems that don't run in real time. You would undoubtedly be displeased if it took your TV remote control more than 5 seconds to transmit a control signal to your TV and another 5 seconds for the embedded device in your TV to change the station for you. Consumers should reasonably anticipate that a TV will react to a remote control event in less than a second. These restrictions, however, are only a gauge of system performance. Automotive, avionics, industrial process control, digital signal processing, multimedia, and real-time databases are some of the traditional application fields for real-time embedded systems. Nonetheless, real-time embedded applications will be present in every item that can be made smart due to the continuous fast advancement of information and communication technology, the rise of the Internet of Things, and ubiquitous computing.

Examples Include Automobile Antilock Brakes

Automobiles are a notable real-time embedded system application field. To manage the engine, automatic transmission, steering, brakes, suspension, exhaustion, and other automotive components, embedded systems are devised and developed. The instrument panel, key, door, window, lighting, air bag, and seat bag are just a few examples of the body electronics that employ them. The ABS is explained in this section. A vehicle safety system is an ABS. It is intended to protect a vehicle's wheels from locking when the brake pedal is pressed, which might happen suddenly in an emergency or while stopping quickly. Moving cars will lose tractive contact with the road surface and start to slide uncontrolled if their wheels suddenly lock up. Because of this, ABS is also short for anti-skid braking system. The primary advantage of ABS functioning is, in very rare instances, keeping the vehicle's directional control during hard braking.

Slip Rate and Brake Force

The wheel velocity (the tangential speed of the tire surface) and vehicle velocity both drop when the brake pedal is applied while driving. Nevertheless, the drop in vehicle speed is not always timed with the decrease in wheel speed. When the amount of friction between a tire and the road surface reaches its maximum, applying more brake pressure won't increase the braking force, which is calculated by multiplying the vehicle's weight by the friction.

ABS Components

There are four parts that make up the ABS. They consist of pumps, valves, speed sensors, and an electronic control unit (ECU). Hydraulic control devices often include valves and pumps (HCUs).

Sensors

The ABS makes use of a variety of sensors. A wheel speed sensor is a sender tool used to measure the rate of rotation of a vehicle's wheels. This object is an electromagnet. The electromagnet's coil experiences an AC voltage as a result of the sensor rotor's rotation. The amplitude and frequency of the induced voltage rise together with the sensor rotor's rate of rotation. The rate of the vehicle's deceleration is measured using a deceleration sensor. It is a sensor of the switch kind. It makes use of phototransistors, which are light-activated. Two Light-Emitting Diodes (LEDs) in a deceleration sensor aim at two phototransistors that are divided by a slit plate. The slit plate swings from the back to the front of the vehicle as the rate of deceleration varies. The phototransistors are made visible to the light from the LEDs by the slits in the slit plate. The phototransistors are turned ON and OFF by moving the slit plate in this manner. The ABS ECU receives signals that categorize the rate of deceleration into four categories based on combinations created by the two phototransistors going ON and OFF.

A steering angle sensor (SAS) calculates the angle and rotational speed of the steering wheel. A sensor cluster in the steering column houses the SAS. For redundancy and data confirmation, the cluster always incorporates more than one steering position sensor. For the steering wheel position to be confirmed, the ECU module has to receive two signals. Often, these signals are not in phase with one another. The body motion sensors and SAS provide the ABS control module with information on how and where the driver is steering the car. A gyroscopic instrument called a yaw-rate sensor detects a vehicle's angular motion around its vertical axis. Slip angle, which is connected to yaw rate, is the angle between the direction of the vehicle's heading and its actual movement. During real braking, a brake pressure sensor records the dynamic pressure distribution between a brake pad and the rotor surfaces.

Pumps and Valves

Hydraulic fluid is commonly used in auto brakes. When the pedal is pressed, the master cylinder of the brakes produces fluid pressure. The hydraulic control solenoid valves in the HCU of a typical ABS system employ electricity to regulate the braking pressure applied to certain wheel brake circuits. A plunger valve that is electronically opened and closed is known as a solenoid valve. A magnetic coil is ignited when electricity is provided to the solenoid, which causes it to move the plunger. Inside the HCU, there are several hydraulic circuits, and each hydraulic circuit is in charge of two solenoid valves: an isolation valve and a dump valve. The valves may be operated in three different ways: apply, hold, and release. Both valves are open in the apply

mode, allowing brake fluid to freely flow via the HCU control circuit and into the designated brake circuit. Via the master cylinder, the driver has complete control over the brakes in this mode. The master cylinder is cut off from the braking circuit when both valves are in the closed position (hold mode). In the case that the driver applies more force to the brake pedal, this prevents the brake pressure from rising higher. Up until the solenoid valves are instructed to alter their position, the braking pressure applied to the wheel is maintained at that level. In the release mode, the dump valve is open while the isolation solenoid valve is closed to release part of the pressure from the brake and enable the wheel to resume rolling. The dump valve creates a pathway back to the accumulator, where braking fluid is kept until it can be restored by an electric pump to the master cylinder reservoir. The ABS's beating heart is the pump. Without the hydraulic ABS pump, antilock brakes would not be possible. When excessive braking causes a wheel to slide, the HCU's pump returns the brake fluid to the master cylinder, which forces one or both pistons to move rearward in the bore. To provide the proper level of pressure and minimize sliding, the controller modifies the pump's condition. All valves are open while braking normally. Reduce the amount of braking force applied to a wheel when it locks until it starts to spin again. The locking wheel's solenoid valve is closed by the ABS hydraulic unit, which lessens the braking force applied to the wheel. In this manner, the rate of wheel deceleration is slowed to a safe level. When that level is reached, the solenoid opens once again to carry out its typical task.

Unit of Electrical Control

The ABS's brain is the ECU. In the automobile, there is a computer. It keeps an eye on all the sensors that are attached to it and manages the valves and pumps. Simply defined, ABS intervenes within milliseconds by regulating the braking pressure at each individual wheel if the sensors installed at each wheel detect a lockup. Hence, the ABS ensures steerability, stability, and the shortest braking distance possible by preventing the wheels from locking up during braking. The ECU continuously polls the sensor data to assess if any anomalous wheel deceleration has taken place. In ideal circumstances, an automobile would typically take 5 seconds to stop from 60 mph, but in the event of a wheel lockout, the time might be as little as 1. Thus, a sharp decrease in wheel speed is a sure sign that a lockout is happening. The ECU sends a control signal to the HCU to lower the braking pressure when it detects a quick deceleration. It boosts pressure when it detects an acceleration of the wheels until it detects a slowdown once again. The valves open and close swiftly throughout the deceleration-acceleration cycle, which continues until the tires slow down at the same pace as the vehicle. Certain ABS systems have a 16 second cycle rate.

ABS Management

Designers have particular difficulties while creating ABS brake controls. Strong nonlinearity and unpredictability in the system to be managed are the fundamental design challenges for any ABS controller. The interaction between the tire and the road surface is first and foremost exceedingly complicated and poorly understood. The majority of friction models in use today are approximations of highly nonlinear events based on experimental data. The whole vehicle's dynamics are similarly nonlinear, and they even change over time. Moreover, since ABS actuators are discrete, only three kinds of control commands—build pressure, retain pressure, or lower pressure—can be used to achieve control accuracy (recall the three operation modes of solenoid valves). With ABS, several different control strategies have been created. The hunt for

better control strategies is currently ongoing. Threshold control, which is as basic as bang-bang control, was the strategy used in early systems. As controllable variables, it makes use of wheel acceleration and wheel slip. The brake pressure is ordered to rise, keep steady, or fall after the computed wheel deceleration or wheel slip exceeds one of the threshold values. Wheel speed oscillations over time are harder to regulate since the braking pressure is cyclically modified based only on the binary values of the input variables.

A cascade closed-loop control structure is used by a variety of increasingly sophisticated control techniques. The command signal (Vwd) for the inner wheel velocity loop is provided by the outer loop, which also calculates required slip and estimates vehicle velocity (Vv). Many methods for inner-loop control have been put forward. For instance, the PID controller is one of these techniques. Proportional-integral-derivative is referred to as PID. PID controllers, which often appear in industrial control systems, follow the feedback control law. The difference between a desired set point and a measured process variable is used by a PID controller to continually compute an error value. Three control policies have been included into it. By using proportional control (P), the output of the controller is proportionate to the error, in this instance Vwd. With integral control (I), the error finally becomes zero since the controller output is proportionate to the duration of the mistake. The derivative control (D) shortens the reaction time by making the controller output proportional to the rate of change of the error. It has been shown that the traditional PID control algorithms may provide satisfactory results even for complicated and varying surface types. In recent years, the resilience, self-tuning, or adaptation to nonlinear models of other control systems have been integrated with the well-known characteristics of the classic PID, greatly enhancing the ABS performance. The feedback control loop may be constructed as an infinite periodic loop regardless of the control law chosen: Set a timer to interrupt every T seconds with a length of time, wait FOREVER for an interrupt, read sensor data, calculate a control value, and output that value. The time T is a constant in the majority of applications. It is a crucial engineering variable. If T is too little, there will be too much processing; if T is too high, the control variables will not be altered rapidly.

Features of Real-Time Embedded Systems

The ABS example ought to have provided us with some insight into the appearance of a real-time embedded system and how it interacts with the broader system it is a part of to perform the required function. The properties of common real-time embedded systems are covered in this section.

System Architecture

A real-time embedded system communicates immediately and constantly with its surroundings. The system needs sensors in situ to collect data from the target it monitors or regulates, which is its surroundings. For instance, the ABS contains a variety of sensors, such as brake pressure sensors, deceleration sensors, and wheel speed sensors. Contrarily, the majority of data in the actual world is represented by analog impulses. Analog data must be transformed into digital signals in order for a microprocessor to read, comprehend, and act upon it in order to alter the data. As a result, there must be an analog-to-digital converter (ADC) between the sensor and the CPU. The controller, an embedded computer made up of one or more microprocessors, memory, a few peripherals, and real-time software, acts as the brain of an embedded system. Depending on the complexity of the embedded system, the software is often made up of a collection of real-time activities that operate simultaneously. These tasks may or may not be supported by a real-

time operating system. Via actuators, the controller affects the target system. Hydraulic, electric, thermal, magnetic, or mechanical actuators are all possible. The HCU, which has valves and pumps, is the actuator in the case of ABS. The actuator is a physical device that can only respond to analog input, while the CPU only produces digit signals as an output. To apply the microprocessor output to the actuator, a digit-to-analog conversion (DAC) must be carried out. All of these system components' relationships.

Respond in Real Time

A real-time system or application must do certain tasks within predetermined deadlines. This characteristic sets real-time systems apart from non-real-time systems. A typical real-time system is the ABS. The system must operate promptly to prevent the wheels from locking up when the sensors detect a sharp deceleration of the wheels; otherwise, calamity may strike. In addition, control law computing is also real-time; if it is not completed before the start of the subsequent cycle, the amount of data that has to be processed will accumulate. A missile guidance system may strike the incorrect target if it does not promptly change its attitude. Position estimates based on a GPS satellite's signal will be inaccurate if it doesn't maintain a very accurate time measurement.

The needed responsiveness of the sensors, actuators, and target dynamics that the embedded system controls determines how quickly real-time activities must be completed. All jobs must be completed by their deadlines for real-time systems. Real-time does not, however, equate to "real quick" or "the faster, the better." Consider a cardiac pacemaker as an example. In course, the patient's heart may enter fibrillation if it fails to produce electricity through the heart muscle at the appropriate period. But, it will also pose issues if it generates current more quickly than the heart's natural beat.

High-Restrictive Environments

As real-time embedded systems are often used in contexts with limited resources, designing them and improving their performance may be difficult. Many embedded systems only use 8-bit processors, despite the fact that certain embedded systems, including air traffic control and wireless mobile communication systems, operate on relatively powerful CPUs. Examples include the electronics found inside of microwaves, coffee machines, dishwashers, and digital watches. The majority of embedded systems have limits on their user interface, memory, and CPU performance. Several embedded systems work in hostile, uncontrolled environments. They must endure extreme temperatures, dampness, stress, vibration, and even rust. Some examples are the embedded systems in automobiles that manage ignition, combustion, and suspension. To fit into the computer environment and carry out their functions, embedded systems must be optimized in terms of size, weight, reliability, performance, cost, and power consumption. So, compared to desktop apps, embedded systems often need much more optimization.

Concurrency

Several calculations running concurrently and possibly interacting with one another are referred to as concurrency. By design, embedded systems have a tight relationship with their physical surroundings. By the ABS analysis, we have shown this property. Several processes happen simultaneously in a physical context by nature. For instance, the following ABS events may take place concurrently:

1. Wheel speed sensor event
2. Deceleration sensor event
3. Brake pedal event
4. Solenoid valve movement
5. Pump operation

Brake pedal activity, wheel speed sensor activity, acceleration sensor activity, solenoid valve movement, and pump functioning. The reaction times for almost all of these situations are strictly regulated. Every deadline must be reached. Several real-time embedded systems are multirate systems as a result of the presence of numerous control processes, each of which may have its own control rate. For instance, real-time surveillance systems must analyze audio and video inputs, but they do so at various speeds.

Predictability

A real-time system must exhibit predictable behavior under all timing conditions. For example, it must be mathematically predicted if a certain work can be finished by a certain date. System workload, processor power, run-time operating system support, process and thread priority, scheduling algorithm, communication infrastructure, and other factors are taken into account while making this assessment.

Since human lives are on the line, real-time systems like an ABS or an airplane's flight control system must always be 100% predictable. Many real-time embedded systems use distributed computing via international communication networks and comprise diverse computer resources, memory, bus architectures, and operating systems. Events in these systems will inevitably have latency and jitter. As a result, relevant limits need to be defined and maintained. Otherwise, these systems can start to behave erratically. Determinism is a concept associated with predictability. The ability to guarantee the execution of an application without worrying that other circumstances, like as unanticipated occurrences, would interfere with the execution in unexpected ways is known as determinism. In other words, the program will function as intended in both Hard and Soft terms. Real-Time Embedded Systems that are flawless in terms of performance, functionality, and reaction time.

Security and Dependability

Certain real-time embedded systems must have great dependability because they are safety-critical. Flight control systems and cardiac pacemakers are two examples. The definition of safety is "freedom from accidents or losses," and it often refers to both single-point and multiple-point fault situations. The capacity of a system or component to carry out its intended functions within certain parameters over a predetermined period of time is referred to as reliability. A stochastic measure of the proportion of the time the system provides services is how it is described. Embedded systems are often found in devices that must operate constantly and error-free for long periods of time. Certain systems, like those in space and cables beneath the sea, are even unreachable for maintenance. As a result, hardware and software for embedded systems are often created and tested with more care than those for general-purpose computer systems. In terms of failures per million operational hours, reliability is often assessed. For instance, 0.12 failures per million working hours is the standard for automotive microcontrollers. An automatic oil pump has a measurement of 37.3. Failures could be brought on by software bugs, mechanical "wear-out," or cumulative run-time errors.

Real-Time Embedded Systems: Hard and Soft

There are two types of real-time systems: hard and soft. The majority of timing restrictions in a real-time system are considered to be hard. The majority of the timing restrictions in a system are soft in a soft real-time system. A system has to comply with a hard real-time limitation. Missing the deadline will either result in a system failure or render the service provided useless. A soft constraint, on the other hand, is a requirement that a system should satisfy, but if the deadline is infrequently missed, it won't have a terrible effect and the service is still somewhat functional. An expression of a hard constraint is often deterministic. With the ABS, for instance, we may have the following limitations: Every 15 milliseconds, the wheel speed sensors must be polled. The wheel speed control rule calculation must be completed once per cycle in less than 20 milliseconds. The wheel speed prediction has to be completed every cycle in 10 ms. These restrictions are difficult since the sensor data, control value, and forecast wheel speed value are all necessary for the ABS to operate properly. It is also due to the regularity of these incidents. When a cycle's deadline is missed, the following cycle instantly begins, rendering the late result meaningless.

Statistics are often used to represent soft restrictions. With an automated teller machine (ATM), we may, for instance, have the following limitations: The likelihood that the ATM will ask the customer to input a password within one second of inserting a credit card or debit card should be no less than 95%. The ATM shall distribute the requested quantity of cash in 3 seconds at a probability of no less than 90% after receiving a favorable answer from the bank that issued the card. Due to the fact that just the degree of consumers' dissatisfaction with the system is impacted when deadlines are missed, deadlines with these two limitations are soft. The reaction time value function of a real-time activity. If the job has a strict deadline, if that deadline is missed, the task's value is zero. If a task has a flexible deadline, its value drops when it is missed, but not immediately to zero. Soft restrictions are present in many hard real-time systems, and vice versa. A thorough validation is needed when a temporal restriction is specified as hard.

Hardware Elements

Real-time embedded system hardware components are introduced in this chapter. There is a significant difference in the hardware components utilized for real-time embedded systems since they may be as little as digital watches and coffee makers or as large and complex as train control systems and mobile communication switches. A collection of common embedded system hardware components.

Processors

According to each embedded application's need for computing power, different processors are employed in embedded systems. Nonetheless, they may be divided into two broad types. Special-purpose processors and general-purpose microprocessors are the two types. Among special-purpose processors, microcontrollers and application-specific integrated circuits (ASICs) are the most common.

Microprocessors

General-purpose microprocessors are often used in real-time embedded systems. A microprocessor is an integrated circuit that houses a computer processor. It has all or the majority

of the capabilities of the central processing unit (CPU). A group of components required for microprocessors to operate. A microprocessor is made to carry out arithmetic and logic operations using registers, which are tiny storage devices. It contains a control unit that is in charge of instructing the processor to execute program instructions that have been saved. It connects with the memory as well as the arithmetic logic unit (ALU). The instruction register stores binary values for each instruction that is read from memory. After reading such values, the instruction decoder instructs the ALU which computational circuits to activate in order to carry out the desired function. Integer arithmetic and bitwise logic operations are carried out by the ALU. The instructions that make up the microprocessor's design produce these operations. Federico Fagin and his team created the first microprocessor for commercial use, the Intel 4004, in the early 1970s. The Intel 4004 is a 4-bit Microprocessor that the company produced. Prior to it, compact computers were constructed utilizing racks of circuit boards that included many small- and medium-sized integrated circuits (ICs). They were merged by microprocessors into one or more large-scale ICs. All previous central processing unit implementation strategies were rapidly surpassed by this strategy. While 128-bit microprocessors are also available, the majority of contemporary microprocessors are either 32-bit or 64-bit devices. Microprocessors like the Motorola 68HCxxx, Intel 80x86, and SPARC are examples of general-purpose processors.

