

POLITECNICO DI TORINO

Master of Science in Mechatronic Engineering



**Politecnico
di Torino**

ACADEMIC YEAR 2022-2023

Implementation of an Embedded RTOS FW platform for data transceivers on 2G / 3G / 4G / 5G mobile network

Academic Supervisor:
Prof. Massimo VIOLANTE
W.A.Y srl Supervisor:
Ing. Enrico CARTA

CANDIDATE:
Klea META

*To everyone that has
made this entire
journey a bit less stressful
and a bit more joyful.
Thank you.*

Abstract

Vehicles are becoming more complex and technically advanced because of the automotive industry's ongoing evolution. Modern cars heavily rely on specialized computers and embedded systems built for certain tasks that enable a variety of features like engine control, powertrain management, advanced driver assistance systems (ADAS), and infotainment systems. To ensure safety, performance, and user experience, these embedded systems must function with precise nature, resource efficiency, and component coordination. Such embedded systems require the use of real-time operating systems (RTOS), which offer essential features like task scheduling, inter-task communication, memory management, and device drivers to provide safe and deterministic operation.

The scope of this thesis is the implementation and testing of a FW platform, CPU independent, based on an embedded Real Time Operating System. The platform will be used on the hardware devices designed by the company that implement the transmission of operational and diagnostic data of the vehicles using the on-board CAN Bus, GPS constellations and mobile internet networks. This project was carried out while working on my internship and it was divided on three steps:

- 1) the installation of FreeRTOS on the development station
- 2) the studying of the current FW architecture of the company's Mobile Terminal Systems,
- 3) writing in 'RTOS oriented' function some of the basic modules of the transceiver: management of the RS232 serial, 2G modem and multi-constellation GPS
- 4) the implementation of the operation of the Mobile Terminal System with state machine using the new modules with tasks, queues, messages and semaphores.

The thesis will start with an overview of real time operating systems and embedded systems in the automotive industry, including their applications, challenges, and trends. Next, the fundamental concepts of FreeRTOS, including task scheduling, inter-task communication, and memory management, will be discussed in detail.

We will then delve into the work done on the project, starting by introducing the instrumentation used and how they came in hand throughout the entire project. Then we will dive into the technical part which is the writing of the firmware for the above-mentioned modules, GPS, Modem and the serial output.

Finally, the thesis will conclude with a detailed explanation of the migration from one CPU to another and the conclusions reached after hitting our goals.

The programming world merged with embedded systems is as complex as fascinating. I have grown up so much academically on this project and I hope I will be able to transmit even a bit of this knowledge to you as well.

Acknowledgments

Abstract

1. Theoretical Background

- 1.1 Real time systems
- 1.2 Embedded systems
 - 1.2.1 Characteristics of an Embedded System
 - 1.2.2 Basic Structure of an Embedded System
- 1.3 Operating systems
- 1.4 RTOS – Real Time Operating Systems
 - 1.4.1 RTOS Architectures
- 1.5 FreeRTOS
 - 1.5.1 The Advantages of FreeRTOS
 - 1.5.2 Tasks
 - 1.5.3 Queues
 - 1.5.4 Timers
 - 1.5.5 Semaphores and mutexes
 - 1.5.6 Events

2. Enviromental analysis

- 2.1 Software Tools
 - 2.1.1 Linux Ubuntu
 - 2.1.2 Visual Studio Code
 - 2.1.3 C Language
- 2.2 Instrumentation used.
 - 2.2.1 Oscilloscope
 - 2.2.2 DC power supply
 - 2.2.3 JTag
 - 2.2.4 Microcontroller GD32F305

3. GPS Task

- 3.1 UART GPS Handler
- 3.2GPS Pin Init
- 3.3GPS Task function

4. Serial Task

- 4.1 UART Console Handler
- 4.2 Serial Task Function
- 4.3 VCOM_getChar Function

5. Modem Task

- 5.1UART Modem Handler Function
- 5.2 ModemInitPPP Function
- 5.3Modem Init Function
- 5.4 ModemInitLCP Function
- 5.5Modem Task Function
- 5.6Modem Send Command Function

5.7 Modem Debug Command Function

6. Migrating from the LPC17xx to the GD32F305 microcontroller

6.1 CPU_SerialInit

6.2 CPU_UartIntInit Function

6.3 Vuart HANDLER Functions

6.4 SerialGetChar Function

6.5 vGetChar Function

6.6 SerialPutChar Function

6.7 vPutChar Function

6.8 ModemInit Function

6.9 GPSInit Function

7. Conclusions

Bibliography

Figures Bibliography

CHAPTER 1

Theoretical Background

1.1 Real time systems

"Real-time systems (RTS) have been developed and have grown in demand in the market especially in industrial environments." [1]

They are considered to be systems whose behaviour depends on the time elapsed, since they start processing data until the outputs are known. Real time operation is characterised by the explicit involvement of the dimension time, manifesting itself in the fundamental timeliness requirement, which real time systems must fulfil even under extreme load conditions, therefore the response time of these systems must be predictable and limited. Only fully deterministic system behaviour will ultimately enable an effective safety licensing of computers for safety critical applications.

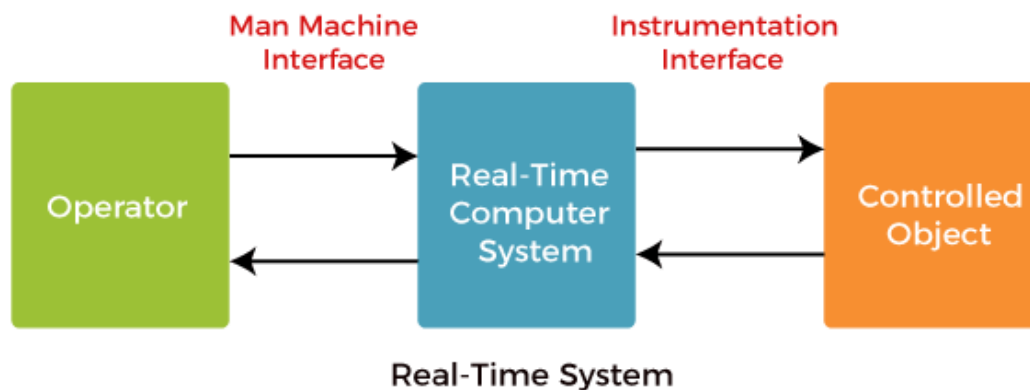


Figure 1.1 Real time system scheme

All across our economic, government, military, healthcare, academic, and cultural infrastructures, real-time systems are used often and in a wide variety of ways. Included are:

- vehicle systems for automobiles, subways, aircraft, railways, and ships
- traffic control for highways, airspace, railway tracks, and shipping lanes
- process control for power plants, chemical plants, and consumer products such as soft drinks and beer
- manufacturing systems with robots
- telephone, radio, and satellite communications
- multimedia systems that provide text, graphic, audio, and video interfaces

RTS can be divided in a number according to the fulfilment of the time restrictions:

- Critical real-time systems
- Non-critical real-time systems

Critical real-time systems

“Critical RTS are systems whose scheduling deadline must be met every single time.” [1] Missing the deadline means the system has failed and there is a possibility of facing fatal consequences. These systems also called **hard real-time systems**. Some examples of these are earthquake alerts, military tactical training, kidnapping, and rescue systems, etc.

Non-critical real-time systems

Non-critical RTS are systems on the other hand, can tolerate missing a deadline occasionally if an average latency is maintained. The scheduling deadline is considered to be more of a goal. A missed deadline from them may lower the quality of service provided but is not catastrophic. They are also known as soft real time systems.

Another classification of real-time systems is given depending on the number of processors that are used for the execution of their tasks. Some types of RTS are:

Conventional real-time systems: systems in which a single processor handles all of their tasks.

Distributed real-time systems: systems where the tasks are running on different processor. “In addition, these systems can be strongly coupled when processors are working with the same operating system.” [1] A few properties of these real-time distribution systems are concurrent systems, reagent systems, systems that perform in challenging environments (such as those with high temperatures or noise control), and systems that are essentially reliable and fault-tolerant.

Efficiency overview

“Real-time systems are time critical, therefore their implementation efficiency is more important than in other systems.” [1] Efficiency can be classified into the following groups: power, memory, and processor cycles. This limitation can influence everything from the choice of processor to the choice of the programming language. One of the main benefits of using a higher-level language is to allow the programmer to abstract away implementation details and concentrate on solving the problem. This is not always true in the embedded system world. Some higher-level languages have instructions that can be an order of magnitude slower than assembly

language. However, if the right techniques are used, higher level languages can be used in real-time systems in an effective way.

Resource management

A system is said to operate in real time as long as it completes its time-critical processes in a reasonable timeframe which include behavioural or “non-functional” requirements for the system that must be objectively quantifiable and measurable. “A system is said to be real-time if it contains some model of real-time resource management which must be explicitly managed for the purpose of operating in real time.” [1]

The degree to which a system is required to operate in real time cannot necessarily be attained solely by hardware over-capacity (such as, high processor performance using a faster CPU). To be cost effective, there must exist some form of real-time resource management. Systems that must operate in real time consist of both real-time resource management and hardware resource capacity and the ones that have interactions with physical devices require higher degrees of real-time resource management. Many of the embedded computers use very little real-time resource management which is usually static and requires analysis of the system prior to it being executed in its environment. In a real-time system, physical time is necessary for real-time resource management to link the events with their exact moment occurrence and also for limiting the amount of time that actions can take, as well as for monitoring the costs incurred as processes get developed.

All real-time systems make trade-offs of scheduling costs vs. performance in order to reach an appropriate balance for attaining acceptable timeliness between the real-time portion of the scheduling optimization rules and the offline scheduling performance evaluation and analysis.

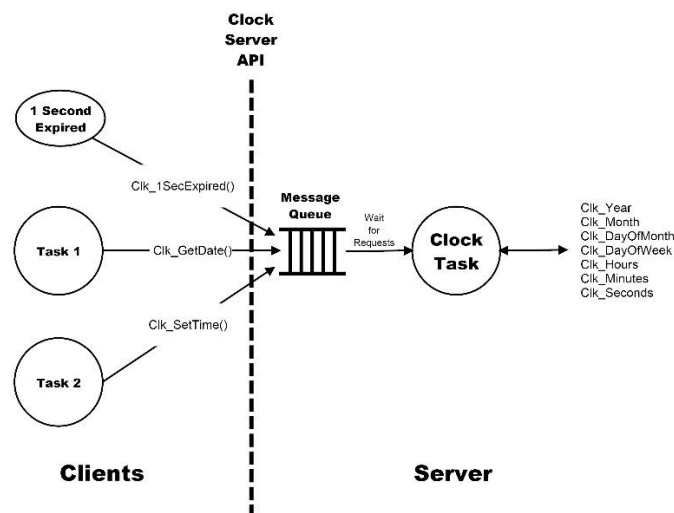


Figure 1.2 Shared resource management

1.2 Embedded Systems

“Real time systems are also classified in two, based on their relationship with the environment they’re in *reactive and embedded*.” [1]

A reactive real-time system is one that has constant interaction with its environment (such as a pilot controlling an aircraft) while an embedded real-time system is used to control specialized hardware that is installed within a larger system (such as a microprocessor that controls anti-lock brakes in an automobile).

“An embedded system can be thought of as a computer hardware system having software embedded in it.” [2] As the title indicates, the term "embedded" refers to something that is joined to another thing. An embedded system can be independent part of a large system. It is a system based on microcontrollers or microprocessors which is designed to perform a specific task. For example, a fire alarm is an embedded system, and its only purpose is to sense only smoke.

An embedded system consists of three elements.

- Hardware component
- Application software.
- Real Time Operating system (RTOS) which we will talk about later.

1.2.1 Characteristics of an Embedded System

- **Single-functioned** – Embedded systems typically perform repetitively the same operation they are specialised in. For example: A calculator always works as a calculator.
- **Tightly constrained** – The constraints on the embedded systems are very tight. “Its implementation features such as the size, power, cost and performance are measured through metrics which should be of a size to fit on a single chip, must perform quick enough to process the data in real time and consume minimum power to extend the battery’s life.” [2]
- **Reactive and Real time** – Many embedded systems must continuously react to changes happening in their environment and compute specific results in real time without any delay. For example let’s consider the insulin pump which is a medical device that should continuously monitor the patients’ blood sugar levels and respond immediately to the patients change of glucose level by adjusting the insulin dosage amount in real time.

- **Microprocessors based** – The system must be based on microprocessors or microcontrollers.
- **Memory** – Since the system's software usually embeds in ROM, it must also have a memory, therefore it will not need any secondary memories in the computer.
- **Connected** – It must have connected peripherals to connect input and output devices.
- **HW-SW systems** – An embedded system is made of a combination of HW and SW which are used for performance and security and providing more features and flexibility respectively.

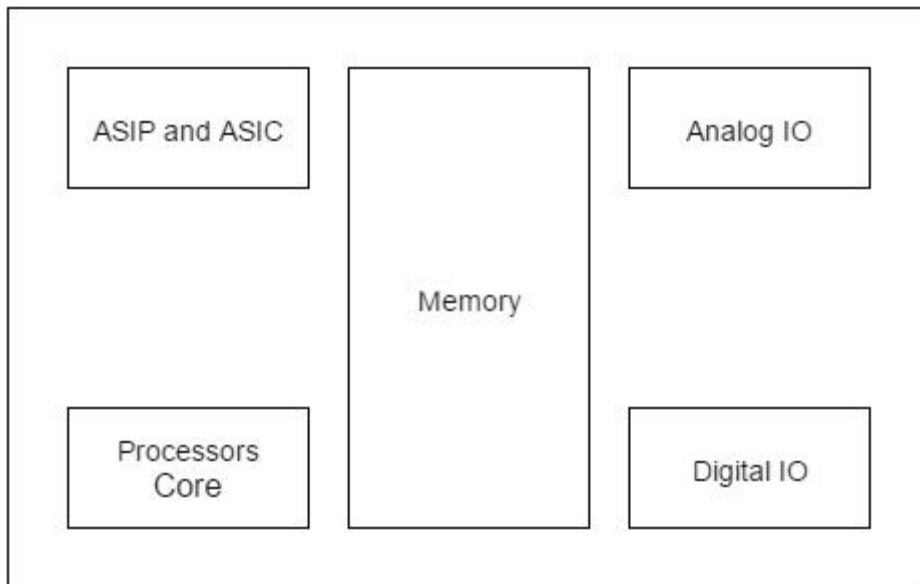


Figure 1.2.1 Composition of embedded systems [3]

Some of the advantages and disadvantages of an embedded system include:

Advantages:

- Low power consumption
- Low cost
- Enhanced performance
- Easily customizable

Disadvantages:

- High development effort
- Larger time to market.

1.2.2 Basic Structure of an Embedded System

The basic structure of an embedded system is reported as below:

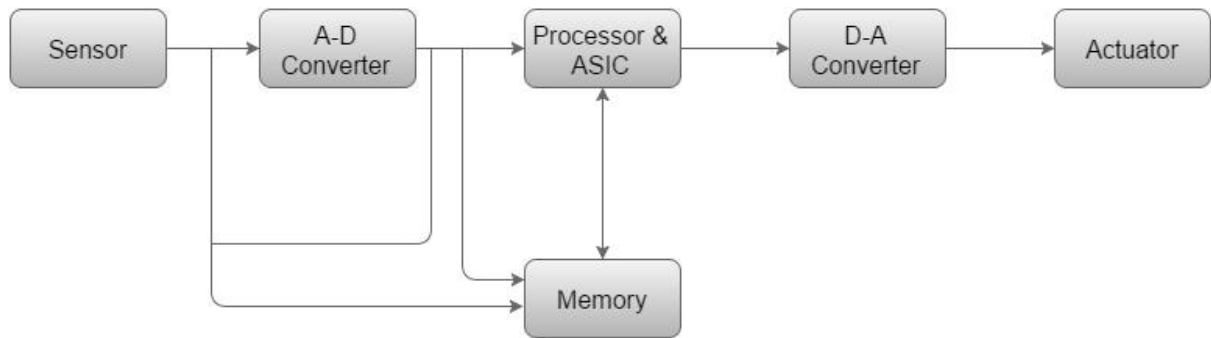


Figure 1.2.2 Structure of an embedded system

- **Sensor** – A sensor measure the physical quantity and transforms it into an electrical signal which can then be read by an electronic instrument like an A2D Converter then stores the measured quantity into the memory.
- **D-A Converter** – The digital data fed by the processor is transformed to analog data by a digital to analog converter.
- **A-D Converter** – The analog signal which is sent by the sensor is transformed into a digital signal by an analog to digital converter
- **Actuator** – An actuator stores the approved output after comparing the D-A Converter's output to the actual (supposed) output that was previously stored in it.
- **Processor & ASICs** – The data is processed by the processor in order to measure the output and have it stored in the memory.

