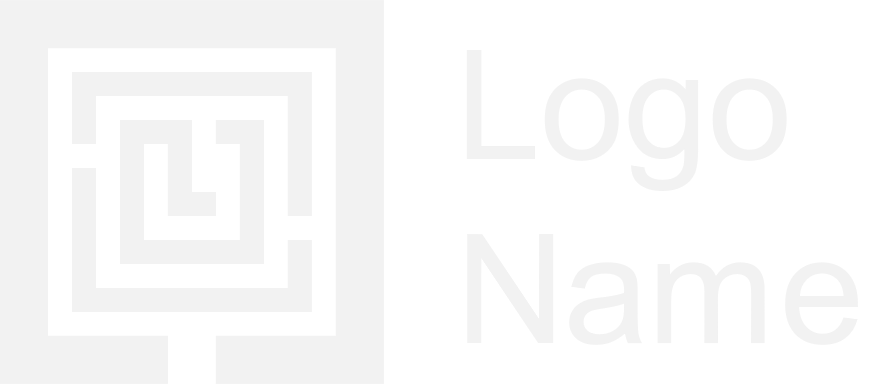
|  |
| --- |
| ***Squad real time operating system*** ***Supervisor: Dr.Omnia El-Barbary***  ***Graduation year: 2023-2024***  ***Department: Information Technology***  Faculty of Computer and Information technology  Team: Mahmoud Sofar Adham Emad Ahmed Ehab  Ali Ashraf Sohaila Mohammed Esraa Elzebedy Rawan Emad Sohaila Khalaf  Grade: Senior level |
|  |

Logo

Description automatically generatedLogo

Description automatically generated

**Content:**

**Introduction**

* ***What is an Embedded system?***
* ***What is an Embedded system mechanism?***
* ***Benefits of an Embedded System***
* ***Structure and Types of Embedded systems***
* ***Brief history of embedded systems***

# **Section1: Real Time Operating System**

* ***1.1 What is an operating system?***
* ***1.2 What is Real Time Operating System (RTOS)?***
* ***1.3 What is the difference between OS and RTOS?***
* ***1.4 Types of Real Time Operating System (RTOS)***
* ***1.5 What is Assembly?***
* ***1.6 What is SquadRTOS ?***
* ***1.7 What is a Task?***
* ***1.8 What is Super loop? And why are we using it ?***
* ***1.9 Interrupts usage***
* ***1.10 Kernel***
  + - * + ***1.10.1 Dispatcher and Scheduler.***
        + ***1.10.2 What is interrupt handling?***
        + ***1.11.3 Memory Management***
* ***1.11 Interrupt handling***
  + ***1.11.1 Memory management***
* ***1.12 Communication Stack (LWIP)***
* ***Section2: Communication***
  + - * + ***2.1 What is IP address?***
        + ***2.2 IP versions.***
        + ***2.3 What is iPv4?***
        + ***2.4 What is ipv6?***
        + ***2.5 OSI model and TCP/IP model***
        + ***2.6 THE REASONS WHY THE OSI MODELS IS USED***
        + ***2.7 The OSI layers***
        + ***2.8 LWIP***
        + ***2.9 UART***
        + ***2.10 Ethernet Stack***
* ***Section3: Webpage*** 
  + - * + ***3.1 Brief of webpage development***
        + ***3.2 Design***
        + ***3.3 Languages used to build RTOS webpage.***
        + ***3.3.1 JavaScript***
        + ***3.3.2 CSS***
        + ***3.3.3 HTML***
        + ***3.4 Benefits of RTOS webpage.***

***Section 4: Desktop Application***

* + - * + ***4.1 Libraries used in the Desktop application.***
        + ***4.2 Languages used in Desktop application.***
* ***4.2.1 Python***
* ***4.3 Desktop design***
* ***4.4 Benefits of Desktop application***

# ***What is an Embedded system ?***

***A computer hardware and software combination created for a particular purpose is an embedded system. Additionally, embedded systems may operate as part of a bigger system. The systems may be programmable or may only perform certain functions. An embedded system may be found in industrial machinery, consumer electronics, agricultural and processing sector equipment, vehicles, medical devices, cameras, digital watches, home appliances, aircraft, vending machines, toys, and mobile devices.***

***Even though they are computer systems, embedded systems can have simple graphical user interfaces (GUIs), like those seen in mobile devices, or they can have no user interface (UI), like those found in devices meant to execute a specific job. Button, LED (light-emitting diode), and touchscreen sensing are examples of user interfaces. Some systems also employ remote user interfaces***.

***By 2025, the embedded market, according to MarketsandMarkets, a business-to-business (B2B) research company, will be worth $116.2 billion. Numerous well-known technology companies, including Apple, IBM, Intel, and Texas Instruments, are among the chip producers for embedded systems. The continuous investment in artificial intelligence (AI), mobile computing, and the demand for high-level processing processors are all contributing factors to the anticipated rise.***

***Various embedded system examples***

***Many different technologies in a variety of industries use embedded systems. Several instances include:***

***Automobiles. In most modern automobiles, there are several computers—up to 100, sometimes—or embedded systems that are used to carry out various functions. Some of these systems carry out fundamental utility tasks, while others offer entertainment or user-facing features.***

***Cruise control, backup sensors, suspension control, navigation systems, and airbag systems are a few embedded technologies found in consumer automobiles.***

***Cellular phones. These are made up of a variety of embedded systems, such as operating systems (OSes), GUI software and hardware, cameras, microphones, and USB I/O (input/output) modules.***

***industrial equipment. They may both be embedded systems themselves as well as incorporate embedded systems like sensors. Embedded automation systems that carry out certain monitoring and control tasks are frequently found in industrial machinery.***