If tasks are vague, microprocessors may be used. They may be used, for instance, in the creation of papers, games, websites, and software. In many situations, the connection between the input and output is not clear. They need a lot of resources, including I/O ports, Memory, and ROM. For certain functions that are intended for an embedded system, the embedded software may be customized.

Microcontrollers

A microcontroller is a standalone device having peripherals, memory, and a CPU that is designed to carry out certain functions as opposed to a general-purpose microprocessor. In systems where the link between the input and output is often well defined, microcontrollers are employed. A computer mouse, washing machine, microwave, automobile, cell phone, and digital watch are among examples. Applications may be integrated on a single chip with the CPU since they are highly specialized and place less demand on resources like RAM, ROM, and I/O ports. The size and price are thereby decreased. Microprocessors cost ten times more to replace than a microcontroller, which is inexpensive. Additionally, complementary metal-oxide-semiconductor technology is often used in the construction of microcontrollers (CMOS). In comparison to previous methods, this technology is a capable fabrication system that consumes less power and is more resistant to power surges. For medium-sized embedded systems, examples of regularly used 16-bit microcontrollers include the PIC24 series, Z16F series, and IA188 series. RAM typically comes in capacities of 1.5, 2, 4, 8, 16, and 32 kB.

Integrated Circuits for Particular Applications (ASICs)

An ASIC is a highly customized device made only for a single application. In lieu of general-purpose logic circuitry, it is employed. It minimizes the total number of circuits required by combining numerous functionalities into a single chip. ASICs are exceedingly costly to produce, and once they are manufactured, there is no way to change them or make improvements since the most expensive and fixed-cost components are the metal interconnect mask set and its

development. You must change the actual silicon IC layout if you wish to change the instruction set or do anything similar. ASICs are not appropriate for usage in the prototype phase of the system design cycle due to their high cost and lack of programmability. Mobile phones, network routers, and gaming consoles are just a few examples of devices that employ ASICs extensively. ASICs make up the majority of SoC (Systems-on-a-Chip) semiconductors. A microcontroller may be thought of as a particular kind of ASIC that runs programs and can do general tasks as a consequence. ASIC solutions are often more efficient than alternatives based on software operating on microprocessors for a specific application.

FPGAs, or Field-Programmable Gate Arrays (FPGAs)

A programmable ASIC is an FPGA. It features a regular grid of quickly reconfigurable logic cells, allowing for the rapid prototyping of embedded systems. FPGAs are often used for designing systems. Because to their increased performance and cheaper price, they are often replaced in the finished product with specialized circuitry, such as ASIC chips. FPGAs do show up in the finished product when configurability is a crucial component of a real-time embedded system's operation. Xilinx created the first FPGA for commercial use in 1985. Current FPGAs may be used to design extremely high performance systems since they are constructed utilizing cutting edge technologies. For instance, the most recent Xilinx Vertex Ultra- Scale, which was unveiled in May 2014, is based on 20-nm technology. Up to 4.4 million logic cells may be found in a 3D or layered design used by the Ultra Scale. Every computable issue can be solved with an FPGA. Digital signal processing, software-defined radio, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation, radio astronomy, metal detection, and a growing number of other applications are just a few of the specific fields where FPGAs are used. FPGA sales were $5.4 billion in 2013; by 2020, they are anticipated to reach $9.8 billion.

CHAPTER 11

DIGITAL SIGNAL PROCESSORS (DSPS)

Asha. KS, Associate Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-ks.asha@jainuniversity.ac.in

DSPs are designed for calculations requiring high data rates. DSPs provide great performance in repetitive and numerically demanding jobs by implementing algorithms in hardware. In signal processing applications, such as audio, video, and communication applications, DSPs are two to three times quicker than general-purpose microprocessors. Costs are often expensive, which is a disadvantage. Another finding from a recent research is that many commercially accessible DSPs don't have good compiler support.

Processors for Application-specific Instruction Sets (ASIPs)

A new design paradigm called ASIPs provides a middle ground between ASICs and programmable processors. ASIPs often have custom integrated circuitry that is integrated with a set of instructions designed to be advantageous to a particular application. This core specialization offers a compromise between the performance of an ASIC and the flexibility of a general-purpose processor. High performance and enhanced design flexibility are benefits of ASIPs since they can support future design modifications by upgrading the application software running on the ASIP.

CPUs with Several Cores

Since the number of transistors that can fit on a chip and the processor clock speed are closely related, when transistor shrinking technology started to slow down, improvements in enhanced processing speed also started to slow down. The power wall is also a problem. In other words, as computers become more powerful (with denser transistors on a chip), they also produce greater heat and need more energy. Multicore processing has thus become a developing industrial trend. Multicore systems are the norm nowadays. A multicore processor is an integrated circuit to which two or more processors have been connected in order to improve performance, lower power use, and more effectively handle numerous tasks at once. All of the cores of a multicore CPU share the same memory. Yet normally, each core has its own private cache memory. Parallel processing is used by threads on several cores. Threads are time-sliced inside each core.

Harvard Architecture and Von Neumann Architecture

The von Neumann architecture and the Harvard architecture are the two basic computer architectures. Both designs make use of the stored program mechanism, which stores data and program instructions in read-write random access memory (RAM). The von Neumann architecture, in contrast, stores both data and instructions in shared memory; as a result, a data operation and an instruction fetch cannot take place simultaneously since they share the same bus. Contrarily, the Harvard design separates the storage of instructions from that of data. In this manner, the CPU is able to read an instruction and access data memory simultaneously both architectural styles. As compared to the Harvard design, the von Neumann architecture performs

poorly since it only has one common bus for data and instructions. It's often referred to as the von Neumann bottleneck. There are many strategies for overcoming the von Neumann bottleneck. Use of cache memory is one. Later, we'll talk about it. Moreover, because data memory and program memory are physically stored on the same chip, inadvertent program memory corruption may happen while using the von Neumann architecture. Yet, because program memory and data memory are kept physically separate in to obtain better and more consistent memory bandwidth, the majority of DSPs adopt Harvard architecture for streaming data.

Complex Instruction Set Computing and Reduced Instruction Set Computing

A collection of instructions that may be fed into a processor is known as an instruction set. While manipulating data, the CPU is directed by these instructions. An instruction normally consists of zero or more operand specifiers, which may indicate registers, memory addresses, or literal data, as well as an opcode that describes the action to do, such as add contents of memory to register. The interface between the programmer and the hardware is provided by the instruction set architecture. It gets the processor ready to comply with all user requests. Complex instruction set computing (CISC) and reduced instruction set computing are the two most used instruction set designs (RISC). Complex instructions are executed by CISC processors, where a single instruction may carry out a number of low-level actions. The main objective of CISC design is to do a job with the fewest number of lines of assembly instructions. Say that our goal is to multiply two numbers.

A specific instruction, such as MULT, would be available on a CISC CPU by default. When this instruction is performed, the two numbers will be loaded from the main memory into two different registers, multiplied in the execution unit, and the result will either be stored in the right register or returned to memory. Hence, using a single instruction like MULT A, B to multiply two values completely solves the problem. A challenging instruction is MULT. It doesn't need the programmer to explicitly call any loading or saving routines since it acts directly on the computer's memory banks. A complicated instruction could need many clock cycles to complete. Contrarily, RISC processors only use straightforward instructions that may be carried out in a single clock cycle. For instance, the straightforward instructions needed to conduct a multiplication operation are as follows:

Storage and Cache

One of the fundamental parts of embedded systems is memory. Memory cells are the fundamental building blocks of memory. It is necessary to set the memory cell to store a logic 1 (high voltage level) and reset it to store a logic 0 in order for it to store one bit of binary information (low voltage level). Up until a set/reset operation changes its value, it remains in that state. Programs and data are both stored in memory. Memory may be seen as a matrix of bits, where each row's length corresponds to the memory's addressable unit.

The memory's capacity is represented by the total number of rows. A register implements each row, giving it a distinct address. Memory addresses normally begin at 0 and increase in size. Memory is typically byte-addressable, which means that each register contains 8 bits. The memory is implemented using 32-bit registers, and certain computers are capable of processing 32 bits at once. The memory is described as 32-bit word-addressable.

Read-Only Memory (ROM)

Programs are kept in ROM. The information in the program memory won't alter while the program is executing. ROM is a kind of nonvolatile memory, therefore when the ROM is switched off, the recorded software is retained. After production, ROM cannot be altered since it is hardwired. In contrast to ROM, PROM (Programmable Read-Only Memory) may be programmed. We may purchase a blank chip and have a PROM programmer customize it to our specifications. Nevertheless, once we program it, we are unable to alter it.Another kind of nonvolatile memory is EPROM (Erasable Programmable Read-Only Memory). A key distinction between an EPROM and a ROM or PROM is that, once programmed, an EPROM can be unprogrammed by exposing it to a powerful ultraviolet light source (such one produced by a mercury-vapor lamp), and a new program may then be put into it.Like the hard disk in a personal computer, EEPROM (Electrically Erasable Programmable Read-Only Memory) is used to store settings that may periodically change and that need to be remembered the next time the computer starts up. In essence, it may be electronically written, erased, and redone. The data may be erased with no special processing necessary.

The newest ROM and most widely utilized technology in contemporary embedded architecture is flash. Flash memory is an electrically erasable and reprogrammable nonvolatile electronic (solid-state) storage media. Technically speaking, flash memory is a kind of EEPROM. Flash memory got its name from how quickly it can completely erase all of the data from a semiconductor chip, much like the flash of a camera.RAM is the most straightforward and widely used kind of data storage. Regardless of where in the memory the data is physically located, RAM enables data objects to be read or written in virtually the same amount of time. Since RAM is volatile—as opposed to ROM—a power outage will completely delete all of its contents. The two most popular forms of RAM are static RAM (SRAM) and dynamic RAM (DRAM). Companies use several methods for storing data. Whereas DRAM only employs one transistor per memory cell, SRAM utilizes six transistors per memory cell. SRAM is hence more costly to manufacture. SRAM cells, on the other hand, are a specific sort of flip-flop circuit that are often implemented using field-effect transistors with high input impedance. While not in use, it uses very little electricity.

Capacitive storage is used in DRAM. The transistor serves as a switch that enables the control circuitry on the chip to read the capacitor's state of charge or modify it. The capacitor may hold a high or low charge (1 or 0, respectively), and it is switched by the transistor. DRAM must be routinely updated since the capacitor is susceptible to charge loss. DRAM becomes increasingly complicated and demanding of power as a result. Nonetheless, this kind of memory is the most common type utilized in embedded systems since it is less costly to make than SRAM.A form of DRAM that is synced with the system bus is called SDRAM (synchronous DRAM). It is a catch-all term for multiple DRAM types that are synced with the microprocessor's preferred clock speed. Every clock cycle, SDRAM may take one command and send one word of data. It is capable of operating at the standard clock frequency of 133 MHz.

Cache Memory

Recently, there has been a huge improvement in processor speed. On the other hand, memory advancements have mostly focused on density, or the capacity to store more data in a given amount of space, rather than transfer speeds. Since the processor's real speed is limited by the rate of data transmission from the memory, even a fast processor working with a sluggish

memory results in a slow overall speed. Consequently, a quicker CPU just implies that it is idle more. The solution to this issue is cache memory technology. A microprocessor can access cache memory, a form of RAM, much more quickly than it can ordinary RAM. This memory is often built into the CPU chip itself or mounted on a separate chip that is coupled to the processor via a separate bus. Program instructions that are regularly used by software during operation are stored in cache memory, while less frequently utilized data is kept in a large, slow memory device. While processing data, the CPU first checks the cache memory; if the instructions are there, it may skip the longer process of reading data from ordinary memory. If not, a block of main memory made up of a certain number of words is read into the cache before the processor receives the word. Due to the phenomena of locality of reference, the other words in the block are probably going to be spoken in the near future. The total speed of the program execution is accelerated by such a design.

Processors with multilayer cache memory are available. Level 1 refers to level one, Level 2 to level 2, Level 3 to level 3, and so on. L1 cache is directly used by a Processor. The L1 cache's objective is to align with the CPU so that it gets the best possible data to work with throughout each processor cycle. It is a tiny, quick cache built straight into the processor's chip. The L1 cache generally employs high-speed SRAM and has a size range of 8 to 64 KB. Every few processor cycles, the L2 cache is anticipated to send data to the L1 cache, while the L3 cache may feed data to the L2 cache at an even slower pace. Based on CPU needs, the three tiers exchange data. A dual-core CPU with two layers of cache memory. While they share the L2 cache and, of course, the main memory, each core has its own private L1 cache.

Interfaces for IO

Using I/O interfaces, embedded CPUs interact with the outside world. I/O interfaces are electrical components that are available to designers as pins on a chip and have connections on both the processor's and input/output devices' sides. A microwave's keypad is an example of an electrical gadget that allows users to communicate with the microwave. The interface is a specific circuit that connects the keypad to the integrated microcontroller. A user presses a key, which is detected by the circuit, which then transforms the event into a distinct, recognized binary number and sends it to the processor's I/O port. The number is read from the port by the processor, who then interprets it. In the case of analog I/O, as shown by the ABS example, a D/A converter (DAC) is required to decode digital outputs and an A/D converter (ADC) is required to encode analog inputs (often voltage) into a digital word (typically 8 bits or 16 bits). Let VRefL and VRefH represent the minimum and maximum voltages that an ADC can encode, respectively. The ADC converts all conceivable inputs into a range of numbers between 00 and FF (8 bits) or between 0000 and FFFF (16 bits). The resolution of the conversion, which reflects the size of the quantization error, is based on the number of bits in the digit word. An analog input may be encoded to one of 256 different levels, for instance, using an ADC with a resolution of 8 bits. Volts may also be used to represent the ADC resolution. The least significant bit (LSB) voltage is the smallest voltage change necessary to ensure a change in the output code level. The LSB voltage is the same as the ADC's resolution Q. The LSB voltage is equal to half of the greatest quantization error. I/O interfaces link various I/O peripherals to the integrated CPU in most cases. Each peripheral is given a unique address to identify it. The process issues a command containing the address of the target peripheral whenever it performs an I/O-related instruction. As a result, each I/O interface must analyze the address lines to ascertain if the

command is for it or for another interface. There are two complementing techniques to map the I/O in a system where the CPU, main memory, and I/O peripherals share a common bus:

Sensors and Actuators

Isolated I/O, also known as port-mapped I/O, and memory-mapped I/O. Dedicated I/O buses or an additional I/O pin on the physical interface of the CPU are used to provide port-mapped I/O, which utilizes a different address space from main memory. Using a specific set of CPU instructions, I/O devices are accessible. This is frequently referred to as isolated I/O since the address space for I/O is separate from that for main memory. Memory-mapped I/O involves integrating the memory of the I/O peripherals with the primary memory map. This means that certain locations in the processor's memory will actually belong to peripheral memory rather than RAM. As compared to port-mapped I/O, memory-mapped I/O is more straightforward to build since it doesn't need any additional processor pins or a separate bus like port-mapped I/O does. Moreover, memory-mapped I/O is more effective than port-mapped I/O.

The capabilities of port-mapped I/O instructions are quite restricted; they are often only offered for basic load-and-store operations between CPU registers and I/O ports. Due of this, adding a constant to a device register that is port-mapped would take three instructions: reading the port into a CPU register, adding the constant to the CPU register, and writing the outcome back to the port. All of the CPU's addressing modes are accessible for both I/O and memory, however, and instructions that perform an ALU operation directly on a memory operand may also be used with I/O device registers since memory mapper I/O uses ordinary memory instructions to address the devices.

Actuators and Sensors

A sensor is an embedded system's input device. This device is a transducer that transforms energy for use in measurement or control. For instance, a camera is a sensor that converts photon energy to electrical charge that represents the photon flux for each image element in an array. An accelerometer translates acceleration to voltage. An ultrasonic sensor transforms ultrasound waves to electrical signals. Sensors come in a variety of forms. Among the most popular ones are displacement sensors, pressure sensors, humidity sensors, acceleration sensors, gyro sensors, temperature sensors, and light sensors. A good sensor should be impervious to interference and sensitive to the property being measured. The sensor's output is AC voltage, and its input is the rotation of the sensor rotor. The rotational speed is inversely related to the voltage magnitude and frequency. Almost every physical or chemical quantity, including weight, velocity, acceleration, electrical current, voltage, temperatures, and chemical compounds, may be measured using a sensor. The construction of the sensors makes advantage of several physical phenomena. For instance, the induction effect is used in automobile wheel speed sensors to create an electromotive force when a magnetic field interacts with an electric circuit. The piezoelectric effect—in which certain materials produce an electric charge in response to applied mechanical stress—is the basis on which the car airbag sensor was created. Both passive and active sensors are available. An external energy source is necessary for an active sensor to function. Radar, sonar, GPS, and X-ray are among examples. Passive sensors, on the other hand, only detect information from the physical world and react to it. A passive sensor is, for instance, the wheel speed sensor. Without using any energy or sending any signals to the wheel, it can detect and measure the rotation of the wheel. Another example of a passive sensor is a temperature sensor. The following factors mostly determine how well sensors perform:

An actuator is a transducer used to move or control a system by converting electrical energy into another kind of energy, such as motion, heat, light, or sound. It serves as the impetus for a number of human-made and natural necessities. Hydraulic, pneumatic, and solenoid actuators are common types. A cylinder or fluid motor that employs hydraulic power to enable mechanical action makes up a hydraulic actuator. A pneumatic actuator transforms the energy produced by a vacuum or highly compressed air into a linear or rotational motion. A solenoid is a particular kind of electromagnetic actuator that creates a linear motion by converting an electrical signal into a magnetic field. The valves are moved by soldering in the ABS hydraulic control unit. Shape-changing materials provide the foundation for certain recently invented actuators, including piezoelectric, shape memory alloy, and magnetostrictive devices. They are used more often in creative applications. For instance, high-speed precision ceramic actuators called piezoelectric actuators turn electrical energy into linear motion with great resolution. Several contemporary high-tech fields, including microscopy, bio nanotechnology, and astronomy/ aerospace technologies, utilise these actuators.

