KAUNAS UNIVERSITY OF TECHNOLOGY

ELECTRICAL AND ELECTRONICS ENGINNERING FACULTY

VIGNESH TIRUMALAI GOVINDASAMY

"RESEARCH OF EMBEDDED OPERATING SYSTEMS FOR A WIRELESS CELLULAR IOT SYSTEM"

Master's Degree Final Project

Supervisor
Prof. Vytautas Markevičius

KAUNAS, 2017

KAUNAS UNIVERSITY OF TECHNOLOGY
ELECTRICAL AND ELECTRONICS ENGINEERING FACULTY
DEPARTMENT OF ELECTRONICS ENGINEERING

"RESEARCH OF EMBEDDED OPERATING SYSTEMS FOR A WIRELESS CELLULAR IOT SYSTEM"

Master's Degree Final Project

Electronics Engineering (621H61002)

Supervisor
(signature) Prof. Vytautas Markevičius
(date)

Reviewer
(signature) Assoc. Prof. Mindaugas Knyva
(date)

Project made by
(signature)Vignesh Tirumalai
Govindasamy
(date)

KAUNAS, 2017

KAUNAS UNIVERSITY OF TECHNOLOGY

Electrical and Electronics Engineering Faculty
(Faculty)

Vignesh Tirumalai Govindasamy
(Student's name, surname)

Electronics Engineering 621H61002
(Title and code of study programme)

"Embedded Operating Systems and Security Solution for a Wireless Cellular IoT System"

DECLARATION OF ACADEMIC INTEGRITY

13 June 20 17
Kaunas

I confirm that the final project of mine, **Vignesh Tirumalai Govindasamy**, on the subject "**Research of Embedded Operating Systems for a Wireless Cellular IoT System**" is written completely by myself; all the provided data and research results are correct and have been obtained honestly. None of the parts of this thesis have been plagiarized from any printed, Internet-based or otherwise recorded sources. All direct and indirect quotations from external resources are indicated in the list of references. No monetary funds (unless required by law) have been paid to anyone for any contribution to this thesis.

I fully and completely understand that any discovery of any manifestations/case/facts of dishonesty inevitably results in me incurring a penalty according to the procedure(s) effective at Kaunas University of Technology.

(name and surname filled in by hand) *(signature)*

Tirumalai Govindasamy Vignesh. Research of Embedded Operating Systems for a Wireless Cellular IoT System. Final Project of Electronics Engineering, *Master's degree* / supervisor Prof. Vytautas Markevičius. Kaunas University of Technology, Faculty of Electrical and Electronics Engineering, department of Electronics Engineering.

Research area and field: Electrical and Electronics Engineering, Technological Sciences

Key words: NB-IoT, MBED OS, FreeRTOS, ARM Cortex M33

Kaunas, 2017. 94 p.

SUMMARY

The master degree final project focusses on Embedded Operating Systems for a Wireless Cellular IoT System. Internet of Things (IoT) is a great chance for upcoming devices to be intelligent, more robust and efficient. This tremendous pathway has become available due to regular cost degradation of various separate systems and accessories like sensors, computing devices, communication methods, the cloud and the big data paradigms. Connectivity is the base for IoT and the type of access needed will focus on the nature of the application. Thus, the target is now on Narrow Band IoT, which is a Low Power Wide Area Network (LPWAN) radio technology standard that has been developed to contribute a wide range of devices and services to be connected via cellular telecommunications bands. Accordingly, the thesis works on ARM MBED OS, an embedded operating system, which is a platform as well as operating system for internet connected devices for 32-bit ARM cortex-M microcontrollers which is needed for NB-IoT system. First, MBED OS is designed and implemented on ARM cortex -M prototyping system MPS2+ as a real-time operating system by bringing latest version of CMSIS-RTOS with RTX as kernel on Cortex-M4 as well as its successor Cortex-M33 contained on MPS2+ hardware board to examine different RTOS parameters such as memory, heap, stack, hardware and software impacts. Next, these obtained parameters for MBED OS is compared with other RTOS, say FreeRTOS on MPS2+ board. Thus, the final outcome would be how cellular IoT system will change when a new embedded operating system will be incorporated into Corelink SSE 200 IoT subsytem and fulfil the requirements for NB-IoT standard.

Tirumalai Govindasamy Vignesh. Tyrimai įterptines Operacines sistemas skirtus belaidžio korinio daiktų interneto (DI) sistemoms. Elektronikos inžinerijos Magistro baigiamasis projektas / vadovas doc. Vytautas Markevičius. Kauno technologijos universitetas, fakultetas Elektros ir elektronikos inžinerija, Elektronikos inžinerijos katedra.

Mokslo kryptis ir sritis: Elektros ir elektronikos inžinerija, Technologiniai mokslai

Reikšminiai žodžiai: NB-IoT, MBED OS, FreeRTOS, ARM Cortex M33

Kaunas, 2017. 94 p.

SANTRAUKA

Galutinis magistro tezių projektas orientuotas į įterptines Operacines sistemas skirtus belaidžio korinio daiktų interneto (DI) sistemoms. Daiktų internetas yra puiki galimybė būsimiems elektronikos prietaisams būti protingais, stabilesniais ir veiksmingesniais. Šis milžiniškas kelias tapo prieinamas dėl įvairių atskirų sistemų ir jų priedų reguliaraus kainų mažėjimo, pavyzdžiui, jutikliai, kompiuteriniai įrenginiai, ryšių metodai, debesų ir didelių duomenų technologijos. Ryšiai yra DI pagrindas ir prieigos prie specifinės aplikacijos galimybė. Taigi, tikslas yra siauros dažnių juostos DI, kuris yra mažo galingumo plačios aprėpties tinklo (LPWAN) radijo technologijos standartas, sukurtas siekiant tiekti platų įrenginių ir paslaugų spektrą, sujungus įrenginius korinio ryšio juostose. Darbas skirtas ARM MBED OS, įterptinei operacinei sistemai, kuri yra platforma, taip pat ir operacinė sistema prie interneto prijungtiems 32 bitų ARM Cortex-M mikrokontrolerių prietaisams ir, kaip saugumo sprendimas (TrustZone), kuris yra reikalingas NB- DI sistemose. MBED OS sukurta ir įgyvendinama remiantis ARM Cortex -M prototipų sistemos MPS2 + kaip realaus laiko operacinė sistema. Sekanti, naujausia versija, CMSIS - Realaus laiko OS su RTX kaip branduoliu Cortex-M4, taip pat jo įpėdiniui Cortex-M33. Darbe nagrinėjama MPS2 + Techninės įrangos maketinė plokštė, išnagrinėti skirtingi Realaus laiko OS parametrai, tokie kaip atmintis, statinė ir dinaminė alokacija, steko, techninės ir programinės įrangos poveikiai. Šie gauti parametrai MBED OS yra lyginami su kitomis Realaus laiko OS, tarkime FreeRTOS funkcionuojančiose MPS2 + plokštė aplinkoje. Taigi rezultate bus galima sukurti korinę DI subsistemą su į Corelink SSE 200 įdiegta nauja operacine sistema, tenkinančia DI standarto reikalavimus.

Content

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Overview of NB-IoT

A world where everything associates. Billions of sensors connect seamlessly to cloud based services which in turn drive new levels of efficiency, new businesses and new opportunities, this is the vision of how the Internet of Things (IoT) stands to transform the way we live, work and interact with the world around us [1].

Contributing connectivity which stabilize to multiple devices faces various innovative challenges. The latest technology referred as '5G' blended world assures to develop ever present coverage, unlimited bandwidth and more capacity to our previous emphasized wireless networks. Taking Ericsson Mobility Report into account, by 2022 it is assured that there will be nearly 18 billion IoT linked devices, where there will be tremendous increase in the number of smartphone connections by more than 2:1.

The main aspect of the wireless network is important in facilitating the IoT. With respect to different cellular technologies like steadily built for speech in 2G services, cellular broadband in 3G and 4G networks there arises a main defect to handle the challenge of developing Cellular IoT technology with high scalability, broad coverage and ultra-efficient low power consumption. On the other hand, interfacing a sensor depicts a completely distinct objection in contrast to associating a smartphone. To accomplish a stable and innovative cellular IoT technology for serving various applications, the focus is turned upon low power, wide area networks (LPWAN) to offer wide connectivity. Standards are still emerging, but the main step to this is a latest authorized technology known as Narrowband IoT (NB-IoT) which was affirmed by the global telecoms standards body which was behind 3G and 4G standards, namely 3GPP [1].

For any standard to be outstanding, it must be open, enabling device vendors to develop and interoperate which in turn drive thriving, multi-vendor markets. Accordingly, ARM acquired Mistbase and NextG-Com to develop Narrow Band IoT, which guarantees to deliver a platform of associated devices to be affordable and serving for various applications like smart city, agriculture, medical, automotive as well as more classic consumer based devices and wearables. To accomplish such emerging features needs a broad range of semiconductor solutions which implant NB-IoT connectivity into their core.

Cellular network operators have their respective role to operate however, migrating from a smartphone centric business model to an IoT is not a normal task. Highly efficient networks that deliver wide area coverage for the disparate IoT use cases is a challenge for operators as well as ensuring that security is built into the network for protection of memory and IoT nodes.