1.2.3 Processors in a System

“Processor is the main part of an embedded system. It is the basic unit that takes inputs, processes the data, and produces an output afterwards. A designer of embedded systems must have the knowledge about both microprocessors and microcontrollers.” [2]

A processor has two essential units:

- Execution Unit (EU)
- Program Flow Control Unit (CU)

“The EU has circuits that implement the instructions pertaining to data transfer operation and data conversion from one form to another.” [2] It includes the Arithmetic and Logical Unit (ALU) and also the circuits that execute instructions for a program control task such as interrupt or jump to another set of instructions.

The CU includes a unit for fetching instructions from the memory. The processor runs the cycles of fetch and executes the instructions in the same sequence as they are fetched from memory.

Types of Processors

Processors are divided in a few categories shown above, we will describe more in details the Microcontroller and the Microprocessor ones.

- General Purpose Processor (GPP)
- Microcontroller
- Media Processor
- Microprocessor
- Digital Signal Processor
- Embedded Processor

Microprocessor

A microprocessor is a single VLSI chip which has a CPU. “Additionally, it can also have other units such as caches, floating point processing arithmetic unit, and pipelining units that help the instructions being processed faster.” [2]

Earlier generation microprocessors’ fetch-and-execute cycle was guided by a clock frequency whose order was ~ 1 MHz while nowadays they operate at a clock frequency of 2GHz.

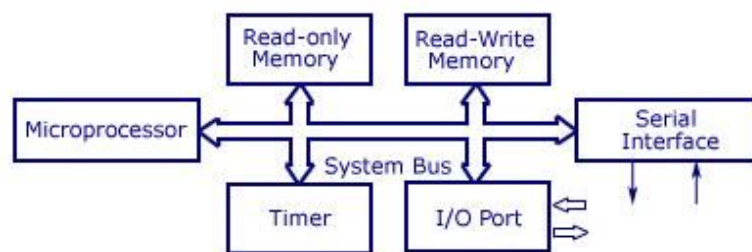


Figure 1.2.3 Schematic arrangement of Microprocessor based system

Microcontroller

A microcontroller is a single-chip VLSI unit (also called **microcomputer**) which, possesses enhanced input/output capability and a number of on-chip functional units, although it has limited computational capabilities,

Microcontrollers are particularly used in embedded systems for real-time control applications with on-chip program memory and devices.

1.3 Operating Systems

An operating system (OS) acts as the link between computer hardware and users. Its purpose is to perform basic tasks like memory, process and file management, as well as handling input and output, and controlling disk drives and printers also known as peripheral devices. Some of the most well-known Operating Systems are Linux, Windows, VMS, OS/400, AIX, z/OS, etc. In the upcoming chapter we will discuss about what Operating system we used in our project and why.

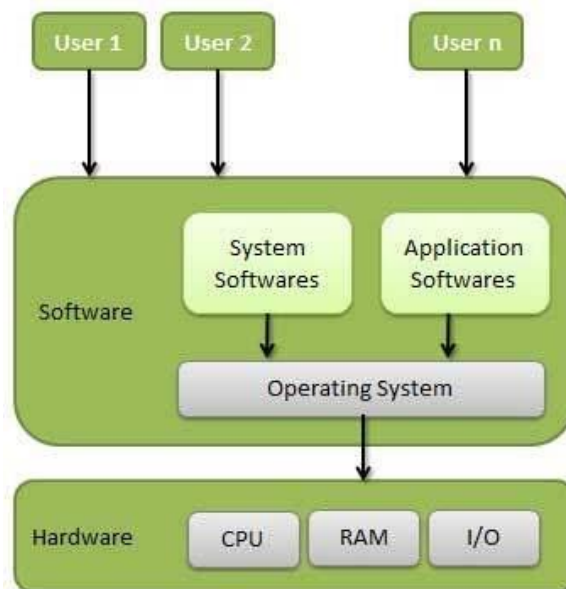


Figure 1.3.1 Architecture of an operating system

Amongst operating system's crucial characteristics are we can mention:

Memory Management which relates to the management of Primary Memory or Main Memory. It is a large array of words or bytes where each has its unique address. Main memory has a fast storage that can be accessed directly by the CPU and is where the program is located if it has to be executed.

Processor Management which relates to the Operating system decision of the time each process stays in the processor. Through the traffic controller the OP keeps track of the processor's status and deallocates it when a process is no longer needed

Device Management which relates to the OS managing the communication of devices via their respective drivers by keeping track of them and deciding which process gets the device and for how much time.

File Management which relates to the OS allocating or de allocating the sources and deciding who gets them as well as keeping track of the files information, their position, status, etc.

1.4 Real Time Operating Systems

Real time embedded systems are common and have real time operating systems as their fundamental elements; they are essentially identical to embedded operating systems, only differing in a few specific features due to the requirements of the project. RTOS is implied in embedded systems that require timing,

The most important factor is time limitations; if a task is finished after its due date or deadline, it will either be useless or have a negative effect on the consumers.

The real-time operating system's time maintenance component is crucial; tasks are identified and prioritized based on their respective deadlines.

The diagram below shows the elements of a real time embedded system.

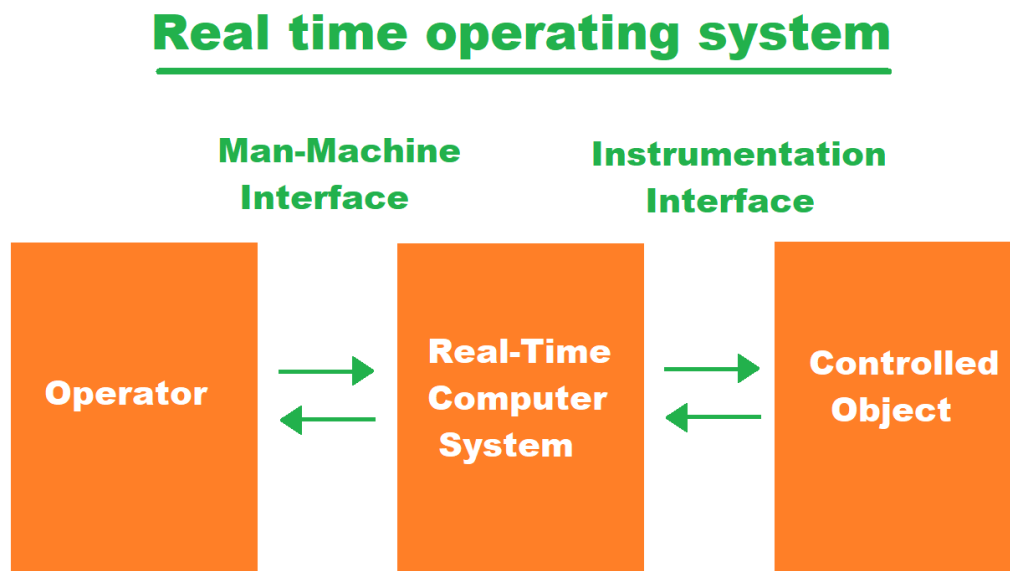


Figure 1.4.1 Elements of a real time embedded system

A real time operating system has many **advantages**. They allow more output to come from all the resources as they maximize the utilization of a system or device and also take way less time for shifting the tasks, while in older systems it used to be around 10 microseconds, now thy take around 3 microseconds to do so. They are more trust worthy as they focus on running the current application and put less importance on the ones that are in the queue and are also considered to be error free. Lastly, they manage the memory allocation better than the OS.

While it is true that RTOS have many advantages, they also have some **drawbacks** such as them having more limited tasks running simultaneously as they concentrate on very few applications to avoid generating errors. Their algorithms can also be very

complicated and hard for the firmware engineers to write on as well as their resources can be quite expensive. Lastly, they perform a minimum number of task switching therefore they don't possess good thread priority as well

1.4.1 RTOS Architectures

“RTOS architectures are a very important element as they affect the reliability of an embedded system and its ability to recover from faults.” [3] There are two RTOS architectures: *monolithic* and *microkernel*.

MONOLITHIC RTOS

Monolithic meaning is “one huge stone.” A monolithic kernel, by definition, executes all parts of the operating system in kernel. For instance, the kernel space of a monolithic RTOS includes device drivers, file management, networking, and a graphics stack while on the other side applications operate in the user space. A single programming error in a file system, protocol stack, or driver can cause the system to crash, even when running user applications like memory-protected processes for example, shield a monolithic kernel from erroneous user code. Moreover, modifying the OS and recompiling it, is required for any change to a driver or system file. “In a monolithic Operating System, a single programming error in a file system protocol stack or driver can crash the whole system.” [3]

MICROKERNEL RTOS

A microkernel RTOS is structured with a tiny kernel that provides the most basic and essential services such as memory management, process management, and inter-process communication. It lacks file systems and many other services which are usually part of an Operating System. Since for such architecture the modularity is the key, the small size comes unavoidably as a side effect.

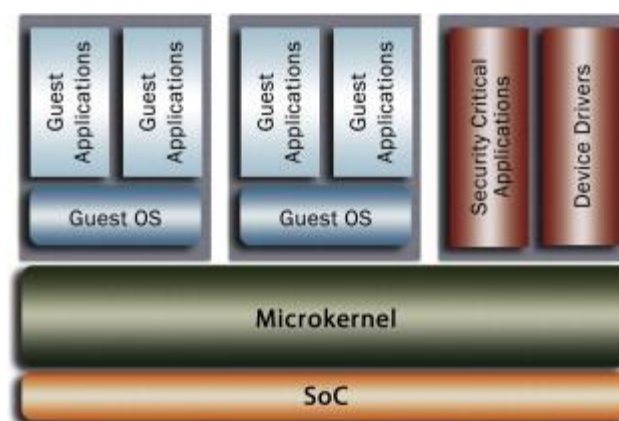


Figure 1.4.2 Microkernel-based architecture

In a microkernel the reliability and security are much more improved as only the core RTOS kernel is the one to have access to the entire system. “The microkernel protects and allocates memory for other processes and provides task switching.” [3] All other components, including drivers and system-level components, are each contained within their own isolated process space, which then prevents errors in a single component to affect the other parts of the system. Such crashes can be also easily found, and the defective component can be quickly restarted while the system is still working (hot restart), resulting in a minimal impact on performance.

1.5 FreeRTOS

FreeRTOS is an open source, real-time operating system for microcontrollers and microprocessors needed to perform tasks deterministically. It has IoT reference immigration and its suitable for over 40 different architectures which were originally developed by Richard Barry that makes it simple to deploy, secure, connect, and manage small, low-power edge devices. It has a kernel and an expanding collection of software libraries appropriate for use in various industry fields and applications. And it is the RTOS we chose to work with while developing this project.



Figure 1.5 FreeRTOS logo

FreeRTOS is intended to be minimal and clear. To make it simple to port and maintain, it is primarily written in the C computer language although when needed, it also includes a few assembly language functions, primarily in scheduler routines that are specific to a given architecture. Along with mutexes, semaphores, and software timers, it offers ways for managing many threads or processes and permits the parallel operation of many tasks. FreeRTOS is designed to run on Microcontrollers and has a scheduler that divides the time between tasks and ensures that each one has enough processing time based on its priority which is assigned by us. Its simple rule is that higher priority tasks will receive preferential treatment. The advantage of FreeRTOS is that it can be used for boards that have two cpu cores or even just one and the scheduler will wrap the switch between tasks giving you the impression that things run in parallel.

FreeRTOS applications are statically allocated, while objects can also be dynamically allocated with five schemes of memory management :

- allocate only.
- allocate and free with a very simple, fast, algorithm.

- a more complex but fast allocate and free algorithm with memory coalescence;
- an alternative to the more complex scheme that includes memory coalescence that allows a heap to be broken across multiple memory areas.
- and C library allocate and free with some mutual exclusion protection.

Device drivers, complex memory management, and user accounts are examples of more sophisticated features that are normally available in operating systems like Linux and Microsoft Windows but are not usually included in RTOS as its compactness and execution speed are mainly prioritized. Although it has a command line interface and POSIX-like input/output (I/O) architecture, we can best think of FreeRTOS as a thread library rather than an operating system.

“FreeRTOS implements many threads, by having the host program call a thread tick method on a regular basis at brief intervals.” [4] By using a round-robin scheduling system and based on job priority, the thread tick mechanism alternates between tasks. The typical interval is 1 to 10 milliseconds, but this interval is frequently altered to suit a particular application.

1.5.1 The Advantages of FreeRTOS

There are several advantages of FreeRTOS and here we will discuss about some of them. Firstly, with its **simplicity** we can refer to the whole OS as a scheduler for threads also known as tasks plus a TCP/IP stack.

Secondly, as FreeRTOS is a real-time operating system, the system developer may guarantee that the processor will handle time-critical events within precisely defined response times.

The work done there is excellent. There are three files that make up the kernel, and together they form a great team: Each line of code has been simplified to only what is absolutely essential so there isn't much that can go wrong as a result. Also, all of the .c files for the entire OS are present in your development tree, making it quite simple to understand if you need to dive into the kernel code to fix an issue.

The size of FreeRTOS is tiny, which pairs perfectly with its simplicity we mentioned in the beginning. So many small microprocessors come with just a little amount of RAM and flash drive. As FreeRTOS is so small, it can significantly reduce your cost of goods. Working on these tiny MCUs is enjoyable when you can save a few bytes since there's never enough room.

FreeRTOS is fully **open source** as well as widely used. Almost any tiny mainstream MCU can run it. That's useful when launching a product since it gives you some hardware independence. Developing your code on top of FreeRTOS makes it slightly more adaptable, which in turn makes you slightly less dependent on the chip manufacturer you have chosen.

As we mentioned before, “FreeRTOS is supplied as a set of C source files”. [5] Some of the source files are common to all ports, while others are specific to a port. All we need to do is build the source files as part of the project we’re developing to make the FreeRTOS API available to our application. “Each official FreeRTOS port is provided with a demo application which is pre-configured to build the correct source files, and include the correct header files.” [5]

A header file called *FreeRTOSConfig.h* is used to configure FreeRTOS for use in a particular application. It includes variables like *configUSE_PREEMPTION*, for instance, whose setting determines whether the pre-emptive or cooperative scheduling algorithm will be used.

All the FreeRTOS ports' source code as well as the project files for all its demo programs are included in a zip file.

The c files for the tasks, list, queue, timers, event groups, and c procedure are located in the source directory. Each of these C files has features that we will discuss in more detail later.

“To make FreeRTOS as easy to use as possible, these kernel objects are not statically allocated at compile-time, but dynamically allocated at run-time.” [5]

FreeRTOS allocates RAM each time a kernel object is created and frees RAM each time a kernel object is deleted. This policy reduces design and planning effort, simplifies the API, and minimizes the RAM footprint.

FreeRTOS has also dynamic and heap memory allocation considered to be part of the portable layer. “Heap memory manager is only required if *configSUPPORT_DYNAMIC_ALLOCATION* is set to 1 in *FreeRTOSConfig.h*, or if *configSUPPORT_DYNAMIC_ALLOCATION* is left undefined.” [5]

FreeRTOS provides five example implementations of heap allocation schemes which we can choose from, but we can as well provide our own. “The five examples are defined in the *heap_1.c*, *heap_2.c*, *heap_3.c*, *heap_4.c* and *heap_5.c* source files respectively, which are located in the *FreeRTOS/Source/portable/MemMang* directory.” [5]

In our project we chose *heap_4.c* which works by subdividing an array which is statically declared, and dimensioned by *configTOTAL_HEAP_SIZE* into smaller blocks. Heap 4 implements the first fit technique to combine neighbouring free memory blocks into a single larger block when allocating memory. This decreases the risk of memory fragmentation and makes it ideal for applications that frequently create and release RAM in varying sizes.