Timers and Counters

Actuators come in several varieties, each with its own unique characteristics. Yet, the following factors essentially determine how well an actuator performs: Highest amount of force or mechanic it can sustainably impose on a system cyclical action Quickness of operation operating circumstances, including temperature, Service life measured in hours or operating cycles.

Counters and Timers

In real-time embedded systems, timing functions are essential. A timer is a unique kind of clock that is used to record durations of time. The number of external events that occur on an external event pin is counted by the counter. A timer and counter function similarly to each other when the event is a clock pulse. As a result, these two words are often used in the same context. A free-running binary counter serves as the timer's primary building block. Each timing pulse that is received causes the counter to increase. Because to its freedom of movement, it can count inputs like clock pulses while the CPU is running the main program. The pulse count accurately estimates the time interval if the input pulses come at a fixed rate. The elapsed time is 1000 microseconds, for instance, if the input pulse rate is 1 MHz and the counter has recorded 1000 pulses. An output signal is asserted when the count exceeds its capacity. The overflow signal may then cause the CPU to receive an interrupt or set a bit that the processor may read. A 16-bit counter that accepts clock cycles as input. The input pulses could not be the same as the clock pulses. Afterwards, pulses are produced using a prescaler. Prescalers are clock-divider circuits that may be customized. Before feeding it to the counter, it divides the fundamental clock frequency by a certain amount. We may allow the counter to count at a chosen pace by using a prescaler. A 16-bit timer may record up to 65,535 x 8 = 524,280 microseconds before it overflows, for instance, if we configure the prescaler to split the 1 MHz of clock frequency by 8. This results in the new rate of the timer being 106/8 = 125 KHz. the free-running counter is often coupled to a capture register. At the occurrence of some event, often a signal to an input pin, a capture register may automatically load the current output of the free-running counter, latch the value into a processor-visible register, and then produce an output signal. Measuring the interval between the leading edges of two pulses is one use of a timer with the capture requester. The program can determine how many clock cycles have passed by reading the value from the capture register and comparing it to an earlier reading.

Timing Events Example

Think of a timer that has a prescaler in its design. The free-running counter has 16 bits, while the prescaler is configured with 3 bits. A clock with an 8 MHz frequency generates timing pulses, which are counted by the timer. Let's say a CPU latches a capture signal with a count of 304D in hex. We are interested in learning how much time has passed since the free counter was last refreshed. Then, we translate the hexadecimal value 304D to its decimal equivalent, which is 12,365. The prescaler divides the clock frequency by 23 since it is set up with 3 bits. As a result, the signals sent into the free-running counter have a frequency of 8 106/23 Hz, or 1 MHz. The time that has passed is thus 12365106, or 0.12365 second, or 12,365 microseconds.

Operating Systems in Real-Time

Real-time operating system is the brain of several computerized embedded systems (RTOS). An operating system known as an RTOS allows the development of applications that must provide logically valid computation results while still adhering to real-time limitations. It offers tools and services for managing resources, scheduling tasks in real time, and coordinating between tasks. In this chapter, we first quickly go through the primary duties of general-purpose operating systems before talking about the features of RTOS kernels. Next we present a few popular RTOS solutions.

General-purpose Operating Systems' Primary Functions

An operating system (OS) is the piece of software that lies in between a computer's hardware and any software programs that are executing on it. An operating system manages and allocates resources. To make the computer system easier to use, it controls the hardware resources and obscures the specifics of how the hardware functions. The CPU, memory, I/O controllers, disks, and other components like terminals and networks make up the majority of a computer's physical resources. A policy enforcer is an OS. To avoid mistakes and incorrect use of the computer, it supervises program execution and specifies the ground rules for interaction between applications and resources. An OS is made up of a number of software components, the central ones of which make up the kernel. The kernel gives the computer's hardware the simplest possible level of control. An operating system's kernel always operates in system mode, whereas all other components and programs operate in user mode. Protection methods are included into the kernel implementation to prevent user space applications from secretly altering kernel functionality. Moreover, the OS offers an application programming interface (API), which specifies the guidelines and interfaces that let applications access OS functions and interact with hardware and other software programs. System calls and message forwarding are two ways that user processes might ask the kernel for services. When using the system call method, the user process applies traps to the OS routine that chooses which function should be called, switches the processor to system mode, and calls the function as a procedure, switches the processor back to user mode after the function is finished, and so on. The user process creates a message that defines the required service and uses a send function to transfer it to an OS process in the message passing approach. The transmit function verifies the requested service that was specified in the message, switches the processor mode to system mode, and sends the message to the process that performs the requested service. The user process uses a message receive operation to wait for the outcome of the service request in the interim. The OS process notifies the user process when the service is finished by sending a message back. The next portions of this section provide a brief overview of some of the primary features of a typical general-purpose OS.

Process Management

An instance of a software in use is referred to as a process. Within the system, it serves as a unit of labor. A process is an active entity, while a program is a passive one. To do its work, a process requires resources like CPU, memory, I/O devices, and files. When an application program is executed, the OS kernel creates a process, allots memory and other resources, determines its priority in multitasking systems, loads program binary code into memory, and starts the application program's execution. The application program then interacts with the user and hardware devices. Any reusable resources are released and given back to the System when a process terminates. The OS has a difficult task when a new process is started, which involves allocating memory, building data structures, and copying code. The fundamental unit to which the OS allots processing time is a thread, which is a route of execution inside a process. A process may have one or more threads running at once.

A thread theoretically has the same capabilities as a process. The job that each one is employed to complete is the key distinction between a thread and a process. Threads are utilized for quick operations, while processes are used for larger, more complex activities, such as running programs. Hence, a thread is sometimes referred to as a lightweight process. Although different processes do not, threads inside the same process do share an address space. Moreover, global and static variables, file descriptors, signal accounting, code areas, and memory are shared across threads. This makes it possible for threads to communicate more easily and enables them to read from and write to the same variables and data structures. As a result, threads use far fewer resources than processes each thread of the same process has its own thread status, program counter, registers, and stack. The lowest unit of work that the OS scheduler manages independently is a thread. Tasks are often referred to as threads or single-threaded processes in RTOSs. For instance, the RTOSs VxWorks and MicroC/OS-III use the word tasks. Tasks and processes (threads) are used interchangeably in this text. Using the right system calls, such as fork or spawn, processes may generate new instances of themselves.

The process that creates is referred to as the parent of the other process, which is referred to as the kid of that process. As a process is formed, it is given an individual integer identifier known as a PID, or process identifier. Arguments are not necessary to establish a process. A parent process often has control over a child process. The parent has various control over the kid, including the ability to give it messages, terminate it, check into its memory, and more. A child process may get part of its parent's shared resources. By using the exit() system function, processes may ask to be terminated on their own. The system may also terminate processes for a number of reasons, such as when it is unable to provide the required system resources, when a kill command is sent, or in reaction to another unhandled process interrupt. All of a process's system resources are released upon termination, and open files are flushed and closed.

There are several reasons why a procedure could be put on hold. A process that is ready to run must be brought in through swapping, which requires the System to release enough main memory, or via timing, which causes a process to pause while it waits for the next time interval. A parent process could also want to pause a child's execution so that it can be examined, changed, or the actions of different children can be coordinated. Processes sometimes need to talk to one another while they are operating. Interprocess communication is what this is (IPC). The OS has IPC support methods. IPC techniques often used include files, sockets, and message queues, pipes, named pipes, semaphores, shared memory, and message passing.

Memory Administration

When it comes to the speed at which applications operate, main memory is the most important component of a computer system. All system memory that is now being used by applications belongs to the OS kernel. Data and instructions are memory-based entities. Every place in memory has a physical address. Data may be accessed 8 bits at a time, regardless of the width of the data and address buses, in the majority of computer architectures because memory is byte-addressable. Memory addresses are digit sequences with a predetermined length. Physical memory may often only be addressed by system software, such as the OS and Basic Input/output System (BIOS). The majority of application programs are unaware of physical addresses. They substitute logic addresses instead. A logical address is the address that a memory region seems to be located at to an application program that is currently running. Due to the use of an address translator or mapping function, the logical address and the physical address may differ.

The phrase "physical address" is often used to distinguish from a virtual address in a computer that supports virtual memory. In specifically, the virtual and physical addresses refer to an address before and after translation carried out by the MMU, respectively, in computers using a memory management unit (MMU) to translate memory addresses. Using virtual memory is beneficial for several reasons. One of them is memory defense. When two or more processes utilize direct addresses while running concurrently, a memory mistake in one process (such as reading a wrong pointer) might wipe out the memory used by the other process, crashing several applications at once. On the other hand, the virtual memory approach may make sure that every process is executing in its own unique address space. A program must be translated into a load module and kept on disk before the loader, an OS component, can load it into memory. The compiler compiles the source code to produce a load module. An object module is created by the compiler. A symbol table section, which contains all external symbols used in the program, a machine code section containing the executable instructions created by the compiler, an initialized data section containing all data used by the program that needs to be initialized, and a header indicating the size of each of the sections that follow. Certain external symbols are utilized in this object module while others are defined in it and will be used by other object modules. When a process is launched, the operating system (OS) allots memory to it before loading the process's load module into the memory it has been given. Executable code and initialized data are transferred from the load module into the process' memory during loading. Memory is also allocated for runtime stack, which is used to store details about each procedure call, and uninitialized data. The stack's starting size is set by default by the loader. As long as the predefined maximum size is not exceeded, more space will be assigned to the stack when it fills up during runtime. Memory allocation while a program is executing is supported by several programming languages. That may be done, for instance, by calling new in C++ and Java or mallow in C. This memory originates from a large memory pool known as the heap or free store. Certain areas of the heap are always in use, while others are vacant and so open for future allocation.

Modern operating systems provide dynamic loading and dynamic linking as two ways to avoid loading large executable files into memory. A program's procedure (library or other binary module), known as dynamic loading, is not loaded until the program calls it. Every procedure is stored on the hard drive in a relocatable load format. The main program is performed after being loaded into memory. On demand, further standard procedures or modules are loaded. Unused routines are never loaded thanks to dynamic loading, which improves memory space usage.

When a significant amount of code is required to address seldom occurring circumstances, dynamic loading might be helpful. Libraries are connected during execution when using dynamic linking. This contrasts with static linking, when libraries are linked at compilation time and a large executable code is produced as a consequence. Dynamic linking is the process of resolving symbols after compilation time by connecting their names to addresses or offsets. Libraries especially benefit from it.

Single contiguous allocation is the most basic memory management strategy. That is, all of the RAM is made accessible to a single application, with the exception of the space set aside for the OS. The MS-DOS operating system employs a method like this. Partitioned allocation is an additional method. Several memory partitions are created, with a limit of one process per partition. As a process begins, the memory management chooses a free partition, assigns it to it, and deallocates it when the process is complete. Some systems let you switch out a partition to secondary storage to free up more RAM. It is then readmitted into memory to continue running later.

The program's virtual address space is divided into pages of the same size, and memory is divided into fixed-size units called page frames. Mapped to frames by the hardware MMU are pages. The address space looks to be contiguous whereas the actual memory may be provisioned on a page basis. Programs and data are not differentiated and protected individually by paging. Programs and data may be divided up into conceptually separate address areas using segmented memory, which facilitates sharing and security. Memory spaces called segments often correlate to logical groups of data like a data array or a code process. A segment table, which typically includes the physical location of the segment in memory, its size, and other information like access protection bits and status, is necessary for supporting segments on hardware (swapped in, swapped out, etc.). The lengthy, continuous free memory space is broken up into ever-increasing contiguous parts when processes are loaded and unloaded from memory. The software may eventually be unable to access huge, continuous blocks of memory. The issue is known as fragmentation. Fragmentation is often decreased by reduced page sizes.

Disputes with Management

A signal signaling an occurrence that requires quick attention may come from a computer's running process or from a device connected to the computer. In response, the processor stops what it is doing, saves its state, and launches an interrupt handler (also known as an ISR or interrupt service routine) to deal with the incident. Interrupts are what drive modern OSs. Almost all activities are started when interruptions arrive. The interrupt vector, which includes the addresses of all the service routines, is used to shift control from the interrupt to the ISR. The address of the interrupted instruction must be saved by the interrupt architecture. When another interrupt is being handled, incoming interrupts are disabled. A system call is a software-generated interruption brought on by a user request or a mistake.

Multitasking

Events in the real world might happen concurrently. The capacity of an operating system to enable numerous separate applications running on the same machine is referred to as multitasking. It is mostly accomplished by time-sharing, in which each application consumes a portion of the computer's processing time. Schedulers, which use scheduling algorithms to choose when to perform which job on which processor, deal with how to divide processors' time

among several tasks. Each task has a context, which is the information that describes its current state of execution and is kept in the task control block (TCB), a data structure that holds all the details necessary for the job's successful completion. When a scheduler switches a task off of the CPU, the task's context must be saved; when the task is chosen to run again, the task's context is restored so that the task may resume execution from the point at which it was last stopped. The cost of multitasking is the expense of context shifts. They often need a lot of calculation. One of the responsibilities of OS design is context switch optimization. This is especially true with RTOS architecture.

File System Administration

The basic abstraction of secondary storage devices is a file. Every file is a designated grouping of data kept on a device. The file system, which offers features for file management, auxiliary storage management, file access control, and integrity assurance, is a crucial part of an OS. In order to store, reference, distribute, and protect files, there must be appropriate methods in place. The file system allots initial space for the data when a file is created. When the file expands, further incremental allocations come next. The space that is made available when a file is removed or has its size reduced is thought to be usable by other files. This results in alternating, variously sized utilised and unused zones. When a file is created, if an area of contiguous space is not immediately accessible, the space must be allocated in pieces. It makes logical to utilize noncontiguous storage allocation algorithms since files do have a tendency to expand or shrink over time, and because users seldom know in advance how big their files will be.

In a computer, files are often arranged into directories, which form a hierarchical system of tree structure. The amount of the data contained in each file, the date and time it was last modified, its owner's user ID and group ID, and its access rights are all commonly stored in a file system. The file system also offers a range of commands for reading and writing a file's contents, setting a file's read/write position, using a protection mechanism, changing ownership, listing files in a directory, and removing a file. A two-dimensional matrix that identifies all users and all files in the system may be used to provide file access control. The matrix's entry at index I j) indicates whether user I is permitted access to file j. This matrix would be quite huge and very sparse in a system with a high number of users and a large number of files. Controlling access to different user classes is a considerably more compact strategy. Role-based access control (RBAC) is a kind of access control that only allows authorized users to access data. RBAC distributes users to distinct roles, and permissions are given to each role in accordance with the demands of the user's position. A user may be given a variety of jobs to carry out daily duties. For instance, a user can need both a developer job and an analyst position. Each role would provide the authorizations required to access certain objects.

Management of I/O

A broad variety of I/O devices are interacted with by modern computers. Among the most popular ones are keyboards, mouse, printers, disk drives, USB drives, displays, networking adapters, and audio systems. Hiding the quirks of hardware I/O devices from the user is one goal of an OS.

Each I/O device in memory-mapped I/O takes up some space in the I/O address space. Physical memory addresses in the I/O address space allow for communication between the CPU and the

I/O device. The CPU obtains data from or issues instructions to I/O devices by reading from or writing to those addresses.

Device controllers are used by most systems. An interface unit is mainly what a device controller is. The device controller facilitates communication between the OS and the I/O device. Almost all device controllers are capable of direct memory access (DMA), which enables them to access system memory without the processor's help. This relieves the CPU of having to transport data to and from I/O devices. Since no I/O transfers need the CPU's immediate attention, interrupts enable devices to let the processor know when they have data to send or when an operation is finished. This frees the processor to carry out other tasks. After finishing each instruction, the CPU detects the interrupt request line. The CPU detects an interrupt raised by a device controller by asserting a signal on the interrupt request line, stores the state, and then transfers control to the interrupt handler. In order to restore control to the processor, the interrupt handler first identifies the reason for the interruption, takes the appropriate actions, and then executes a return from interrupt instruction.

Latencies for I/O operations are often high. The majority of this delay is brought on by the sluggish peripheral device speeds. For instance, data cannot be read from or written to a hard drive until the disk's rotation immediately places the read/write head over the target sectors. Having one or more input and output buffers connected to each device helps reduce latency.

CHAPTER 12

CHARACTERISTICS OF RTOS KERNELS

K. Gopala Krishna, Associate Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-k.gopalakrishna@jainuniversity.ac.in

A general-purpose OS offers a wide range of services that real-time systems also need, but it consumes too much space and has too many features that may not be required for a given real-time application. Moreover, it is not configurable, and the inherent timing uncertainty of the system means that response time is not guaranteed. A general-purpose OS is thus inappropriate for real-time embedded devices. The design of an RTOS must meet three important criteria. Initially, the OS's time behavior has to be foreseeable. The maximum execution time for each service offered by the OS must be known. These functions include managing interrupts and OS calls, among others. Second, the scheduler must be aware of job deadlines, and the OS must control time and scheduling. Lastly, the OS has to be quick. For instance, the context transition overhead should be minimal. A quick RTOS ensures strict deadlines while also assisting with a system's soft real-time limitations. A real-time kernel and additional higher-level services, such file management, protocol stacks, a Graphical User Interface (GUI), and other elements, are often found in an RTOS. I/O devices are the main focus of most supplementary services. A real-time kernel is software that controls a microprocessor's or microcontroller's time and resources while giving applications essential functions like task scheduling and interrupt management. A little piece of code known as a board support package (BSP) is developed for a certain board that complies with a specific OS in embedded systems. It is often constructed with a bootloader that has the bare minimum device support necessary to install the operating system and device drivers for every device on the board. The remainder of this section introduces some of the most significant real-time services for RTOS kernels that are defined in POSIX 1.b.