It is a keynote that three-main support of upcoming generation of ARM-enabled computing SoCs are low-power, high performance and security. Thus, the final master piece for stabilized NB IoT modem would be ARM CORDIO-N IP, which offers on-chip radio connectivity pathway for IoT System on Chip, whereby achieving LPWAN technology via NB IoT. It should be noted that ARM CORDIO-N IP is integrated into CoreLink SSE 200 Subsystem, which provides platform for developing safe IoT node by having dual ARM Cortex M33 processors. Thus, ARM is working to accomplish it by integrating various technologies like MBED OS (main theme of the project) and other power management functions within CoreLink SSE 200 Subsystem, since this subsystem is completely investigated IP, which reduces time to verify its internal characteristics, allowing developers to believe the IP and focus on integration.

Objective

The two-main goal of this project includes

a) To investigate/research/evaluate/study MBED OS, an embedded operating system.

b) To compare and analyse various functionalities of MBED OS with another OS (say FreeRTOS) to validate stable operating system for NB-IoT system.

In this project, the target will be on wireless Cellular IoT system.

The required work to be fulfilled for this thesis to accomplish above objectives includes:

1. Evaluating MBED OS and to examine the different ways of implementing MBED OS on ARMv7-M architecture (ARM Cortex-M4) as well as on ARMv8-M architecture (ARM Cortex M33 processor). Properties like OS-performance, memory footprint, heap and stack consumption will be reconnoitred to have deep knowledge of how a new RTOS will be affecting a cellular IoT system.
2. Next, to compare MBED OS functionalities against another RTOS (say FreeRTOS) to get familiarized with MBED OS pros/cons.

Problem statement

Why Narrow Band IoT?

Cellular IoT is an impressive technology that will have a huge impact on the world in many aspects. The technology had been available for many years using GSM communication technology. However, GSM have been operated greatly in early days, this technology has been overcome by new cellular standards such as NB IoT and LTE Cat-M because of following reasons.

1. **Low power consumption –** Current consumption in order of 1nA, which forces devices to work for maximum of 10 years on single charging cycle.
2. Low cost on device and effective.
3. Upgraded indoor and outdoor coverage compared with old wide area connectivity.
4. Secure connectivity and high authentication.

5. Optimised data switching, where it affords tiny, intermittent sections of data.

The reason why MBED OS is chosen as an embedded operating system and how it fits for NB-IoT is explained in detail in upcoming section.

Organization of thesis

The thesis is organized into 5 chapters, where it will elaborate in detail about this project.

Initially, an overview about this thesis including the objectives and problem statement is elaborated. This overview is used as a guide line to develop MBED OS on Cortex-M prototyping system MPS2+.

Chapter 1 will explain and justify on the literature review, which consists of the currently running system operated earlier and the proposed system for embedded operating system which is realized via standardized IoT protocols.

Chapter 2 elaborates deeply about MBED OS which is main theme of the thesis including architecture and world-wide deployment of NB-IoT. Further, this chapter illustrates services and applications offered by using MBED OS.

Chapter 3 focusses about different testing and results that are conducted to each module of the project. This chapter also concludes performance analysis of MBED OS and FreeRTOS.

Lastly, focussing on the conclusion and future scope for the development of MBED OS as an embedded operating system needed for wireless cellular IoT system(NB-IoT).

1. LITERATURE REVIEW

This chapter discusses about the researcher's review articles and past research about the theory and methods implemented in cellular IoT system and Embedded operating system and proposed system that are needed to fulfil the requirements of NB-IoT standards.

1.1 Researcher's Review

1.1.1 The need for sophisticated cellular IoT system – Enabling path to Narrow Band IoT

In past, Machine-to-machine communication(M2M) is an emerging technology as it defines: two machines communicating or exchanging data without human interaction, which includes serial connection or Powerline connection(PLC) [2]. As businesses have investigated the importance of M2M, it has been popularized on latest technology known as Internet of Things(IoT). The necessity for applications throughout the enterprise to adopt device data to contribute performance improvements, business innovation or other possibilities clearly distinguishes the potential of IoT vs. M2M [3]. This IoT-enabled data is mainly to be contributed to cloud computing, which allows permission by any approved firm application. In contradiction, M2M defines direct end-to-end communication. In addition, the cloud-enabled architecture shapes IoT to act as stable system thereby, eliminating the need of more complicated hard-wired connections and SIM card establishment [3]. This is key where, M2M is called as "**Plumbing"** that focusses to serve telemetry and remote sensing applications. On the other hand, IoT is picturized as a **"Universal enabler"** which contributes highly reliable and wide connectivity with unlimited access for low cost and energy enabled devices**.**

The Internet of Things (IOT) will blend billions of devices. It can be clearly noted that when there arise different deployment plans for IOT devices, the leading wireless technologies will also differ. For instance, home automation resumes powered, immobilized devices with near proximity which adopts Wi-Fi or mesh network technologies, transported by broadband system. On contrast, sensors that are supplied widely serves various applications such as smart metering, water governing and associated cities. However, these Wi-Fi and mesh networking topologies doesn't fit for distributed sensors, since they perform at low data rates packets, have less spontaneity in broadcasting the data low response and high-power consumption (battery operated). This forced a Cellular IOT (CIOT) technology to be a preferable solution, since it overcomes the above defects and became superior, where enormous consumer needs wireless broadband services, which has driven the last decade of telecom standards resulting in LTE advanced 4G multimode devices [4]. In addition, why NB-IoT is chosen as preferred cellular IoT system is explained in detail in upcoming section.

1.1.2 Why ARM MBED OS suited as Embedded operating systems?

An embedded operating system, also referred to as RTOS is a kind of OS that is embedded and essentially developed for a reliable hardware arrangement. Hardware adopting embedded OS is mainly shaped to handle easily signifying less density and highly portable where in turn this system is automatically preferred by various functions defined within non-embedded systems for efficient and stable operations.

Over the past decade, most new open source OS projects have shifted from the mobile market to the Internet of Things. This is to note that billions of IoT devices strongly built by various communication sectors that are working on various operating systems can lead to compatibility and time-to-market concerns. Thus, finally ARM proclaimed MBED IoT Device Platform, a multitasking and open source operating system, which serves to reduce the complexity and fasten the world-wide deployment for IoT enabled products based on the ARM Cortex-M architecture. In addition, MBED OS offers great support for memory management (i.e. main property of any operating system) by consuming below 256 kilobytes of memory. It is mainly designed to integrate Standardized IoT protocols like 6LoWPAN together with TLS/DTLS, MQTT and less weight M2M communication systems and security solutions and other services within single incorporated system which is stabilized for low power and cost effective. On contrast with ARM Cortex-A processors which operates on iOS or Linux operating systems, Cortex M chips don't have such OS standardisation and how it overcomes with other operating systems and becomes superior is explained in existing system. Further, it was defined that MBED OS as event driven operating system that enables device manufactures and developers to fast track designs by not having to worry about core needs such as protection, device management and low power operation as these features are incorporated within MBED OS. With great affordability for these building blocks, ARM strongly agrees that MBED OS will play major role for developers to deliver their IoT products to market.

1.2 Existing System

1.2.1 Cellular IoT sector – An overview

The main key point to note that IoT demands low difficulty, cost effective devices with low power consumption which also defines life cycle of battery to be long as well as wide coverage for long distance communication thereby, tracking locations like dense areas, where there is no possibility for good network coverage. Cellular networks wrap-up the globe with highly securable, more reliability for permitting fast mobile access adopted standards. Thus, a good foundation for IoT with stable technical adaptations and services forces to accomplish the cellular system with cost effective and long-life span of battery. This keynote inspired to investigate and build the necessary tools for Cellular IoT sector. There are different cellular IoT system approved by 3GPP, where they are compared to each other with respect to different parameters (i.e. bandwidth, duplex mode peak rate) and how NB IoT dominates other cellular IoT system and becomes superior for Low Power Wide Area Network (LPWAN) is described below:

1. **Cat-1 – Category 1 –** This was adopted in the LTE considerations earlier on Release 8. By adopting this system, there is a way to produce downlink of 10 Mbps and uplink of 5 Mbps data rates. Although, MIMO (Multiple Input and Multiple Output) is not affordable the User Equipment (UE) still demands 2 receiver antennas. Thus, Cat-1 system is non-supportive UE consideration for LTE-enabled mobile broadband services since it produces poor performance and not attaining 3G accomplishment. But recently it came to attract that Cat-1 is standardized where, it fulfils the considerations of a broad field of MTC applications but it's still degraded by NB-IoT features and its performance, which is described later in same chapter.
2. **Cat-0 – Category 0** – It is recently standardized from Release 12, where its UEs are essential for IoT nodes. It offers equal data rates of 1 Mbps for both uplink and downlink. As stated above User Equipment (UE) is not supportive for Cat-1, this is in contrast, where Cat-0 minimized the difficulty by 50%. In addition, this system demands only one receiver antenna and affords half-duplex system, paving route for the creators to drastically cut-down the cost of modem to be effective.
3. **Cat-M1 – Category M1 (which is also called as Category M)** – This defines newest operation for Release 13, where the remaining complexity appearing on Cat-0 is drastically cut-down by 75-80% and patterned. The main key property is the implementation of the User Equipment (UE) transmitter and receiver models with degraded bandwidth compared to basic LTE User Equipment (UE) running on bandwidth of 20 MHz as tabulated on table 1. Thus, User Equipment (UE) on Cat-M1 works everywhere within an LTE carrier with bandwidth of 20 MHz, where each User Equipment (UE) on Cat-M1 works by allowing maximal bandwidth of 1.4 MHz. Apart from this property, another key-feature to serve for various IoT use cases is wide coverage of greater than 15 dB. It is to observe that Letter 'M' designated in the category name generally refers to **'Machine'**. If the 3GPP tasks on MTC progress, there's possibility to view advanced 'machine' types soon (i.e. Cat-M2 or NB-IoT).
4. **EC-GSM-IoT –** It was previously known as EC-EGPRS, which is abbreviated as Extended Coverage GSM for IoT applications. This falls under Release 13 category, where it contributes special improvements to the GSM as well as EGPRS standards to afford good coverage and other IoT services. EC-GSM-IoT affords 20 dB coverage enhancements and can be deployed within the current GSM networks.