***medical supplies. These might have embedded systems, such as sensors and control systems. Medical devices, like commercial machines, must also be highly user-friendly to avoid machine errors that may have been avoided. This implies that they frequently have a more complicated OS and GUI created for a suitable UI.***

# ***What is an embedded system's mechanism?***

***The word "embedded" refers to the fact that embedded systems always operate as a component of an entire device. Small computers that are embedded in other mechanical or electrical systems are low-cost and power-efficient. They typically include a CPU, a power source, memory, and communication interfaces. Embedded systems employ communication ports to send data via a communication protocol between the CPU and peripheral devices, which are frequently other embedded systems. This data is interpreted by the processor with the aid of simple memory-stored software. Typically, the software is very specialized for the purpose the embedded system serves.***

***A microprocessor or microcontroller might be the processor. Microcontrollers are merely microprocessors with built-in memory and external ports. Memory and peripherals are not built into microprocessors' chips; instead, they are used in separate integrated circuits. Both can be employed, however because microprocessors are less integrated than microcontrollers, they often need additional support circuitry. System on a chip (SoC) is a common phrase. On a single chip, SoCs house several processors and interfaces. For embedded systems with great volume, they are frequently employed. Application-specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs) are a few of examples of SoC kinds.***

***Embedded systems frequently operate in real-time situations and interact with the hardware using an RTOS (real-time operating system). Designers have increasingly determined that near-real-time techniques are appropriate at greater levels of chip capacity and that the jobs are tolerant of minor fluctuations in reaction time. Embedded Java and Windows IoT (formerly Windows Embedded) are two more operating systems that have been scaled down to operate on embedded devices. In these situations, stripped-down versions of the Linux operating system are frequently used.***

# ***Benefits of an Embedded System:***

***Embedded systems' primary attribute is that they are task-specific.***

***The following qualities can also be present in embedded systems:***

***- Comprised of software, firmware, and hardware.***

***- They are designed for specialized duties inside the system, not a variety of jobs, therefore they may be embedded in a larger system to carry out a particular purpose.***

***- Either microcontroller- or microprocessor-based, both of which are integrated circuits that provide the system's computing power.***

***- Are frequently used in internet of things (IoT) devices for sensing and real-time computing, which are internet-connected devices that do not need a user to function.***

***- Can differ in function and complexity, which has an impact on the software, firmware, and hardware they employ.***

***-Are frequently forced to do their task under time pressure to maintain the health of the bigger system.***

# ***Structure and components of Embedded systems:***

***- Although embedded systems' complexity varies, they typically contain three key components:***

***- Hardware: Microprocessors and microcontrollers are the foundation of embedded systems' hardware. Microprocessors and microcontrollers have a lot of similarities. A microprocessor is a central processing unit (CPU) that is combined with other fundamental computing elements including memory chips and digital signal processors (DSPs). All of the parts are housed on a single chip in microcontrollers.***

***- Firmware and software: The complexity of software for embedded devices might vary. However, embedded IoT systems and industrial-grade microcontrollers typically run very straightforward software that uses little memory.***

***- Operating system in real time: Particularly for smaller-scale systems, they are not usually present in embedded systems. By controlling the software and establishing guidelines for program execution, RTOSes specify how the system functions.***

***A fundamental embedded system would include the following hardware components:***

***- Sensors: Data from physical senses is converted into an electrical signal through sensors.***

***- Analog-To-Digital convertor: An analog electrical signal is converted into a digital signal using an analog-to-digital (A-D) converter.***

***- Processors: Digital signals are processed by processors and stored in memory.***

***- Digital-To-Analog convertors: The digital data from the processor is converted into analog data via digital-to-analog (D-A) converters.***

***- Actuators: Actuators choose the appropriate output by comparing it to memory-stored output.***

***The processor converts the information from the processor's readable input, which the sensor reads from external sources, into meaningful output for the embedded system.***

# Types of Embedded systems:

***There are a few fundamental types of embedded systems, and each has unique functional needs. As follows:***

***- Mobile embedded systems: Small systems made to be portable are called mobile embedded systems. This may be seen in digital cameras.***

***- Networked embedded systems: Embedded systems that are networked can output data to other systems through this connection. Point of sale (POS) systems and home security systems are a couple of examples***

***.***

- ***Standalone embedded systems: Independent embedded systems are those that do not rely on a host system. Similar to every embedded system, they carry out a certain function. In contrast to other embedded systems, they are not always a part of a host system. An illustration of this would be a calculator or MP3 player.***

***- Real-Time embedded systems: The needed output is provided via real-time embedded systems in a predetermined amount of time. They frequently perform time-sensitive jobs, making them useful in the medical, industrial, and military fields. An illustration of this would be a traffic control system.***

# ***Brief history of embedded systems:***

***In the 1960s, embedded systems first appeared. In order to lower the size and weight of the Apollo Guidance Computer, the digital system deployed on the Apollo Command Module and Lunar Module, Charles Stark Draper created an integrated circuit in 1961. It assisted astronauts in gathering real-time flight data as the first computer to employ ICs.***

***The cost of integrated circuits fell, and their use increased by the late 1960s and the beginning of the 1970s. Texas Instruments created the first microcontroller in 1971. The TMS1000 series, which went on sale in 1974, included a 4-bit CPU, read-only memory (ROM), and random-access memory (RAM), and it was priced at about $2 per unit when purchased in bulk.***

***The 4004, often regarded as the first commercially available CPU, was also produced by Intel in 1971. The 4-bit microprocessor was intended for use in calculators and other compact devices, but it also needed support chips and perpetual memory. The 16 KB, 8-bit Intel 8008 was introduced in 1972, while the 64 KB, 8-bit Intel 8080 was released in 1974. The x86 series, which replaced the 8080 in 1978, is still mainly in use today.***

***Real-time VxWorks, the first embedded operating system, was published by Wind River in 1987. In 1996, Microsoft introduced Windows Embedded CE. The first embedded Linux products started to appear around the late 1990s. Today, almost all embedded devices run Linux.***

Diagram

Description automatically generated

# ***Section 1:***

- 1.1 What is an operating system?