As we've already mentioned, FreeRTOS has methods to handle numerous threads or tasks, including mutexes, semaphores, and software timers. We will describe these arguments in a more detailed way as they are a crucial part on how we built our project also.

1.5.2 Tasks

A task is an independent thread of execution that can compete with other concurrent tasks for processor execution time. As mentioned earlier, developers decompose applications into multiple concurrent tasks to optimize the handling of inputs and outputs within set time constraints. A task is nothing but a piece of code that is schedulable and it implements a specific functionality in the application. It can be scheduled periodically or aperiodically and it can only be in one of two states, Running or Not Running who on its own is divided into three sub states called Suspended, Ready, and Waiting. To create the Task in FreeRTOS `xTaskCreate()` API function is used and its priority is determined from the beginning based on how important the task is. The maximum number of priorities that can be used is determined by the application-defined `configMAX_PRIORITIES` compile-time configuration constant which is found in `FreeRTOSConfig.h`. The ability to prioritize any number of tasks ensures the greatest degree of design flexibility.

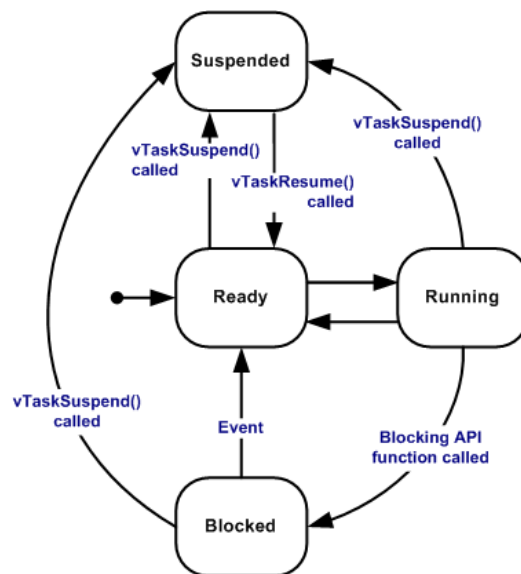


Figure 1.5.1 Valid Task Transitions

Tasks then are scheduled based on their prior assigned priorities. Once the tasks are created, they will enter in the ready task list of the freertos and they will be set to run in the CPU by the what so called scheduler which is a code that is part of the freertos kernel and runs in the privileged mode of the processor.

Pre-emptive and *cooperative scheduling* are two different scheduling strategies that are present in the `freertosconfig.h` file and are both by default set to 1. Pre-emptive scheduling will be the main topic of discussion and it can be broadly classified into two types: *Round Robin pre-emptive scheduling* and *priority-based scheduling*. Since the round robin is a cyclic executive, it schedules the jobs without giving them any precedence and as a result, equal time frames are allocated to each activity in a circular order. The priority of each task determines when and how long it should run

under priority-based scheduling. Context switching, which is handled by PendSV Handler in FreeRTOS, allows the CPU to move from one job to another as it is being executed.

1.5.3 Queues

A queue is a data structure that can hold a finite number of fixed sized data. They are used as first in first out buffers where data is written to the end of the queue and removed from the front of the queue. The queue is used for two main operations: enqueue, which means adding data to the queue, and dequeue, which means removing or deleting data from the queue. The length of the queue and the number of bites each item of the queue takes are the two arguments that the *xQueueCreate* API uses to create the queue in FreeRTOS. The queue create API, like other APIs, dynamically constructs a queue in the ram memory's heap space. Any task or ISR that is aware of the existence of queues can access them as independent objects. “Any number of tasks can write to the same queue, and any number of tasks can read from the same queue.” [5] In practice it is very common for a queue to have multiple writers, but much less common for a queue to have multiple readers.

A task is able to define a "block" time when attempting to read from a queue and if the queue is empty at a certain point, the task will remain in the Blocked state while it waits for data to become available again. When another task or interrupt adds data to the queue, the task that is in the Blocked state and waiting for it to become accessible from a queue is automatically switched to the Ready state.

“The task will also be moved automatically from the Blocked state to the Ready state if the specified block time expires before data becomes available.” [5] Queues have multiple readers, so more than one task can be blocked on it while waiting for data and in such cases only one task with the highest priority will be unblocked when data becomes available. If it happens that the blocked tasks have the same priority, then the one that has been waiting for the longest time will be unblocked.

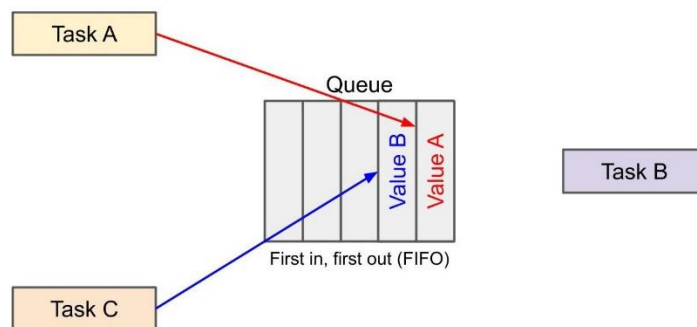


Figure 1.5.2 Example of queue management

A task can also specify a block time when writing to a queue. Queues can have multiple writers, so it is possible for a full queue to have more than one task blocked on it waiting to complete a send operation. In this is the case just like we stated previously only one task with the highest priority will be unblocked when space on the queue becomes available. If they all have equal priority, then the task that has been waiting for the longest time will be unblocked.

1.5.4 Timers

Software timers are used to a function-s execution at a set time in the future, or periodically with a predetermined frequency. “The function executed by the software timer is called the software timer’s call-back function.” [5] The FreeRTOS kernel implements the software timers and manages their operation. They do not depend on hardware support and are not associated to hardware timers or hardware counters. FreeRTOS software timers do not use any processing time until a software timer call-back function is really being executed and its use is optional.

1.5.5 Semaphores and mutexes

As we have mentioned before, FreeRTOS provides semaphores and mutex events which are used for task synchronization such as a task waiting in the blocked state for an event to happen, and unblocked the task when a particular event has taken place. To communicate event occurrence to tasks we use event groups which allow synchronization of a task more than one event. In addition, we can use them to unblock multiple tasks that are waiting for the event to happen.

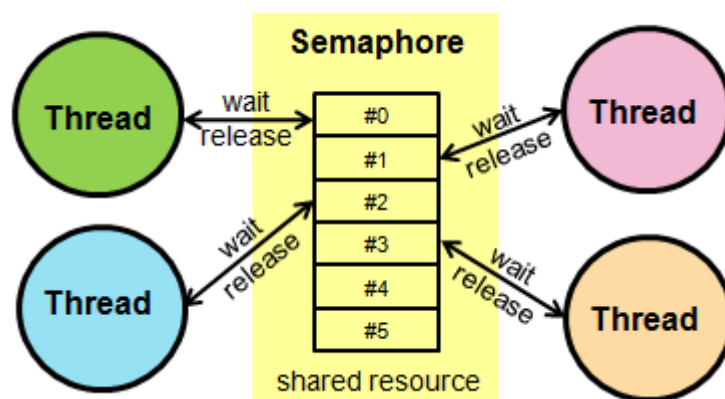


Figure 1.5.3 Semaphore handling example

“A semaphore is a kernel object that one or more threads of execution can acquire or release for the purpose of synchronisation or mutual exclusion.” [6] “Mutexes are binary semaphores that include a priority inheritance mechanism.” [7] The

difference between the two is that we can use binary semaphores for implementing synchronisation between tasks or between tasks and an interrupt, while mutexes are better used to implement simple mutual exclusion.

“Mutexes also allow a block time to be specified which indicates the maximum number of 'ticks' that a task should wait to enter the Blocked state when attempting to 'take' a mutex if it is not immediately available.” [7] If a high priority task blocks while attempting to get a mutex that could be held by a lower priority task, then the priority of the task holding the token is temporarily raised to that of the blocking task which is also called priority inheritance.

A single semaphore can be obtained a finite number of times by the tasks depending on how it is firstly initialized

There are two types of semaphores which are mainly used in RTOS: *binary semaphore* and *counting semaphore*:

Binary semaphore works on only two values 1 or 0, where 1 implies that the semaphore is available and the 0 that it is not available. Binary semaphores are used for synchronization between two tasks or a task and interrupts well as for mutual exclusion.

On the other hand in counting semaphore you can set the semaphore value more than 1 and each time a task acquires the semaphore the number assigned to it decreases by 1 and each time it releases it the semaphore's number increases by one. Counting semaphores are usually used for counting events or for resource management.

1.5.6 Events

“Event groups are another feature of FreeRTOS that allow events to be communicated to tasks.” [5] They allow the task to wait in the Blocked state for a combination of one or more events to occur and unblocks them when the event takes place which makes them useful for synchronizing multiple tasks as well as broadcasting events to more than one *taskEvent*.

Event Flag vs Event Group

An event 'flag' is a Boolean (1 or 0) value which is used to indicate if an event has taken place. “The state of an event flag is stored in a single bit, while the state of all the event flags in an event group in a single variable.” [5] The state of each event flag in an event group is represented by a single bit in a variable of type *EventBits_t* which has made the event flags to be also known as event 'bits'. If a bit is set to 1 in the *EventBits_t* variable, then the event represented by that bit has taken place and vice versa.

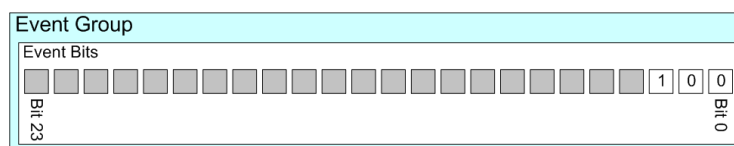


Figure 1.5.4 An event group containing 24-event bits, only three of which are in use.

CHAPTER 2

Environmental Analysis

2.1 Software Tools

2.1.1 Linux

Linux is a Unix-like, open source and community-developed operating system (OS) for computers, servers, mainframes, mobile and embedded devices. It is one of the most used operating systems as it is supported on almost all common computer systems, including x86, ARM, and SPARC.

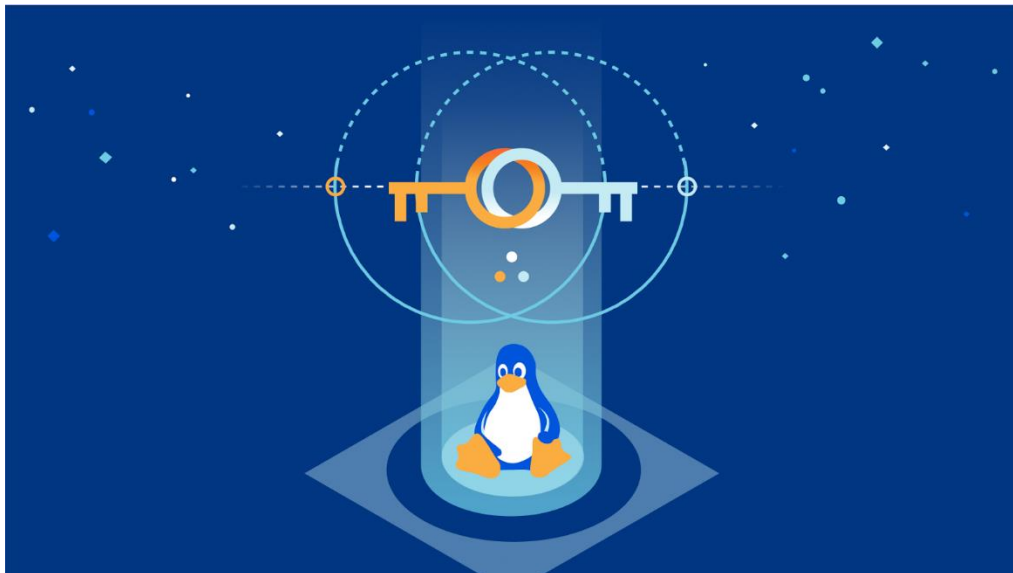


Figure 2.1.1 Linux Logo

Every Linux OS version controls hardware resources, runs and processes programs, and offers a user interface in some capacity. There is a Linux version accessible for almost any task thanks to the large developer community and a variety of distributions making it prevalent in many areas of computing.

The Linux operating system is widely used and supports an extensive variety of use cases. These are some applications that work with it:

Server OS for shared servers of any kind, including web servers, database servers, file servers, email servers, etc.

- **Desktop operating systems** for personal productivity computing.

- **Embedded device** or appliance operating systems for systems with a minimal demand for processing capacity. Household appliances, vehicle entertainment systems, and network file system goods all utilize Linux as their embedded operating system.
- **Software development OS** for commercial software development.
- **Cloud OS** for cloud instances.

Linux offers many advantages to users from which we can list:

- **Open-source software** - The GNU GPL open-source software license governs the distribution of the Linux kernel. The majority of the releases come with hundreds of programs and multiple choices in almost every area. In order to support their hardware, many of the distributions also include proprietary software, such as device drivers from manufacturers.
- **Licence fees** - Linux does not have any sort of licence fees, different from the other commonly used OS such as Windows and Apple MacOS, making many IT Organisations save by moving their software servers from a commercial OS to Linux. Even though the support system provided by them comes with the fee, it is still more affordable than the above-mentioned OS.

Reliability - Linux is viewed as a reliable operating system which receives regular security updates. It can handle unexpected input and software failures and is also regarded as stable making it able to operate under almost any circumstance.

- **Backward compatibility** - Linux and other open-source software are periodically updated for security and functional fixes while being able to maintain their fundamental functionality. Making their configurations and shell scripts to continue functioning as-is even after implementing software upgrades. Linux and open-source software typically don't change their modes of operation with new releases, unlike commercial software manufacturers who release new versions of their OSes along with new ways to work.
- **Variety of choices** - Linux can be optimizable between a variety of distributions and different options for compiling and running it on any hardware device.

One of the most commonly used Linux workstation platforms is Ubuntu desktop thanks to its ease of use, and it is also the distribution we had installed on our pc to develop the project. “Ubuntu Core sets the standard for tiny, transactional systems for highly secure connected devices.” [8] Ubuntu Server is the reference operating system for the OpenStack project, and a hugely popular guest OS on AWS, Azure and Google Cloud. It relies on the architecture of Linux to communicate with a computer's hardware so that software can work properly.

2.1.2 Visual Studio Code

In order to develop the entire code for our project we needed a code editor, so we chose to work with Visual Studio Code.

Visual Studio Code is a streamlined code editor with support for development operations like debugging and task running, used with a variety coding languages such as C, C++, Java, Python, it provides the tools a developer needs for a quick code-build-debug cycle while the more complex workflows are carried on fuller featured IDEs, such as Visual Studio IDE.

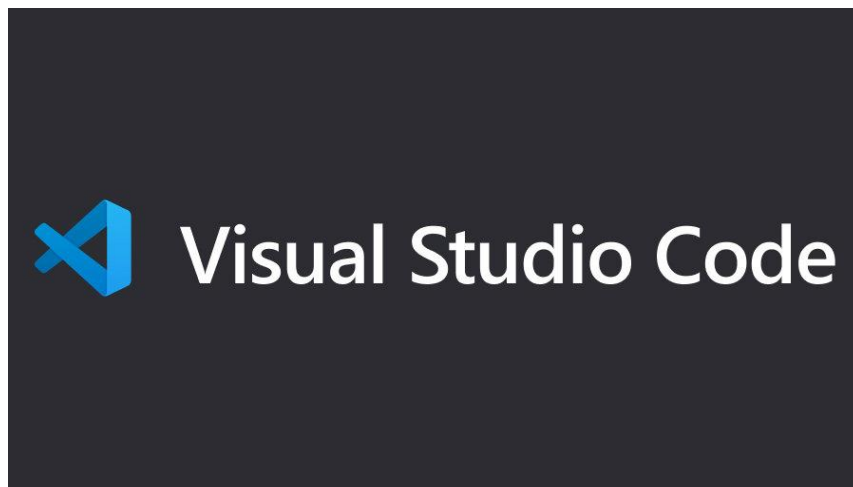


Figure 2.1.2 Visual Studio Code Logo

It is very versatile as it allows users to open one or more directories, which can then be saved in workspaces for future reuse.