A table 1 depicts the comparison of cellular IoT system designed by 3GPP, which differs depending on each parameter defined. From the figure 1 with yellow area, Release. 13 also addresses the greatly cost-effective need of the IoT standardization with a latest radio technology called NB-IoT, which is a heart of the project. It is to assure that NB-IoT always consumes spectrum of below 200 kHz and can be deployed widely in-band by again using LTE resource blocks that are still free or standalone in the spectrum within associated LTE carriers or on complimentary GSM spectrum. The focus is to reconstruct the Cellular IoT system with specialized features to serve wide applications of Cellular IoT system.

In addition, a figure 2 describes road map of cellular IoT, where highlighted block is main theme of the thesis. This NB IoT which fulfils the requirements of Low Power Wide Area Network(LP-WAN) paves the route to fifth generation (5G) radio communications.

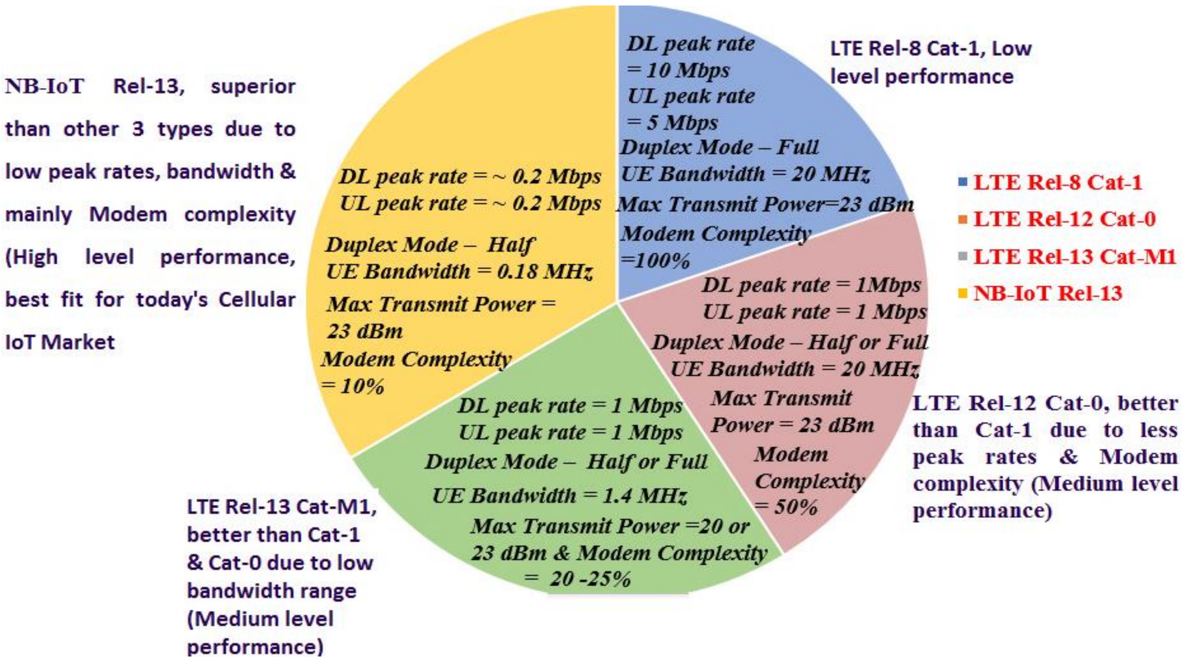


Figure 1: Comparison between different Cellular IoT system – 3GPP

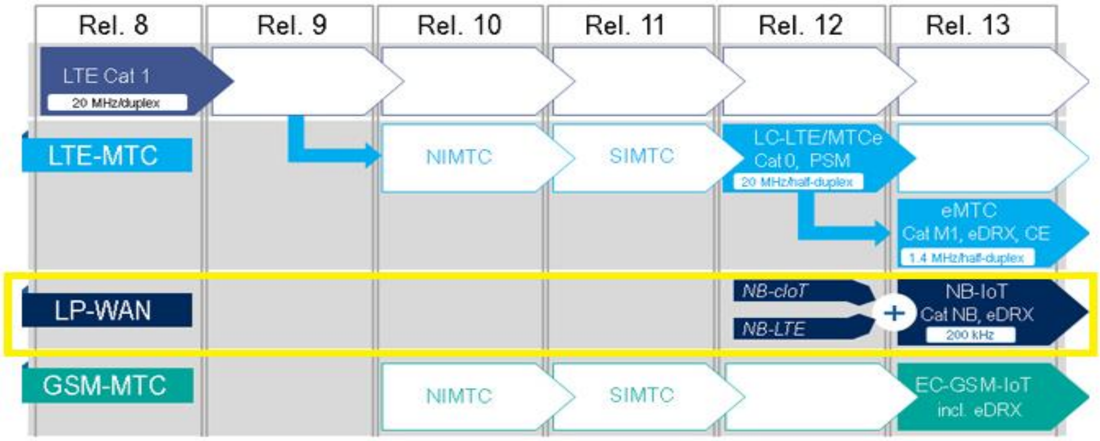


Figure 2: 3GPP IoT standardization route to 5G technology [4]

1.2.2 Embedded operating systems – An overview

Currently it is extremely strange to investigate an embedded system without an operating system as it plays a major role in the market. A entire spectrum of embedded devices can be visualized from figure 3 where software architecture (i.e. complexity) is plotted against CPU architecture.

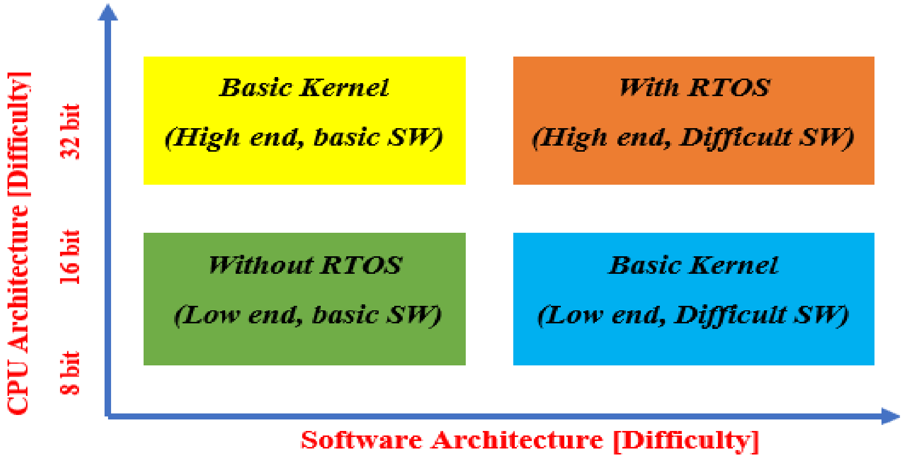


Figure 3: CPU Vs software and operating system Difficulty

It can be seen from figure 3 that blocks have been divided into four sections, where the top right corner defines difficult software to be imported on high-end processor. It can be noted that this block plays an important role which in turn acts as foundation for real-time operating systems (RTOS) and another OS. On the other hand, the block on the bottom right corner defines to adopt basic kernel when the software faces difficult problems. It should be noted that a powerful chip may be needed to operate basic software, where CPU efficiency is needed to attain execution speed. Under this circumstance, it is not necessary to adopt kernel however incorporating it shapes the efficiency of software architecture with high scalability and accommodates a future increase in difficulty, where basic software is running on a low-end device without any kernel [5].

Therefore, it should be noted that operating systems differs mainly in terms of characteristics, where

a) **Kernel layout -** It is given more importance to compare the functionalities of various operating systems and finds a best OS that can withstand cellular IoT system (mainly NB-IoT).

b) **Scheduler -** scheduling directly causes the ability of the operating system to work in real time and affords various priorities to communicate with user. It also faces great impact on power consumption of the device. Further different operating systems possess various scheduling flows and procedures.

c) **Programming structure** – The main key-note is that on different operating systems, the tasks are operated within the same context itself without separating the address region of the memory. Also, some operating systems affords multithreading, where every thread or task operates inside its own thread and possess its own stack memory.