A program that controls computer hardware or acts as an interface is known as an operating system. An operating system loads during the startup phase when you switch on a computer and starts sending orders to various parts.

An operating system (OS) not only enables communication between computer parts but also serves as the user interface. There are other kinds of operating systems, including embedded OS created for dishwashers, HVAC systems, automobiles, and ATM machines. While most people are aware with operating systems used on home computers or game consoles, there are other sorts of operating systems.

Many people think that the first microcomputer, the MITS Altair, was the first to have an operating system. Contrary to popular belief, operating systems have a long history dating back to the 1950s, long before solid-state drives and CD-ROMs were commercially accessible.

- 1.2 What is Real time operating system (RTOS)?

***RTOS stands for real-time operating system, which is a specialized operating system that is designed to provide a reliable and predictable platform for embedded systems that require precise timing and responsiveness.***

***Unlike general-purpose operating systems, which are designed to provide a wide range of features and functionality, RTOS is designed to prioritize real-time performance and ensure that critical tasks are given priority over non-critical tasks. This is achieved through features such as preemptive multitasking, task scheduling, and interrupt handling.***

***RTOS is commonly used in applications such as robotics, control systems, and automotive electronics, where precise timing and responsiveness are critical for the proper functioning of the system. It provides a highly customizable and efficient platform for developing embedded systems, with features such as low-level hardware access, optimized algorithms, and comprehensive libraries and tools.***

***Overall, RTOS is a powerful and specialized operating system that provides a reliable and predictable platform for embedded systems that require real-time performance.***

# ***1.3 Difference between RTOS and OS:***

- While general-purpose operating systems (OS) prioritize the performance of the entire system, real-time operating systems (RTOS) prioritize time-sensitive tasks and ensure that they are completed within predetermined deadlines.

- RTOS supports applications in crucial sectors including aerospace, medical technology, and automotive systems, whereas general-purpose OS supports servers and home computers.

- In contrast to general-purpose OS, which utilizes non-deterministic scheduling methods, RTOS uses deterministic scheduling techniques to ensure that time-critical operations run consistently.

An operating system called a real-time operating system (RTOS) is employed in systems that offer immediate answers to operational problems. It is an operating system used for time-sensitive real-time computing applications. An operating system is a piece of software that manages the hardware and software resources of a computer. It handles input and output and manages files, among other fundamental duties.

The name "RTOS" refers for "real-time operating system," emphasizing this feature: To control planning, RTOS may successfully handle interruptions through the use of priority-based functioning. Contrary to a broad sense OS, an RTOS must meet computational deadlines regardless of how dire the situation.

A crucial element of an RTOS is its dependability with regard to how long it takes to receive and execute a job from an application; this unpredictability is known as "jitter." Alternatively, OS stands for operating system.

The operating system of the entire computer is the most important application that runs on it. Every device that has a CPU and GPU comes with one or more operating systems.

# ***1.4 Types or Real Time operating system (RTOS)***

***- Hard real-time systems and Soft real-time systems***

***Hard real-time systems and soft real-time systems are two categories of real-time systems that are differentiated by their degree of tolerance for missed deadlines.***

***Hard real-time systems are designed to be extremely time-sensitive and must meet strict timing constraints. Failure to meet these constraints can result in catastrophic consequences, such as system failure or loss of life. Examples of hard real-time systems include aerospace control systems, medical devices, and automotive safety systems. In hard real-time systems, missing a deadline can result in system failure, making it critical to ensure that all tasks are completed within the specified time limit.***

***Soft real-time systems, on the other hand, have less strict timing constraints, and missing a deadline may not have catastrophic consequences. Examples of soft real-time systems include multimedia applications, such as video and audio streaming, where occasional delays may be acceptable, but consistent delays can result in a degraded user experience.***

***Overall, the distinction between hard real-time systems and soft real-time systems is based on the degree of tolerance for missed deadlines, with hard real-time systems requiring strict adherence to timing constraints, while soft real-time systems can tolerate occasional delays.***

Table

Description automatically generated

# ***1.5 What is Assembly?***

***Assembly language, also known as assembler language or symbolic machine code, is any low-level programming language with a close resemblance to the architecture's machine code instructions. It is frequently referred to simply as Assembly and is frequently abbreviated as ASM or asm. One statement per machine instruction (1:1) is the norm for assembly language, however constants, comments, assembler directives, symbolic labels for things like memory locations, registers, and macros are often also available.***

***The 1947 book Coding for A.R.C. by Kathleen and Andrew Donald Booth contains the first assembly code in which a language is used to represent machine code instructions. An assembler is a tool that transforms assembly code into machine code that can be executed. However, Wilkes, Wheeler, and Gill used the term to mean "a program that assembles another program consisting of several sections into a single program" in their 1951 book The Preparation of Programs for an Electronic Digital Computer, which is generally where the term "assembler" originates from. Assembling refers to the conversion procedure much as it does to assembling source code. Assembly time is the computing phase that occurs when an assembler processes a program.***

***Each assembly language [nb 1] is unique to a certain computer architecture since assembly depends on the machine code instructions.***