Timers and Clocks

The majority of embedded systems need to keep track of time. In most RTOS kernels, the number of system ticks serves as a representation of time. The RTOS operates by timing all events based on hardware interrupts that are set up to occur frequently, say once per millisecond. For instance, if you provide the argument 20 while using the task delay function in VxWorks, the job will stall until the timer interrupts 20 times. Each tick in the POSIX standard corresponds to 10 milliseconds, and there are 100 ticks in a second. The procedures provided by certain RTOS kernels, like VxWorks, enable users to set and get the value of the system tick. The interrupt is also known as a clock interrupt, and the timer is sometimes referred to as a heartbeat timer.

An ISR increases the tick count and determines whether it is time to unblock or wake up a job at each clock interrupt. If so, it contacts the scheduler to reschedule if necessary. An RTOS kernel, such as the taskDelay function in VxWorks, enables you to call functions of your choosing after a certain number of system ticks based on the system tick. Your function could be immediately invoked from the timer ISR depending on the RTOS. There are further timing services as well. For instance, the majority of RTOS kernels enable programmers to set time limits for how long a

job must wait before receiving a message from a mailbox or queue, a semaphore, and other things. The determinism of real-time applications is enhanced by timers. Applications may schedule events to occur at certain intervals or times using timers. Many timer-related functions were specified by POSIX, including the following:

1. Allocate a timer using the specified clock as a timing foundation using the timer-create() function.
2. Remove a previously made timer using timer-delete ().
3. Get the amount of time left till expiry and the reload value using timer-get time ().
4. Return the timer expiry overrun with timer-get-overrun ().
5. Set the time until the next expiry and arm the timer with timer-set time ().
6. The current job is suspended using the Nanosleep () function until the timer expires.

Priority Planning

All tasks are not equal in terms of how urgently they must be completed since real-time activities have deadlines, whether they are soft or hard. Shorter deadline tasks should be scheduled for execution before longer deadline tasks. As a result, an RTOS usually prioritizes tasks. In addition, the RTOS should instantly schedule the higher priority job on the processor for execution if a higher priority work is released while the processor is servicing a lower priority task. This will guarantee that the higher priority task is completed before its deadline. Pre-emption is the term for this action. Priority-based and preemptive task scheduling are popular in real-time applications. Early deadline first (EDF) and rate monotonic (RM) scheduling are two examples. Real-time systems should not use scheduling algorithms like first-in-first-served and round-robin that do not take task priority into consideration.

By using priority-driven preemptive scheduling, the preemptive scheduler has a clock interrupt job that gives it the ability to switch once the task has had a certain amount of time to run—the time slice. The benefit of this scheduling technique is that it ensures that no job occupies the processor for any longer than the time slice. The dispatcher, which is the module that provides the job chosen by the scheduler control of the CPU, is a crucial part of scheduling. As a consequence of an interrupt or system call, it is given control while operating in kernel mode. It is in charge of changing the context. As it is called every time a job is switched, the dispatcher should be as quick as possible. Context switches should not be used if possible since they practically idle the CPU for a brief period of time. The selection of task priorities is the key to the effectiveness of priority-driven scheduling. Low-priority jobs may suffer from starvation and fail to complete on time if scheduling is based on priorities. Some well-known scheduling algorithms and resource access control mechanisms will be covered in the next two chapters.

Communication and Resource Sharing Between Tasks

A task cannot call another task in an RTOS. Instead, tasks coordinate their execution and access to shared data via real-time signals, mutex, or semaphore objects, as well as information transmission through message passing or memory sharing.

Real-Time Signals

Similar to software interruptions, signals exist. When a child process ends in an RTOS, a signal is immediately transmitted to the parent. Signals are also used for a variety of additional synchronous and asynchronous notifications, including alerting a process to a memory violation

and waking it up when a wait call is made. POSIX extended the signal generation and delivery to improve the real-time capabilities. In real-time systems, signals play a crucial role as the a method of alerting processes to the presence of asynchronous events such the expiry of high-resolution timers, the receipt of quick interprocess messages, the completion of asynchronous I/O, and the transmission of explicit signals.

Semaphores

Semaphores are counters that are used to manage access to shared resources across processes or threads. The value of a semaphore is the number of units of the resource that are currently available. On semaphores, there are two fundamental operations. One is to atomically increment the counter. The alternative is to atomically decrease the counter after waiting till it is not null. A semaphore does not keep track of which resources are free; it merely keeps track of how many resources are free. In situations where a resource may only be utilized by one job at a time, a binary semaphore functions similarly to a mutex.

Message Transfer

Tasks may communicate with one another by sending messages in a structured message passing scheme in addition to signals and semaphores. Information transmission is far more helpful when messages are passed. Moreover, it may be used only for synchronization. Message passing and shared memory communication often coexist. Everything that both parties to a communication can understand might constitute the message's substance. Send and receive are the two essential actions. Direct or indirect message passing are both options. Any process that wants to communicate through direct message passing must clearly identify the communication's receiver or sender. Indirect message passing involves the sending and receiving of messages from ports or mailboxes. Only if two processes share a mailbox are they able to interact in this manner. Both synchronous and asynchronous message passing are possible. The sender process is stopped in synchronous message passing until the message primitive has been executed. Asynchronous message passing gives control back right away to the sending process.

Communal Memory

An RTOS may translate shared physical space into independent, process-specific virtual space by using shared memory. Information (resources) are often shared across many processes or threads via shared memory. Exclusive access must be granted to share memory. As a result, the memory space has to be protected using mutex or semaphores. A task's essential portion is the piece of code that accesses the shared data. A side effect in using shared memory is that it may cause priority inversion, a situation that what a low-priority task is running whereas a high-priority task is waiting.

Asynchronous I/O

I/O synchronization comes in two flavors: synchronous I/O and asyn- chronous I/O. When a user task requests an I/O operation from the kernel in synchronous I/O and the request is approved, the system will wait until the operation is finished before it may go on to other jobs. It is preferable to have synchronized I/O when the I/O process is quick. Also, it is simple to apply. Application processing and application-initiated I/O activities might overlap with the help of an RTOS. This is the RTOS's asynchronous I/O service. When a task requests an I/O operation using asynchronous I/O, other jobs that are not dependent on the I/O results are scheduled to run

in the meanwhile. Tasks that rely on the I/O being finished are halted in the meantime. To increase throughput, latency, and/or responsiveness, asynchronous I/O is employed.

Storage Locking

A process may use the real-time capability of memory locking, which is defined by POSIX, to eliminate the delay of obtaining a page of memory. In order to make the page memory-resident, or to keep it in the main memory, the memory is locked. This enables precise control over which application components must remain in physical memory in order to minimize the cost of data transfers between memory and disk. For instance, memory locking may be used to retain a thread that monitors a crucial process that has to be attended to right away in memory. The locked memory is instantly unlocked as soon as the procedure ends. Moreover, locked memory may be intentionally unlocked. To lock and unlock memory, POSIX, for instance, defined the mlock() and munlock() functions. No matter how many times the mlock function was invoked, the munlock function releases the specified address range. In other words, many calls to the mlock function may lock address ranges, while a single call to the unlock function can unlock the locks. Thus, memory locks cannot stack. When several processes lock the same or an adjacent area, the memory region is locked until all processes have released it.

LynxOS

A Unix-like RTOS made by Lynx Software Technologies is called LynxOS. LynxOS is a POSIX-compliant RTOS that offers deterministic, hard APIs in an integrated kernel with a tiny footprint. It has memory locking, real-time priorities, preemptive scheduling, known worst-case response time, and ROMable kernel. To fully use multicore/multithreaded CPUs, LynxOS offers symmetric multiprocessing capability.

The most recent version, LynxOS 7.0, comes with cross-development host compatibility, new tool chains, and debuggers. From the ground up, the LynxOS RTOS is built to comply with open-system interfaces. For embedded real-time applications, it makes use of the existing Linux, UNIX, and POSIX programming skills. By employing well-known procedures rather than learning proprietary ones, programmers may be more productive and reduce the amount of time spent developing real-time systems. The majority of applications that employ LynxOS are real-time embedded systems in the avionics, aerospace, military, industrial process control, and telecommunications industries. There are already millions of devices using LynxOS.

An abbreviation for the embedded operating system is OSE. The real-time embedded operating system was developed by the Swedish information technology firm ENEA AB. OSE employs signals in the form of messages sent to and received from system operations. Each process has a queue where messages are kept. Signals may be sent between processes running on different computers using a number of transports thanks to a link handler mechanism.

The open-source interprocess kernel design was built on the OSE signaling technology. A high-level programming paradigm and an easy-to-use API are shared by the Enea RTOS family to make programming simpler. It is made up of two items, each of which is tailored for a certain category of applications: A fault-tolerant, distributed, and multicore RTOS with outstanding performance is called Enea OSE. The full-featured ENEA OSE RTOS is available in a compact and multicore DSP-optimized form as Enea OSEck. OSE supports a wide variety of processors, primarily 32-bit ones from the ARM, PPC, and MIPS families.

QNX

The full-featured and durable QNX Neutrino RTOS was created by QNX Software Systems Ltd, a BlackBerry subsidiary. Products from QNX are made for embedded systems that operate on many different platforms, such as ARM and x86, and a variety of boards that are implemented in almost any kind of embedded environment. As a microkernel-based operating system, QNX is built on the concept of executing the majority of the OS kernel as a collection of discrete jobs, referred to as servers. This contrasts with the more conventional monolithic kernel, in which the OS kernel is a single, very large program made up of a sizable number of special-purpose components. Using QNX, developers may disable any feature they don't need without changing the OS; instead, those servers won't operate thanks to the usage of a microkernel. A QNX variant serves as the main operating system for BlackBerry's Playbook tablet computer. QNX is also the foundation for the BlackBerry range of smartphones, which run the BlackBerry 10 OS.

VxWorks

An RTOS called VxWorks was created as a proprietary piece of software by Wind River, an Intel subsidiary that offers runtime software, industry-specific software solutions, simulation technologies, development tools, and middleware for embedded systems. VxWorks is created for use in embedded systems that need real-time and predictable performance, similar to other RTOS offerings.

The Intel, MIPS, PowerPC, SH-4, and ARM architectures are supported by VxWorks. A collection of runtime components and development tools make up the VxWorks Core Platform. Compilers like Diab, GNU, and Intel C++ Compiler (ICC), as well as its build and configuration tools, make up the basis of the VxWorks development environment. Moreover, the system comes with productivity tools including the Workbench development suite, Intel tools, and development support tools for host support and asset monitoring. Using VxWorks, cross-compiling is used. An integrated development environment (IDE), which includes the editor, compiler toolchain, debugger, and emulator, is used to build software on a "host" machine. The "target" system is then used to build software for use there. As a result, the developer may target hardware with more constrained capabilities while using strong development tools. Products from a variety of industries, including aerospace and military, automotive, industrial (including robotics), consumer electronics, medical, and networking, employ VxWorks. VxWorks is also used as the onboard OS in a number of well-known devices. The Mars Reconnaissance Orbiter, the Phoenix Mars Lander, the Deep Impact Space Probe, and the Mars Pathfinder are a few examples of spacecraft.

Embedded Windows Compact

Windows Embedded Compact is an OS subtype created by Microsoft as a member of their Windows Embedded family of technologies. It was formerly known as Windows CE. It has a little impact. For devices with less memory, such as industrial controllers, communication hubs, and consumer electronics like digital cameras, GPS systems, and car entertainment systems, RTOS is optimal. It works with x86, SH (just for automobiles), and ARM.

Scheduling tasks

Some of the fundamental operations of every RTOS kernel include task management and scheduling. A scheduler that is a component of the kernel is in charge of assigning and

scheduling jobs on processors to make sure that due dates are fulfilled. Several well-known and often used methods for job assignment and scheduling are presented in this chapter.

Tasks

A task is a piece of work that is scheduled for CPU execution. It is a real-time application software building component that is supported by an RTOS. A real-time application that makes use of an RTOS may really be organized as a collection of separate jobs. Three categories of tasks exist:

Routine duties. The repetition of periodic tasks occurs once per period, such as 200 milliseconds. They are motivated by time. Regular tasks often result from the collection of sensory input, calculation of control laws, planning of actions, and system monitoring. The application criteria may be used to determine the specific rates at which such activities must be cycled through. Since each instance of a periodic task must finish execution before the next instance is released, periodic tasks have strict deadlines. If not, task instances will accumulate.

Ad hoc duties. Aperiodic jobs are singular actions. They are motivated by events. For instance, when the cruise control system is on, the driver may alter the vehicle's cruising speed. The system periodically receives a speed signal from a spinning driveshaft, speedometer cable, wheel speed sensor, or internal speed pulses created electronically by the vehicle and pulls the throttle cable using a solenoid as necessary to maintain the speed set by the driver. When a user manually adjusts the speed, the system must adapt to the change while continuing to function normally. Tasks that are aperiodic either have no deadlines or have flexible deadlines.

Random tasks. Events are also used to drive sporadic tasks. While the minimal interatrial time must be met, the arrival timings of sporadic task occurrences are not known in advance. Sporadic tasks have strict deadlines, in contrast to aperiodic jobs that do not. For instance, the speed control system must react quickly to the event (a strong foot on the break) when a driver of a vehicle notices a hazardous scenario in front of him and applies the brakes to stop the car.

Task Specification

The following temporal parameters may be used to define tasks in real-time systems:

Release period. The moment a task is made available for execution is known as the release time. At any moment during or after the release time, the work may be scheduled to be completed. It may not be carried out right away if, for instance, a job with a greater or equivalent priority is already occupying the processor. A task Ti's release time is indicated by the symbol ri.

Deadline. The point in time by which a task's execution must be finished is known as the deadline. di indicates the time limit of Ti.

Relative time frame. A task's relative deadline is the one calculated in relation to the time it will be released. A task's relative deadline, for instance, is 200 milliseconds if it is released at time t and has a deadline of t + 200 milliseconds. Di indicates the relative deadline of Ti.

Elapsed time. When a job is executed independently with all necessary resources available, its execution time is the length of time needed to finish the task. The intricacy of a job and the processing speed are the two key factors that affect how long it takes to complete. ei stands for the execution interval of Ti.

Time to respond. A task's reaction time is the amount of time that passes between when it is released and when its execution is complete. The greatest permitted response time for a job with a hard deadline is the task's relative deadline.

Period. A periodic task's period is the distance between the release timings of two successive occurrences of the job. We suppose that throughout the book, every interval is the same size. Pi stands for the time of Ti.

Phase. The first instance's release time is the phase of a periodic job. Usage. Ti's phase is represented by i. A periodic task's usage, ui, is equal to the ratio of its execution time to its period, or ei/pi. The task execution time, release time, deadline, relative deadline, and reaction time all relate to the occurrences of a periodic task. We assume that throughout the book, every occurrence of a periodic job has the same execution time. We outline an ongoing job as follows:

$$Ti = (ei.Di, I\ pi)$$

For instance, a task with the parameters (2, 10, 3, 9) would release its first instance at time 2, followed by instances at times 12, 22, and so on. Each instance takes 3 units of time to execute. An instance must be performed nine units after it is released. The job is specified with Ti = if the phase is 0. (pi, ei.Di).

If the task's period and relative deadline coincide, we just supply two parameters:

$$Ti = (pi, ei)\ (pi, ei).$$

We can determine their hyperperiod, indicated by H, given a collection of periodic tasks Ti, where I = 1, 2,..., n. When I = 1, 2,..., n, H is the least common multiple (LCM) of pi. Prime factorization is one method of calculating H. This method is based on the idea that there is only one possible way to express any positive integer bigger than 1 as a product of prime integers. The prime numbers may be thought of as the building blocks that, when put together, form a composite number.

In order to reuse the same schedule for subsequent hyper periods, we just need to calculate the schedule for the first hyper period of a collection of periodic activities. Tasks include functional attributes in addition to time factors, which are crucial for task scheduling.

Criticality. In a system, not all tasks are equal in importance. The nature of the tasks themselves and the condition of the regulated process at the time determine their respective priority. The importance of a task in relation to other tasks is indicated by its priority.

Preemptively. The carrying out of duties may be interspersed. The scheduler has the option to pause a task's execution in favor of another that is more urgent. After the more important work is finished, the suspended task may begin again. Preemption is the term used to describe this kind of job interruption. If a job can continue running after being interrupted, it is preempt- able. To put it another way, there is no need to start again. A CPU processing work serves as an illustration. On the other hand, a work is not preemptable if it must be carried out uninterruptedly from beginning to end. They must be restarted from the beginning if they are stopped in the midst of execution. Partially anticipatory tasks are possible. The crucial component of a job is nonpreemptable, while the remainder of the work may be, for instance, if it just uses common resources.

Job States

A job may be in one of the following three states in real-time systems:

Running. A task is considered to be in the running state while it is actively working. The processor is now being used by it. Just one job may be active at any one moment in a single-processor system.

Ready. A job that is capable of running but is not doing so right now is said to be in the ready state if another task with an equivalent or higher priority is already running. Apart for the processor, ready jobs are equipped with all necessary resources. When a job in this state gains the greatest priority among all other tasks in this state and the processor is freed, it may begin to execute. The status of any number of tasks is possible.

Blocked. When a job is awaiting either a temporal or an external event, it is said to be in the blocked state. For instance, if a task uses taskDelay(), it will stop itself until the timer's expiry, which is a temporal event, has occurred. When a user enters anything, that is an external event, the task in charge of processing user inputs does nothing. Moreover, tasks may pause while they await RTOS kernel object events. The blocked state prevents the scheduling of tasks. Other tasks may also be in this stage. A new task is added to the ready state queue when it is generated. Depending on its importance and the priority of other jobs that are in the ready state, the task may be scheduled for execution right away or later. In this stage, all tasks must be finished for the CPU to operate. A task moves into the running state when it is sent for execution on the processor with the greatest priority. A job in the running state may be preempted by a task with a higher priority if it is pre-emptible and the scheduler is priority-preemption based. The RTOS kernel adds it to the ready queue when it is preempted. A task that is currently running may also transition to the blocked state for a number of reasons. Now, let's discuss the blocking scenario that results in priority inversion. Imagine that two tasks, A and B, share a memory space that is intended to be utilized just for these two processes. A is given precedence over B. A is released and preempts B while B is active and using shared memory. A is prevented from executing its code when it tries to access shared memory because shared memory is not accessible. Now B is sent out to run in the ready condition. Here, a lower priority activity is being carried out while a higher priority task waits.