There are many existing open sources embedded operating systems available in market and some of them are described briefly and compared below. In addition, how MBED OS fulfils the requirements of NB-IoT by dominating all other operating systems listed below is explained within this chapter.

1. **Contiki** – It is open source OS configured for IoT, which consumes 10KB RAM and 30KB flash memory. With such memory management, Contiki OS shows poor performance as compared to Tiny OS or RIOT OS which contributes real-time mechanism. However, the popularly adopted Contiki offers great wireless networking support, with an IPv6 stack developed by Cisco. The OS delivers an essential development tools which also adds a dynamic module loading Cooja Network Simulator for debugging wireless networks. Contiki is touted for efficient memory allocation [6].

   Event driven systems handle processes as event handlers which takes the control of CPU until it gets approved. But this also faces the defects within event driven systems where the CPU is unfit to acknowledge an external stimulus when the event that it is progressing takes more time (i.e. long cryptographic or security procedures). These defects can be resolved by adopting an OS on preemptive scheduling. Thus, Contiki resolve this issue by incorporating a hybrid model which adopts an event-driven kernel. Therefore, preemptive multi-threading is developed within this kernel as an application library and this library files are ported into program when it is needed as it's designed as optional case. A figure 4 above depicts Contiki OS system, which consists of core and loaded programs, which is decided during compilation. On other hand, the core contains kernel, program loader, the most popularized section of language run-time and support libraries.

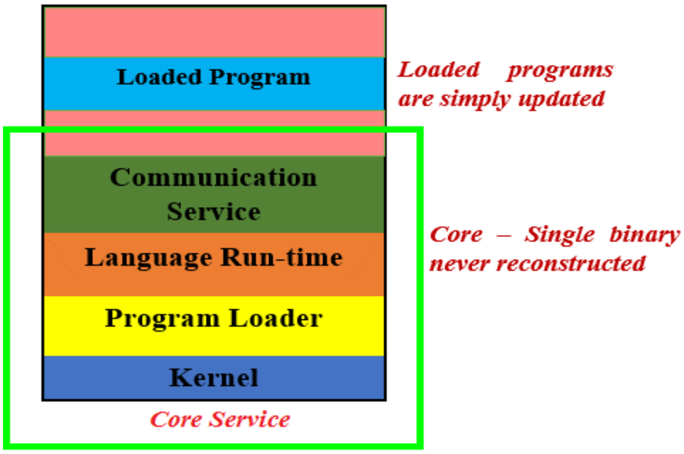


Figure 4: Contiki OS Architecture

2. **RIOT OS** – This is next open source microkernel adopted 8 years old operating system. It is well suited for its efficient power management and great support for wireless connectivity. It affords broad field of devices within IoT system. This open source OS contributes hardware needs with 1.5KB RAM and 5KB flash memory. Thus, it represents small memory footprint in contrast to Tiny OS. In addition, it also contributes properties such as multi-threading,

dynamic memory management, hardware abstraction, partial POSIX compliance, and C++ support [6], which are essential parameters of Linux OS rather than lightweight RTOS. Additional properties provide a less interrupt latency of about 40 clock cycles, and priority-enabled scheduling. It is also possible to implement within Linux or other operating systems and contributes wide deployment to embedded devices via a native port. This open source OS is picturized on figure 5, which consists of Kernel, that acts as core, platform specific code for CPU and boards, device drivers and other functions, needed for demo applications and examples thereby illustrating properties described above and testing.

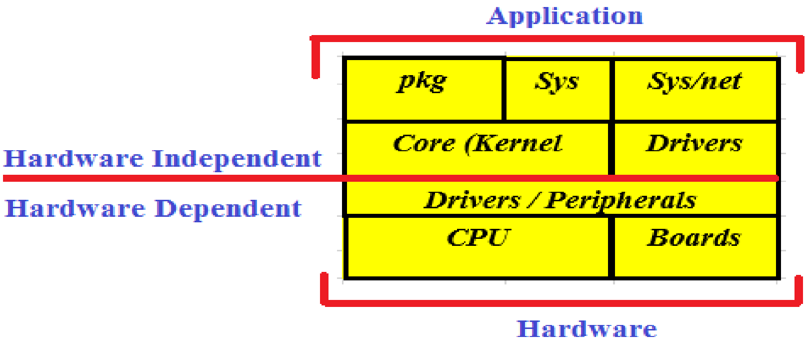


Figure 5: RIOT OS System

3. **FreeRTOS** - FreeRTOS is reaching to compete with Linux among various embedded development foundations, and it's typically famous for developing IoT end products. However, FreeRTOS cannot withstand properties defined within Linux like device drivers, user accounts, and advanced networking and memory management [6]. But it possesses less memory footprint in contrast to Linux. It is to note that FreeRTOS is structured not with prevailing RTOS (for instance, VxWorks), and it is fully authorized by GPL license, which is open source. FreeRTOS consumes partial amount of RAM (i.e. half Kilobytes) and 5-10KB of ROM, while with a TCP/IP stack, the memory consumption is high which is about 24KB of RAM and 60KB flash memory. An architecture of FreeRTOS is depicted in figure 6, which is divided into hardware and software. In addition, software handles drivers, kernel, which is core and corresponding applications to be carried out.

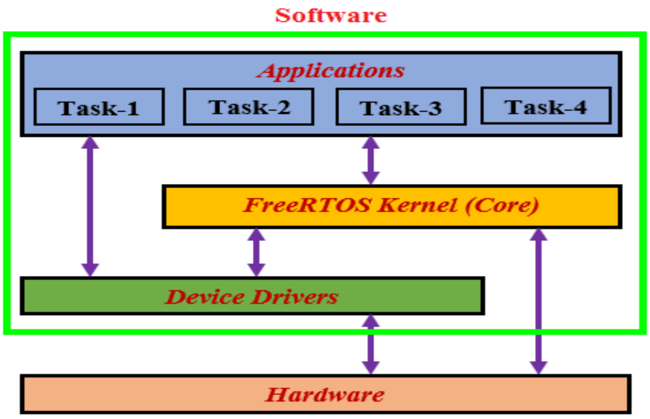


Figure 6: FreeRTOS Architecture

This is to note that the project targets to compare MBED OS with FreeRTOS for investigating each of their functionalities and operations, which is explained in detail on upcoming chapters.

4. **Tiny-OS** -- This is another kind of open source OS authorized by BSD-license. As the name itself defines the meaning that this OS is tiny, contributing low power system on MCU devices with code memory consuming few kB of RAM and a few tens of kB. This is scripted in a C dialect language which is referred as nesC. By the project's own admission, "computationally-intensive applications can be difficult to write". The project is working on Cortex-M3 support, but for now it's still designed for lower-end MCUs and radio chips [6].

   A basic block diagram of Tiny OS is depicted in figure 7 and subdivision of each block is picturized in figure 8, where application configuration, interfaces and specific components are handled by user application, which sends set of commands to TOS components and Hardware Abstraction Layer(HAL), scheduler and library components are built inside Tiny OS, which sends events to be performed.

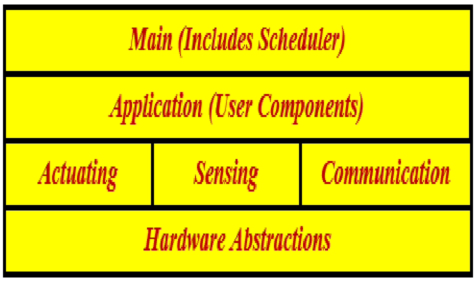


Figure 7: Tiny-OS Block diagram

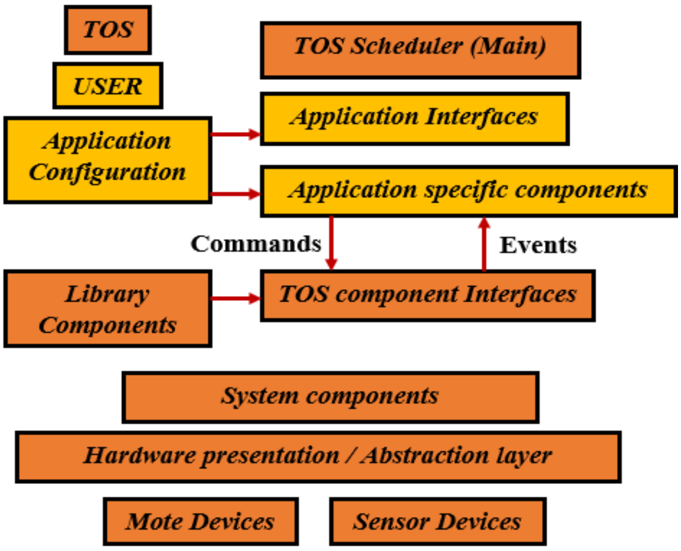


Figure 8: Tiny-OS Architecture layers

5. **uClinux** – An esteemed and simple uClinux is a kind of Linux OS that operates mainly on MCUs, and next on specific Cortex-M3, M4, and -M7 processors. It is to note that as this OS needs more RAM than it's available on Cortex-M cores, it demands MCUs with built-in memory controllers which adopts an external DRAM chip to fulfil its RAM needs. This OS is now ported into the mainline Linux kernel, where uClinux OS gets benefitted from the continuous wireless affordability found on Linux. However, latest MCU-enabled OSes such as MBED OS are closing the gap quickly on wireless, and are effortless to configure for stable NB-IOT system [6]. A figure 9 depicts uClinux environment which is divided into Kernel space – supervisor mode and user space – user mode.