***Sometimes there are many assemblers available for a given architecture, and other times one assembler is only compatible with a certain operating system. Most assembly languages don't have a specific syntax for operating system calls, and they can be used with any operating system [nb 2] because they give access to all the processor's actual capabilities, which are ultimately what all system call mechanisms depend on. While most high-level programming languages require interpreting or compiling, which are much more difficult processes than assembling, they are typically portable across various architectures in contrast to assembly languages.***

***Both systems programming and application programming were frequently carried out fully in assembly language during the early years of computers. While still indispensable for some tasks, higher-level interpreted and compiled languages are now used for most programming. The consequences of moving away from assembly language programming were summed up by Fred Brooks in "No Silver Bullet" as follows: "Surely the most effective stroke for software productivity, dependability, and simplicity has been the increased adoption of high-level languages for programming. The majority of observers attribute that progress to advances in dependability, simplicity, and understandability that increase production by at least a factor of five.***

***For performance reasons or to connect directly with hardware in ways the higher-level language does not provide, it is common practice today to employ tiny amounts of assembly language code within larger systems developed in higher-level languages. For instance, less than 2% of the Linux kernel source code in version 4.9 is written in assembly, whereas more than 97% of it is written in C.***

# ***1.6 What is SquadRTOS?***

SquadRTOS is a real-time operating system designed to provide a reliable and efficient platform for embedded systems. It is specifically tailored for small to medium-sized embedded systems, and its highly customizable architecture can be adapted to meet the specific needs of each project.

One of the key features of SquadRTOS is its preemptive multithreading kernel, which allows multiple tasks to run concurrently while ensuring that critical tasks are given priority over non-critical tasks. This makes it an ideal choice for applications that require precise timing and responsiveness, such as robotics, control systems, and automotive electronics.

Another advantage of SquadRTOS is its comprehensive set of libraries, drivers, and tools that simplify the development process. This helps to reduce development time and costs, while improving the quality and reliability of the final product.

Overall, SquadRTOS is a powerful and flexible real-time operating system that provides an efficient and reliable platform for embedded systems. Its customizable architecture, real-time capabilities, and development tools make it an ideal choice for a wide range of applications in various industries, including aerospace, medical devices, and industrial automation. With its proven track record of success, SquadRTOS is a trusted and reliable choice for developers looking to build high-performance embedded systems.

# ***1.7 What is a task ?***

***In an SquadRTOS (Real-Time Operating System), a task refers to a unit of execution that can run independently and concurrently with other tasks. A task is essentially a piece of code that can perform a specific function, such as reading data from a sensor, processing a set of instructions, or generating an output signal.***

***Tasks are created by the application and managed by the SquadRTOS kernel. Each task has its own stack and execution context, and can be scheduled to run at a specific time or in response to an event or interrupt. Tasks can communicate with each other through message passing, semaphores, or other synchronization mechanisms.***

***One of the key benefits of using tasks in an SquadRTOS is that they allow for efficient use of system resources by enabling concurrent execution of multiple tasks. This can help improve system responsiveness and reduce latency in real-time applications.***

***Tasks in an SquadRTOS can have different priorities, which determine their order of execution. Tasks with higher priorities are executed first, and lower priority tasks are executed only when higher priority tasks are blocked or suspended.***

***Overall, tasks are a fundamental concept in SquadRTOS programming, and understanding how to create, manage, and schedule tasks is essential for developing efficient and responsive real-time applications.***

# ***What is Super loop and why are we using it ?***

A super loop is a simple programming structure that can be used to create tasks and manage their execution. A super loop consists of an infinite loop that repeatedly executes a sequence of tasks, each of which performs a specific function.

To implement tasks with a super loop, you can define each task as a separate function, and then call these functions from within the super loop. Each task can be executed sequentially, one after the other, or concurrently, by using interrupts or other synchronization mechanisms.

Here's a simple example of how to create tasks with a super loop:

void task1(void) {

// Code for task 1

}

void task2(void) {

// Code for task 2

}

void main(void) {

while (1) {

task1(); // Execute task 1

task2(); // Execute task 2

}

}

In this example, we have defined two tasks, `task1()` and `task2()`, which are called sequentially from within the super loop in the `main()` function. Each task can perform a specific function, such as reading data from a sensor, processing a set of instructions, or generating an output signal.

Note that in this example, the tasks are executed sequentially, which means that `task2()` will not be executed until `task1()` has completed. If you want to execute tasks concurrently, you can use interrupts or other synchronization mechanisms to ensure that each task is executed at the appropriate time.

The super loop structure is a simple and flexible way to create tasks in an embedded system, and can be easily adapted to suit a variety of applications. However, it may not be suitable for more complex systems that require advanced scheduling or synchronization mechanisms. In such cases, an SquadRTOS may be a better option for managing tasks and ensuring system responsiveness.

Sure, I'd be happy to discuss the advantages and disadvantages of using a super loop in an embedded system.

Advantages of a Super Loop:

1. Simplicity: A super loop structure is easy to understand and implement, making it a popular choice for small and simple embedded systems.

2. Efficiency: A super loop can be very efficient, as it avoids the overhead of task switching and context switching that is required in more complex scheduling mechanisms.

3. Flexibility: The structure of a super loop is flexible, allowing the programmer to easily add, remove or modify tasks as needed.

Disadvantages of a Super Loop:

1. Limited concurrency: A super loop structure can only execute one task at a time, which can limit the concurrency and responsiveness of the system.

2. Lack of priority: A super loop does not provide a mechanism for task prioritization, which can be important in systems that require real-time performance.