Visual Studio Code has a source control built-in feature with a dedicated tab inside of the menu bar where you can access its settings and view changes done on the current project the user is working on. It can also be used in any platform and for any given coding language as it allows the user to set the code page in which the active document is saved, the newline character, and the programming language of the active document.

2.1.3 C Programming Language

C is a procedural programming language initially developed by Dennis Ritchie in the year 1972 mainly as a system programming language to write an operating system. From its most important features we can mention are the low-level memory access, a simple set of keywords, and a clean style making it suitable for programming an operating system or compiler development.

It is to no surprise why we chose to work with C in our project as it is the most suitable one when it comes to Embedded Systems. Despite being invented more than 30 years ago, when it comes to Embedded Systems, there is no other programming

language which even comes close to C. Let us try to understand the reasons behind why C is the most preferred language for embedded systems:

It is **Processor independent**: C language is machine independent which means it is not specific to any system making it able to work on different hardware configuration. We can compile and execute a C program written for one hardware in another one with almost no change and if there is one required, it is related to the OS specific functions and system calls. While writing a program in C, user doesn't have to know the underlying hardware specific details of the machine, since the compiler takes care of it. When a C program is compiled for a particular hardware, respective assembly code is generated which is then linked with static and dynamic libraries to provide an executable file.

Its **Portability** is also another important feature of C as the code developed can be compiled in other platforms with few changes. It is also efficient, easy to understand, maintain and debug.

Performance

C code gets compiled into raw binary executable which can be directly loaded into memory and executed. It provides optimized machine instructions for the given input, which increases the performance of the embedded system. Most of the high-level languages rely on libraries, hence they require more memory which is a major challenge in embedded systems, but C on the other hand does not.

Memory management

As we mentioned before, C provides direct memory control without having to give up on the benefits of other high-level languages. We can directly enter in the memory with the help of pointers and perform various operations using them and also allocate it dynamically or statically depending on whether the memory requirement is known or not.

2.2 Instrumentation used

In the previous chapter we discussed about the SW tools we used to develop our project, now we will focus on the hardware ones. An Oscilloscope, DC Power supply, JTag as well as the main component the CPU with the GD32F305 microcontroller. We will focus separately on each to give you a general explanation about their functionalities and their use in our project.

2.2.1 Oscilloscope

“An **oscilloscope** is a type of electronic test instrument that graphically displays varying electrical voltages as a two-dimensional plot of one or more signals as a function of time.” [9] It is the most popular and available MCU debug tool for the embedded developer as it captures repetitive analog signals as well as the repetitive MCU I/O digital signals. Today’s MCU has a significant number of pins, busses, and I/O, that require a high-channel-count measurement system with sufficient memory for real-time debugging. In Embedded systems different types of examination examinations are performed such as the one of baud rate, output level status and the wait time. “Its main purpose is to show repetitive or single waveforms on the screen which are hard to be perceived by the human eye”[9] which can then be analysed for properties such as amplitude, frequency, rise time, etc. In the past such properties would require measures to be done manually against the scales built into the screen of the instrument but nowadays modern digital instruments are able to calculate and display these properties directly.



Figure 2.2.1 Oscilloscope RIGOL DS1104Z

“Oscilloscopes are used in different fields such as engineering, science, automotive industry and are divided in general-purpose instruments which are used for maintenance of electronic equipment, and special-purpose oscilloscopes that can be used to analyse an automotive ignition system or to display the waveform of the

heartbeat as an electrocardiogram, for example.” [9] In our project we used the RIGOL DS1104Z Oscilloscope

2.2.2 DC Power Supply

Embedded systems that require a significant amount of processing power and that need to interface with analog components require important embedded system power supply guidelines in order to ensure power and signal integrity.

An embedded system consists of many different peripherals that can operate from a wide range of power supply. So, to power the entire system, multiple **DC-DC voltage converters** are used.



Figure 2.2.2 DC Power supply

A DC power supply also known as a “bench power supply” is a power supply that gives direct current (DC) voltage to power a device from the AC power supply of the outlet.

For example, household power outlets in Japan are provided with 100V AC, but minute voltage fluctuations actually occur due to losses during the power supply and changes in power consumption of devices connected to the same power source.

Fluctuations in the voltage supplied to home appliances with relatively simple structures, such as vacuum cleaners and fans, do not pose much of a problem. However, for precision devices such as electronic equipment, a slight change in voltage can cause malfunction.

A DC stabilized power supply is also used to convert AC power to DC since this last one is required to run electronic devices. Stabilized DC power supplies are divided into two types depending on the output method: constant voltage power supplies and constant current power supplies. In constant-voltage power supplies, the output

voltage is controlled so that it remains constant even if the power supply load changes. On the other hand, constant-current power supplies are controlled so that the output current remains constant.

2.2.3 JTag Connector

Joint Test Action Group, also known as JTAG, is the common name for IEEE standard 1149.1. This standard defines a particular method for testing board-level interconnects, which is also called Boundary Scan. In short, JTAG was created as a way to test for common problems, but lately has become a way of configuring devices. The JTAG hardware interprets information from five different signals: TDI (Test Data In), TDO (Test Data Out), TMS (Test Mode Select), TCK (Test Clock), and TRST (Test Report-optional).

“The primary advantage of boundary-scan technology is the ability to observe data at the device inputs and control the data at the outputs independently of the application logic.” [10] Simple tests can find manufacturing defects such as unconnected pins, a missing device, an incorrect or rotated device on a circuit board, and even a failed or dead device

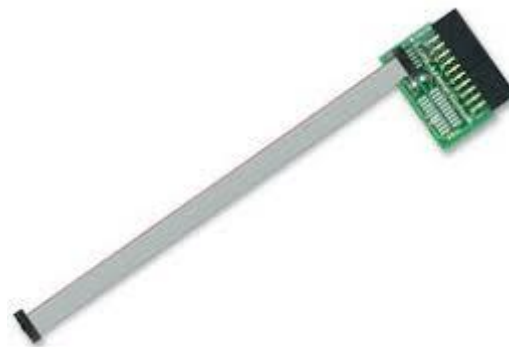


Figure 2.2.3 JTag Connector

“JTAG allows device programmer hardware to transfer data into internal non-volatile device memory (e.g. CPLDs).” [11] Some device programmers serve a double purpose for programming as well as debugging the device. In the case of FPGAs, volatile memory devices can also be programmed via the JTAG port, normally during development work. In addition, internal monitoring capabilities (temperature, voltage and current) may be accessible via the JTAG port.

As a practical matter, when developing an embedded system, emulating the instruction store is the fastest way to implement the "debug cycle" (edit, compile, download, test, and debug). This is because the in-circuit emulator simulating an instruction store can be updated very quickly from the development host via, say, USB. Using a serial UART port and bootloader to upload firmware to Flash makes this debug cycle quite slow and possibly expensive in terms of tools; installing

firmware into Flash (or SRAM instead of Flash) via JTAG is an intermediate solution between these extremes.

One basic way to debug software is to present a single threaded model, where the debugger periodically stops execution of the program and examines its state as exposed by register contents and memory (including peripheral controller registers). When interesting program events approach, a person may want to single step instructions (or lines of source code) to watch how a particular misbehaviour happens.

So, for example a JTAG host might HALT the core, entering Debug Mode, and then read CPU registers using ITR and DCC. After saving processor state, it could write those registers with whatever values it needs, then execute arbitrary algorithms on the CPU, accessing memory and peripherals to help characterize the system state. After the debugger performs those operations, the state may be restored, and execution continued using the RESTART instruction.

Debug mode is also entered asynchronously by the debug module triggering a watchpoint or breakpoint, or by issuing a BKPT (breakpoint) instruction from the software being debugged. When it is not being used for instruction tracing, the ETM can also trigger entry to debug mode; it supports complex triggers sensitive to state and history, as well as the simple address comparisons exposed by the debug module. Asynchronous transitions to debug mode are detected by polling the DSCR register. This is how single stepping is implemented: HALT the core, set a temporary breakpoint at the next instruction or next high-level statement, RESTART, poll DSCR until you detect asynchronous entry to debug state, remove that temporary breakpoint, repeat.

2.2.4 Microcontroller

The microcontroller used on our project is the GD32F305xx device which belongs to the mainstream line of GD32 MCU Family. “It is a new 32-bit general-purpose microcontroller based on the Arm® Cortex® -M4 RISC core whose features implement a full set of DSP instructions to address digital signal control markets that demand an efficient, easy-to-use blend of control and signal processing capabilities.” [12] It also provides a Memory Protection Unit (MPU) and powerful trace technology for enhanced application security and advanced debug support. The GD32F305xx device provides up to 1024KB on-chip Flash memory and 96KB SRAM memory. An extensive range of enhanced I/Os and peripherals connected to two APB buses.

The devices offer up to two 12-bit 2.6 MSPS ADCs, two 12-bit DACs, up to ten general 16-bit timers, two 16-bit PWM advanced timers, and two 16-bit basic timers, as well as standard and advanced communication interfaces: up to three SPIs, two

I2Cs, three USARTs and two UARTs, two I2Ss, two CANs and a USBFS. The device operates from a 2.6 to 3.6 V power supply and available in -40 to $+85$ °C temperature range. Several power saving modes provide the flexibility for maximum optimization between wakeup latency and power consumption, an especially important consideration in low power applications. The above features make GD32F305xx devices suitable for a wide range of interconnection and advanced applications, especially in areas such as industrial control, consumer and handheld equipment, communication networks, embedded modules, human machine interface, security and alarm systems, graphic display, automotive navigation, IoT and so on.



Figure 2.2.4 Microcontroller GD32F305

CHAPTER 3

GPS Task

We will now get more technical with our work and focus on each module of the CPU separately to give you a very thorough and detailed explanation on what the work consisted on.

We will start with the GPS Module which was the first module we worked on that also gave us the bases and a general idea on how to divide the project later.

Global Positioning System (GPS) is a technology that uses signals from GPS satellites to determine the location of a device on the Earth's surface. GPS can be implemented on a microcontroller using a GPS module.

A GPS module consists of a GPS receiver and an antenna. The GPS receiver receives signals from GPS satellites and uses the information contained in those signals to determine the device's location, velocity, and time. The GPS receiver then sends this information to the microcontroller through a serial communication interface such as UART.

The microcontroller can then process the GPS data and perform various tasks based on the device's location, such as navigation, tracking, and mapping. It can also be used in conjunction with other sensors, such as accelerometers, to provide more accurate and comprehensive information about the device's position and movement.

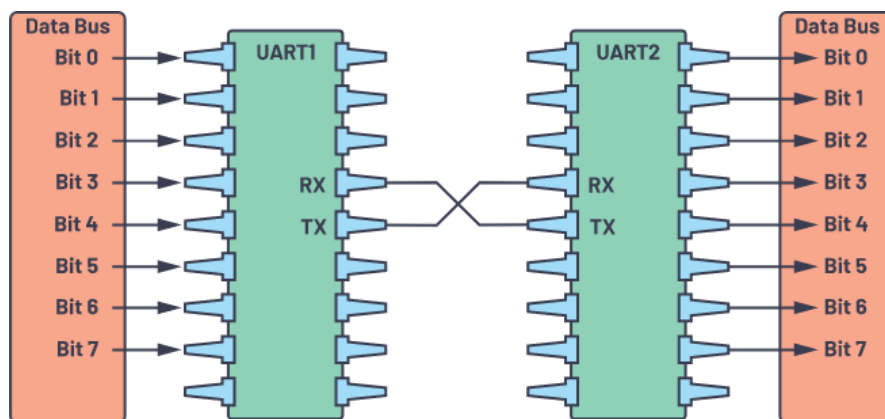


Figure 3 Uart with data bus

As we mentioned previously, to establish the communication between the microcontroller and the GPS Receiver we need to use the UART interface.

“By definition, UART is a hardware communication protocol that uses asynchronous serial communication with configurable speed.”[13] Asynchronous means there is no clock signal to synchronize the output bits from the transmitting device going to the

receiving end.

The interface of a UART device is done by the two signals named:

- Transmitter (Tx)
- Receiver (Rx)

The main purpose of a transmitter and receiver line for each device is to transmit and receive serial data intended for serial communication. Uart is connected to a controlling data bus that sends data in parallel form. From this, the data will be transmitted on the transmission line(wire) serially, bit by bit to the receiving UART.

To establish this communication in our project we created a function called

v_UART_GPS_Handler

3.1 UART GPS Handler

This part of the code defines an ISR that is triggered when data is received by one of four UART peripherals. The function retrieves the received data, sends it to a queue, and sets a bit in an event group. If any of these operations cause a higher-priority task to become ready to run, the function sets a flag to indicate that a task switch should occur.

The function begins by initializing a flag variable called ***!HigherPriorityTaskWoken*** to ***pdFALSE*** which is later used to determine whether a higher-priority task has become ready to run during the execution of the ISR and needs to be woken up.

The second variable, ***send***, is used to store data that is received by the UART peripheral while the third one called ***error_code***, is used to store the value of the Interrupt Identification Register (IIR) for the UART peripheral that corresponds to the ***vUART_GPS*** variable. The switch statement is used to We then determine which UART peripheral should be read based on the value of ***vUART_GPS*** by using the *switch* statement. Depending on its value the function reads the Interrupt Identification Register (IIR) of one of four UART peripherals, each identified by a number from 0 to 3.

We then enter an if loop to check whether the value of IIR is indicating that data has been received by the UART. If yes, then the data is retrieved by calling the ***vGetChar()*** function and storing the data in the variable ***send***.

The function then sends the received data to a FreeRTOS queue called ***QUEUE_GPS_RX*** which was created previously by using the ***xQueueSendFromISR()*** which is a function used to send data from an Interrupt Service Routine(ISR) to a FreeRTOS Queue. If sending the data to the queue causes a

higher-priority task to become ready to run, the **lHigherPriorityTaskWoken** variable is set to **pdTRUE**.

“Finally, the function calls “**xEventGroupSetBitsFromISR()**, which is a version of *xEventGroupSetBits()* that can be called from an interrupt service routine (ISR).”[14] It means that setting bits in an event group will automatically unblock tasks that are blocked waiting for the bits. “Setting bits in an event group is not a deterministic operation” [5] because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow non-deterministic operations to be performed in interrupts or from critical sections. Therefore *xEventGroupSetBitFromISR()* sends a message to the RTOS daemon task to have the set operation performed in the context of the daemon task - where a scheduler lock is used in place of a critical section.

This function sets a specific bit (GPS_BIT_0) in an event group called **xEventGroupGPS** which was previously created. If this operation causes a higher-priority task to become available, the **lHigherPriorityTaskWoken** variable is set to **pdTRUE**.

The function ends by calling **portEND_SWITCHING_ISR()**, which is a macro used to indicate that a task switch should be performed if the **lHigherPriorityTaskWoken** variable is **pdTRUE**.

```
void vUART_GPS_Handler() {
    long lHigherPriorityTaskWoken = pdFALSE;
    unsigned char send;
    int error_code ;
    //EventBits_t uxBits;

    switch(vUART_GPS){
    case 0:
        error_code=vUART0->VIIR;
        break;
    case 1:
        error_code=vUART1->VIIR;
        break;
    case 2:
        error_code=vUART2->VIIR;
        break;
    case 3:
        error_code=vUART3->VIIR;
        break;
    }
}
```

Figure 3.1 Code snippet

3.2 GPS Pin Init

The second thing we worked on while building the GPS Module was the GPS Init function. This part initializes the GPS device by configuring its pins, turning it on, initializing the UART interface, and configuring the device to output a PPS signal, so in other words it will configure the PIOs to turn the GPS on.