A table 1 illustrates overview of different open source operating systems for IoT sensor nodes, where the highlighted box focusses on ARM MBED OS, which is main theme of the project and why it has chosen is explained previously and how it dominates with other IoT OSes and provides a valuable solution for NB-IoT will be discussed in upcoming chapters.

Table 1: Overview of Embedded Operating Systems [7]

Features	**Contiki**	**RIOT**	**FreeRTOS**	**TinyOS**	**uClinux**	***MBED***
Architecture	monolithic	Microkernel RTOS	Micro-kernel RTOS	monolithic	monolithic	***monolithic***
Scheduler	Cooperative	Preemptive tickless	Preemptive, optional tickless	Cooperative	Preemptive	***Preemptive***
Programming model	Event-driven, protothreads	Multi-threading	Multi-threading	Event-driven	Multi-threading	***Event-driven, multi-threaded***
Supported MCU families or vendors	AVR MSP430, ARM Cortex-M, PIC32,6502	AVR MSP430, ARM Cortex-M, x86	AVR MSP430, ARM, x86, 8052, Renesas	AVR MSP430, px27ax	ARM7, ARM Cortex-M	***ARM Cortex-M***
Programming script	C	C, C++	C	nesC	C	***C, C++***
License	BSD	LGPLv2	Modified GPL	BSD	GPLv2	***Apache License 2.0***

1.3 Problem Analysis in Existing System

The main problem which are drawn from the conclusions laid by various researchers are analysed, which is divided into 2 sections – Cellular IoT and Embedded Operating Systems. In addition, how NB-IoT is decided as preferable solution for LPWAN by dominating all other cellular IoT systems is depicted in figure 9. Further, how MBED OS fits for NB-IoT as Embedded OS with low power consumption, large coverage and security solutions is discussed.

a) Cellular IoT system:

1. **Performance -** Maximum coupling loss (MCL) is stated as the maximal total channel loss between User Equipment (UE) and eNodeB (eNB) antenna ports at which the data service can still be received. Practically, it provides antenna gains, path loss, shadowing and any other

impairments. The higher the MCL, the more robust the link is [27]. With respect to 3GPP, the MCL for CAT-M1 is 155.7 dB whereas NB-IoT is 164 dB – an ultimate difference of more than 8 dB. Overall, this would represent a significant advantage for NB-IoT's performance.

2. **Peak data rate -** Both Cat-M1 and Cat-M2 or NB-IoT devices will have reduced peak data rates compared to regular LTE devices (e.g. Cat-1 or Cat-0). Cat-M1 has limited throughput of up to 1 MBps in both downlink and uplink directions [28], while NB-IoT still degrades peak data rate down to 10's of kbps. This degraded peak data rates enables for both processing and memory savings in the device hardware, where NB-IoT acts superior.
3. **Bandwidth -** LTE aids adaptable carrier bandwidths between 1.4 MHz to 20 MHz, applying 6 to 100 resource blocks. For LTE Cat-M1, the device bandwidth is controlled to 1.4 MHz only to withstand the lower data rate. On contrast, the bandwidth of NB-IoT still falls to 200 kHz. The bandwidth degradation for Cat-M1 needs a new control channel to change the legacy control channels, which makes unfit inside the narrower bandwidth. Thus, for Cat-M2 or NB-IoT, a new firm of NB-IoT synch, control, and data channels are offered to withstand the narrower bandwidth thereby, NB-IoT still dominates other cellular IoT systems.
4. **Duplex Modes:** Due to the less frequent and latency-tolerant nature of IoT data transmissions [27], LTE IoT devices can minimize complexity by only aiding half-duplex communications, where neither the transmit nor receive path is active at a given time. It can be seen from table-1 that Cat-M1or Cat- 1 devices can afford either half-duplex or full-duplex FDD, whereas NB-IoT devices is designed with only half-duplex FDD. This makes the device to execute an effortless RF switch rather than full duplexer that is more difficult and expensive.

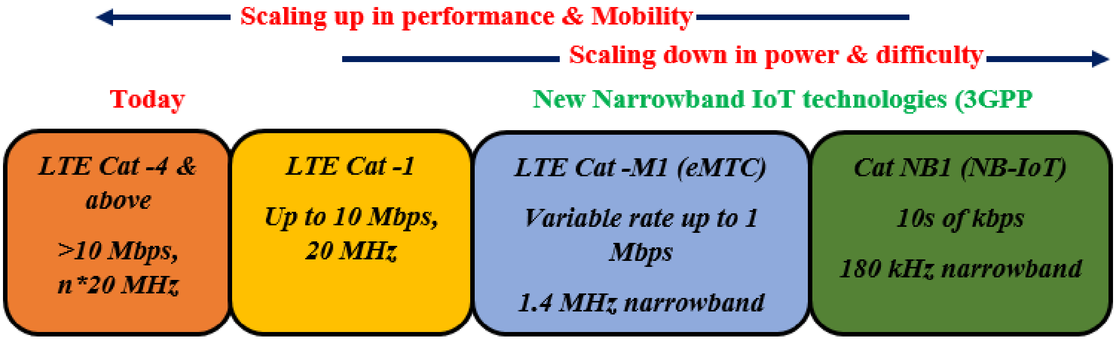


Figure 9: NB-IoT Roadmap

b) Embedded OS

1. **RAM requirements** – As stated earlier, uClinux requires more RAM to run than its available on M-class cores. The number of RTOS objects (like tasks, mutex, semaphore, mailbox etc.) used by any specific application will affect RTOS RAM usage, hence each object needs some RAM space. It can be noted that higher-end M-class cores, contains built-in memory

controllers so use of an external DRAM chip to meet uClinux's minimum RAM requirements leads to increase in cost, power consumption and size, which makes advantageous for ARM MBED OS.

2. **Cost and size considerations -** In technical terms, each embedded system is different. CPU, memory, and peripherals and others differ from one device to another. But systems vary commercially too and the final price of a piece of equipment, along with the volumes produced, affect the OS licensing options. In certain cases, a few dollars per device is acceptable. In others, where very large volumes are anticipated, a royalty free business model might be ideal. For several reasons, **size** plays a major role when an MCU RTOS or a small MPU RTOS is considered. Initially, the program should fit in the MCU. If it's unfit, there is demand for slower and expensive external memory. This size criteria applies to program memory, initialized variables (in Flash and RAM), and program RAM, which contains stacks and the heap for the system. Smaller sizes with the same features minimize bill of materials (BOM) costs and improve product margins. Implementation sizes sometimes vary dramatically when comparing identical features. This causes were eliminated in MBED OS, where it acts as open source and supports nearly 74 targets from different vendors.
3. **Memory footprint -** It can be observed that the specification of the RTOS differs mainly due to design size. Also, many RTOSes have greater scalability, where their memory consumption is decided by the functions or services adopted by the specific application. However, considering high scalability other services mainly networking have large impact on affecting code size. Hence memory footprint of OS needs to be small. As compared to other RTOS like Contiki, FreeRTOS or others, Tiny-OS has no memory protection, which may lead to crash or corrupt the system easily. On contrast, MBED OS has small footprint, which is a key feature.
4. **Energy efficiency -** It can be noted that for battery operated IoT devices, usage of efficient energy is crucial. This is because sensor nodes in IOT devices are characterized by low power draws and prohibitively expensive to replace batteries. On the other hand, MBED OS is purely energy efficient, which is targeting towards 10 years of operation on AA battery.
5. **Security** – This is main factor for embedded OS, since IoT devices are synchronised to internet. It is vital that the operating system needed for IoT stick to severe protection expectations and meet stringent requirements imposed by deployments in sensitive and critical settings [7]. Thus, MBED OS affords greatly for security solutions as compared to other embedded operating systems.

2. PROPOSED METHODOLOGY

This section illustrates brief description of LPWA and its key benefits. Further this chapter describes how Narrow Band IoT paved a route for low power solutions. In addition, it describes the heart of the project - MBED OS, as a proposed embedded operating system which is implemented on ARM cortex M33 processor contained on ARM MPS2+ hardware board. In addition, MBED OS is also implemented on ARM Cortex M4 (i.e. successor of ARM Cortex M33) to investigate how superior is ARM Cortex M33 than M4, which is briefly explained in the upcoming sections. Further to evaluate the functionalities of MBED OS, FreeRTOS is considered for comparison and how MBED OS dominates FreeRTOS in terms of memory management, stack consumption and performance are described in detail via simulation results.