3. Difficulty in managing timing: A super loop structure can make it difficult to manage the timing of tasks, especially if the tasks have different execution times or require precise timing.

4. Lack of synchronization: A super loop structure does not provide a mechanism for task synchronization, which can be important in systems that require coordination between tasks.

Overall, the choice of whether to use a super loop structure in an embedded system depends on the specific requirements and constraints of the system. While a super loop can be simple and efficient, it may not be suitable for more complex systems that require advanced scheduling or synchronization mechanisms. In such cases, an RTOS or other more advanced scheduling mechanism may be a better option.

Basic Super Loop

Task1

Task2

# ***1.8 Interrupt usage***

***Programming using interrupts is a technique commonly used in embedded systems to handle events and tasks in a timely and efficient manner. Interrupts are signals that are generated by hardware or software events and cause the processor to temporarily suspend the current task and execute a specific interrupt service routine (ISR) that handles the event.***

***To program using interrupts, the programmer needs to define the ISR for each event and register it with the system. When the event occurs, the processor interrupts the current task, saves its context, and executes the ISR to handle the event. Once the ISR is completed, the processor restores the context of the interrupted task and resumes its execution.***

***Programming using interrupts has several advantages over other techniques, such as the super loop. Interrupts provide faster response times and more efficient use of system resources, as the processor can perform other tasks while waiting for an event to occur. Interrupts also allow the system to handle multiple events simultaneously, making it more scalable and adaptable to a wider range of applications.***

***However, programming using interrupts also requires a higher level of expertise and attention to detail, as the programmer must ensure that the ISR is properly designed, tested, and integrated into the system. Improperly designed or poorly implemented ISRs can lead to performance issues, system crashes, or other unexpected behaviors.***

***Advantages:  
  
Faster response times: Interrupts provide faster response times to events than other programming techniques, such as the super loop. This is because the processor can immediately suspend the current task and execute the interrupt service routine (ISR) to handle the event, without having to wait for the next iteration of the main loop.  
  
Efficient use of resources: Interrupts allow the processor to perform other tasks while waiting for an event to occur, which makes better use of system resources and reduces the overall processing time.  
  
Scalability: Interrupts can handle multiple events simultaneously, making them more scalable and adaptable to a wider range of applications.  
  
Deterministic behavior: Interrupts provide deterministic behavior, which means that the response time of the system can be accurately predicted and controlled.  
  
Disadvantages:  
  
Complexity: Programming using interrupts is more complex than other programming techniques, such as the super loop. This is because the programmer must design, test, and integrate the ISR into the system, which requires a higher level of expertise and attention to detail.  
  
Debugging: Interrupt-driven programming can be difficult to debug since the execution order of the tasks is not determined by the programmer. This can make it harder to isolate and fix bugs in the code.  
  
Overhead latency: Interrupts have an overhead latency, which is the additional time taken to perform tasks that are not directly related to their main code execution, such as context switching and interrupt handling. This overhead latency can impact the overall performance and responsiveness of the system.  
  
Unpredictability: The execution order of the tasks is determined by the events that trigger the interrupts, which can make the behavior of the system harder to predict and control. This can lead to unexpected or undesirable behaviors in the system.***

***Diagram

Description automatically generated***

***Making system using super loop and interrupts :***

***to achieve concurrent execution of tasks, you can also use interrupts to trigger the execution of specific tasks in response to external events or input signals.***

***To implement tasks with a super loop and interrupts, you can define each task as a separate function, and then register these functions as interrupt service routines (ISRs) for the corresponding interrupt sources. When an interrupt occurs, the ISR is automatically executed, allowing the associated task to be performed in a timely and efficient manner.***

***Here's a simple example of how to create tasks with a super loop and interrupts:***

***void task1(void) {***

***// Code for task 1***

***}***

***void task2(void) {***

***// Code for task 2***

***}***

***void isr1(void) {***

***// Call task 1***

***task1();***

***}***

***void isr2(void) {***

***// Call task 2***

***task2();***

***}***

***void main(void) {***

***// Register ISR 1 for interrupt source 1***

***register\_isr(isr1, INTERRUPT\_SOURCE\_1);***

***// Register ISR 2 for interrupt source 2***

***register\_isr(isr2, INTERRUPT\_SOURCE\_2);***

***// Enable interrupts***

***enable\_interrupts();***

***while (1) {***

***// Super loop***

***}***

***}***

***In this example, we have defined two tasks, task1() and task2(), and two ISRs, isr1() and isr2(), which are registered for the corresponding interrupt sources. When an interrupt occurs, the associated ISR is automatically executed, allowing the corresponding task to be performed.***

***Note that in this example, the super loop is empty, because the tasks are executed in response to interrupts. The super loop may still be used to perform other tasks that do not require interrupt-driven execution.***

***The combination of a super loop and interrupts provides a simple and efficient way to create tasks in an embedded system that require concurrent execution. However, it may not be suitable for more complex systems that require advanced scheduling or synchronization mechanisms. In such cases, a SquadRTOS may be a better option for managing tasks and ensuring system responsiveness.***

***Advantage:***

***Interrupts allow for a more complex programming model, which can handle a greater number of events and tasks, without overloading the processor. This can lead to more efficient and responsive systems, especially in larger and more complex embedded systems.***

***Disadvantage:***

***However, the increased complexity of interrupt-driven programming can also make it more difficult to debug and maintain the code. This is because the execution order of the tasks is determined by the events that trigger the interrupts, which can be unpredictable and harder to understand than the deterministic execution order of the super loop.***

***On the other hand, the super loop provides a simpler programming model that is easier to understand and maintain, especially for small and less complex systems. However, this simplicity comes at the cost of reduced efficiency and scalability, which can limit the performance and responsiveness of the system as it grows in complexity.***