When a job cannot be completed because other than the processor, it is blocked. The needed resources and the time delay are two examples of the criteria. The job will be set to ready when all the requirements are satisfied. In the case of priority inversion, task B will notify the RTOS kernel when it leaves the shared memory access. Right away, task A will take control of job B and continue working on it. The changes in a task's status. Take note of the additional states for jobs that various RTOS kernels defined. Five task states, such as running, ready, waiting, suspended, and waiting-suspended, are defined by the T-Kernel RTOS, for instance. A job in VxWorks may also be in the pending, suspended, delayed, or a mix of these stages in addition to the running and ready states.

Precedence Constraints

In a real-time application, tasks may be subject to priority restrictions in addition to timing restrictions. A precedence constraint specifies the order in which two or more tasks must be completed. It reflects dependencies between jobs in terms of data and/or control. For instance, in

a real-time control system, the calculation of the control laws must wait for the outcomes of the polling of sensor data. As a result, in each control cycle, the job for fetching sensor data shall execute before the process for computing control laws. We utilize A B to describe the relationship between tasks A and B in terms of their immediate priority. To make the precedence restrictions among a group of activities more understandable, we may utilize a task graph. Tasks make up the nodes of the graph, and an edge that is directed from node A to node B signifies that task A is the job that came before task B.

Assignment of Tasks and Scheduling

The tasks are allocated or assigned to the different processors and scheduled on them after taking into account all task timing factors and functional characteristics. Heuristic methods are often used since multiple processor scheduling and assignment issues are NP-complete. The method of creating a multiprocessor schedule typically involves two steps: first, tasks are allocated to processors, and then tasks are scheduled individually for each processor. If one or more of the schedules are impractical, we either rerun the job distribution or apply a different scheduling technique. There may not be a workable assignment or timeline for a certain issue. Take note that this chapter's scheduling methods are entirely uniprocessor-based. Techniques for task assignment are covered later in this chapter. Every work is carried out within a certain setting. On a CPU, only one job from the program may be running at any one moment. An assignment of tasks to available processors is called a schedule. An RTOS's scheduler is a module that carries out the scheduling (assignment) algorithms.

A schedule is considered to be legitimate if all priority and resource use restrictions are satisfied, and no job is either under- or overscheduled (given insufficient time to complete) (the task is assigned more time than needed for execution). If every job on the timetable is completed by the due date, the schedule is considered to be practicable. If there is a workable timetable for a system (a collection of tasks), the system is said to be schedulable. Finding workable schedules makes up the majority of real-time scheduling labor. If an algorithm consistently generates a workable schedule for any collection of activities, it is considered to be optimum. We evaluate a scheduling algorithm's effectiveness using schedulable utilization (SU). The SU is the highest overall utilization of jobs that the algorithm can safely arrange. The scheduling method performs better the greater the SU. A timetable might be calculated beforehand or discovered dynamically while the system is running.

Time-Based Scheduling

In clock-driven scheduling, choices are made on scheduling at certain time instants that are normally selected in advance before the system starts working. It often works for deterministic systems when all task parameters remain constant during system operation and tasks have strict deadlines. It is possible to construct clock-driven schedules off-line and store them for use at runtime, which significantly reduces runtime scheduling overhead. If a realistic schedule for a group of periodic activities with specified parameters already exists, drawing one for the clock-driven technique is relatively simple. Typically, one may use this method to create a variety of workable timetables. The user must decide which option meets the best criteria before making their choice.

The system of three periodic tasks has two schedules that are workable: T1 = (4, 1), T2 = (6, 1), and T3 = (12, 2).The schedules are in the system's first hyper-period, which is the range of 0 to

12 units of time. There are three examples from T1, two instances from T2, and one instance from T3 throughout this time frame. Let's examine the first schedule and determine its viability. Think about T1 first. T1 releases its first instance at time 0 with a deadline of time 4. It will start to run at time 0 and end at time 1, respectively. It therefore fulfills the deadline. The second instance of T1 is issued at time 4, with an expiration date of time 8. It will begin to be carried out from time 4, and end at time 5. The deadline has reached. With a deadline of time 12, the third instance of T1 is issued at time 8. It will begin to be carried out from time 8, and end at time 9. The deadline has reached. Let's now examine T2. At time 0, the first instance of T2 is made available. It runs out at time six. Time 1 is designated for execution, while Time 2 is designated for completion. It therefore fulfills the deadline. The second T2 instance is issued at time 6, with a deadline of time 12. It will begin to be carried out at time 6, and end at time 7. The deadline has reached. The lone instance in T3 has a deadline of time 12 and is released at time 0. It will begin to be carried out from time 2, and end at time 4. It therefore fulfills the deadline.

The timetable is achievable since every task instance that was released during the system's first hyperperiod met its deadline. It will be determined by a similar study that the second timetable is likewise doable. In reality, by rearranging the job execution slots a little, one may get different workable schedules. It is crucial that you take into account every circumstance within the time frame and ensure that each one meets its deadline. Why we only drew the timetable up to time 12 may have puzzled you. The hyper period for these three activities is twelve therefore that explains the situation. In clock-driven scheduling, a group of tasks' main cycle is sometimes referred to as its hyper period. It can be shown that each task's main cycle is always its LCM, independent of the task's phase. All scheduling choices in a timetable may be listed using a schedule table.

CHAPTER 13

STRUCTURED CLOCK-DRIVEN SCHEDULING

Shweta Gupta, Associate Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-shwetagupta832000@gmail.com

The schedules in Example are workable, but the decision points are dispersed at random. In other words, the times at which new tasks are chosen for execution do not follow any pattern. With a structured clock-driven scheduling technique, choices about scheduling are made at regular intervals rather than at random periods. In this manner, a timer with a predetermined duration that expires on a regular basis might be used for decision times.

Frames

The scheduling decision periods in a structured timetable divide the time line into units called frames. Each frame has a length f, also known as frame size. Due to the fact that scheduling choices are only determined at the start of each frame, preemption does not occur inside a frame. Each periodic task's phase is a nonnegative multiple of the frame size to facilitate the creation of the schedule. In other words, each periodic duty is released for the first time at the start of a frame. Just now, we said there shouldn't be any preemption inside of a frame (otherwise, it is not a structured schedule). Preemption should be avoided since it entails context shifts. As a result, the frame size we choose should be sufficient for each activity to be completed inside the frame. Imagine that a system has n periodic jobs. This restriction may be mathematically represented as

maxf ei, I = 1, 2,..., n

The selected frame size should split the main size in order to reduce the number of entries in a schedule table and save storage space. Otherwise, since the schedule won't just repeat from one major cycle to the next, holding the schedule for just one major cycle won't be enough, necessitating the need for a bigger scheduling table. This restriction may be expressed as H mod f = 0.H is a multiple of each task's period, therefore if f can divide H, f must also divide at least one task's period in the system. Hence, pi mod f = 0, I 1, 2,..., n.

You should be aware that this restriction is crucial since most embedded devices have a limited amount of storage. We should make sure that every task instance can complete its deadline while selecting the frame size. This restriction mandates that there be at least one complete frame between the time a task instance arrives and its due date. Because it won't be scheduled until the start of the frame when a task instance arrives immediately after a frame begins.

Slice Your Tasks

There may not be a suitable frame size that satisfies all three conditions for a group of periodic jobs. This usually occurs as a result of one or more jobs having lengthy execution durations. In such instance, the first requirement and the third constraint may not agree. Slice one or more large jobs (those with longer execution durations) into a number of smaller ones as a possible solution to the problem. To demonstrate the concept, we provide an example.

Task Slicing Example

Considering the three periodic tasks T1 = (4, 1), T2 = (5, 1), and T3 = (10, 3),Finding a workable frame size for a cyclic scheduler is our goal. The first restriction calls for f 3. The second restriction limits the size of the frame to 4, 5, or 10. The three tasks resemble those in Example 4.4, as you can see. The execution of T3 was altered from 2 to 3 units of time, which is the sole difference. We may infer from the study in Example 4.4 that none of the frame sizes of 4, 5, or 10 is practical. As a result, there isn't a frame size that satisfies all three requirements. The answer is simple: T3 is a large chunk job that we have.

Aperiodic Tasks Scheduling

Aperiodic jobs are often the outcome of outside occurrences. There are no strict deadlines for them. Nonetheless, in order to decrease latency and boost system performance, it is preferable to serve aperiodic jobs as soon as feasible. Typically, aperiodic jobs are planned to run in the background of periodic tasks. Aperiodic activities are thus carried out during the processor's idle time intervals. The timetable, as an illustration. These periods of idleness are known as slacks. If they are waiting to be executed, they may be utilized to carry out aperiodic activities. Remember that for instances of periodic tasks, all we need to do is make sure their deadlines are fulfilled; there is no advantage to finishing their executions sooner. So long as they can still fulfill their due dates, we may attempt to postpone the execution of periodic task instances as much as feasible. In this manner, we may complete periodic duties as soon as feasible. This method is known as slack theft.

Planning Random Tasks

There is no way for the scheduler to ensure that sporadic jobs will complete by their hard deadlines since their parameters (release time, execution time, and deadline) are unknown in advance. When a sporadic job with a known deadline is released, it is standard procedure for the scheduler to assess whether it can release the work at a time when all periodic tasks and maybe other sporadic tasks have already been scheduled. The newly released sporadic task will be scheduled for execution if it successfully passes the test. If not, the scheduler rejects the job and notifies the system, allowing it to take the appropriate corrective action.

Round-Robin Methodology

An algorithm for time-sharing scheduling uses the round-robin method. A time slice is allotted to each work in the round-robin scheduling method. Tasks are carried out in any order. All jobs are kept in a FIFS (first-in-first-service) queue when they are ready. The work at the front of the line is always taken out of the queue and sent to be completed. The job is moved to the back of the FIFS queue to wait for its next turn if it is not completed within the designated time slice. The time slice is often brief, giving the impression that every action is executed practically instantly when it is ready. All tasks are presumed to be pre-emptable under this scheduling method since each time they are performed, they are only partially completed. This results in frequent context shifts. Implementing the round-robin scheduling method is easy. Using the CPU is equitable for all jobs. The main disadvantage is that it may result in tasks missing their due dates and delays the completion of all activities. It is a poor choice for planning projects with strict deadlines. Weighted round-robin is an enhanced variant of round-robin. With the weighted round-robin strategy, different jobs may have different weights, the fractions of processor time, as opposed to

all ready tasks receiving an equal portion of the processor time. We may speed up or slow down the execution of a job by changing the weights.

Algorithms for Priority-Driven Scheduling

A priority-driven scheduling algorithm makes scheduling choices when a new task (instance) is released or a job (instance) is finished, as opposed to clock-driven scheduling algorithms that schedule work at certain time points off-line. Decisions are made in real time since the schedule is online. Each job has a priority assigned to it. When the system is functioning, priority assignment may be done statically or dynamically. Static-priority or fixed-priority scheduling algorithms are described as prioritizing jobs statically, whereas dynamic-priority scheduling algorithms prioritise tasks dynamically. Implementing priority-driven scheduling is simple. It doesn't need to know in advance when tasks will be released and when they will be completed. Only once a job is released do the parameters of that task become known to the online scheduler. The only choice for a system whose future workload is uncertain is online scheduling.

Automata with Fixed Priorities

The rate-monotonic (RM) algorithm is the most well-known fixed-priority algorithm. Based on the duration of the jobs, the algorithm assigns priority. If pi is greater than pj and there are two jobs, Ti and Tj, Ti has higher priority than Tj. The RM technique makes it very simple to schedule periodic tasks: when a new task instance is released, if the processor is idle, it runs the task; otherwise, the scheduler compares the tasks' priorities. If the new task has a higher priority, it preempts the one that is now being executed and begins to run on the processor. The preempted work is added to the ready task queue.

The Planning of Recurring Tasks

Tasks that are aperiodic either have flexible deadlines or none at all. The easiest algorithm for scheduling aperiodic jobs is to run them in the background of periodic tasks, or during the slow periods of the periodic task schedule. Aperiodic jobs are given the lowest priority by the algorithm. As a result, periodic jobs' scheduleability won't be impacted, but there is no assurance that aperiodic tasks' response times will be met. The slack stealing approach may be used to enhance the scheduler's performance with relation to responding to irregular jobs. According to the logic underlying slack-stealing, the goal of scheduling periodic tasks is to fulfill their due dates; there is no advantage to finishing their execution sooner. As long as their deadlines are satisfied, we may postpone their implementation as much as feasible. By doing so, we can carry out aperiodic activities as soon as feasible. Since a clock-driven schedule is generated off-line and we know precisely when the processor is available for aperiodic activities, how much slack time there is in each frame, and the maximum that a schedule may be moved, slack stealing can be performed with ease. Slack theft is substantially more challenging to implement in a priority-driven scheduling system since runtime scheduling choices must be made. Polling is a more widely used method for planning aperiodic jobs. This method involves adding a polling server, also known as a poller, to the system as a periodic task Ts = (ps, es), where ps is the polling period and es is the job's execution duration. The polling server is given relative priority depending on its polling interval by the scheduler, which regards it as another periodic job. The poller examines the aperiodic task queue while it runs. The poller runs the job first in the queue if it is backlogged. The poller waits for the next cycle after ceasing operation after having ran for

es units of time or until the aperiodic job it was doing has completed. But, if there are no aperiodic jobs available for execution at the start of a polling session, the poller suspends itself right away. It won't be prepared for use until the subsequent polling interval. In accordance with this polling policy, an aperiodic job must wait until the start of the subsequent polling period before being taken into account for execution. a postponeable server The bandwidth-preserving polling server Tds = (ps, es) maintains the server's execution time or budget until it is depleted by carrying out aperiodic activities during the current period or expires at the end of the period. In this approach, if the server's priority allows, an aperiodic job that enters an empty aperiodic task queue after the start of a polling period may still be completed inside the period. Deferrable servers have a straightforward concept that speeds up the execution of aperiodic operations.

Planning for Random Tasks

Unpredictable assignments are released. There must be a minimum interarrival time between any two consecutive occurrences of a sporadic job even if it doesn't have a period. Otherwise, it would be difficult for a scheduler to ensure that every one of its instances would finish on time. One approach to dealing with sporadic jobs is to consider each one as a periodic task with a period equal to its shortest interval between arrivals. Another option is to use a deferrable server, which is how we now manage aperiodic jobs. An acceptance test should be carried out before a sporadic job is scheduled for execution since sporadic tasks have severe deadlines. It should be refused if it cannot be scheduled.

Practical Factors

The assumptions underlying the scheduling algorithms we have so far are that every task is preemptible at any time, that once a task instance is released, it never suspends itself, remaining ready for execution until it is finished, and that context switch overhead is insignificant in comparison to task execution time. Yet, these presumptions are not necessarily true in applications in the actual world. In this part, we go through how to assess a system's scheduleability when these practical considerations cannot be disregarded.

Non-preemptivity

A task or a specific section of a task could not be preemptible. One technique to prevent unbounded priority inversion, for instance, is to make a task nonpreemptable while it is in its crucial part. The next chapter will go into further depth on this. If preemption is too expensive, tasks are also designed to be unpreemptible. Consider a job that bundles and transmits a control signal to an external device as an example. The process of transmitting a control signal must restart if it is interrupted.

Self-Suspension

When a task waits for an external action, such as an I/O operation or Remote Procedure Call (RPC), to finish on another processor, the task suspends itself. When a task self-suspends, the scheduler assigns it to a blocked queue and removes it from the processor pool. In course, the tasks in their nonpreemptable sections may stop the blocked task when it attempts to reacquire the processor. The execution of lower priority tasks could be delayed if a task is suspended. The picture displays two tasks: T1 = (5, 2) and T2 = (2, 6, 2.1, 6).

Immediately after starting its execution, the first instance of T1 suspends itself for 3 units of time, delaying the first instance of T2's deadline until time 8. Suppose that we are aware of the maximum duration of external operation and that self-suspension has a finite time. The length of Ti's longest self-suspension is denoted by i. Indicate by bi(ss) the task's blocking time.

Contextual Changes

Each task has a context, which is the information describing the task's state throughout execution and kept in the task control block (TCB), a data structure that holds all the details important to the task's execution. When a task is chosen to run again, its context is recovered so that it may continue from where it left off when the job was switched out of the CPU. When a scheduler moves a task out of the CPU, its context must be saved. If there is no self-suspension, each job in a fixed-priority system preempts no more than one lower priority task. As a result, each task instance experiences two context switches: one when execution begins and one when it is finished. As a result, we may double the execution time for each job by include the context transition time. If CS is the greatest amount of time the system may spend on a single context transition, then if there is no self-suspension, we must add 2CS to the execution time of each job when analyzing the schedulability of a system. We increase the execution time of a job by 2(ki +1)CS if it can self-suspend up to ki times.

Assignment of Work

All of the scheduling strategies described in the sections before use a single processor. As a result of a single processor's limitations, many real-time embedded systems actually use several processors to function. A multiprocessor scheduling issue is often solved using uniprocessor scheduling, as was described before in this chapter, by allocating jobs in a system to each processor individually first. We present a few well-known job assignment techniques in this section.

Bin-Packing Methods

The most crucial aspects to take into account when assigning tasks are task execution durations, communication costs when jobs are allocated to different processors, and resource allocation. Bin-packing algorithms are focused on usage and do not take into account the cost of transmission. In certain real-time applications, processes distributed across several CPUs communicate through shared memory. They incur very little communication expense. Bin-packing algorithms work well for assigning tasks in these kinds of systems. In addition, even though communication and resource access costs may be significant, we may prefer to disregard them early in the system design process. Based on each task's complexity, we estimate its execution time and generally calculate its utilization. Next, we allocate tasks depending on each task's usage. The first stage in task assignment is determining the required number of processors. On the basis of the entire task usage, this may be done basically. Consider a system with n independent, preemptible, periodic jobs that are to be planned using the EDF algorithm, and whose relative deadlines are equal to their respective periods. Knowing the efficiency of each activity allows us to quickly calculate the system's overall efficiency, or U. We round up U to the next integer, let's say k, because the SU of the EDF is 1. The system's absolute minimum need for processors is k. We may divide the set of tasks into k groups and assign all the tasks in each group to the same processor in order to distribute the work among the processors. All we need to do is make sure that each group's overall task utilization does not exceed 1. Of fact, attempting to

use all of the processor time for recurring chores would not be a wise idea. We may set the threshold for processor usage to some value U′ 1 to reserve a certain amount of processor time for the execution of irregular and aperiodic operations.