2.1 Increasing demand for Low Power Wide Area Services(LPWA)

Mobile network operators are performing latest possibilities to afford popularized IoT and M2M services where, LPWA networks plays a major role in IoT market for linking a broad field of IoT devices that possess low power consumption, wide coverage cost effective and security solutions. The LPWA connections are mainly to function a diverse range of vertical industries and cover a wide range of applications and world-wide deployment scenarios for which existing mobile network technology such as GSM or 2G/3G system remains unfit to offer wide connectivity [8]. This pave a route for an emerging technology –NB-IoT, which satisfies the needs of LPWA networks.

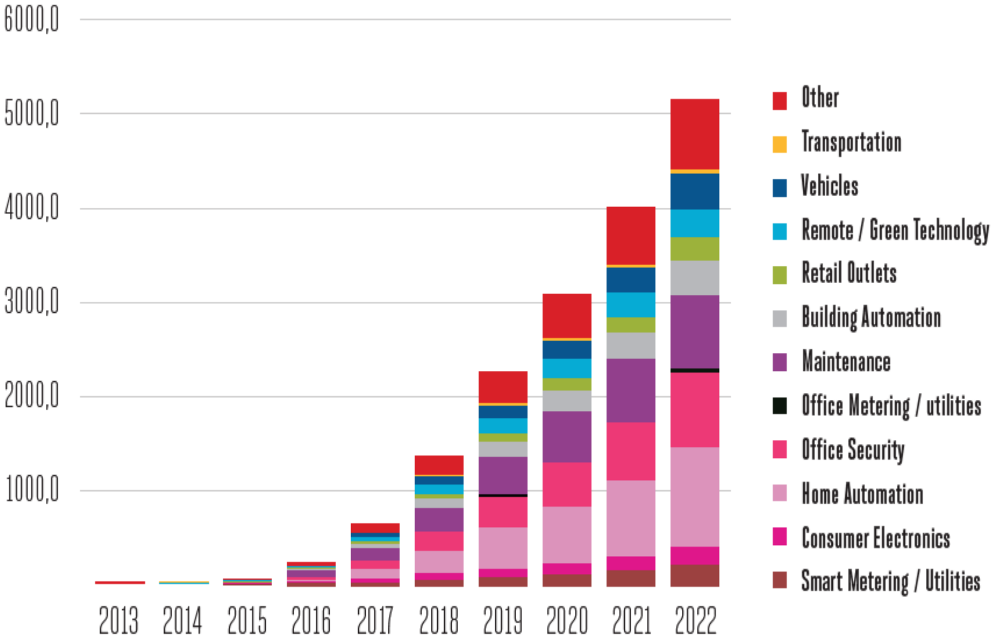


Figure 10: Global LPWA connections shared by applications (in millions) [8]

It can be noted that analysts anticipate the number of LPWA connections is likely to grow gradually over the next two years before surging in 2018 as standardised LPWA technologies gain economies of scale and are proven in the marketplace [8] which is depicted in figure 10.

2.2 LTE Cat -M2 or NB-IoT – Paving a route to 5G technology

It was observed earlier from section 1.3 how NB-IoT dominates other cellular IoT system and became superior in terms of performance bandwidth and duplex modes.

The 5G vision emerges to enhance existing wireless networks by three key factors.

a) The ability to contribute massive mobile broadband (mMBB). Driving networks to contribute ubiquitous GB (i.e. Gigabyte) connectivity is stable in operating latest mobile use cases like augmented reality (AR) and virtual reality (VR) which requires high resolution, engaging user interfaces (UI's) to enrich the user experience [1].

b) Next is massive machine type communications (mMTC) which arises to deliver the plumbing for the IoT and the need of networks to massively scale and merge the billions of sensors to cloud based services.

c) Lastly, implementing on mMTC where, we have ultra-reliable machine type communications or uMTC which is targeted at contributing a new breed of ultra-low latency applications such as industrial control, automotive and remote healthcare.

NB-IoT was developed as a 'clean sheet' technology within 3GPP and as such is built from the ground-up to be optimized for mMTC, setting the foundations for 5G IoT connectivity. As the standards progress over the next couple of years we will see further enhancements brought into NB-IoT to further enable the 5G vision of massive machine type communications.

2.3 Physical Baseband Architecture for NB-IoT

There has been a massive evolution of the cellular system in the last 30 years. It can be noted that the first generation is related to only analog elements, where second generation deals with digital processing. In the third and fourth generation, in depth of digital processing was established as CMOS development with higher density, low power consumption and higher frequency. The modems currently used in smartphones contains extremely complex hardware and software systems that support several standards such as GSM, HSDPA and LTE. The complexity is mainly driven by the increased data rates that reach 800Mbps today and will exceed 1Gbps soon.

The cellular platform based industries are steadily challenged by superior demands on new functionality that needs higher data rates. Latest but, faster general-purpose processors (GPP) are developed along with powerful digital signal processors (DSP) to keep up the pace with the new requirements. Figure 11 shows a typical system architecture for an LTE modem. Multi-core considerations for both GPPs and DSPs provide another dimension to the complexity in these systems. Custom HW accelerators are also modernized and developed to cope up with new demands. Although these new specifications forces technology to new limits, another significant development called, the IoT development, is challenging system architecture in a distinct way. The focus is changed from high data rates to low cost and low power. The system architecture for IoT should be lightweight and simple to meet these needs of low-cost and low-power.

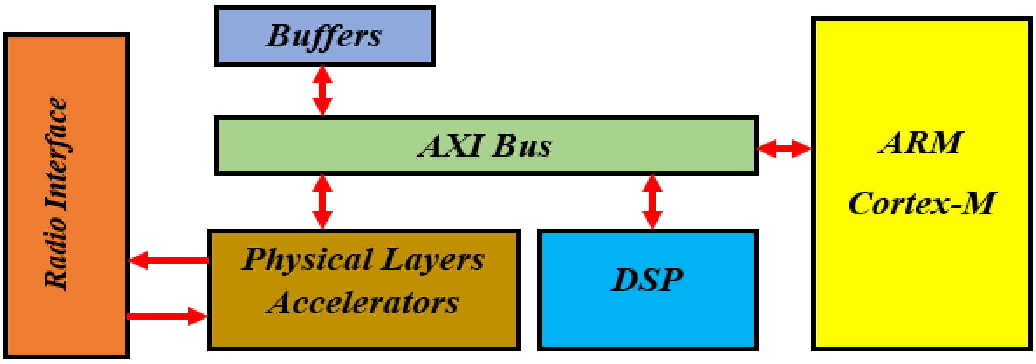


Figure 11: Typical LTE system architecture

Shrinking a full LTE modem (figure 13) that needs to afford only NB-IoT can degrade the whole design requirements. However, the destination architecture, which depends upon a full LTE architecture, is not suitable when the cost and power management is considered, since the structure of full LTE modem is totally different from NB-IoT modem. Minimizing the number of GPP and DSP cores with less difficulty ARM and DSP cores, degrading clock frequency and memory are essential designs when shortening a full LTE modem towards a NB-IoT modem. To accomplish low power and cost effective a diverse method might be beneficial. A modern approach to ignore complexity is the removal of the dedicated DSP, which is depicted in figure 12 contrasting figure 11.

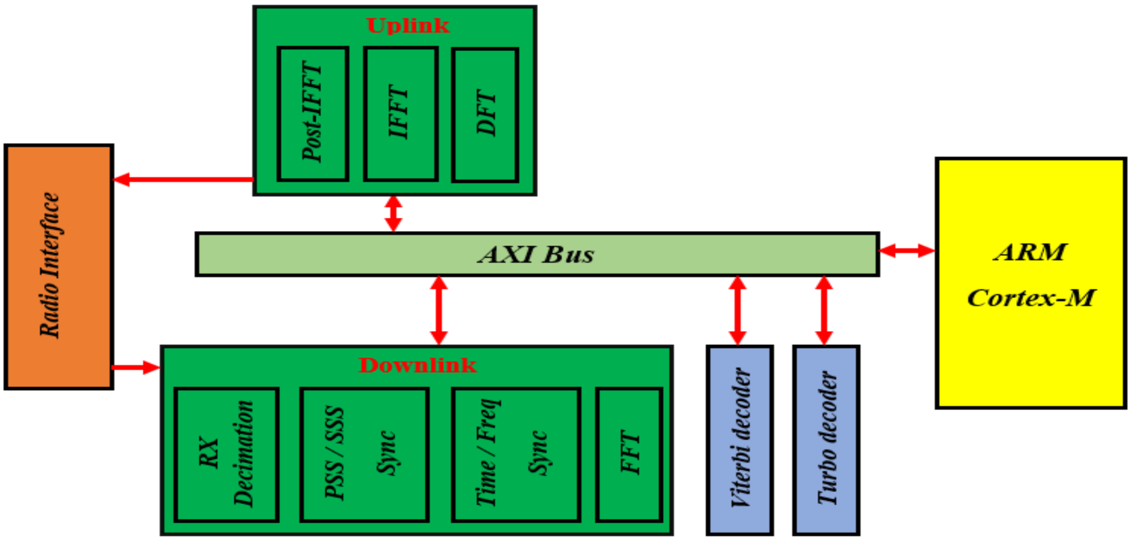


Figure 12: New LTE system architecture

It can be viewed from figure 12 that the thesis focusses only on ARM cortex-M (yellow box), which is main block to be considered in LTE modem. It should also be noted that this deals with latest M-core on which both ARM MBED OS and FreeRTOS security is implemented. This is explained in detail in upcoming section.