***Diagram

Description automatically generated***

***Why are we made Real-Time Operating system (SquadRTOS vs Super loop and Interrupts )?***

***In the context of embedded systems development with SquadRTOS, real-time operating systems (RTOS) offer several advantages over super loops and interrupts. RTOS provides a preemptive multitasking kernel, allowing multiple tasks to run concurrently while ensuring that critical tasks are given priority over non-critical tasks. This makes it ideal for applications that require precise timing and responsiveness.***

***Super loops, on the other hand, can be simple to implement but may not be able to handle complex systems with multiple tasks. Interrupts can be used to handle events but may result in high system overhead and increased complexity.***

***SquadRTOS provides a highly efficient and reliable real-time operating system that offers minimal system overhead and simplified development. This includes features such as optimized algorithms, low-level hardware access, and full control over system resources.***

***The main differences between tasks in SquadRTOS and super loops and interrupts are as follows:***

***1. Private Stack (Tasks isolated ): Each task is assigned its own private stack, which is not shared with any other task in the system. This allows each task to have its own call stack without interfering with the execution of other tasks, unlike a super loop which shares the system stack.***

***2. Priority Assigned: Each task is assigned a priority, which enables the scheduler to make decisions on which task should be running at any given time. The goal is to ensure that the highest priority task in the system is always doing useful work.***

***Advantages :***

***Memory protection: Each task's private stack is protected from other tasks in the system, preventing memory corruption and other errors that can occur when multiple tasks share the same stack.***

***Improved robustness: By having its own stack, each task is less likely to be affected by stack overflows or other stack-related errors that can occur when multiple tasks share the same stack.***

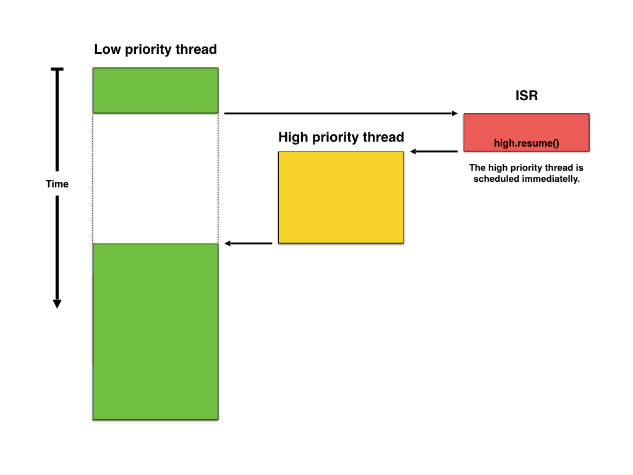
***Better context switching: Context switching between tasks is faster and more efficient when each task has its own stack. This is because the processor does not have to save and restore the entire stack for each task switch, but only the necessary parts of the stack for the current task.***

***Improved debugging: Debugging is easier when each task has its own stack, as it is easier to isolate and identify errors that occur within a specific task.***

***Overall, while super loops and interrupts can be useful for simple systems, SquadRTOS provides a more reliable and efficient platform for larger and more complex systems that require precise timing and responsiveness.***

***Super loop Tasks with priorities if task have Priority higher than current thread it will work***

***Chart, waterfall chart

Description automatically generated ***

# ***1.10 Kernel***

***- 1.10.1 Dispatcher and schedular***

***A dispatcher in a real-time operating system (SquadRTOS) is responsible for selecting which task or thread should execute next, based on scheduling policies and algorithms. The dispatcher is a key component of the SquadRTOS kernel and plays a critical role in ensuring that real-time tasks are executed in a timely and predictable manner.***

***The dispatcher works in conjunction with the scheduler, which is responsible for managing the allocation of system resources, such as CPU time, memory, to different tasks or threads. The scheduler uses scheduling policies and algorithms to determine which task or thread should execute next, based on factors such as task priority, deadline, and available resources. The dispatcher then selects the next task or thread to execute, based on the scheduling decision made by the scheduler.***

***The dispatcher typically operates at a lower level than the scheduler ,and is responsible for performing context switches between tasks or threads. When the dispatcher selects a new task or thread to execute, it saves the context of the current task or thread and restores the context of the selected task or thread. Context switching involves saving and restoring the contents of CPU registers, program counter, stack pointer, and other state information, and can have a significant impact on system performance.***

***In a typical SquadRTOS, the dispatcher uses a priority-based scheduling algorithm, where tasks or threads with higher priority are executed first. The dispatcher may also use other scheduling policies, such as round-robin scheduling or earliest deadline first (EDF) scheduling, depending on the specific requirements of the system or application.***

***The dispatcher must be designed to respond to scheduling decisions in a timely and predictable manner, to ensure that real-time tasks are executed in a timely and reliable manner. The dispatcher must also be designed to handle interrupts and other asynchronous events, which can affect scheduling decisions and require immediate attention.***

***The dispatcher is a crucial part of real-time operating systems, and it is imperative to carefully design and implement it to guarantee the timely and predictable execution of real-time tasks. The dispatcher works in tandem with the scheduler and other components of the RTOS kernel to ensure that real-time systems fulfill their performance and reliability objectives***.