It is trivial to frame these work assignment challenges as conventional bin-packing puzzles. Objects or things must be packed into a finite number of bins or containers of the same volume in a manner that minimizes the number of bins utilized in a bin-packing issue.

Access Restriction and Resource Sharing

All of the scheduling methods discussed in this Chapter work on the presumption that each job is independent of the others. Since tasks often have explicit or implicit dependencies among one another in real-time systems, we disprove this claim in this chapter. The task precedence graph may be used to specify explicit dependencies. Sharing information or resources puts implicit dependencies on the actions that use those resources. Several shared resources don't permit concurrent access. Due to possible priority inversions and even deadlocks, jobs that share resources may experience scheduling anomalies. The execution behavior and scheduleability of activities are discussed in this chapter along with how different resource access control mechanisms operate to lessen the unfavorable effects of resource sharing. Priority-driven and single-processor systems are our main interests.

Resource Exchange

Data structures, variables, main memory, files, registers, and I/O units are a few examples of common resources. A job could need additional resources in addition to a processor to advance. A computer work could, for instance, exchange data with other computational activities, with semaphores protecting the shared data. A resource is each semaphore. A task must lock the semaphore before entering the crucial code that allows it to access the shared data when it needs to access the shared data protected by a semaphore R. Throughout the length of the crucial phase, we state that the job needs the resource R in this instance. We only take serially reused resources into account. A resource that may be utilized securely by one job at a time without being exhausted is known as serially reusable. It is not an issue to access the resource at a later time if a job needing it is preempted. Devices, files, databases, and semaphores are a few examples of serially reused resources. Not all resources that are serially reused are preemptable. Examples of nonpreemptable resources are a CD and a cassette. A corrupted CD will arise from abruptly transferring the CD recorder from one process to another if that process has already started to burn a CD-ROM. It is not possible to preempt a job that is utilizing a nonpreemptable resource. To put it another way, once a job receives a unit of resource, that unit is not again made accessible to other tasks until that task releases it. If not, the resource may become inconsistent and cause a system failure.

Resource Access Control

In real-time systems, resource sharing can result in severe issues. To control who has access to shared resources, we need rules. To tackle priority inversion and deadlocks brought on by resource sharing, a number of well-known resource access control protocols have been created. When and under what circumstances each request for a resource is allowed, as well as how activities needing resources are planned, are determined by a resource access control protocol. Deadlocks can be avoided by using an access control protocol that is well-designed. Priority

inversion cannot be eliminated, however, by any protocol. Access control should aim to minimize the amount of time that higher priority actions are blocked.

Critical Section Protocol with No Preemption

As we previously said, constrained priority inversion won't generally harm an application as long as the crucial portion of the lower priority job runs without delay. Unbounded priority inversion, in which a medium-priority job preempts the lower priority task while the latter is working on its crucial phase, is the true problem. Making all crucial portions non-preemptable is a straightforward method to avoid unbounded priority inversion. To put it another way, a task that locks a resource operates at a priority greater than all other tasks' priorities until it unlocks the resource (or completes the execution of its critical section). Non-preemptive Critical Section (NPCS) Protocol is the name of this protocol. Circular waiting is impossible with this protocol since no job can be preempted while it is holding a resource. A deadlock is thus not feasible.

With the help of Mars Pathfinder, prioritize

The events surrounding the Mars Pathfinder mission in July 1997 are a well-known illustration of priority inversion. The small rover from the Pathfinder mission, which sent back high-definition color images of the Martian landscape, is what most people remember it for. The mission software used on the Martian surface was the source of the issue. Through a MIL-STD-1553 data channel, multiple spaceship components talked with one another. Two high-priority jobs were in charge of controlling activity on this bus. Using a conduit, a bus management task and a low-importance job for meteorological research (task ASI/MET) might interact. On Earth, the program largely functioned without any problems. But, an issue on Mars became severe enough to need a number of software resets during the mission. The low-priority scientific work was preempted by a few medium-priority tasks while it was holding a mutex associated to the pipe, which started the chain of events leading to each reset. The high-priority bus distribution manager (task bc dist) attempted to deliver additional data to the low-priority job over the same pipe while it was being preempted. The bus distribution manager had to wait since the scientific job was still in control of the mutex. The other bus scheduler soon started operating. It caused a system reset after determining that the distribution manager hadn't finished its task for that bus cycle.

Protocol for Priority Inheritance

The Priority Inheritance Protocol is another straightforward protocol that prevents unbounded priority inversion. The priority of the blocke[1]d higher priority task is inherited by the lower priority task under this protocol when it blocks the higher priority task. In this manner, the lower priority task can swiftly finish the execution of its crucial section. Any task with a priority between the two cannot preempt the lower priority task because of priority inheritance. Unbounded priority inversion is thus prevented.

Multiple tasks that are waiting for a resource may be blocked by a task that is holding it. In that case, the blocking task inherits the highest priority since the most recent blocked task must have it. The task returns to its original priority level once its critical section has been completed and the resource has been released. We refer to the priority that is assigned to a task in accordance

with a scheduling algorithm (for example, rate-monotonic) as its assigned priority because the priority of a task is modifiable under the priority inheritance protocol. The assigned priority of a task is a fixed value in a fixed-priority scheduling system. A task may be scheduled at a priority that is different from its assigned priority because a task can inherit the priority of other tasks. We can define priority as the task's current priority.

Protocol for Priority Inheritance Rules

The priority inheritance protocol is governed by the following three rules:

Planning Guideline: Ready tasks are preemptively scheduled on the processor based on their current priorities. Unless it is in its critical section and prevents higher priority tasks from being completed, a task's assigned priority is its current priority.

CHAPTER 14

PROPERTIES OF PRIORITY INHERITANCE PROTOCOL

Ryan Dias, Assistant Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-ryan.dias@jainuniversity.ac.in

We previously noted that the NPCS protocol can prevent both deadlocks and unlimited priority inversion. Priority inheritance protocol's ability to prevent unlimited priority inversion has previously been defended. The obvious issue is: Can deadlocks also be removed by the priority inheritance protocol? Sadly, the answer is no. The answer is straightforward: it is unable to prevent resource waiting in circles.

Have a look at the impassed schedule. There are no rules of the priority inheritance protocol that are broken by this schedule. Thus, the priority inheritance protocol is ineffective in avoiding deadlocks. To further demonstrate this truth, we utilize another example. The higher priority job is unblocked and allowed to run when the lower priority activity has finished running its crucial phase and released the resource. Even if the lower priority job has many important portions, it cannot be executed until the higher priority task has finished running.

Protocol for Priority Ceilings

In order to avoid deadlocks and shorten blocking times, the priority ceiling protocol is an upgrade on the priority inheritance protocol. This protocol makes the assumption that each task's resource requirements are known before any task execution begins. Each resource has a priority ceiling assigned to it, which is the highest priority job that might possibly consume that resource. As is well knowledge, anytime a resource request is made in the priority inheritance protocol, the resource is provided to the requesting task if it is available.

Even if the resource is free, such a request under the priority ceiling protocol could or might not be approved. More specifically, the protocol assures that a lock is successful only if the priority ceiling of the new resource is greater than that of any preempted resources when a task preempts the critical section of another task and locks a new resource.

Priority Ceiling Protocol Guidelines

The priority ceiling protocol's scheduling, resource distribution, and priority inheritance guidelines are as follows:

Planning Guideline: Ready tasks are proactively scheduled on the processor based on their current priority. Unless a job is in its critical section and blocks higher priority tasks, its assigned priority equals the task's current priority.

Assigning Rule: If a resource R is locked by another job at the moment t when a task T locks it, L(R) is rejected and T is halted.

If T's current priority is greater than t and R is free, T is given R, and the lock L(R) is successful.

If T is the task holding the resource(s) whose priority limit is equal to (t), R is only allotted to T if T is the task with a current priority of lower than or equal to (t). If not, the lock fails and T is blocked.

Priority Inheritance Rule: Whenever a lower priority job obstructs a higher priority task, the blocked higher priority task's current priority is carried over to the lower priority task until the blocked higher priority task has finished executing its essential portion and released the resource. After then, it goes back to the original priority. The scheduling rule and the priority inheritance rule are identical to those in the priority inheritance protocol, as you can see. The allocation rule is the only factor that differs. Let's use an example to demonstrate how the protocol works before we go through the advantages of this new rule.

The Priority Ceiling Protocol's characteristics

The priority ceiling protocol may prevent unbounded priority inversion, which is a trait that the priority inheritance protocol also shares.

The worst time to block

Direct blocking, priority inheritance blocking, and priority ceiling blocking are the three forms of blocking that may occur when resource accesses are governed by the priority inheritance protocol. When a job with a higher priority demands a resource that is being used by a task with a lower priority, direct blocking occurs. The length of the relevant critical part of the lower priority job is the maximum blocking time. The higher priority job's priority is inherited by the lower priority task when it blocks the higher priority process. As a result, any tasks whose priorities fall between the two tasks' allocated priorities are blocked by priority-inheritance. The length of the relevant critical part of the lower priority job is the maximum blocking time. Any higher priority job that does not have a priority greater than the priority ceiling of R is priority-ceiling prevented from successfully locking further resources once a lower priority task successfully locks resource R. The length of the relevant critical part of the lower priority job is the maximum blocking time.

Observe that only when activities seek resources do direct blocking and priority ceiling blocking take place for tasks that do not consume resources. A job can only be stopped for the length of one crucial part, even if it conflicts with other activities on many resources. The worst-case blocking time of a job under the priority ceiling protocol is then computed using an example.

Protocol for Stack-Sharing Priority Ceiling

The priority ceiling protocol is made simpler by the stack-sharing priority ceiling protocol, which offers stack-sharing functionality. The worst-case blocking time of a task under this protocol is identical to the worst-case blocking time under the priority ceiling protocol, however this approach has less context transition cost. The stack is shared across jobs, which prevents stack space fragmentation and conserves memory. Each job typically has a separate runtime stack where its local variables and return address are kept. When a job is created, memory is automatically allocated for it to utilize. To lower the overall memory demand when there are too many processes in a system, it may be required for numerous tasks to share a single runtime stack, as shown in.

Stack-Sharing Priority Ceiling Protocol Guidelines

When tasks share a runtime stack, the task at the top of the stack is the one that is executed. After a job is finished, the space used for it is released. A job Tj is given the space above another task Ti when one preempts the other. A preemption job may only be resumed when it is back at the top of the stack, meaning that all tasks occupying space above it have been finished. It goes without saying that such a space-sharing approach prohibits any type of blockage.

The Stack-Sharing Priority Ceiling Protocol's characteristics

The stack-sharing priority ceiling protocol is an upgraded version of the basic priority ceiling protocol that was discussed in the preceding section. It has each benefit provided by the priority ceiling protocol: no chained blocking, no deadlocks, and no unbounded priority inversion. The stack-sharing priority ceiling protocol states that blockings may only happen when jobs are released. A job can only be preempted after it has begun running; it cannot be blocked before it is finished. No task ever experiences more than two context transitions as a result. A job that wants to access resources may experience four context flips under the priority ceiling protocol, two of which are brought on by preemption and two by blocking. The stack-sharing priority ceiling protocol may be modified for non-stack-sharing systems even though it was designed with stack sharing in mind. The following are the guidelines for the revised protocol:

Schedule Regulations: If a job does not have resources, it is completed at the priority that was given to it. The same-priority tasks are scheduled in FIFO order. Each task holding resources has a priority that is the highest among all priority ceilings for those resources. Assigning Rule: When a task requests a resource, the resource is given to the task.

Parallel Programming

The theory and ideas of real-time tasks, task scheduling, and resource access control were explored in the earlier chapters. This chapter moves to real-time embedded software implementation, with an emphasis on the synchronization and communication protocols between systems. The Code:: program tests each programming example provided in this chapter. Blocks Ubuntu's Integrated Development Environment.

Potential parallelism may be expressed using concurrent programming. It separates a larger calculation into smaller computations that may run simultaneously, i.e., at the same time as other computations. Almost all real-time systems are inherently concurrent because in the actual world, devices work in parallel. For instance, a washing machine must routinely do many tasks at once: taking water, checking the water level and temperature, scheduling a wash cycle, dispensing detergent, managing the agitator, rotating the tub, and so on. A thermostat is always monitoring the room temperature, the timer, and polling the keypad, at least three different jobs. Thus using concurrency to program the controllers of these devices makes sense. Both programmers and consumers of applications may benefit greatly from concurrent programming. First of all, it makes an application more responsive. While several costly computations are being run simultaneously, the system seems to reply to every user request right away. Moreover, it raises CPU usage. Every time a job is prepared to run, other tasks compete for the processor's attention and keep it occupied. Other tasks may take over if one gets stuck. Also, it offers a useful framework for failure isolation. You may be wondering whether we can manage concurrent processes using sequential programming approaches. To manage the numerous

concurrent operations that are fundamentally unrelated to one another, you must design the system, if you decide to do so, in a way that requires the cyclic execution of program sequences. The resultant programs will be less clear and eloquent, which will complicate the problem's decomposition.

It is important to note that concurrent programming and parallel programming are two ideas that are similar but not the same. Concurrent programming aims to handle problem-solving complexity by allowing jobs to multiplex their executions on a single processor. Yet, with concurrent computing, only one job is active at any one moment. In contrast, parallel computing physically does many tasks at the same time, for instance, on various processors of a multiprocessor system, with the aim of accelerating calculations. A single processor can only handle one calculation at once, hence parallel computing is not conceivable.

Threads in POSIX

The example of POSIX threads, or Pthreads, is used to demonstrate the problems with concurrent programming and how to solve them. For real-time embedded systems, POSIX offers a collection of industry-standard application programming interfaces (APIs). While many RTOS providers have their own proprietary programming interfaces, POSIX conformity is necessary to allow applications to work with a variety of hardware and operating systems. We are not going into great depth on Pthreads since this chapter's objective is not to educate Pthreads. The only routines presented are those that are required to briefly demonstrate the ideas of concurrent programming. The programs provided in this chapter should be understandable to readers with C, C++, or Java programming knowledge. Pthreads are described as a collection of C language programming types and method calls that are implemented using a thread library and the header file pthread.h. For thread management, Mutexes, conditional variables, and thread synchronization with read/write locks and barriers, Pthreads API methods are offered. The new thread's opaque and specific identifier is contained in the routine's first parameter thread. The attributes for the new thread are chosen from the contents of the pthread attr t structure that the attar argument corresponds to. Arg is supplied as the only parameter to start routine(), which is called by the new thread to begin execution (). The call returns 0 in case of success and an error number in case of failure.

Synchronization Primitives

Concurrent applications should be carefully written so that processes or threads may communicate with one another without interfering with one another or causing problems. They often have to adhere to rules on how their executions should be spaced apart. Due to race situation behavior, this form of synchronization is required. The majority of RTOS kernels include synchronization objects to aid programmers in creating synchronization in order to effectively prevent multiple jobs from accessing shared data. The most well-known kernel objects are semaphores, condition variables, and mutexes. The principles of race conditions and critical sections are first explained in this part, after which the different approaches to process synchronization are presented.

Crucial Parts and Racial Conditions

Several processes or threads are active at once while computing concurrently. Programmers often have little control over how often processes are switched. Preemption or swapping is

managed by the operating system's scheduler, not the programmer. After receiving any command, every process may be halted. When two or more processes interact using shared data and the result is dependent on the precise order in which the processes execute their instructions, this is known as a race situation. Imagine an ATM application with two processes: P1 and P2, which credit and debit money from respective accounts respectively. The three following actions are carried out consecutively by each process: reads an account's balance, changes the balance, and returns the balance in writing. Let's say a couple's joint account has a balance of $1,000. The woman is taking out $200 from the account at one ATM while the husband is adding $200 at another ATM:

Communal Memory

A low-level method for activities to interact with one another is via shared memory. By putting it on memory pages shared by many jobs, data is communicated. All of the activities involved map shared memory into their own address areas. Shared memory changes are instantly apparent to other jobs when one puts a value into a specific byte of the memory pool. As a result, shared memory is a faster inter-task communication method than message queues since data transmission via it is not handled by the kernel.

This is due to the fact that under the message queue technique, senders must transfer messages from local memory into kernel space, while receivers must copy messages into local memory from kernel. Of course, additional safeguards must be employed to prevent access to the shared data, such as mutexes, conditional variables, and/or semaphores. We can store any kind of data structure in a shared memory space, just as with message queues. Files are used to implement POSIX shared memory objects. The objects are persistent in the kernel, meaning they remain in place until they are specifically destroyed or the system is restarted. Opening a shared memory object is required before using the generated descriptor to map the object into a task's address space to create a shared memory area. The procedure used to establish a new shared memory object or open an existing one is

The routine's first parameter name is a reference to the shared memory object. The construction requirements for a shared memory object name must be the same as those for a typical file path name, which must begin with a "/" and cannot include further "/."

The shared memory object's opening is controlled by the second parameter oflags. O RDONLY is for read-only and O RDWR stands for read and write. Shared memory does not support write-only. When the flag is OR'd with O CREATE, a shared memory object creation method is called. The final parameter mode is only essential when O CREATE is being used. Moreover, additional flags may be OR'd with it. If the object referred by name already exists and O CREATE is set, but O EXCL is not set in the shm open() call, the call will simply return the descriptor of the existing object. For instance, you may provide O EXCL to change the behavior of O CREATE. Both flags must be unset for the call to succeed. The shm open () call's return value, the file descriptor fd, is used as a reference to an area for which a virtual memory mapping is created. The portion of shared memory starting at offset offset and ending at length length will be translated into the process' virtual address space, which is determined by addr. The access mode specified in the shm open () procedure must be consistent with the memory protection mode specified by the third parameter, prot. The single possible value for the parameter flags is MAP SHARED, which makes changes made to the mapped memory by the caller accessible to other processes mapping the same object.

Sharing Memory Security

To prevent one process from impeding another's progress in the shared memory, processes that access it must be synchronized. The synchronization between threads of a single process is the sole situation where the mutex, condition variable, and semaphore techniques described previously in this chapter are appropriate. Using named semaphores is one method for synchronizing data exchange across processes.