We started by checking the MTS-2046 datasheet to know the pin number for turning on the GPS and the PPS* signal. Essentially the PPS signal also known as *Pulse Per Second* is generated by the GPS receiver to be used for the synchronization between separate stations in the communications system or the measurement system. It normally stays on stage '1' and goes to '0' every second to signalise the drop of the second.

After checking the pins directions and functions we go the datasheet of the microcontroller. The pins signal is as a “general-purpose input/output also known as GPIO is on an integrated circuit or electronic circuit board which may be used as an input or output, or both, and is controllable by software” [15]. GPIOs have no predefined purpose and are unused by default. If used, the purpose and behaviour of a GPIO is defined and implemented by the designer of higher assembly-level circuitry: the circuit board designer in the case of integrated circuit GPIOs, or system integrator in the case of board-level GPIOs.

We start by working on the G_FON Signal and initially we should set pin 1.0 as GPIO. The value of its function is 00 so we go to the Pinsel3 section in the table to check the pins whose function is equal to zero. The bits of pin 01 are two, 0 and 1. As stated on the table, we need the two bits to be equal to 00. We start by using 2 bits equal to 11 and we express them in the decimal form which is equal to 3. We shift it to 0 bits by using the '<<' operator which significantly means that we are shifting the 0 bits of 3 to the left. So, we go from ...00000011 to ...11000000. We then perform the Bitwise complement Operation through the xor denoted by (~), which is a unitary operator, meaning that will work only on one operand, and it will change the 1s to 0 and the 0s to 1. So, the value we had ...11000000 will shift to ...00111111.

After setting pin 1.0 as GPIO we then worked on setting it as an output as its direction stated on the table. We do this by using the vGPIO1->vFIODIR |= to set the direction of the pin.

We then move forward to the PPS signal and set the pin 1.18 as GPIO in the same way we did with pin 1.0. Finally, we turn the GPS on by using the vFIOCLR function to clear the bits.

Generic Name	Description	Access	Reset value ^[1]	PORTn Register Name & Address
FIODIR	Fast GPIO Port Direction control register. This register individually controls the direction of each port pin.	R/W	0	FIO0DIR - 0x2009 C000 FIO1DIR - 0x2009 C020 FIO2DIR - 0x2009 C040 FIO3DIR - 0x2009 C060 FIO4DIR - 0x2009 C080
FIOMASK	Fast Mask register for port. Writes, sets, clears, and reads to port (done via writes to FIOPIN, FIOSET, and FIOCLR, and reads of FIOPIN) alter or return only the bits enabled by zeros in this register.	R/W	0	FIO0MASK - 0x2009 C010 FIO1MASK - 0x2009 C030 FIO2MASK - 0x2009 C050 FIO3MASK - 0x2009 C070 FIO4MASK - 0x2009 C090
FIOPIN	Fast Port Pin value register using FIOMASK. The current state of digital port pins can be read from this register, regardless of pin direction or alternate function selection (as long as pins are not configured as an input to ADC). The value read is masked by ANDing with inverted FIOMASK. Writing to this register places corresponding values in all bits enabled by zeros in FIOMASK. Important: if an FIOPIN register is read, its bit(s) masked with 1 in the FIOMASK register will be read as 0 regardless of the physical pin state.	R/W	0	FIO0PIN - 0x2009 C014 FIO1PIN - 0x2009 C034 FIO2PIN - 0x2009 C054 FIO3PIN - 0x2009 C074 FIO4PIN - 0x2009 C094
FIOSET	Fast Port Output Set register using FIOMASK. This register controls the state of output pins. Writing 1s produces highs at the corresponding port pins. Writing 0s has no effect. Reading this register returns the current contents of the port output register. Only bits enabled by 0 in FIOMASK can be altered.	R/W	0	FIO0SET - 0x2009 C018 FIO1SET - 0x2009 C038 FIO2SET - 0x2009 C058 FIO3SET - 0x2009 C078 FIO4SET - 0x2009 C098
FIOCLR	Fast Port Output Clear register using FIOMASK. This register controls the state of output pins. Writing 1s produces lows at the corresponding port pins. Writing 0s has no effect. Only bits enabled by 0 in FIOMASK can be altered.	WO	0	FIO0CLR - 0x2009 C01C FIO1CLR - 0x2009 C03C FIO2CLR - 0x2009 C05C FIO3CLR - 0x2009 C07C FIO4CLR - 0x2009 C09C

[1] Reset value reflects the data stored in used bits only. It does not include reserved bits content.

Figure 3.2 GPIO register map (local bus accessible registers - enhanced GPIO features) from the LPC17xx Family_UserManual_1008

We then initialize the UART (serial communication interface) to enable the communication with the GPS device by using the **SerialInitWithInterrupt** function. It has three arguments: **vUART_GPS**, which specifies the UART module to use; **vUART_GPS_BAUDRATE**, which specifies the baud rate of the communication; and **vUART_GPS_PRIORITY**, which specifies the priority of the UART interrupt.

Then we create a task called **vTaskDelay** which blocks the other tasks for a specific period of time, in our case it waits for 300 milliseconds before continuing execution.

By using the **vPutString** we then send the NMEA string to the GPS device using the UART interface. NMEA stands for National Marine Electronics Association, and an NMEA string refers to a data sentence that conforms to the NMEA standard. It is widely used in the marine industry surveying, aviation, and agriculture, where accurate positioning and navigation are essential.

The NMEA standard specifies a communication protocol for marine electronics such as GPS receivers, sonar, and autopilots, among others. The string contains information about a particular aspect of the navigation system, such as the GPS position, speed, depth, and course. Each NMEA sentence begins with a dollar sign "\$" and ends with a checksum and is made up of alphanumeric characters. It also has a unique identifier known as a "talker ID" that indicates the type of device sending the sentence.

Lastly we raise a flag called **initGPSReady** to indicate that GPS initialization has been completed.

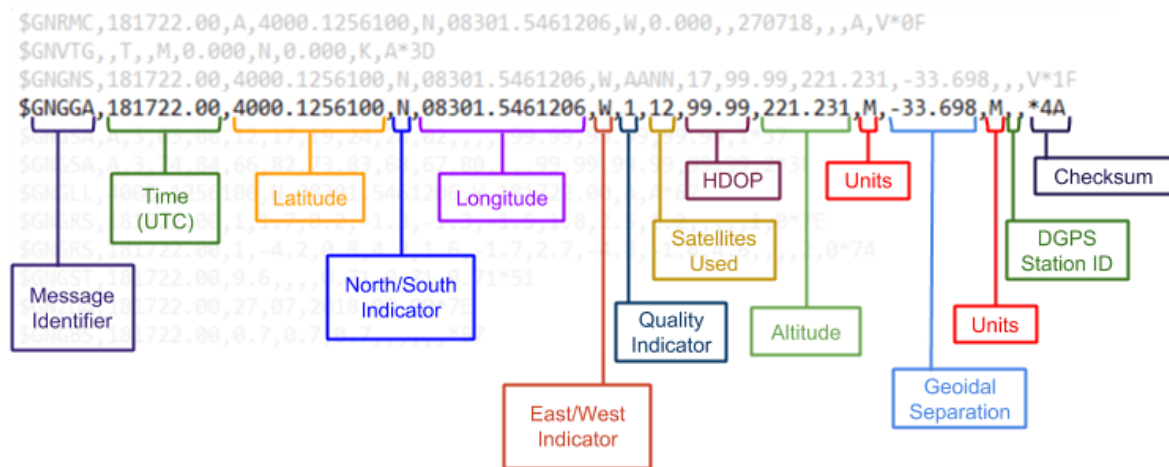


Figure 3.2.1 Example of NMEA String

3.3 GPS Task function

Afterwards we started with the most demanding part of the GPS Module, which is writing the function of the GPS Task. We started from the template of the LPC13xx project and continued modifying it. As stated before, through the code already written it was not able to make the communication between the tasks and specifically the gps task was not able to receive the messages through queues but only one character at a time. So, we started by creating an algorithm map which consisted in 2 queues through which the NMEA would send its string with the information from the gps. The NMEA code was already written by the firmware engineer of the company, so we had to take care only for the GPS Task part. After creating the Algorithm Map, we translated it in C code.

First, we introduced 2 strings called *str_1* and *str_2* with length equal to 256 which was defined in the beginning of the **GPSTask** file. These strings are going to store the NMEA message. Then if a character is received, the code checks if it is the ASCII character '\$' and sets the variable **char_count** to zero, toggles the **use_string2** variable (which determines which buffer to write incoming characters to), and sets the

store_string variable to **true**. This indicates that the GPS module is starting to send a new message, so the program should start storing the incoming characters in a buffer.

If the received character is not '\$', the code checks if it is not a carriage return (**\r**) or a line feed (**\n**) and if it is not, and if the **store_string** variable is **true**, it stores the character in the appropriate buffer (either **str_1** or **str_2**, depending on the value of **use_string2**) and increments the **char_count** variable.

Since we are using two strings, we used a Boolean variable called **use_string2** initially equal to **true**. This means that if the Boolean variable was true, string 2 was going to be used to store the NMEA message, otherwise if string 2 was full, the Boolean variable would be **false** therefore string 1 was going to be used. The characters coming from the GPS are represented with **gps_char**. Moving forward we call the function **GPSInit** which was described in the previous paragraph and through **printf** we print a statement which will let us know that the task of GPS is being initialised.

Finally, the program repeats the loop and waits for the next character to be received on the queue.

```
if (gps_char == '$') { // check if the char is $
    char_count = 0; //initialise the count
    use_string2 = ((use_string2==true)? false:true); //switch the string (toggle)
    store_string = true;
}else if((gps_char != '\r') && (gps_char != '\n')) {
    if(store_string == true){
        if(use_string2 == true){
            str_2[char_count] = gps_char;
        } else {
            str_1[char_count] = gps_char;
        }
    }
    char_count ++;
}else if (store_string){
    if(use_string2 == true){ // check if we are using string2, if so write on it
        str_2[char_count] = '\0';
        printf("GPS2 string %s\n", str_2 );
    } else {
        str_1[char_count] = '\0';
        printf("GPS1 string %s\n", str_1 ); // store_string
    }
    store_string = false;
}
```

Figure 3.3.1 Code snippet

So far, we have implemented the GPS task on its own and its whole function is to display the two strings in the terminal. The purpose of this entire project however is to make the GPS Module communicate with the Console. Therefore we will go back to the **GPS_Task** and modify it.

Inter task communication is done by the help of queues, semaphores and mutexes which we explained in the first chapter. To manage these features of FreeRTOS the Events will come in hand. The purpose of an event in FreeRTOS is to allow tasks to communicate and synchronize with each other based on the occurrence of a specific condition or event. Events can be used to signal when a resource is available, a

specific action has been taken, or when a task has completed its execution. They are an essential mechanism for task synchronization, allowing tasks to wait for specific events to occur before proceeding with their execution. This helps to ensure that tasks are executed in a coordinated and controlled manner, preventing race conditions, deadlocks, and other synchronization problems.

We start with the same variable declarations as we previously did, including the two strings to store NMEA messages from the GPS, and then initialize the GPS module. We then call the event that will enable the communication between the GPS and Console through the "xEventGroupWaitBits()" function which waits for one or more bits to be set in a given event group ("xEventGroupGPS"), and the specific bits to wait for are defined by the "GPS_BIT_0 | MESSAGE_BIT_1" argument. The third argument, "pdFALSE", indicates whether all or any of the bits specified in the second argument must be set before the function returns. In this case, only one of the bits needs to be set for the function to return. The fourth argument, "pdFALSE", specifies whether the bits that caused the function to return should be cleared automatically or not. Finally, the last argument "xTicksToWait" specifies the maximum amount of time the function should wait for the bits to be set before returning.

We then enter the main loop, where we wait for two possible events using an event group: either a GPS character is received, or a message is sent to the GPS task. If a GPS character is received, the character is processed to form an NMEA message and stored in one of the two message strings, depending on which string was used last. If a message is received, the task sends the message to the console task and sets a flag in the console event group.

In the first case "***if(uxBits & GPS_BIT_0)***" we check if a specific bit (GPS_BIT_0) is set in a variable called uxBits. If the bit is set, it means that there is new data available from the GPS module and the code should proceed to read a character from the GPS module.

We then clear the GPS_BIT_0 bit by using the *xEventGroupClearBits* function in an event group called *xEventGroupGPS*, indicating that the data has been read.

Then we enter a loop that reads characters by using the *xQueueReceive()* function from a queue called *QUEUE_GPS_RX* which was defined previously. The read character is then stored in a variable called *gps_char*, the remaining part of this process is the same as the one we described previously when the GPS task whole function was to display the two strings in the terminal.

```
Chr from Console
A message came for GPS Task
GPS1 (1) GNRMC,000400.095,V,,,,,0.00,0.00,060180,,,N*54
0x20004f00:GNRMC,000400.095,V,,,,,0.00,0.00,060180,,,N*54
```

Figure 3.3.2 Output shown on the Command Prompt

In the second case “*if(uxBits & MESSAGE_BIT_1)*” we check if the *MESSAGE_BIT_1* is set in a variable called *uxBits*. This means that a message has been received in a queue called *QMsgGPS*. The code then will retrieve the message from the *QMsgGPS* queue using the *xQueueReceive()* function which is then printed to the console using the *printf()* function. Before printing the message, we take a binary semaphore called *serialSemaphore*. As we have mentioned previously a semaphore is a synchronization mechanism that allows multiple tasks to access a shared resource (in this case, the console or serial port) in a mutually exclusive way. The *(TickType_t)TICKWAIT_SEMA* argument specifies a maximum amount of time (in system ticks) to wait if the semaphore is not available. After printing the message we release the *serialSemaphore* allowing other tasks to access the console.

```
if( uxBits & MESSAGE_BIT_1 )
{
    xQueueReceive(QMsgGPS, (void *)&msg, (TickType_t)10); // Receive the char from console
    xSemaphoreTake(serialSemaphore, (TickType_t)TICKWAIT_SEMA);

    printf("0x%08x:%s\n", msg, (char *) msg);
    xSemaphoreGive(serialSemaphore);

    printf("A message came for GPS Task\n");
    xEventGroupClearBits( xEventGroupGPS, MESSAGE_BIT_1);
}
```

Figure 3.3.3 Code snippet

We then check whether a flag we have mentioned previously called *use_string2* is true or false. If yes, the code sends the message to another queue called *QMsgConsole* using the *xQueueSend()* function. If instead *use_string2* is false, the message is sent to a different queue, *QMsgGPS*, while using the same function. After the message is sent to the appropriate queue, another bit is set, *MESSAGE_BIT_2*, in a different event group called *xEventGroupCONSOLE* using the *xEventGroupSetBits()* function. We will discuss about this in detail on the next chapter.

```
if( uxBits == 0 )
{
    xSemaphoreTake(serialSemaphore, (TickType_t)TICKWAIT_SEMA);
    printf("GPS:No message or char received.\n");
    xSemaphoreGive(serialSemaphore);
}
```

Figure 3.3.4 Code snippet

If `uxBits` is equal to 0, a message is printed to the console indicating that no message or character has been received. The code first takes the **serialSemaphore** to ensure that it has exclusive access to the console. It then prints a debug message to the console using the **printf** function. Finally, the code releases the **serialSemaphore** using the **xSemaphoreGive** function to allow other tasks to access the console.

CHAPTER 4

Serial Task

We will now move to another part of the project which is the Console Module. In the previous chapter we worked on the GPS Module and its purpose was to establish a communication with the Console. Based on the work done on the GPS Module we will continue to work on the Console.