Text

Description automatically generated with medium confidence

- 1.10.2 Schedular:

***The scheduler is a component that manages the allocation of system resources, such as CPU time, memory, and I/O devices, to different tasks or threads with real-time requirements.***

***Schedular responsibility:***

***The scheduler is responsible for ensuring that tasks or threads are executed in a timely and predictable manner, according to their criticality and deadline requirements.***

**S*quadRTOS Schedular overview:***

***SquadRtos schedulers typically use scheduling policies and algorithms that prioritize tasks based on their criticality and deadline requirements. For example, tasks with hard real-time requirements, such as control tasks in a robotic system, may be given higher priority than tasks with soft real-time requirements, such as user interface tasks.***

***Real-time schedulers may also include features such as preemption, where a higher-priority task can interrupt a lower-priority task, and interrupt handling, where the scheduler can respond to hardware or software interrupts in a timely manner.***

***Real-time schedulers are critical components of real-time systems, where tasks or threads must be executed in a predictable and reliable manner, often with strict timing requirements. By managing the allocation of system resources and prioritizing tasks based on their criticality and deadline requirements, real-time schedulers can ensure that real-time systems meet their performance and reliability goals.***

***Preemptive-based:***

Preemptive-based scheduling with round robin is a type of CPU scheduling algorithm that combines the features of both preemptive and round-robin scheduling.

Preemptive scheduling means that the operating system can interrupt a running process and switch to another process at any time, based on priority or other scheduling criteria. This allows higher-priority processes to execute before lower-priority processes, which can improve system responsiveness and reduce wait times.

Round-robin scheduling is a time-sharing algorithm that assigns a fixed time slice to each process in a circular queue. When the time slice expires, the current process is interrupted and moved to the back of the queue, and the next process is executed. This ensures that each process gets a fair share of the CPU time and prevents any single process from monopolizing the CPU.

In preemptive-based scheduling with round robin, the time slice assigned to each process is smaller than in pure round-robin scheduling, typically in the range of a few milliseconds. This allows the operating system to switch between processes more frequently, based on their priority or other scheduling criteria. If a higher-priority process becomes available, the operating system can preempt the current process and switch to the higher-priority process immediately, ensuring that critical tasks are completed as soon as possible.

Preemptive-based scheduling with round-robin is a versatile and effective algorithm that strikes a balance between the requirements of multiple processes and the timely execution of critical tasks.

Advantages and disadvantages of both preemption round-robin scheduling:

Advantages:

Both preemption and round-robin scheduling can provide fair access to system resources among all tasks, preventing starvation.

They can both be efficient and effective in meeting the performance and timing requirements of different types of tasks.

They can both be combined with other scheduling techniques to create hybrid scheduling algorithms that take advantage of the strengths of multiple techniques.

Disadvantages:

Both preemption and round-robin scheduling can increase system overhead and complexity by adding scheduling and context-switching overhead.

They can both result in priority inversion and race conditions, which can lead to missed deadlines or system failure.

They may not be suitable for all types of systems or applications and may require careful consideration of the specific requirements of the system.

In summary, the selection of a scheduling technique, whether preemption or round-robin, should be based on the particular needs of the system or application. By thoughtfully weighing the pros and cons of each technique, and exploring the possibility of hybrid scheduling algorithms, the system can be optimized to achieve its performance and reliability objectives while minimizing unnecessary overhead and complexity.

# ***1.11 Interrupt handling:***

***Interrupt handling is a critical component of SquadRTOS, a real-time operating system, that is responsible for managing the handling of hardware and software interrupts. Interrupts are signals from hardware devices or software events that temporarily halt the normal execution of a task, allowing the system to respond to the interrupt. Interrupt handling involves several components, including the interrupt service routine (ISR), interrupt controller, and interrupt dispatcher.***

***The interrupt service routine (ISR) is a small piece of code that is executed when an interrupt occurs. The ISR is responsible for handling the interrupt, performing any necessary operations, and returning control to the interrupted task. The ISR is typically designed to be as small and efficient as possible, with minimal overhead, to ensure that it can respond quickly to interrupts.***

***The interrupt controller is a hardware component that is responsible for managing the flow of interrupts in the system. The interrupt controller receives interrupts from hardware devices and software events and determines which ISR should be executed in response to the interrupt. The interrupt controller can also prioritize interrupts based on their importance and can mask or disable interrupts to prevent interference with critical operations.***

***The interrupt dispatcher is a component that manages the scheduling of interrupt service routines. The dispatcher determines the priority of each interrupt and schedules the corresponding ISR to run based on the interrupt priority. The dispatcher is responsible for ensuring that the system responds to interrupts in a timely and efficient manner, without introducing delays or conflicts.***

***Interrupt handling can introduce overhead and complexity into the system, as interrupts must be handled quickly and efficiently to ensure that the system remains responsive and reliable. Interrupt handling can also introduce synchronization issues, as multiple tasks may attempt to access shared resources concurrently. To mitigate these issues, SquadRTOS provides a range of features and mechanisms to manage interrupt handling, including interrupt priorities, interrupt masking, and interrupt nesting.***

***Interrupt priorities allow the system to prioritize interrupts based on their importance, ensuring that critical events are handled quickly and efficiently. Interrupt priorities are typically assigned based on the type of interrupt and the importance of the event that triggered the interrupt. For example, a system may assign a higher priority to interrupts from critical hardware devices, such as a watchdog timer or a real-time clock, than to interrupts from less critical devices, such as a keyboard or a mouse.***

***Interrupt masking allows the system to temporarily disable interrupts to prevent interference with critical operations. Interrupt masking is typically used when a critical operation, such as a system update or a data transfer, is in progress, and interrupts could cause the operation to fail or introduce errors. Interrupt masking can also be used to ensure that interrupts are handled in a specific order, to prevent conflicts or race conditions.***

***Interrupt nesting is a feature that allows the system to handle multiple interrupts simultaneously or in rapid succession. Interrupt nesting allows the system to prioritize and schedule interrupts based on their importance and to avoid conflicts or delays. Interrupt nesting can be challenging to implement and requires careful design and testing to ensure that it is reliable and efficient.***