2.4 ARM cortex-M: Ultra low power empowering NB-IoT

ARM cortex M-33 is the most supportive and applicable of all cortex-M based processors. It is designed with latest technology which is based on current ARMv8-M architecture with ARM security solutions and DSP. The design aims on efficient energy by succeeding high compute performance with low-power consumption. It affords many configurable options to facilitate deployment in a broader application, but it is basically structured as the processor to accomplish a whole NB-IoT modem. It should be noted that cortex-M33 is the successor of the Cortex-M4 with a latest pipeline that offers higher performance with exceptional energy efficiency and the other differences are depicted on figure 13.

A single Cortex-M33 can operate the entire NB-IoT modem, where an IoT platform (i.e. ARM MBED) and the IoT application itself – all without the need for a separate DSP processor. This is main target that the project focusses to achieve it by implementing MBED OS on ARM cortex M-33 processor, which is operated on ARM MPS2+ hardware board. The Cortex M33 processor succeeds and shapes upon the existing large ecosystem of Cortex-M processors in software and tools. This forces NB-IoT products to be developed faster and released into market.

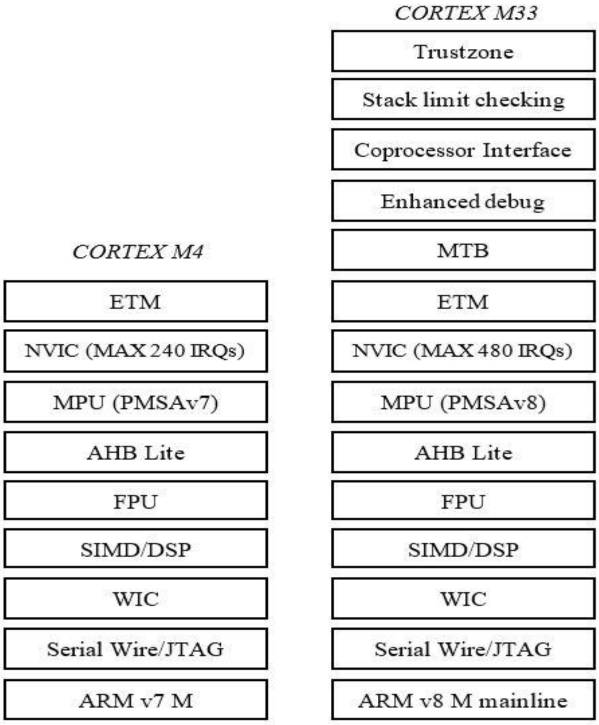


Figure 13: Development of ARM Cortex M33 over M4

2.5 Hardware system – Cortex -M Prototyping System +

As stated earlier that MBED OS is to be implemented on ARM Cortex-M prototyping system**,** which is built on IoT Subsystem for V2M-MPS2+. Also, FreeRTOS is to be implemented on the same hardware board for comparison. This board is mainly designed for prototyping and evaluation of Cortex-M processor family accounting the latest ARM Cortex M7 processor. It is more supportive motherboard, accessible as function of the ARM Versatile Express field of development boards. This board has a feature to contribute two FPGAs for prototyping Cortex-M based designs and a range of different debug options [10]. The main key-point to note regarding the project is that the board is initially designed to be supportive till ARM cortex M7 (i.e. from ARM Cortex M0 to Cortex M7). Hence to bring ARM Cortex M33 for a subsequent operation on this board, a simple boot operation is performed on MPS2+ board, where the required boot files known as IoT FPGA image for MPS2+ that supports ARM Cortex M33 is offered by the product. Once the booting is done, the board is configured for Cortex M33. Next the board support packages needed for ARM cortex M33 contained on MPS2+ is installed from ARM Keil (i.e. Keil:: V2M-MPS2_IOTKit_BSP). Additionally, this hardware board has a feature of rollback to previous Cortex M processor series (ideally ARM Cortex M4). Thus, the MPS2+ product has significant feature of affording IoT subsystems for Cortex M processors and this subsystem is linked with peripherals that are available on MPS2 to enable foundation on software development to configure IoT subsystems with MBED OS support.

2.6 Prelude of ARM MBED Operating System

ARM MBED OS is a popular open source embedded operating system, which is released under Apache 2.0 license. It is designed mainly for things to be connected to internet or cloud. For the MBED OS component, the OS is a lightweight, low-power kit OS structured to run on Cortex-M processors. ARM for their part shapes necessary hardware features and even some common libraries, with an aim on offering building blocks for developers looking to design scalable products. This is depicted in figure 14, which describes MBED OS stack. It should be noted that MBED OS does not cover all its microcontrollers like more featured Cortex-R, but only its hugely famous M class. It seems to be primarily targeted at offering standardized connectivity and IoT smarts to the lower end models, which are mainly designed into billions of devices. It constitutes standards for security, device management, and connectivity. It is to note that the project focusses on latest version of MBED OS 5 (typical version – MBED OS 5.4), where it is constructed from previous versions (i.e. adding features and other services of MBED OS 2 and MBED OS 3 together), thus it acts as sustainable operating system by fulfilling the requirements of Narrow Band IoT system.

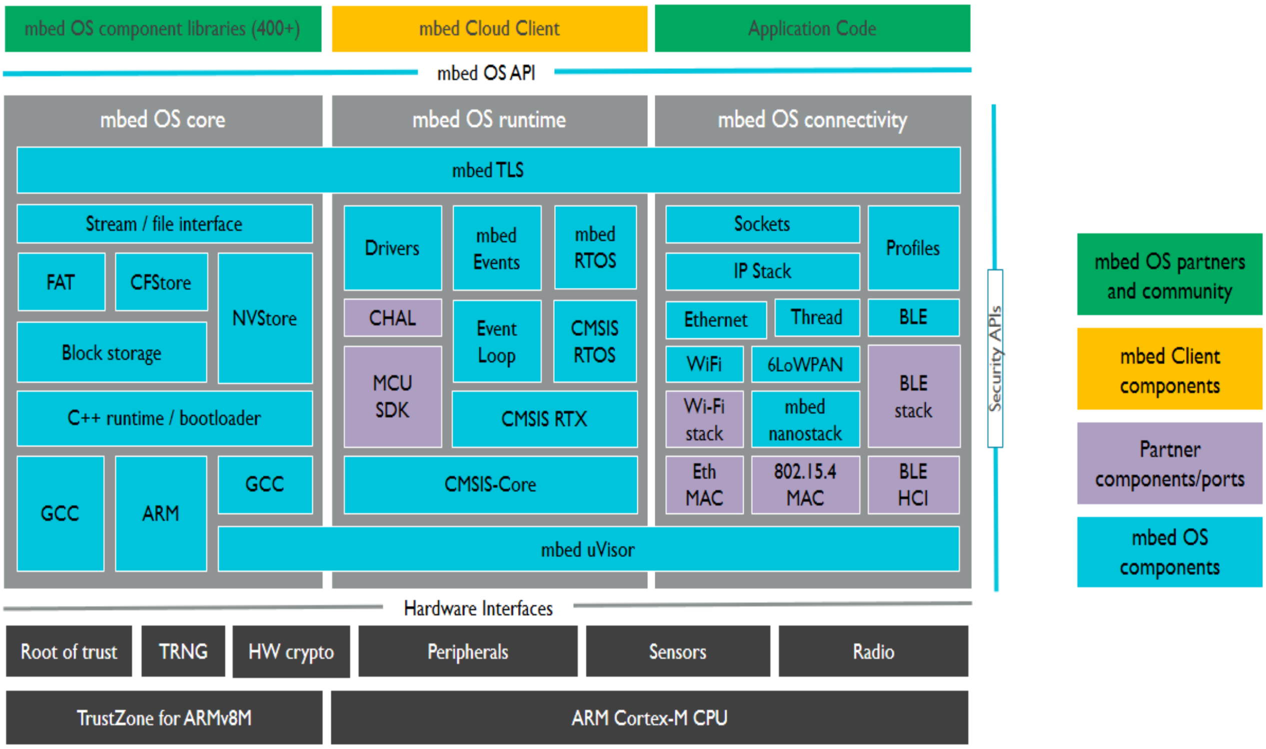


Figure 14: MBED OS stack [11]

2.6.1 Review of ARM MBED IoT device platform

a) what's the background of ARM MBED and why it was created?

ARM MBED was built to facilitate developers and OEMs in deploying and bringing IoT devices to the market quicker and enables system integrators and enterprises to deploy scalable, secure IoT solutions. With a global partnership of 65+ partners and 250,000 developers, it is privilege that ARM MBED is one of the world's leading IoT solutions today.

b) What are the benefits of using ARM MBED for the design of IoT devices?