4.1 UART Console Handler Function

Just like for the GPS the first thing we do is establish the Console UART Interrupt by creating a function called `vUART_CONSOLE_Handler()` which will handle the interrupt that is triggered when data is received on a UART channel. Its logic is that new data is received, it checks whether it indicates the start of a debug mode. If the debug mode is not active, it will send the received data to a console receive queue and set an event group bit to notify other tasks that new data is available. Otherwise, if debug mode is active, it will simply ignore the received data.

```
if(error_code & 0x4){
    send=SerialGetChar();
    if(!debugActive)
    {
        xQueueSendFromISR( QUEUE_CONSOLE_RX, &send, &lHigherPriorityTaskWoken);

        xEventGroupSetBitsFromISR( xEventGroupCONSOLE, CONSOLE_BIT_0,
                                   &lHigherPriorityTaskWoken);
    }

    else
    {
        xQueueSendFromISR( QUEUE_CONSOLE_RX, &send, &lHigherPriorityTaskWoken);
    }

    portEND_SWITCHING_ISR( lHigherPriorityTaskWoken );
}
```

Figure 4.1 Code snippet

We will break down the function to explain more in detail how the code works. The function starts by declaring as variable which is used to track whether or not a higher-priority task has been woken by this function "*lHigherPriorityTaskWoken*"

and initializing it to *"pdFALSE"*. We then declare the variables **send** and **error_code** which are used to store the character that is received from the UART channel and read the interrupt identification register for the UART channel respectively.

The function then reads the interrupt identification register (*vIIR*) of the UART channel specified by *vUART_CONSOLE* which contains information about the type of interrupt that triggered the handler function.

If the 4th bit (bit 2) of the interrupt identification register is set, it means that there is new data available to read from the UART channel. The function reads the data using the *"SerialGetChar()"* function and stores it in the *"send"* variable.

Then the code checks whether *debugActive* is not set, and sends the received data to the console receive queue using **xQueueSendFromISR()** and also sets an event group bit using **xEventGroupSetBitsFromISR**.

If instead *debugActive* is set, it means that the system is currently in debug mode and normal data should not be sent to the console receive queue.

If the 4th bit of **error_code** is not set, it means that there is no new data available to read from the UART channel. In this case, the code simply reads a character from the UART channel using **SerialGetChar()** and stores it in the **send** variable.

4.2 Serial Task Function

In the previous chapter we established the communication between the GPS and the console, from the GPS Side. Now we will continue working on it from the Console side.

We start by creating a function called **vSerialTask**, and then we declare the variable *uxBits*, which is used to store the event bits returned from the *xEventGroupWaitBits()* function. We then declare the variable *xTicksToWait*, which is used to set the maximum number of ticks to wait before timing out, in our case is 5000 milliseconds.

The function then checks if two queues, *QUEUE_CONSOLE_RX* and *QUEUE_CONSOLE_TX*, are not null. If either of these queues are null, the function will print a message indicating that the Serial Task has been deleted and call *vTaskDelete()* to delete the task, otherwise the function will print a message indicating that the **Serial Task** has been initialized and proceed to the main loop of the task.

We enter the main loop of the function by using the FreeRTOS API function **xEventGroupWaitBits** to wait for specific bits to be set in the event group called **xEventGroupCONSOLE**. The second parameter passed to the function is a bitmask of the bits being waited for, which in this case is the logical OR of two bit values: **CONSOLE_BIT_0** and **MESSAGE_BIT_2**. The third and fourth parameters are both set to **pdFALSE**. The third parameter specifies whether all bits in the bitmask should be set (**pdTRUE**) or any of the bits (**pdFALSE**) before the wait is considered successful while the fourth parameter specifies whether the function should clear the

bits in the event group that were waited for (**pdTRUE**) or not (**pdFALSE**). The last parameter passed to the function is **xTicksToWait**, which is the amount of time the function should wait for the bits to be set before timing out.

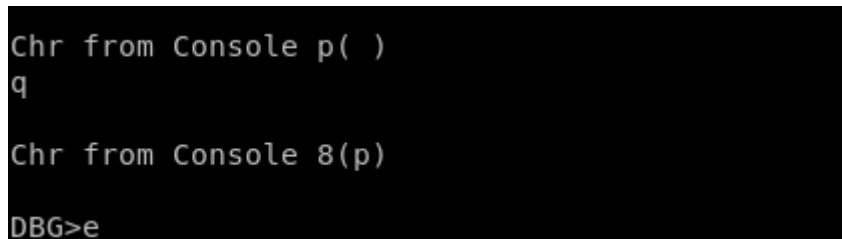
Moving forward, there are two possible events that can take place, either the Console sends a message, or otherwise it receives one.

In the first case “**if(uxBits & CONSOLE_BIT_0)**” the event is set when a character is received from the serial console. The code block checks whether the *CONSOLE_BIT_0* bit is set in *uxBits*. If so, the bit is then cleared in the *xEventGroupCONSOLE* event group.

Then it reads the characters from the *QUEUE_CONSOLE_RX* queue by using a *while* loop which will continue to run as long as *xQueueReceive* returns *pdPASS* which indicates that a character was successfully read from the queue.

We then print a message to the console with the received character (*c*) and the last character that was received (*last_char*) and make sure to check that the received character is not equal to the end-of-file character (EOF).

Moving forward, once the character is received the code checks whether it is equal to '8' with the previous character being 'P' or 'p' and that the debug mode is not already active. This is done because the command “P8” or “p8” is set to be the one that activates the debug mode. If the conditions in the previous line are met, a semaphore named *debugSemaphore* is given and the *last_char* variable is set to a space character. We will discuss later what happens when we enter the debug mode and what is its purpose.



```
Chr from Console p( )
q
Chr from Console 8(p)
DBG>e
```

Figure 4.2.1 Output shown on the Command Prompt

If the character instead is “G” or “g” the code then sets the first character of the command array to 'G' and sends a message to the queue named *QMsgGPS* and sets the *MESSAGE_BIT_1* bit of the event group named *xEventGroupGPS*.

Now we are starting to establish the communication between the GPS and the console. Once the Console send the character to the GPS Module, it will reply back by sending the NMEA String containing the coordinates data.

```
Chr from Console g(8)
h
0x20000e70:G
A message came for GPS Task
```

Figure 4.2.2 Output shown on the Command Prompt

Lastly if the character received is “M” or “m” the code will set the first character of the command array to 'M' and send a message to a queue named *QMsgMODEM*, and sets the MESSAGE_BIT_3 bit of an event group named *xEventGroupMODEM*. We will discuss in the upcoming chapter, where we talk about the Modem Module and its purpose more thoroughly about the purpose of part of the code.

```
Chr from Console m(8)
n
0x20000e70:M
A message came for MODEM Task
```

Figure 4.2.3 Output shown on the Command Prompt

In the second case *if(uxBits & MESSAGE_BIT_2)* the event is set when a message is received from another task and sent to the *QMsgConsole* queue. The code attempts to receive a message from the *QMsgConsole* queue using the *xQueueReceive* function. If the message is received within 10 ticks (as specified by the *(TickType_t)10* argument), the code will proceed to acquire a lock on the *serialSemaphore* semaphore using *xSemaphoreTake* which is done to ensure that no other task or thread can access the serial port while the current task is using it. Once the semaphore is acquired, the code prints the received message to the serial port using *printf*. The message is printed in hexadecimal format using the *%08x* format specifier, and the actual message string is cast to a *char** type. After printing, the semaphore is then released using *xSemaphoreGive* and the MESSAGE_BIT_2 bit is cleared in the *xEventGroupCONSOLE* event group using the *xEventGroupClearBits* function.

Finally, if no bits are set in the *uxBits* variable, the console task waits for the specified time period and then prints a message indicating that no message or character was received.

```
CNS:No message or char received(0).  
CNS:No message or char received(0).  
CNS:No message or char received(0).
```

Figure 4.2.4 Output shown on the Command Prompt

4.3 VCOM_getChar Function

As we have previously mentioned, in an embedded system, it is common to use queues to communicate between different tasks or interrupts. A queue is a data structure that allows multiple tasks or interrupts to add data to it and multiple tasks or interrupts to receive data from it. When a task or interrupt wants to send data to another task or interrupt, it simply adds the data to the queue. When a task or interrupt wants to receive data, it waits for the data to become available in the queue.

In order to read single characters from the queue we create a function called *getChar* whose purpose is to read a single character from a queue called **QUEUE_CONSOLE_RX**. When a character is available, the function retrieves it from the queue and returns it as an integer.

```
int VCOM_getChar(void){  
    unsigned char c;  
    xQueueReceive( QUEUE_CONSOLE_RX, &c, portMAX_DELAY );  
    return c;  
}
```

Figure 4.3 Code snippet

We then use the **portMAX_DELAY** parameter in the **xQueueReceive ()** function to indicate that the function should block indefinitely until a character is received from the queue. This means that the function will wait until a character is available before returning, which could cause the task or interrupt that is calling this function to block indefinitely if there are no characters being sent to the queue.

Therefore, when using a blocking function like **xQueueReceive()** with an indefinite delay, it is important to make sure that data is being sent to the queue as expected and

that the queue has enough space to store the data. If the queue is full and cannot accept any more data, then the **xQueueReceive()** function will not be able to receive any data, and the task or interrupt that is calling this function will block indefinitely.

CHAPTER 5

Modem Task

The last module we worked on was the Modem. The purpose of a modem on a microcontroller is to enable the microcontroller to communicate over a network or other communication channel that requires modulation and demodulation of data signals. It also allows a microcontroller to send and receive data over a communication channel that uses analog signals, such as telephone lines or radio frequency channels. The modem modulates digital data into an analog signal that can be transmitted over the communication channel and demodulates received analog signals back into digital data that the microcontroller can process.

Modems are often used in applications that require remote monitoring and control, such as industrial automation, security systems, and remote data acquisition. By using a modem, a microcontroller can communicate over long distances and across different types of communication networks, such as the public switched telephone network (PSTN) or the internet.

In addition, modems on microcontrollers can support different communication protocols, such as dial-up, cellular, or satellite communications, making them versatile and adaptable to various communication requirements.

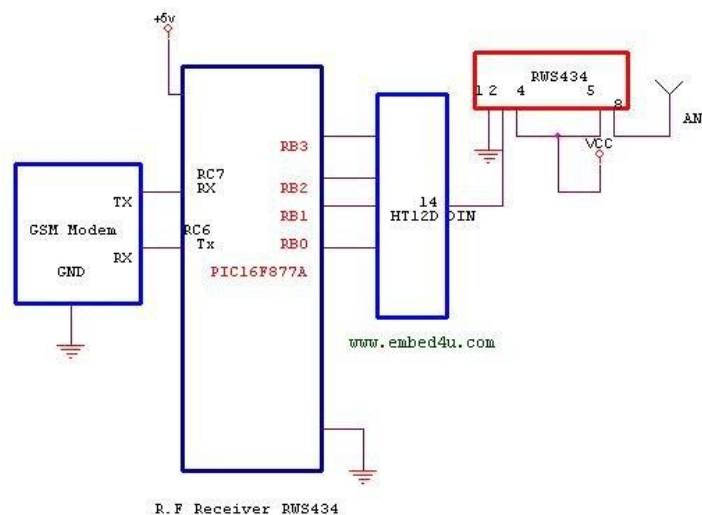


Figure 5.1 Example of a GSM Modem Interface with a Microcontroller

A modem is typically implemented on a microcontroller by connecting a hardware modem chip to the microcontroller's serial port or universal asynchronous receiver-transmitter (UART) interface. The modem chip typically contains the necessary

analog-to-digital (ADC) and digital-to-analog (DAC) converters, as well as the signal processing circuitry needed to modulate and demodulate data signals.

To use a modem chip with a microcontroller, the microcontroller sends data to the modem chip over its UART interface. The modem chip then modulates the data into a suitable analog signal that can be transmitted over a communication channel, such as a phone line or radio frequency channel. When receiving data, the modem chip demodulates the analog signal and sends the resulting digital data to the microcontroller.

The microcontroller can then process the received data and send responses back to the modem chip for transmission. In addition, the microcontroller can also control the modem chip by sending commands and configuration settings over the UART interface. To establish this communication in our project we created a function called *v_UART_Modem_Handler*.

5.1 Uart Modem Handler Function

The purpose of this code is to handle incoming data from a UART module and store it in a queue for further processing. The code is an ISR that is triggered when data is received by one of four UART peripherals. The function retrieves the received data, sends it to a queue, and sets a bit in an event group. If any of these operations cause a higher-priority task to become ready to run, the function will then set a flag to indicate that a task switch should occur.

A switch statement is used to determine which UART module (0-3) triggered the interrupt. Depending on the value of *vUART_MODEM* which is a variable set on another part of the code, the corresponding UART module's *vIIR* (Interrupt Identification Register) value is assigned to *error_code* which indicates the cause of the interrupt.

The code will then check if bit 2 (represented by the bitmask 0x4) which is used to indicate if there is data available to be read from the UART's receive buffer of *error_code* is set.

```
if(error_code & 0x4){
    send=vGetChar(vUART_MODEM);
    xQueueSendFromISR( QUEUE_MODEM_RX, &send, &lHigherPriorityTaskWoken );
    if(debugActive){
        if(!hexData){
            if(send=='\n'){
                dataReady++;
            }
        }
    }
}
```

Figure 5.1.1 Code snippet

The **xQueueSendFromISR** function is then called to add the received character to a queue called **QUEUE_MODEM_RX**. The **&send** argument will pass a pointer to the **send** variable so that its value can be copied to the queue. The **&lHigherPriorityTaskWoken** argument is then used to pass a pointer to the **lHigherPriorityTaskWoken** variable, which will be set to **pdTRUE** if a higher priority task needs to be woken up.

If a debug mode is active the code will check if the received data is in hexadecimal format. Otherwise it will check if the received character is a newline (**\n**) character. If it is, the **dataReady** variable will be incremented.

If instead the debug is not active the **xEventGroupSetBitsFromISR** function will be called, to notify the modem that the Bit was increased.

The function ends by calling the interrupt service routine (ISR) and telling it to perform any necessary context switching through the *lHigherPriorityTaskWoken* variable if a higher priority task needs to be woken up.

5.2 ModemInitPPP Function

The **ModemInitPPP** function was already written by one of my colleagues therefore we will not get very much into details on it however we will break it down briefly to give an insight on its purpose.

The function initializes a modem for a Point-to-Point Protocol (PPP) connection. PPP is a protocol used to establish a direct connection between two nodes in a network, such as a modem and a server.

It was first proposed as a standard by the Internet Engineering Task Force (IETF) in 1989 and became a working standard in 1994. The IETF specification for PPP is RFC 1661. PPP is a protocol most widely used by Internet service providers (ISPs) to enable dial-up connections to the Internet. It facilitates the transmission of data packets between point to point links.

PPP uses Link Control Protocol (LCP) to establish a session between a user's computer and an ISP. LCP is responsible for determining if the link is acceptable for data transmission. LCP packets are exchanged between multiple network points to determine link characteristics including device identity, packet size, and configuration errors.

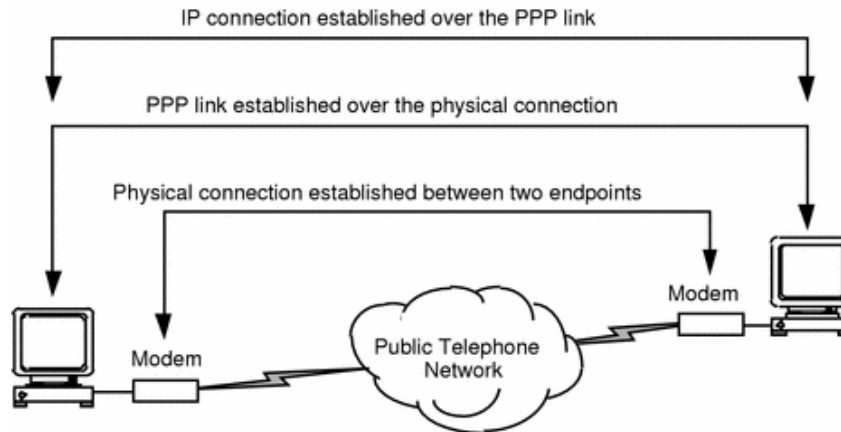


Figure 5.2 Established IP over PPP Links

Going back to the code, as we said, it initializes a modem for a PPP connection by checking the SIM card status, modem status, and signal quality before connecting to the GPRS network and checking the connection status. It starts by printing a status message to indicate the progress of the modem initialization process.

Then through a function called **ModemSendCommand()** we pass a command to the modem accompanied by a Boolean value which indicates whether the response needs to be show on the screen or not.

The first command, **"AT+CPIN?"** checks if the SIM card requires a PIN.

The second command, **"AT+COPS?"**, checks the status of the modem.