***Interrupt handling is also important for ensuring the safety and security of the system. Interrupt handling can prevent errors and failures caused by hardware or software events and can detect and respond to security threats or attacks. Interrupt handling can also provide diagnostic information and error messages that can be used to identify and resolve system issues.***

***In summary, interrupt handling is a complex and critical aspect of operating system design, requiring careful consideration of system requirements, hardware capabilities, and software implementation. By effectively managing interrupts, SquadRTOS can provide fast, reliable, and efficient responses to external events and requests, enabling a wide range of applications and use cases. Interrupt handling is also essential for ensuring the safety and security of the system and can provide valuable diagnostic information and error messages.***

***- 1.11. Memory Management :***

***Memory management is an essential component of any operating system that is responsible for managing the allocation, utilization, and release of system memory. The operating system has to manage the limited memory resources available in the system and allocate these resources efficiently to support the execution of multiple programs simultaneously.***

***Memory management involves various tasks, such as memory allocation, memory protection, memory swapping, and garbage collection. Effective memory management ensures that the operating system can run multiple programs simultaneously without running out of memory or causing memory-related errors.***

***Memory allocation is the process of assigning memory to programs or processes. The operating system must provide a mechanism for allocating memory dynamically as processes are created, and freeing up memory when processes are terminated. Memory allocation can be done using various algorithms, such as first-fit, best-fit, and worst-fit, depending on the size of the available memory and the requirements of the processes.***

***Garbage collection is the process of reclaiming memory that is no longer being used by a program or process. This is accomplished by identifying memory that is no longer referenced by the program and freeing it up for use by other processes.***

***Effective memory management can improve overall system performance by reducing the amount of time spent managing memory resources and minimizing the risk of memory-related errors, such as memory leaks and segmentation faults. Memory leaks occur when a program fails to release memory when it is no longer needed, leading to a gradual loss of available memory and eventually causing the system to crash. Segmentation faults occur when a program attempts to access memory that it is not authorized to access, leading to system instability and potential data loss.***

***Memory management is particularly important in real-time systems, where response time and predictability are critical. In real-time systems, memory allocation and deallocation must be performed in a predictable and deterministic manner to avoid delays and ensure that critical processes have sufficient memory resources available.***

***In summary, memory management is a critical component of any operating system that is responsible for managing the allocation, utilization, and release of system memory. Effective memory management can improve overall system performance and stability, minimize the risk of memory-related errors, and ensure that critical processes have sufficient memory resources available. By providing efficient and reliable memory management mechanisms, operating systems can support the execution of multiple programs simultaneously, enabling a wide range of applications and use cases.***

# ***Communication stack***

***The Lightweight Internet Protocol (lwIP) stack is a free and open-source networking stack that is widely used in embedded systems and real-time operating systems (RTOSs). It provides a range of networking protocols and services, including IP, TCP, UDP, HTTP, and DNS, and is designed to be highly configurable, efficient, and portable.***

***lwIP was developed by Adam Dunkels in 2001 and has since become one of the most popular networking stacks for embedded systems and RTOSs. The stack is optimized for low memory usage and low processing overhead, making it well-suited for use in resource-constrained systems, such as microcontrollers and embedded devices.***

***lwIP stack provides a range of networking protocols and services, including IP, TCP, UDP, HTTP, and DNS. It is designed to be highly configurable, allowing developers to customize the stack to meet their specific requirements. The stack can be configured to support a range of network interfaces, including Ethernet, Wi-Fi, and cellular, and can be adapted to work with various hardware platforms and architectures.***

***One of the key features of lwIP is its support for zero-copy networking, which allows data to be transferred between network interfaces and application buffers without needing to copy the data between memory locations. This can significantly reduce the processing overhead and memory usage associated with network communication, making it well-suited for use in embedded systems and RTOSs.***

***lwIP also supports a range of advanced networking features, such as Quality of Service (QoS), Multicast, and IPv6. These features can help to optimize network performance and improve the reliability and security of network communication.***

***Another important feature of lwIP is its support for thread-safe operation, which allows multiple threads to access the networking stack simultaneously without causing conflicts or synchronization issues. This can be particularly important in real-time systems, where multiple tasks may need to access the networking stack concurrently.***

***lwIP is designed to be highly portable and can be ported to a wide range of hardware platforms and architectures. It provides a range of platform-specific APIs, allowing developers to integrate the stack with their specific hardware platform and operating system.***

***lwIP is a highly configurable, efficient, and portable networking stack that is well-suited for use in embedded systems and real-time operating systems. It provides a range of networking protocols and services, including IP, TCP, UDP, HTTP, and DNS, and supports advanced networking features such as QoS, Multicast, and IPv6. Its support for zero-copy networking and thread-safe operation can significantly reduce the processing overhead and memory usage associated with network communication, making it an ideal choice for resource-constrained systems.***

***In conclusion, the lwIP stack is a highly versatile and efficient networking stack that is well-suited for use in embedded systems and real-time operating systems. Its support for a wide range of networking protocols and services, advanced networking features, and zero-copy networking and thread-safe operation make it an ideal choice for resource-constrained systems. With its large and active development community and extensive documentation, lwIP is sure to remain a popular choice for networking in embedded systems and RTOSs for many years to come.***

***To implement the website and send HTTP requests, we first created a web server using lwIP. We configured the stack to support the HTTP protocol and set up a socket to listen for incoming HTTP requests. When a request was received, we used lwIP's APIs to parse the request and generate an appropriate response. We also implemented a simple web page using HTML and CSS to serve as the website.***