As with message queues, named semaphores adhere to all name rules. By passing the semaphore's name and common flags as parameters to the sampan function, we may open an existing semaphore:

Real-Time Resources

The bulk of jobs in a real-time system are recurring ones. We need a reliable method of keeping track of the passing of time in order to carry out these activities. In order to prevent job overruns, this is also crucial. We present commonly used real-time signals and timers in this section. We also go through how to build recurring activities using these kernel features.

Real-Time Indications

Signals are a crucial component of the multitasking of several real-time kernels, much as mutexes, conditional variables, semaphores, message queues, and shared memory. Signals may be used for many different things, including managing exceptions, stopping processes in unusual situations, and even intertask communication. In this section, we concentrate on timer expiry as an example of an asynchronous event that should be processed for notification. The software counterpart of an interrupt is a POSIX signal.

A signal is an asynchronous notification that is delivered to a process or to a particular thread within a process to let it know that an event has occurred. Asynchronous in this context denotes that the event may happen at any moment and may not be tied to how the process is being carried out. An example is the CTRL+C key combination. The kernel, a terminal driver, or other processes may send out signals. For instance, the Unix command $ kill -KILL 1234 causes a process with the ID 1234 to get a SIGKILL signal. Numbers are used to identify signals. A list of signal numbers is supported by each POSIX compatible system. Typically, the head file signal's defines symbolic names for signals. Every signal number has a distinct significance and impact on the process that receives it. For example, if you click CTRL+C, a signal of SIGINT from the OS is produced. The OS becomes aware of an unauthorized memory reference made by a process, which causes it to instantly suspend the application process and send out a SIGSEGV signal. The default SIGSEGV signal handler will receive the signal and output an error message before terminating the process.

Via the use of the kill () or sigqueue() calls, user processes may communicate with other processes. A process has three options once it receives a signal: to handle, block, or ignore it. Nevertheless, certain signals, like SIGKILL (which ends a process) and SIGSTOP (which pauses a process), must be paid attention to. There is no server process to deliver the signals when real-time signals are produced by a POSIX timer, when asynchronous I/O completes, or when a message arrives on an empty message queue. Instead, the data structure sigevent is used to specify how signals should be provided as part of the startup of the timer, the asynchronous I/O, or the message queue.

Residual Number System Overview

Energy economy and quick calculation are required by today's embedded devices and wireless applications. From the transistor level to the level of the architecture, significant advancements have been achieved in many parts of electronic systems in this approach. Long carry-propagation chains continue to be a problem for arithmetic level computations, nevertheless. The switch from the conventional weighted two's complement number system to an effective alternative number system can have a significant impact on the performance of computation systems, even though many computer arithmetic methods, such as parallel-prefix computations, have been introduced to reduce delay. For more than 50 years, the residue number system (RNS) has been the most intriguing and difficult alternative number system in computer arithmetic. The capacity to do addition, subtraction, and multiplication without carry-propagation across residues is its most intriguing property. This extraordinary skill makes the goal of the computer arithmetic researcher—carry-free arithmetic—possible. To perform division, residue to binary conversion, sign detection, magnitude comparison, and overflow detection effectively, there are various challenges. RNS has thus mostly been employed in fields like digital signal processing and encryption where addition, subtraction, and multiplication are the major operations. RNS is also ideal for other types of applications, such as computer networks, DNA arithmetic, and cloud storage. These other intriguing qualities include error detection/correction and fault-tolerant capabilities. Many improvements have been made recently in RNS-targeted design and software. Defining the dynamic range to cover, or the range of numbers that must be handled, is the first step in designing an RNS-based digital system. Finally, based on that dynamic range and the kind of applications that are being targeted, an effective moduli set should be chosen. RNS must be connected to the weighted representation of numbers via a forward converter. The residues are then subjected to concurrent calculations using separate modulo arithmetic channels. The resultant RNS number is then converted by a reverse converter into a weighted representation that is employed by traditional digital systems. A sign detector, a magnitude comparator, and a scaler are examples of optional RNS components that may be utilized in the RNS datapath depending on the needs of the application. The remainder of this chapter will highlight current work and developments while introducing several approaches for developing all RNS sections and components. Also, a step-by-step RNS teaching approach is presented for the first time, integrating RNS's mathematical foundation with hardware architectures, implementations, and applications. This kind of instruction may make it easier for electrical and computer engineering researchers and students to comprehend RNS principles and designs and inspire them to look into new RNS architectures and use them in more real-world settings.

RNS Organization

Selecting an appropriate moduli set is the first stage in developing an RNS system, taking into account the dynamic range (the product of all moduli) and the level of parallelism necessary (corresponding to the number of moduli in the set). The pair-wise relatively prime numbers are included in the moduli set. The two primary categories of RNS moduli sets are arithmetic-friendly and conversion-friendly moduli sets. Contrary to arithmetic-friendly sets, conversion-friendly moduli sets provide simpler reverse converter structures. Arithmetic-friendly should be used if the target application's internal arithmetic operations to conversions ratio is high, and vice versa. You may find some tips and relevant information regarding the sample moduli sets and the technique of selection here. The most popular and effective moduli are the 2n and 2n 1 moduli because they make it possible to create simple hardware arithmetic units. Several specific moduli

sets have been created for RNS based on these well-known integers. The conventional moduli set 2n 1, 2n, and 2n 1 is the most well-known. While this three moduli set has received the bulk of attention in RNS research, the lack of available parallelism prohibits its use in modern high-performance computer systems. Hence, expanded RNS moduli sets have been suggested, including 2n 1, 2n, 2n 1, 2nC1 1, and 2n 1, 2n, 2n 1, 2n—1 1, 2nC1 1. It should be noted that general RNS moduli sets exist in addition to specific RNS moduli sets, allowing the designer to favorably choose each modulo. For instance, in prime moduli sets, where all the moduli are prime, modulo multiplications may be converted to ads based on an isomorphism. Keep in mind that bigger moduli sets are often taken into account for cryptography applications since they work on extremely large integers.

It shows a typical RNS structure. The forward converter outputs residues after receiving weighted integer operands. Consider the moduli set f2n, which includes the numbers 1, 2,n, 2n C 1g, and n D 4. The dynamic range is 4080, and the residue representation of the originally weighted input X 20 may be obtained by finding the remaining product of dividing X by the various moduli, which gives rise to X. (5, 4, 3). In reality, just modular additions are necessary to generate the residues for certain moduli sets; the complete division procedure need not be computed. Forward converters are therefore parallel architectures made up of separate units that each compute a single residue using adders for the various residues. Several publications, including, have tackled the forward converter design. Moreover, designing forward converters for universal RNS moduli sets has lately been taken into consideration.

There are five steps in the design of a reverse converter. A conversion method is chosen in the first step based on the characteristics of the moduli set. The values of the multiplicative inverses and moduli are taken into account in the conversion formulae at the next stage. The conversion equations should then be worked on to simplify the process using residue arithmetic principles. The streamlined equations are then put into practice utilizing arithmetic circuits like adders and logical gates. The Chinese remainder theorem (CRT), mixed-radix conversion (MRC), and the New CRTs-I and CRT-II, which were evolved from the original CRT, are the most well-known conversion methods. Each of these algorithms' specifics and illustrations may be found in. With RNS moduli sets, there are several reverse converter designs. As the structure of the reverse converter depends on the moduli sets, effective reverse converters are often designed using specific moduli sets with a constrained number of moduli to optimize the design for various cost functions, such as performance, cost, and power consumption. Since they are employed in both the forward and reverse converters in addition to the parallel modulo arithmetic channels, modulo adder and multiplier designs are also a significant area of study in RNS. There are several architectural designs for adders and multipliers, as well as general and particular moduli, with the majority of them being for the set of moduli 2n 1, 2n 1, and 2n (for this last modulo regular binary adders and multipliers can be used. There are several further RNS processes that the applications may need. The detection of signs is one of them. In the RNS, sign detection is difficult and necessitates comparing numbers with half of the dynamic range; integers in the first half of the dynamic range are thought to be positive, and all the other half to be negative. This is in contrast to the weighted number system, where the sign can be detected by the most significant bit (MSB). A further challenging RNS operation is magnitude comparison. Typically, weighted equivalents of the RNS number are created by semi-transforming it, which is how RNS sign detectors and magnitude comparators are created. Scaling and division are other challenging tasks. In order to prevent overflow in a sequence of RNS multiplications and additions, which

are common operations of digital filtering, these operations are crucial. To simplify the procedure, the scaling factor is often thought of as the power of two modulo. Also difficult to detect in RNS is the overflow, which happens when an operation's outcome cannot be expressed within the dynamic range. Introduces the most recent and effective sources for hardware design information for each RNS component.

CHAPTER 15

RNS TEACHING METHODOLOGY

Vinay Kumar S B, Assistant Professor,
Department of Electronics and Communication Engineering, Faculty of Engineering and Technology, Jain (Deemed to be University) Bangalore, India
Email Id-sb.vinaykumar@jainuniversity.ac.in

The learner for the embedded systems courses must be familiar with the most recent advancements and developments at the arithmetic level in order to teach RNS-based embedded systems. The RNS potential and the idea of residue number-based embedded systems are not well-known in this field. While the instructors who regularly follow publications may be aware of its potential, it is challenging to incorporate RNS in embedded systems courses in a systematic and coordinated fashion. The educator should be knowledgeable with the capabilities and characteristics of each RNS component, as well as how to apply it to embedded systems. Just giving students a brief introduction to the emerging method of RNS-based embedded system design can confuse them and not teach them how to use modular arithmetic in RNS-based embedded systems due to the multidisciplinary nature of RNS, which is based on mathematical formulation, digital design, and computer architecture.

Moreover, since RNS has so many applications, it may not be practical to cover them all. RNS is applicable to both the hardware and software components of embedded systems. Extensions to the instruction-set architecture (ISA) that provide single-purpose and application-specific processor architecture may include RNS structures. Also, as shown by RNS-based encryption implemented on GPU, RNS is efficiently employed in parallel programming of embedded devices. In this part, a systematic and structured methodology is provided with the goal of instructing students on the RNS technique for creating embedded systems and inspiring them to work and do research on it. The two primary stages of this technique are as follows: first, students should familiarize themselves with the idea and framework of RNS. Making sure students are acquainted with a precise and well-fitting framework for embedded system applications is crucial.

The subject could get dry if students are just given a ton of references without being shown an example of a chosen architectural style. Finding the ideal architecture for embedded systems and the course requires a teacher to search through the whole collection of RNS papers, which takes time. In order to provide students a thorough grasp of how to construct the core of the arithmetic route of embedded processors based on RNS, this section simplifies the procedure and presents a more straightforward structure. For researchers in the field of embedded systems who wish to dig further and launch research projects, we also provide some tips and present some references. Since the circuits for this RNS set are simple and there are publications proposing embedded processors based on it, we chose the moduli set of 2n 1, 2k, 2n 1 as the basic set to teach RNS to students. Even though these conference papers serve as excellent examples of embedded systems built using RNS, there are several variations between them and the architecture we use here. As they are studying embedded systems and this is not a course on computer math, students may not be familiar with carry-save arithmetic. The second component of the suggested technique is learning the embedded processor architecture based on RNS, discussing the findings, and using

experimental data to demonstrate RNS' superiority in real-world case studies. In order to increase performance with great energy efficiency, students are taught to integrate their expertise from the embedded system course with modular arithmetic circuits.

Standard Concepts

Reviewing the foundational congruence theorems and formulae is the focus of stage one. It should be noted that numerical illustrations may be quite helpful in helping pupils grasp the topic. Every RNS components should include numerical examples so that students may comprehend how each RNS component works before reaching hardware implementations. It is advised that instructors choose a sample moduli set first, and then display the RNS residues in a table. The table may then be used to demonstrate the accuracy of each operation once it has been completed. This is crucial while studying the complex ideas of the RNS process. The teacher may choose two RNS numbers after presenting the dynamic range table and ask the students to compare them and identify their signs without displaying the table. Students get acquainted with RNS hard operations thanks to these simple questions. In addition, the absence of a sign bit, significance, and link between residues may be used as examples to highlight the challenges associated with putting inter-modulo operations into practice. To help students learn, both contents offered in and numerical examples may be employed. Students should be able to comprehend the mathematical foundation of RNS and the operations involved in each section, the selection process for moduli sets, and how to create condensed formulae for forward and reverse conversions for carefully chosen moduli sets at the conclusion of this stage. Also, it is advised that educators provide a short introduction to RNS floating-point numbers. This point may be very useful for energy-efficient floating-point devices if it is developed correctly. Researchers have already identified this curricular element. In this instance, it is enhanced by the inclusion of several significant aspects including a discussion of challenging RNS operations principles, the use of a dynamic range table, and floating-point operations. Students' misconceptions regarding RNS capabilities, which are covered in greater depth in the next sections, may be caused by teachers focusing only on mathematical explanations without taking into account specifics of hardware implementations.

Modular Multipliers and Adders

The second step of the process deals with RNS-compatible modular adders and multipliers. It is advised to discuss adders first, followed by multipliers. As they are important and useful in actual RNS implementations, moduli 2n 1 and 2n 1 adders are described in this article along with a detailed explanation of how they work. Ripple-carry adder (RCA)-based hardware designs may be utilized to teach these adders because of their simplicity, particularly for newcomers. Papers and are useful resources at this point. It is preferable to begin with other moduli that have lately drawn research attention, such as 2n 3, which may continue to be a focus for the course's projects. More complex structures, such hybrid adders and high-radix booth recoded modulo multipliers, may be shown if more effort can be spent on this step. In addition, discussing the characteristics of various circuit architectures and the effects of employing various structures should unquestionably be taken into account at this time. Tutors should educate students on the benefits and drawbacks of the various architectures and have them ready to choose components for certain applications. As an instance, an instructor may discuss parallel-prefix modulo adders, which are fast adders but have high power requirements. Conversely, ripple-carry adders with end-around carry use far less energy but have a longer delay. The influence of adder structures

on the design and effectiveness of circuits may be demonstrably shown by describing the hybrid structures and utilizing the experimental data to illustrate the trade-off between delay and power consumption. You may learn about multipliers from similar cases. In order to save space and energy anytime the module 2n 1 multiplier is not on the critical path, the authors of this study increase the delay of the multiplier. For students, it is intriguing to see how circuit settings might alter the multiplier's design.

Converter to Forward

After pupils are comfortable with adders and multipliers, one may describe how to use the forward converter. Contrary to what was first stated, it is advised to teach this section after discussing adders since the forward converter is often made up of multi-operand modular adders. The instructors can readily demonstrate the whole designs of forward converters for various moduli since it is easier to grasp forward converter hardware if you have a solid knowledge of modular additions.

Backward Converter

The next step is to discuss one of RNS's most important and challenging operations. It is essential that students become knowledgeable about the development and use of reverse converters. The moduli set is very important for reverse converters. The performance of the reverse converter is also influenced by several other variables, including the algorithm and components used. Students learned about moduli set selection and various reverse conversion mathematical procedures throughout the first phase. Here, we'll demonstrate to the pupils how to use modular adders to solve simplified final reverse conversion equations. Due to the dependence between the converter architecture and the chosen moduli set, this section cannot be taught in a generalized manner with all of its subtleties. As a result, while teaching this area, papers with particular moduli sets might be a useful option. Picking up another particular moduli set and asking students to locate and use the streamlined formulae is a good exercise for this section.

RNS-Based Embedded Systems Design Application

It is feasible to demonstrate to students how RNS may be utilized in embedded system design after introducing them to the intriguing characteristics and structures of RNS. An analog-to-digital converter (ADC), a digital-to-analog converter (DAC), a DSP, cryptographic processing cores, and memory may be used by the instructor to first illustrate the structure of a typical embedded system architecture.

Then he may explain how to improve the embedded system using the relevant RNS structures. In the past, ordinary ADCs were used to convert analog signals to digital ones, and forward converters were used to turn the digital signal into the remaining signals. To directly convert analog signals to digital residues, however, analog to residue converters (ARC) have been developed lately. This method allows for the inclusion of the forward converter delay and cost in the ADC. Yet, the system's components should all work with residues. Similar to this, a digital data to analog signal converter (RAC) that does away with the requirement for a reverse converter has been suggested. There are many RNS-based DSPs, cryptographic processing circuits, and RNS-based memory options that may be included into the design of an embedded system to accomplish a variety of applications. The principles needed to comprehend RNS as

part of a course on computer algebra and embedded systems are now explained. The two following elements may also be added to the program if additional time can be devoted to teaching RNS.

Intensive RNS Operations

Further procedures including sign detection, magnitude comparison, overflow detection, scaling, and division are needed in certain computationally expensive applications. Previous curricula and publications did not address these difficult RNS procedures. Despite the fact that solutions to the problems have been proposed, they have not yet been successfully implemented. Thus, these domains are the subject of ongoing study. It is advised that the teachers describe how each of them is designed using current academic articles. Explaining tough RNS operations has the benefit of encouraging students to do more study on these difficult concepts in addition to providing answers to their queries about them.

Implementation of ASIC/FPGA

The proposed methodology's final stage, which is optional but also encouraged, demonstrates the entire process of designing a hardware implementation of an RNS system, including design, HDL coding, functional verification using simulation tools, synthesis of the logical circuits, and experimental evaluation. Students might be inspired and given a practical perspective by examples used to illustrate these stages. Students are expected to have experience designing digital circuits and writing HDL code. Nonetheless, a quick explanation of the various HDL coding types and how to choose between structural and behavioral modeling depending on the circumstance might be useful. Additionally, this stage can be a good time to discuss how application-specific integrated circuits (ASIC) or field-programmable gate arrays (FPGA) can achieve appropriate trade-offs between area, delay, and power consumption through the careful and intelligent placement of various hardware components in the structure of RNS circuits (FPGA). Due to the time commitment, it should be noted that not all RNS components must be implemented. It is sufficient to provide a sample, such as ready HDL codes for a key component of RNS, such as the reverse converter, which may be supplemented by the results of ASIC or FPGA implementation.