The third command, **"AT+CSQ=?"**, checks the signal quality of the modem.

Finally, the fourth command, **"AT+CGDCONT=1,\"IP\",,\"shared.tids.tim.it\""**, connects the modem to the GPRS network and after, the function prints the current connection status by sending the command **"AT+CGDCONT?"** to the modem.

5.3 ModemInit Function

The Modem Initialization, just like for the GPS is done by configuring the necessary pins and performing a startup sequence.

Just like for the GPS module, we started by checking the MTS – 2046 datasheet to know the pin numbers we need to work on and their direction to establish the network connection, such as a cellular connection, which can be used to transmit data or make voice calls.

We will repeat the same process we did for the GPS Initialisation. We start by configuring the first pin as Gpio. After having checked the Pinsel3 section in the table to know the pins function, we express them in decimal form and then shift the pin to the right number of bits needed by using the '<<' operator. Finally, we perform the Bitwise complement Operation through the xor denoted by (~), which is a unitary

operator, meaning that will work only on one operand, and it will change the 1s to 0 and the 0s to 1.

After setting pin 1.19 as GPIO we then worked on setting it as an output SOFT ON as its direction stated on the table. We do this by using the `vGPIO1->vFIODIR |=` to set the direction of the pin. Finally, we set it as an Open Drain through the “`vPINCON->vPINMODE_ODI |=`” operation.

We perform the same operation to set the pin 2.6 as Gpio then we clear the bit 6 of the FIODIR register on port 2, which sets pin 2.6 as an input pin which is used to detect when the modem is receiving an incoming call or message, also known as the RING Signal.

Lastly, after setting the pin 1.10 which is used to send a "modem wake-up" signal to the modem, necessary after the modem has been in a low-power state, as GPIO, we set its direction as an output through the FIODIR Operation.

After we have configured the various pins, we move forward on checking whether the Modem is awake and if so we turn it off by clearing one of the pins.

A delay of 100ms is introduced through `vTaskDelay` to give the modem time to initialize before any commands are sent to it.

We end this function by setting an **initReady** flag to 1 to indicate that the modem has been initialized and is ready to use.

5.4 ModemInitLCP Function

Moving forward we define a function called **ModemInitLCP()** whose purpose is to that initialize the Link Control Protocol (LCP) for a modem. The LCP is a protocol used in Point-to-Point Protocol (PPP) to establish, configure, and test the data-link connection between two communication endpoints. “It also imparts negotiation for set up of options and use of features by the two endpoints of the links.”[16] When PPP tries to communicate, it sends out LCP packets prior to the establishment of connections over the point – to – point link who then check the communication line to ascertain whether it can sustain the data volume at the required speed. Accordingly, it agrees upon the size of the data frame. It also identifies the linked peer and detects configuration errors. If the LCP accepts the link, it establishes and configures the link so that communication can proceed. If the LCP concludes that the link is not working properly, it terminates the link.

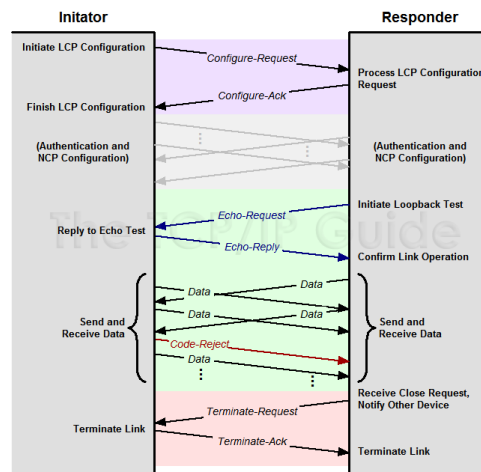


Figure 5.4 PPP Link Control Protocol (LCP) Message Exchanges

We start the function by declaring two variables **flagReceived** and **pos** which are both initialized to zero. They are used to keep track of how many LCP flags have been received and the position in the **LCPData** buffer where the received data will be stored, respectfully.

We then check if there is any data available in the **QUEUE_MODEM_RX** queue using the *xQueueReceive()* function. If yes, one byte of data will be read and stored in the *temp* variable which is then stored in the **LCPData** buffer at position **pos** whose value is then incremented.

Moving forward we check whether the byte that was just read is an LCP flag (**PPP_LCP_FLAG**) and if so the **flagReceived** counter is incremented.

Then though a *for* loop we go through each byte of data that were received and stored in the **LCPData** buffer one at a time, and then send them to a serial port.

5.5 ModemTask Function

In the previous chapter we established the communication between the Console and the Modem, from the Console Side. Now we will continue working on it from the Modem side.

We start by creating a function called **vModemTask** whose purpose is to receive data from the modem, parse the data, and send the data to the console task.

The function is made to communicate with the modem and receive messages from it. It uses an event group to keep track of when characters are received from the modem, and a queue to store those characters until they can be processed. When a new message is received (indicated by the '+' character), the code stores the following characters in a message string buffer until the end of the message is reached (indicated by a carriage return or line feed character). The code then processes the message and prepares to receive the next one.

Firstly we initialise some variables to such as, **modem_char** which is the character coming from the modem, **mchar_count** is a counter for the number of characters received from the modem, **store_mstring** is a flag that enables writing a string.

We then define an event bit mask using the FreeRTOS **xEventGroupWaitBits()** function which waits for one or more bits to be set in the **xEventGroupMODEM** event group with a timeout of 5000 milliseconds.

Then we call the **ModemInit()** function to initialize the modem and print a message to let the user know that the Modem task is starting.

We then set an event group using the **xEventGroupWaitBits()** function, which takes several parameters. The first parameter **xEventGroupMODEM** handles the event group being used. To indicate the specific bits that need to be set in the event group for the function to return we use two bit masks called **MODEM_BIT_0** and **MESSAGE_BIT_3**.

In order to avoid waiting for all the bits to be set **pdFALSE** is passed as the third parameter which indicates that any of the specified bits being set will cause the function to return.

The fourth parameter is also **pdFALSE** and it indicates that the bits will be cleared automatically once the function returns. Lastly we decide on the maximum amount of time that the function will wait for the specified bits to be set before timing out and returning.

```
uxBits = xEventGroupWaitBits(  
    xEventGroupMODEM,  
    MODEM_BIT_0 | MESSAGE_BIT_3 ,  
    pdFALSE, // pdTRUE,  
    pdFALSE,  
    xTicksToWait);
```

Figure 5.5.1 Code snippet

Just like for the previous modules, the task will wait for two different events: **MODEM_BIT_0** and **MESSAGE_BIT_3**. **MODEM_BIT_0** indicates that a character has been received from the modem, and **MESSAGE_BIT_3** indicates that a message has been received from the console.

In the first case “**if(uxBits & MODEM_BIT_0)**”, the event is set when a character is received from the modem. If the **MODEM_BIT_0** flag is set, we clear the flag using the *xEventGroupClearBits* function which is used to clear one or more bits in an event group.

We then enter a while loop that reads the characters from the queue we have called **QUEUE_MODEM_RX**. This is done by using the **xQueueReceive** function which reads a character from the queue and stores it in the *modem_char* variable we mentioned previously. The **xQueueReceive** is a FreeRTOS function which takes

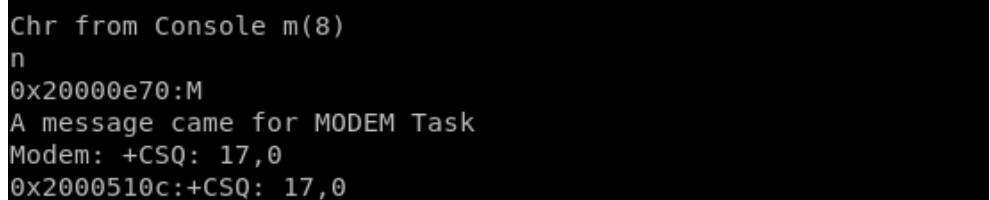
three arguments: the queue to read from (*QUEUE_MODEM_RX*), a pointer to the variable where the read data will be stored (*&modem_char*), and the amount of time to wait for data if the queue is empty (10 ticks in our case). We check that the data was successfully read from the queue using the *pdPASS* value.

So far, we have worked on enabling the task to read a character from the queue. We will then check whether the character read from the modem is the “+” sign which signifies that we are in the beginning of a string. If so, we will set the **mchar_count** variable to 0 to indicate that a new message is being received and the **store_mstring** variable to true to indicate that the characters that follow should be stored in the string pointer buffer. Finally, we add the “+” character to the message buffer to keep track of the current position in the buffer.

We also check whether the character coming is a carriage return or line feed character (*\r or \n*) and if not we proceed on adding the character to the message pointer buffer to keep track of the current position in the buffer.

On the other hand, if the character read from the modem is actually a carriage return or line feed character (*\r or \n*) and *mchar_count* is greater than 0 which indicates that some characters have been stored in the message buffer the message is terminated by adding a null terminator (*\0*) to the end of the message buffer.

In the second case “**if(uxBits & MESSAGE_BIT_3)**”, the event is set when a message has been received by the modem so the task will read the message from the *QMsgMODEM* queue and store it in the message variable. The function waits up to 10 ticks for a message to be available in the queue. We then take a semaphore called **serialSemaphore** which as we have stated before it is used to prevent multiple tasks from accessing the console output at the same time and then we print the value of the **message** variable as both a hexadecimal value and a string to the console using **printf()** and then give the **serialSemaphore** semaphore back to the system.



```
Chr from Console m(8)
n
0x20000e70:M
A message came for MODEM Task
Modem: +CSQ: 17,0
0x2000510c:+CSQ: 17,0
```

Figure 5.5.2 Output shown on the Command Prompt

We then create an integer pointer **pstrm** that points to the memory address of the **str_m.message** string and the send its value to the *QMsgConsole* queue, which is used by another task to receive messages. We again take the **serialSemaphore** semaphore and print the value of *str_m.message* to the console and then give it back to the system.

Finally, we check if none of the bits in the *uxBits* variable are set. If so we assume that either a message or a character is expected to be received from the modem.

A screenshot of a terminal window titled "GtkTerm - /dev". The window has a menu bar with "File", "Edit", "Log", "Configuration", "Controlsignals", "View", and "Help". The terminal output consists of five identical lines: "MODEM:No message or char received.".

Figure 5.5.3 Output shown on the Command Prompt

In this case we will send the *"ATE0;+CSQ\r"* string to the modem via the *vPutString* function using the *vUART_MODEM UART* port. This will disable the echoing of characters on the modem and retrieve the signal quality value of the network.

A message is also printed to the console using to indicate that no message or character has been received from the modem.

```
if( uxBits == 0 ) // not bits were set, so message or char will come
{
    vPutString("ATE0;+CSQ\r",vUART_MODEM);

    xSemaphoreTake(serialSemaphore,(TickType_t)TICKWAIT_SEMA);
    printf("MODEM:No message or char received.\n");
    xSemaphoreGive(serialSemaphore);
}
```

Figure 5.5.4 Code snippet

5.6 ModemSendCommand Function

This purpose of this next function is to send a command to the modem, wait for a response, and then print the response to the console for debugging or monitoring purposes. The function takes two arguments: a character pointer "command" that represents the command string to be sent to the modem, and an integer *"muxTrue"* that indicates whether the command requires a multiplexing command to be sent after it. The *"vPutString"* function is used to send the command string to the modem with

the UART port `"vUART_MODEM"`. If `"muxTrue"` is true, two extra characters, `'\r'` and `'\n'`, are also sent to terminate the command. The function then delays the task for 500 milliseconds using `"vTaskDelay"` to give the modem time to process the command.

After that, we use the `"vGetString"` function to read the modem's response to the command into the `"receive"` character array. It will continue reading the characters from the UART buffer until a newline character is encountered or the buffer is full.

Finally, the modem's response is printed to the console using `"printf"`. There is another delay of 1000 milliseconds using the FreeRTOS function `vTaskDelay` to allow time for the response to be printed to the console before the function returns.

5.7 Modem Debug Command Function

The last function introduced in this chapter is called **ModemSendDebugCommand** and its purpose is to send debug commands to the modem and receive a response from it. The communication with the modem is done by using a UART interface to waiting for a response using a timeout mechanism. Once the response is received, it is then printed to the console.

The function starts by taking two arguments, a pointer to a character array **command** and an integer **muxTrue**. The **command** argument contains the debug command to be sent to the modem while the **muxTrue** argument is a flag that indicates whether the command is being sent over a multiplexer, which is a device that enables multiple signals to be transmitted over a single channel. The multiplexer, shortened to “MUX” or “MPX”, is a combinational logic circuit designed to switch one of several input lines through to a single common output line by the application of a control signal. They operate like very fast acting multiple position rotary switches connecting or controlling multiple input lines called “channels” one at a time to the output. They can be either digital circuits made from high speed logic gates used to switch digital or binary data or they can be analogue types using transistors, MOSFET’s or relays to switch one of the voltage or current inputs through to a single output.

```
vPutString(command,vUART_MODEM);
if(muxTrue){
    vPutChar('\r',vUART_MODEM);
    vPutChar('\n',vUART_MODEM);
}
while(dataReady!=2 && (timer<TIME_OUT_VALUE || dataReady!=0)){/

    timer++;
    if(dataReady!=0){

    }

}
```

Figure 5.7.1 Code snippet

The function first initializes a timer and a character array **receive**, which will be used to store the response from the modem and then sets a global variable **dataReady** to 0. Afterwards it prints the command to the console and sends the command to the modem using the **vPutString** function. If the **muxTrue** is true, the function also sends a carriage return and line feed character to the modem using the **vPutChar** function.

Moving forward the function enters a loop that checks the **dataReady** variable to see if any data has been received and waits for a response from the modem. If no data has been received and the timer has not yet reached a predefined timeout value (**TIME_OUT_VALUE**), the loop will continue to wait for data. If data is received or the timer exceeds the timeout value, the loop then exits.

If the data is received within the timeout period, the response from the modem is read into the **receive** array using the **vGetString** function, and the response is printed to the console otherwise if the timeout period was exceeded and no data was received, the function prints the message "Time out" to the console.

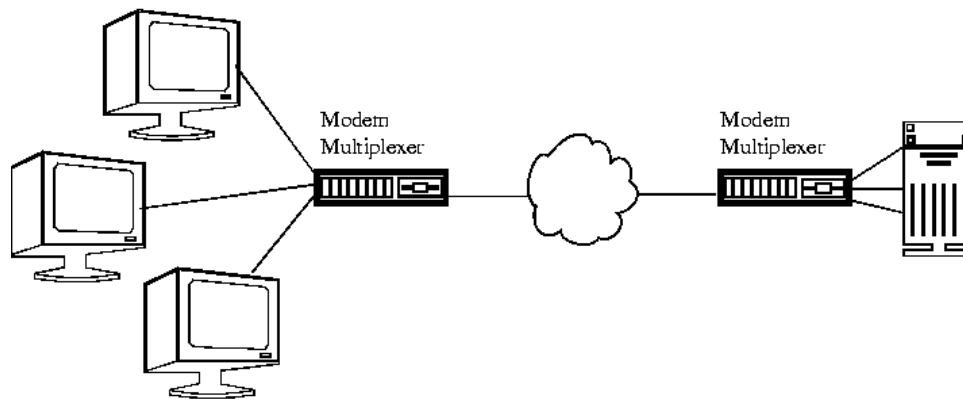


Figure 5.7.2 Single line Multiplexer Connection

CHAPTER 6

Migrating from the LPC17xx to the GD32F305 microcontroller

As it was mentioned in the beginning of this project, the scope was to write the firmware for a GD32F305 Microcontroller. We started from the LPC17XX as the project had already started before I joined the company for the internship, so we decided it was for the best to continue the project where it was left, then after implementing all the modules we would migrate the project from one microcontroller to the other.

The scope was not only migrating to the previously mentioned microcontroller, but also to separate the specific code needed for a specific microcontroller to the general one. This way in the future we would be able to change the microcontroller to another one and focus on some very few specific changes to be made, instead of having to start writing the firmware from the scratch.

So the next step to be done was to group all the functions that were written to match the specifications of a certain microcontroller in one .C file that we called “Subfunctions”.

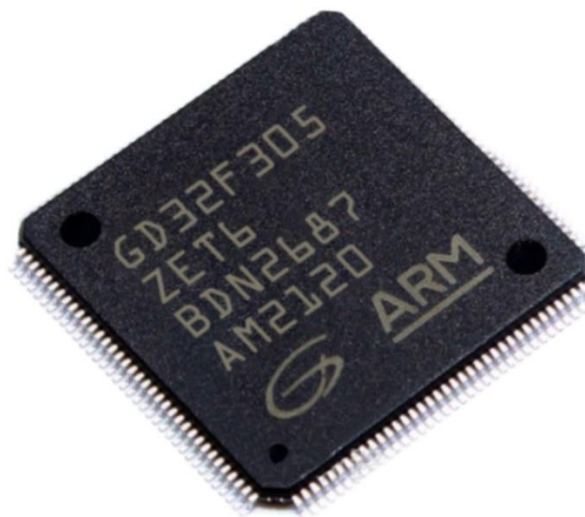


Figure 6 GD32F305 Microcontroller

6.1 CPU_SerialInit

In the previous chapters we explained thoroughly how we established the UART Communication between the Modules we worked on and the microcontroller. When we migrated from LPC17XX to GD32F305 we decided to rewrite these functions based on the pre-defined functions which are part of the firmware written by the developers of the GigaDevice Semiconductor Inc.

We start by defining a function we named **"CPU_SerialInit"** that is going to be used to initialize and configure the different UART devices for serial communication on a microcontroller. The function takes two parameters, the first one is an integer that specifies the UART device to be initialized, and the second one is the baud rate to be used for serial communication. We then use a switch statement to select the appropriate UART device based on the input parameter. There are three possible cases: USART0, USART1, and UART4 used for the GPS, Modem and the Console.ù

```
void CPU_SerialInit (int UART,int baudrate) {  
  
    switch(UART){  
        case USART0:  
            // All clock GPIO already enabled into prvSetupHardware  
            //rcu_periph_clock_enable(VUSART0_SET_CLK);  
            rcu_periph_clock_enable(RCU_USART0);  
            //gpio_pin_remap_config(GPIO_USART0_REMAP, ENABLE);  
            /* configure USART Tx as alternate function push-pull */  
            gpio_init(VUSART0_PINSEL, GPIO_MODE_AF_PP, GPIO_OSPEED_50MHZ, VUSART0_PINSEL_TX);  
            /* configure USART transmitter */  
            usart_transmit_config(USART0, USART_TRANSMIT_ENABLE);  
            /* configure USART receiver */  
            usart_receive_config(USART0, USART_RECEIVE_ENABLE);  
            /* enable USART */  
            usart_enable(USART0);  
            break ;  

```

Figure 6.1 Code snippet

For each case of the UART device we enable the peripheral clock and configure the GPIO pins for the UART transmission with the specified settings: alternate function push-pull mode, 50MHz output speed, and the selected pin. We then use the *"usart_transmit_config"* and *"usart_receive_config"* functions to enable the USART transmitter and receiver for the selected UART device, respectively. This is done to enable the specified USART peripheral to transmit and receive data over the serial interface.

We repeat the same process for the other two UART Cases and then finally, the *"usart_baudrate_set"* function is used to set the baud rate for the selected UART device which determines the rate at which data is transmitted over the serial interface and must be set to the same value on both the transmitting and receiving devices.

6.2 CPU_UartIntInit Function

In order for the microcontroller to efficiently receive data from the above-mentioned devices and also allow it to process and respond to it in a timely manner, we created another function called *CPU_UartInit*. The purpose of this function is to enable USART interrupts for each specified UART module and establish the communication with the GPS, modem, and console, which are sending and receiving data in a serial format.

By using the **usart_interrupt_enable** function which is part of the already written firmware for the microcontroller we enable the USART interrupts for the specified UART modules and allow the interrupt to be triggered when the receive buffer is not empty. This means that the UART will generate an interrupt whenever a new byte is received and stored in the receive buffer.

```
void CPU_UartIntInit(int UART)
{
    usart_interrupt_enable(UART, USART_INT_RBNE) ;
}
```

Figure 6.2 Code snippet

6.3 Vuart HANDLER Functions

As we have mentioned in the previous chapters , for each device we need to set up the interrupt service routine (ISR) for the UART Handlers. The ISR on its own is a function that is automatically executed when an interrupt is generated and whose purpose is to handle incoming data on the UART interface, which is typically used for serial communication.

The Uart Handler functions for each device are the same as the ones we mentioned in the previous chapter, they were just moved in the Subfunctions file as they were specifically made for the modules that belong to the microcontroller we were working on.

The only difference is that they are introduced inside an if loop “**if(RESET != usart_interrupt_flag_get(vUART_GPS, USART_INT_FLAG_RBNE))**” where we check if there is data available in the receive buffer of the UART module by using the **usart_interrupt_flag_get** function and execute the block if there is any data available

6.4 SerialGetChar Function

In order to provide a simple interface for retrieving characters from the console UART we create a function called *SerialGetChar* whose purpose is to return a character received on the console UART.

Inside we call the *vGetChar* function and pass the **vUART_CONSOLE** variable, which is a pointer to the UART Peripheral, responsible for receiving and sending data, as an argument.

6.5 vGetChar Function

The purpose of this next function is to read a single character from any of the UART peripherals. We start by defining the function which we named *vGetChar()*.

Essentially what the function will do is take an integer from the UART as an input and return a single character from it.

Inside we call a function named **usart_data_receive()** which is part of the pre-defined functions of the GD32F305 firmware library which interacts with the hardware of the microcontroller and passes the UART integer as an argument.

```
char vGetChar(int UART)
{
    return ((char)usart_data_receive(UART));
}
```

Figure 6.5 Code snippet

6.6 SerialPutChar Function

In the previous chapters we have mentioned how we have established the communication between the three modules, where messages and characters are exchanged back and forward between them. We have talked about how we have sent specific characters to the Uart Console and it would return back another character as an input. This is all done by the function we have called **SerialPutChar()**

Inside of it we call the function named **vPutChar()** which we will explain more in detail afterwards, where we pass a character called *c* as an argument and to specify the UART Console peripheral we use the **vUART_CONSOLE** variable

The **vPutChar()** function will return a character value indicating whether the character was successfully transmitted over the UART.

6.7 vPutChar Function

As we mentioned before in the previous paragraph, we use the function **vPutChar** to return a character value indicating whether the character was successfully transmitted over the UART.

Inside the function we firstly call the **usart_data_transmit()** which is a function from the GD32F305 Firmware Library whose purpose is to pass the *c* character and the *UART* integer as an argument.

```
char vPutChar(char c,int UART)
{
    usart_data_transmit(UART, c);
    while(RESET == usart_flag_get(UART, USART_FLAG_TBE));
    return (c);
}
```

Figure 6.7 Code snippet

Next, we enter a while loop to wait for the transmit buffer to be empty before continuing to execute the rest of the code. Inside this loop we call the **usart_flag_get()** function to check whether the transmit buffer is empty in order to ensure that the character is transmitted before the function returns.

6.8 ModemInit Function

In the previous chapter we initialized the modem by configuring the necessary pins and setting them as GPIOs and deciding their directions through different operations such as PINCON, FIODIR, etc.

As we mentioned before, the microcontroller contains a firmware library where hundreds of important functions are written, including the ones that helped us initialize the Modem module and the GPS one as we will see in the next paragraph.

We start by defining a function named **ModemInit()** as we did previously. Inside we configure the two Modem Signals we got from the company's datasheet as GPIO pins by using the **gpio_init** function which is part of the firmware library for the

microcontroller. Inside the function we declare its arguments, starting by the *gpio_periph* , the signals direction and mode as an output in open-drain mode and an output in push-pull mode respectively.

We then check whether the modem is on by using the **GPIO_ISTAT()** macro to verify the status of the pin, and if not we turn it on by using the **GPIO_BC()** macro on the pin. To give the modem the necessary time to wake up we use the **vTaskDelay()** for 100ms.

If instead we want to turn the modem off we use the **GPIO_BOP()** macro to toggle the state of the pin.

We wait again for 300 milliseconds using the **vTaskDelay()** function and then set a flag called **initReady** to 1 to indicate that the modem has been initialized.

6.9 GPSInit Function

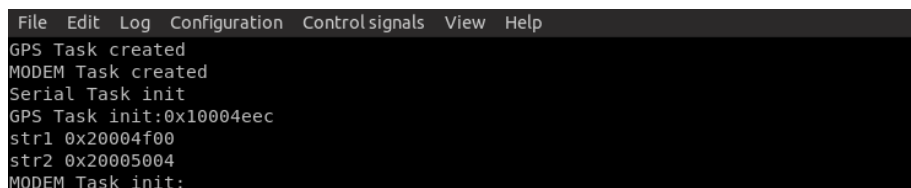
The GPS Module initialization is done following the same procedure as we did for the Modem. In essence we configure its power state and send a command to set it up in the following way.

Firstly, we turn the GPS on by using the (*GPIO_BC(GPIOB) = GPIO_PIN_0;*) command. We checked the signals' function and direction on the Datasheet of the company as we did previously for the Modem.

By using the *gpio_init* function we then configure the pin as an output with push-pull mode and a speed of 50MHz (*gpio_init(GPIOB, GPIO_MODE_OUT_PP, GPIO_OSPEED_50MHZ,GPIO_PIN_0);*).

We then delay the function for 300 Ms to give time to the GPS module to power up and get stabilized.

Finally we send to the GPS Module a command which sets up the GPS module to output GPRMC and GPGGA sentences at a rate of 1Hz in order to have it configured. Afterwards we set the flag *tiu1* to indicate that the initialization has been completed.



```
File Edit Log Configuration Controlsignals View Help
GPS Task created
MODEM Task created
Serial Task init
GPS Task init:0x10004eec
str1 0x20004f00
str2 0x20005004
MODEM Task init:
```

Figure 6.9 Snippet from the command prompt when the CPU is running.

CHAPTER 7

Conclusions

Embedded systems play a crucial role in various industries. In order to guarantee reliable performance, they must be capable of providing multitasking, real-time processing, and seamless module integration. In this thesis, we focused on developing firmware for embedded systems using FreeRTOS, an open-source real-time operating system. We started from an already existing project with an ideal to make it CPU independent. We worked on the GPS, Modem and Console module and were able to establish the communication between them. We were able to migrate the project from one CPU to another all thanks to RTOS and as a result of our research done throughout this entire project, we have drawn several conclusions.

Firstly, we found that FreeRTOS provides an efficient and practical approach to firmware development in embedded systems. Thanks to the management features of FreeRTOS, we were able to provide efficient multitasking, making it possible to execute multiple tasks with different priorities simultaneously. This aided the integration of modules such as GPS, modem, and serial output, resulting in better performance and reliability. The handling of communication protocols and data exchange between modules was also made easier thanks to FreeRTOS, contributing to the overall efficiency of the embedded system.

Secondly, we found that FreeRTOS enabled the development of real-time capabilities in the embedded system. The scheduling and prioritization features of FreeRTOS ensured timely and deterministic execution of critical tasks, such as communication with the GPS and modem modules. This provided the system to be more safe and reliable, making it suitable for time-critical applications where deadlines must absolutely be met.

As we just mentioned the reliable features of FreeRTOS, such as memory protection and error handling, helped in isolating and mitigating software faults which increased the stability and resilience of the system against unexpected events or failures. The error handling capabilities of FreeRTOS allowed the system to recover from errors, preventing it from crashing or responding in an unintended behaviour.

Lastly, our findings suggest several future directions for research and development in the field of embedded systems and RTOS. For instance, further optimization of power consumption could be explored, considering the increasing demand for energy-efficient embedded systems in various applications. Additionally, the integration of additional modules or sensors, implementation of advanced communication protocols, or application of real-time scheduling algorithms could be investigated to further improve system performance. The potential for customization and

extensibility offered by FreeRTOS makes it a viable platform for future innovation in the field of embedded systems.

In conclusion, our thesis enhances how effective FreeRTOS can be when it comes to firmware development in embedded systems. The use of FreeRTOS enabled efficient firmware development, demonstrated portability and scalability, provided real-time capabilities, enhanced system reliability, and opened up possibilities for future research and development. The findings from this thesis will contribute to the companies continuity on this project and pave the way for an overall migration to Real Time Operating Systems. The use of FreeRTOS as an RTOS for firmware development in embedded systems holds promise for addressing the challenges associated with developing reliable and efficient embedded systems in various domains.

Bibliography

- [1] www.sciencedirect.com | Real Time Systems - an overview | ScienceDirect Topics
- [2] www.tutorialspoint.com | Embedded Systems Overview
- [3] www.blackberry.qnx.com | Ultimate Guide to Real-Time Operating Systems
- [4] www.en.wikipedia.org | FreeRTOS – Wikipedia
- [5] www.freertos.org | Mastering the FreeRTOS Real Time Kernel Documentation
- [6] www.guru99.com | Mutex vs Semaphore – Difference Between Them
- [7] www.freertos.org | FreeRTOS Mutexes
- [8] www.ubuntu.com | The story of Ubuntu
- [9] www.wblog.wiki | Oscilloscope
- [10] www.sciencedirect.com | Boundary Scan
- [11] www.en.wikipedia.org | JTag – Wikipedia
- [12] www.arm.com | Cortex-m4
- [13] www.analog.com | UART: A Hardware Communication Protocol
Understanding Universal Asynchronous Receiver/Transmitter
- [14] www.docs.espressif.com | FreeRTOS Overview
- [15] www.en.wikipedia.org | General-purpose input/output
- [16] www.tutorialspoint.com | Link Control Protocol (LCP)

Figures bibliography

- Figure 1.1 - www.javatpoint.com | Hard and Soft Real-Time Operating System
- Figure 1.2 - www.iar.com | Resource Sharing in RTOS-based Designs
- Figure 1.2.1 - www.iotdunia.com | Embedded Systems Basics
- Figure 1.2.2 - www.tutorialspoint.com | Basic Structure of an Embedded System
- Figure 1.2.3 - www.circuitstoday.com | Microprocessor and Microcontroller – The difference
- Figure 1.3.1 - www.tutorialspoint.com | Operating System - Overview
- Figure 1.4.1 - www.theengineeringprojects.com | Real Time Embedded Systems: Definition, Types, Examples and Applications
- Figure 1.4.2 - www.sciencedirect.com | Microkernel
- Figure 1.5 - www.hackaday.com | Getting started with FreeRTOS and Chibios
- Figure 1.5.1 - www.freertos.org | Tasks
- Figure 1.5.2 - www.digikey.com | Introduction to RTOS - Solution to Part 5 (FreeRTOS Queue Example)
- Figure 1.5.3 - www.keil.com | Semaphores
- Figure 1.5.4 - www.freertos.org | Event Bits (or flags) and Event Groups
- Figure 2.1.1 - www.blog.cloudflare.com | The Linux Kernel Key Retention Service and why you should use it in your next application
- Figure 2.1.2 - smartstore.com | Utilizzo di Visual Studio Code come editor di codice sorgente
- Figure 2.2.1 - Oscilloscope RIGOL DS1104Z
- Figure 2.2.2 - MAISHENG DC 0 ~ 30 V, 0 ~ 10A MS3010D DC Power Supply
- Figure 2.2.3 - Segger JTag Connector
- Figure 2.2.4 - Microcontroller GD32F305
- Figure 3 - www.analog.com | UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter
- Figure 3.2 - GPIO register map (local bus accessible registers - enhanced GPIO features) from the LPC17xx Family_UserManual_1008
- Figure 3.2.1 - www.brandidowns.com | Decoding NMEA Sentences
- Figure 5.1 - www.pic-microcontroller.com | GSM Modem Interface with PIC 18F4550 Microcontroller
- Figure 5.2 - www.docs.oracle.com | Introducing Solstice PPP
- Figure 5.4 - www.tcpipguide.com | PPP Link Control Protocol (LCP)
- Figure 5.7.2 - www.telecomworld101.com | Multiplexing
- Figure 6 - GD32F305 Microcontroller