RNS-Based Embedded Processor Design

With a broad variety of items, digital consumer electronics comprise a significant part of the global economy today. In many digital consumer devices, including those for telecommunications, digital audio, video, and image, speech processing, encryption, and multimedia, digital signal processors (DSPs) play a crucial role. During the last ten years, the design of DSPs has advanced quickly because to the constant requirement to combine power consumption, flexibility, and the integration of new features with performance improvement.

Traditional carry propagation arithmetic, which uses a weighted number system as its foundation, is a commonly used and well researched method. The construction of arithmetic units with higher performance and increased efficiency is, however, prevented because of dependencies and the need to carry out the entire weighted propagation of the carry. As a result, the Residue Number System (RNS) has been suggested as a substitute arithmetic system for applications that need a lot of calculation. The remainders of the division by co-prime moduli, which make up a moduli set, are used by RNS, a non-weighted numbering system, to represent

an integer value. By lowering the amount of hardware needed to process the data, the many and smaller values utilized in the RNS encoding enable parallelism, high-speed, and low energy processing. The fact that the multiplications and adds are carried out individually on each separate residue channel is what causes the increased parallelism. A general, effective, and scalable RNS architecture enabling moduli sets with an arbitrary number of channels is needed to make RNS use, adaptation, and application easier. The design of RNS with any moduli set of the type 2n k0; ; 2n kj, where j NC0, is made possible by the proposed unified structure for a scalable RNS processor based on 2n ki (ki 2n=2 1) moduli channels in this chapter. The investigated moduli set permits arbitrary changes to the number of RNS channels, which increases the Dynamic Range (DR), or changes to the channel width, which decreases latency and area cost. The proposed RNS architecture offers a comprehensive processing system that supports the calculation of conversions, including base extension, RNS-to-RNS conversions, and binary-to-RNS conversions. The suggested architecture enables the creation of a more compact RNS processor by having the capacity to compute the conversion using the arithmetic units of each channel. At the channel level, addition, subtraction, and multiplication are enabled along with the ability to accumulate results. Two procedures are taken into consideration for the reverse conversion: the Mixed Radix Converter and the Chinese Remainder Theorem (MRC). A general RNS design is also suggested, together with an Instruction Set Architecture (ISA) to give a straightforward interface. The suggested RNS architecture and the resultant processor implement the ISA without the requirement to specify the number of channels directly, enabling the generation of generic code irrespective of the DR, making this apparent to the programmer.

Architecture for Processors

Targeting demanding arithmetic tasks, the RNS processor is intended to be utilized as a coprocessor of generic Central Processing Units (CPUs). The conversion from binary to RNS comes first in an RNS calculation process, which is then followed by the arithmetic operations carried out in each channel, and ultimately the conversion from RNS to binary. As a result, receiving code and data (often in binary) from the CPU and sending it back after processing is one of the key responsibilities of the RNS processor. The technique herein provided takes into account that the number of needed conversions is smaller than the number of arithmetic operations conducted in the channels, which is characteristic of numerous applications, including cryptographic operations, in an effort to create a more compact structure. Moreover, because conversion processes may be carried out using the hardware resources of the modular channels, this enables more compact topologies. As a result, these conversions are carried out in order.

Blocks for channel arithmetic, an RNS-to-binary converter, and a global control unit are all included. The modular additions, subtractions, and multiplications are carried out on each channel by the channel arithmetic blocks. These arithmetic operations were selected as the modular operations because they are fundamental to applications like asymmetric cryptography and digital signal processing. The binary-to-RNS conversion is likewise carried out using these arithmetic blocks.

Here, a two-step technique is used to convert RNS data to binary data. The RNS format is first transformed into a Mixed Radix System (MRS) representation, and then the MRS values are afterwards transformed into binary values. The arithmetic moduli channels are used to completely calculate the MRS conversion. The RNS-to-binary converter module, which contains the extra logic necessary for the conversion, is used to calculate the second step of the reverse

conversion, which cannot be completed using just the arithmetic operations accessible in the RNS channels. Here, base extensions are also carried out using the MRS operation in order to execute the conversion between moduli sets. The basic extension enables number representation conversion between two moduli sets without converting from RNS to binary in the source moduli set and from binary to RNS in the target moduli set. When calculating asymmetric cryptographic algorithms using RNS, one of the key operations is the Montgomery Modular Multiplication, which heavily utilizes the base extension operations. Since each RNS processor only has one moduli set, these base extension procedures need the calculation of several moduli sets, necessitating many processors for the proposed RNS computation. As a result, the suggested RNS design permits the use of several RNS processors inside the RNS computing system. Fist in First out (FIFO) buffers may be utilized to create communication between the RNS architecture and generic CPUs since they provide communication and data flow control while also serving as synchronization points. The studied ISA and the suggested architecture are described in the parts that follow, together with the intended computing logic for the binary-to-RNS conversion, modular arithmetic operations, and RNS-to-binary conversion.

Instruction Set Architecture

An ISA is developed to provide a straightforward and independent interface to the proposed RNS architecture. Without the need to explicitly specify the number of channels for the implemented RNS and resultant DR, this ISA takes a scalable RNS architecture into account. This makes it possible to produce transparent to the programmer generic code. Without having to be familiar with the specifics of RNS arithmetic, the programmer may write programs for a normal arithmetic binary processor, with the exception of the conversion procedures. As further outlined in Section 2.4.2, the specified ISA supports the following arithmetic operations: addition, subtraction, and multiplication, both with and without accumulation capability. In addition to these arithmetic operations, base extension, binary conversion operations, and base extension are also taken into consideration and are further described. As arithmetic calculations are used to carry out these conversion procedures, the number of computations and thus the amount of clock cycles rely directly on the quantity of channels. Certain constants are needed for the conversion calculation; these constants are dependent on the moduli set being utilized and remain constant throughout the RNS processor's lifetime. Overhead is added owing to data transfers if they are defined programmatically and transferred from the CPU to the coprocessor. These constants and the operation micro-code are included into the RNS processor in light of this and to retain an easy-to-use interface. Using this, it is feasible to calculate binary conversions to and from other formats as well as potential base extensions without the need for outside assistance. As previously mentioned, the conversion processes are carried out sequentially, necessitating the completion of many cycles. Due to this, there are two primary groups of instructions for the RNS processor: single cycle instructions and multi-cycle instructions. Although the multi-cycle instructions directly rely on the number of RNS channels, imposing a number of cycles equal to the number of channels, the first one just needs one clock cycle to perform the arithmetic calculation. Here, a 32-bit instruction size is taken into consideration since it is the length that is now most often utilized on embedded Processors. The suggested instruction structure is separated into four fields: operation code (opcode), destination register (rd), register source one (rs1), and register source two (rs2). There are 256 registers that may be addressed thanks to the 8-bit length of each register field. Two sub-fields make up the operation code field: the first is needed to enable the RNS processor, and the second specifies the arithmetic operation.

Using processor enabling flags, the RNS architecture permits the use of up to three RNS processors in a single RNS computing system. Given the prevalent RNS-based cryptographic implementations, this upper limit is taken into account. The arithmetic operations (opAU) are given by four bits, plus one bit to permit accumulation. In multi-cycle operations like the base extension, this extra bit is also utilized to specify whether the internal shared bus propagates the data, enabling data transfers between two RNS processors.

Single-cycle instructions and multi-cycle instructions make up the proposed ISA. The first three operations (1–3) perform modular addition, subtraction, and multiplication between rs1 and rs2, storing the result in the designated rd and, if the accumulator flag and is active, in the accumulator. The next three operations (4-6) that are preceded by "a" carry out the identical tasks as the ones before them, with the exception that the accumulator value is also added. The final value is once again saved in the destination register and, if flag an is set, in the accumulator. The next three single cycle instructions (7-9), denoted by "s," behave in the same way as the preceding ones, except that they deduct the result of the operation between rs1 and rs2 from the accumulator rather than adding it. The flag a serves the same purpose in these instructions; if it is set, the operation's outcome is also saved in the accumulator in addition to the destination register.

The first two multi-cycle instructions (10–11) change the moduli set in the RNS processor from binary to a mixed-radix representation, respectively. These instructions only need the destination register to be defined since they receive data through the shared bus. A source register and a destination register are needed to carry out the conversion from RNS to MRS in the third multi-cycle instruction (mToRNS rd,rs1). Be aware that after this conversion, the RNS representation is replaced by a Mixed Radix System. The second stage of the conversion from RNS-to-binary, which was initiated by an RNS-to-MRS instruction, is carried out by the instruction mToBin (MRS-to-binary). Keep in mind that the RNS-to-binary conversion is seen as a two-step process in this context, as further explained in. As the calculated values are pushed to the output data buffer instead of a target register, the MRS-to-binary instruction just needs to define rs1. Each RNS channel is divided into four primary components to support this ISA: (1) the register bank and accumulator for data storage; (2) the RNS Arithmetic Unit (AU) for the execution of modular arithmetic operations; and (3) the control unit (CTR). The global control unit's behavior is determined by the data information flow from and to the CPU. Keep in mind that one of the key functions of the RNS architecture, which is to be utilized as a coprocessor of generic CPUs, is to accept code and data from the CPU and deliver it back after processing. Buffers are employed in this instance as synchronization points to streamline communication and data flow management. As a result, anytime necessary data or code is lacking, the RNS processor stalls. Moreover, the RNS processor stops when the output data buffers are full, preventing the loss of processed data. One unidirectional and one bidirectional FIFO buffers are taken into consideration for code and data, respectively, in order to construct this data control flow, as shown in the diagram.

Mathematical Operations in RNS

This section discusses the arithmetic operations in each moduli channel, the RNS-to-binary conversion, the base extension operation, and the formulation and resultant modular hardware structures needed to calculate the binary-to-RNS conversion, RNS-to-binary conversion, and these operations.

RNS to Binary Conversion

The conversion to the RNS representation is the first usual operation in an RNS processor. Calculating the residual of each operand's division by 2n ki is done to accomplish this. Calculating the integer division of the input value X by 2n ki will provide the input value's residue modulo 2n ki. Yet this method of getting the rest is an expensive one. The residue xi of an integer X with.N n/-bit inputs may be computed as follows when this procedure is done using solely modular addition operations:

Calculation Channels

One of the most crucial considerations for computation optimization in the Residue Number System is the arithmetic operations. Several smaller arithmetic channels make up an RNS. The modular arithmetic structures used to build these channels are often more complicated than their binary counterparts with the same bit width. The conversion overheads may be offset by an effective implementation of these arithmetic structures, which can result in overall gains. Given their simplicity, moduli sets with modulo 2n k channels and unlimited k values may be used to define RNS with bigger moduli sets, even if the majority of the state of the art still concentrates on modulo channels of the type 2n 1. Given that the operands in each channel need fewer bits, this enables the creation of arithmetic systems with improved performances. The concept and resultant structures for executing addition, subtraction, and multiplication with and without accumulation for modulo f2n — kg and f2n C kg are shown in the following.

Control Modules

Two different kinds of control units are employed in the RNS processor here: a global control unit for processor control and a distributed control block (CTR) for each arithmetic channel. The fetching of code, the decoding of the corresponding instructions, and the control of transfers to and from the data buffers are all managed by the global control unit. Without touching the data bus or constant memory, the single cycle arithmetic instructions compute the result using data from registers and save the result into a destination register.

A step counter that is increased after each repetition of the operation is used to manage the multi-cycle instructions. The multi-cycle instruction is finished when this counter reaches the number of iterations necessary to finish a particular instruction, which depends on the number of channels in the RNS processor. After that, another instruction may be received from the code buffer. Arithmetic operations that are only partly supported by arithmetic channels are used to implement the multi-cycle instructions.

The micro-code that will be executed defines these multi-cycle instructions. On the specific micro-code memory, this micro-code is kept. the following operations are carried out on the arithmetic channels for multi-cycle instructions: (1) binary-to-RNS (mnemonic bToRNS); (2) MRS-to-RNS (mToRNS), which is utilized to convert from a mixed-radix representation to the RNS processor moduli set; (3) RNS-to-MRS (rnsToM); and (4) MRS-to-binary (mToBIN). The necessary micro-code may be condensed to a relatively tiny memory, since there are only three different kinds of arithmetic operations that must be performed, and they only vary in the initial stages of each instruction. 4. The only difference between the micro-codes for the instructions bToRNS and mToRNS is the set of constants they utilize. The same thing occurs when using the rnsToM and mToBIN commands.

The operands source is described in addition to the arithmetic operations utilized in each multi-cycle instruction. These sources include the data buffer (for external data), internal shared bus, register file, and internal memory (for the constants stored in memory). The data buffer and internal memory are used as sources one and two, respectively, by the binary-to-RNS and MRS-to-RNS conversion instructions. The internal shared bus is used as the source for the first operand of the RNS-to-MRS conversion instruction, while the internal memory is used as the source for the second operand. The information shown on the internal bus is determined by the operation step and derived from the arithmetic channels. This is how the mixed-radix digits zi are propagated across the other RNS channels.

The register file and internal memory serve as the operand sources for the last multi-cycle instruction, which performs an MRS-to-binary calculation. Because no registers need to be updated, this last instruction also turns off the register update flag. This conversion procedure, as defined in, solely computes X using the binary multipliers in the channels (2.23). Remember that the RNS-to-binary conversion is calculated using two multi-cycle instructions: the first one converts RNS to MRS, calculating the mixed-radix dig- its, and the second one translates MRS to binary. With the help of the RNS-to-MRS instruction in the source moduli set and the MRS-to-RNS instruction in the destination moduli set, the architecture herein provided enables the implementation of Base Extension operations. The conversion instruction solely relies on the step counter and the stored constants for the selected moduli set when the proposed technique is used. As a result, the suggested architecture supports moduli sets of the type f2n kig and is made general and scalable. The constraint of wk log2.k/ n=2 and the requirement that the moduli must be co-prime, however, place a cap on the number of moduli that may be included in the moduli set. As a result, the number of co-prime integers and moduli that may coexist increases with each channel's width, n. how many co-prime integers may be specified in relation to the permitted channel width.

State-of-the-Art Analysis

The relevant state of the art, which consists of several programmable RNS architectures, is examined here in order to compare the scalable and programmable RNS processor that is detailed here. The Residue Digital Signal Processor (RDSP), a general processing architecture proposed in, the well-known Cox-Rower architecture proposed in, and a more recent work implementing a unified computing system all meet this requirement. In these systems, the code is defined as data transfers between registers and functional units. The computation of asymmetric encryption techniques like the RSA and ECC algorithms is the focus of the following two designs.

Modern Technology

A 32-bit pipelined RISC-based processor with five pipeline stages, the RDSP uses RISC technology. Arithmetic units supporting the balanced moduli set 2n 1, 22n, and 2n 1 with n 8 carry out the calculations. This processor's ability to handle binary arithmetic operations is one of its unique features. The necessary conversion procedures from binary to RNS and RNS to binary are supported by this CPU. Specialized arithmetic units are used for this. The processor supports both accumulation-based and accumulation-free modular addition and multiplication operations. Moreover, this processor includes flow control instructions (such as brunch instructions), a functionality not found in the processor thus described or in the remaining state of the art. Different RNS moduli sets, such as f2n — 3; 2n — 1; 2n C 1; and 2n C 3g, could be considered

for this processor in order to increase the DR and/or to reduce the width of the RNS channels. This architecture's main disadvantage is that the supported moduli set only permits the parallelism of three channels, providing a relatively small DR. As the modular reduction of f2n 3g consumes more space and imposes greater delay costs, this moduli set has the drawback of requiring more complicated modulo arithmetic.

Computation That Is Fault-Tolerant in a Redundant Residue Number System

The size of device features has rapidly decreased over the last several decades to keep up with the increasing demand for electronic items that are smaller, quicker, and less power-hungry. The off-leakage current of the whole chip will grow significantly before the scaling of device feature size hits the ultimate 5 nm physical limit for implementability. Device dependability suffers as a result of the current rate scaling of the devices. The device is increasingly prone to parameter fluctuation as the feature size decreases. The dependability of the device will then become more unpredictable as a consequence of the failure mechanisms that are amplified by parameter uncertainty.

The stress factor will also rise as a result of the device node decreasing. The oxide layer will encounter larger electric fields as a result of the operating voltage scaling's somewhat slower rate, which will make the layer more susceptible to device failure. Transistors with a size of a few tens of nanometers and less are subject to single event upsets and intermittent mistakes while operating at low supply voltage because aging, noise, radiation, heat, and other dynamic circumstances may readily perturb a device out of specification. Designing for fault tolerance must greatly improve the dependability of digital systems. Error repair code, self-checking logic, module replication, and reconfiguration are examples of traditional fault-tolerant approaches, however they are prohibitively costly and not scalable. Several of these techniques to fault detection and reconfiguration that combat soft mistakes need enormous test and reconfiguration times or vast trustworthy memory.

By virtue of its parallel and modular arithmetic operations, the Residue Number System (RNS) might possibly alleviate the dependability requirement of low-level circuits. Variations in circuit power, timing perturbations, and transient perturbations during calculations may change the states of numerous devices nearby due to the isolation provided by carry forwarding in RNS, but cluster faults that occur in one modulus channel will not spread to the others. A Redundant RNS (RRNS), which has error detection and repair capabilities by maintaining the valid range for routine information processing, is created by adding redundant moduli to an existing moduli set. By determining if the matching size of the received residue representation falls inside the illegal range, the residues mistake is identified. The residual mistakes must first be found once their presence has been established before being fixed. In contrast to two's complement number system (TCS), error detection and correction techniques based on RRNS may also fix arithmetical processing mistakes brought on by purposeful fault injection, manufacturing flaws, process, voltage, and temperature changes, and circuit noise.

RNS-based error detection and correction algorithms often need lengthy calculation times and huge hardware implementation areas, despite their error robust properties. This is because systematically locating the incorrect residue digits requires repetitive calculations. The size of the moduli set affects how many calculations are required. Multiple error detection and correction require numerous iterative computations because the size of the moduli set grows as the number of correctable residue errors increases. As a result, their direct hardware

implementation is just as challenging as, if not more challenging than, the implementation of several inter-modulo operations in RNS. The idea of identifying, finding, and fixing residue mistakes in duplicate RNS is presented in (RRNS). Based on the amount of residue mistakes that can be corrected, the current residue error detection and correction methods in RRNS.