ARM MBED is categorized into two product lines: MBED OS and MBED Cloud. As stated earlier, ARM MBED OS is an open source operating system that enables the creation and deployment of commercial, standards-based IoT solutions achievable at scale. On other hand, ARM MBED Cloud is an IoT device management that offers secure and reliable connectivity, updating and provisioning of any devices, on any cloud. A table 2 depicts main key benefits of using ARM MBED. It should be noted that the thesis focusses on MBED OS as RTOS which is to be implemented with help of CMSIS-RTOS with RTX kernel on ARM Cortex M33 as well as on ARM Cortex M4, which is highlighted in table 2 and this will be discussed briefly with simulation results in upcoming sections.

Table 2: Benefits and features of ARM MBED OS [12]

Parameters	**Description [Benefits]**
Device and component affordability [12]	It affords wide range of ARM Cortex -M based devices where, developers can prototype IoT applications quickly on low-cost development boards.
Real time software execution	***With an RTOS core based on the broadly used open-source CMSIS-RTOS-RTX, MBED OS affords deterministic, multithreaded real-time software execution. This RTOS primitives are anytime available, enabling drivers and applications to cope up with features such as threads, semaphore, mutexes, timers, queue, memory pool etc.***
Ease of use	With modular library structure, the needed support for any specific application will be automatically updated on device. By using MBED OS API, an application code remains pure, portable and simple, whilst taking influence of security and communications.
End to End Security	This addresses security mainly in device hardware, software, communications and in lifecycle of device, where: **Hardware empowered security** – At ground level of MBED OS, it's available to use supervisory kernel known as uVisor to provide isolated security domains, which blocks access to memory and peripherals **Communications security** – This takes standard protocols of SSL and TLS for offering security to communications on internet and enables to be included in MBED project with basic API.
Drivers and supported libraries	Include broad range of drivers support for standard MCU peripherals, which contains digital and analog IO, interrupts, ports and bus IO, PWM, SPI, I2C and serial.

2.6.2 ARM MBED OS-5 Architecture

The thesis addresses MBED OS-5 to be implemented on ARMv-8 architecture i.e. on cortex M33 processor as well as on its predecessor, ARMv-7 architecture [i.e. on ARM Cortex M4]. The reason why MBED OS -5 is chosen is because of following key features:

a) MBED OS core – Incorporated RTOS with CMSIS-5 libraries for multithreading which offers CMSIS-RTOS kernel. This is explained briefly in section 3.6.2.1.

b) **Tools and workflows** – An easy and simple workflow and component packaging tool to afford incorporation with broad variety of commercial third party tools.

c) **Core technology investments** – Impressive ARM embedded ecosystem with upgraded connectivity with simpler API's to boost productivity and portability and security solutions with dedicated life cycle (secure and non-secure modes).   .

A figure 15 describes ARM MBED OS 5 architecture with built-in structures mainly MBED OS core, where RTOS kernel is incorporated plays major role and is to be developed on ARM cortex M, which is a goal of the project.

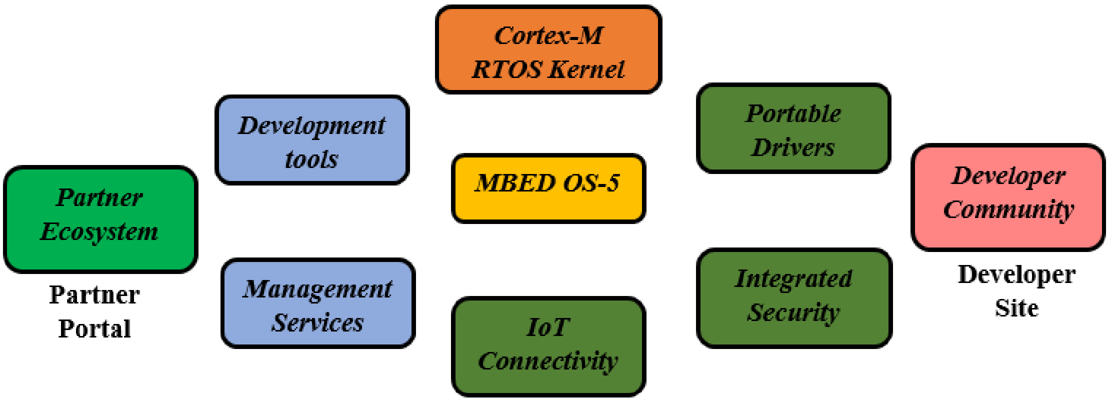


Figure 15: ARM MBED OS -5 Architecture

ARM MBED OS-5, which is platform OS for IoT devices should fulfil the following needs:

1) **Fasten the development of IoT devices:**
   a) Pre-integrate the required connectivity and software components for constrained IoT devices(MCUs).
   b) To implement latest development system and option to MCU to shape productivity.
   c) Offer OS functionality and APIs across wide range of vendor solutions for choice.
2) **Speed up the deployment of IoT devices:**
   a) Contribute standardised connectivity to the cloud across various transports.
   b) Provide solutions for device management problems.
   c) Offer manageability from cloud to enable route for opportunities and minimize cost/risk.
3) **Take advantage of ecosystem scale:**
   a) Unlicensed and simpler accessibility to avoid barriers to entry and enable maintenance.
   b) In cooperation to contribute superior gearing and pace.
   c) Modern tools and web infrastructure to afford an ecosystem and enable network effects.

2.6.2.1 ARM MBED OS-5 Core

This acts as heart of MBED OS, which is a consistent kernel and hardware abstraction contributing application portability. This is picturized in figure 16. The main properties of MBED OS core are as follows, which is presented on stack of MBED OS core (refer figure 17).

1) **Contains an RTOS kernel:**
   a) Developed on open source CMSIS-RTOS-RTX, which is broadly established RTOS kernel. The thesis addresses CMSIS RTOS version-2 with RTX5 kernel as an operating system for ARM MBED, which is discussed briefly further.

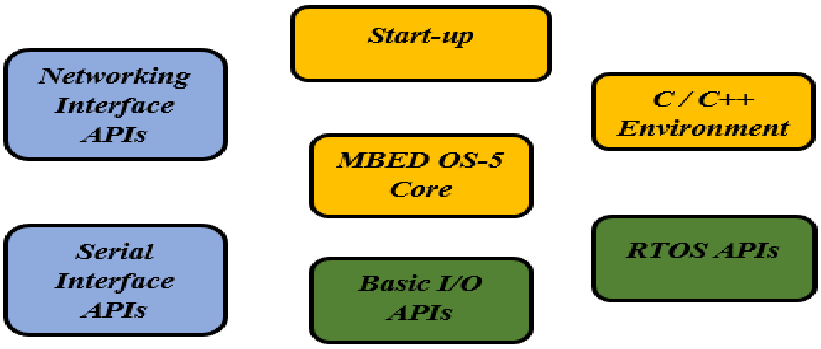


Figure 16: ARM MBED OS-5 Core

2) Incorporates peripheral driver APIs steadily across devices:
   a) Initialisation of Start-up and environment.
   b) Afford memory maps and cross-toolchain and its integration.
   c) Driver APIs for all universal peripherals, which is affordable across all MCUs
3) Unchangeable application and component libraries can be developed:
   a) Offers portability for developers, which in turn service network effects.

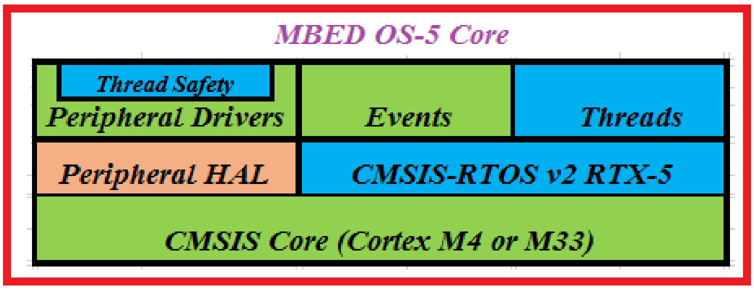


Figure 17: ARM MBED OS-5 stack

It can be observed that one of the major enhancement developed in mbed OS 5 is a special programming model adopted on a real-time operating system (RTOS). Thus, previous versions of MBED OS (i.e. MBED OS-2 or MBED OS-3) had optional affordability for an RTOS. On contrast, with MBED OS-5, RTOS affordability is a standard property of the platform, so developers have great privilege of a more flexible programming model adopted on multiple threads.

2.7 KEIL Pack – Cortex Microcontroller Software Interface Standard

This defines vendor-independent hardware abstraction layer designed to support mainly for the Cortex-M processor series and describes generic tool interfaces. The CMSIS offers steady device affordability and easy software connection to the processor and the peripherals, facilitating software reusability, making effortless learning tool for various microcontroller developers, and saving the time for latest devices to be released into market. A figure 18 depicts CMSIS structure, where the required components are described as follows. In addition, the project concentrates on CMSIS 5 version, which affords both ARMv7-M architecture (Cortex M4) and ARMv8-M architecture (Cortex M33), which is main part of the project.

It should be noted that CMSIS is described to interact with different silicon vendors and offers unique decision to connect to peripherals, RTOS and middleware components.

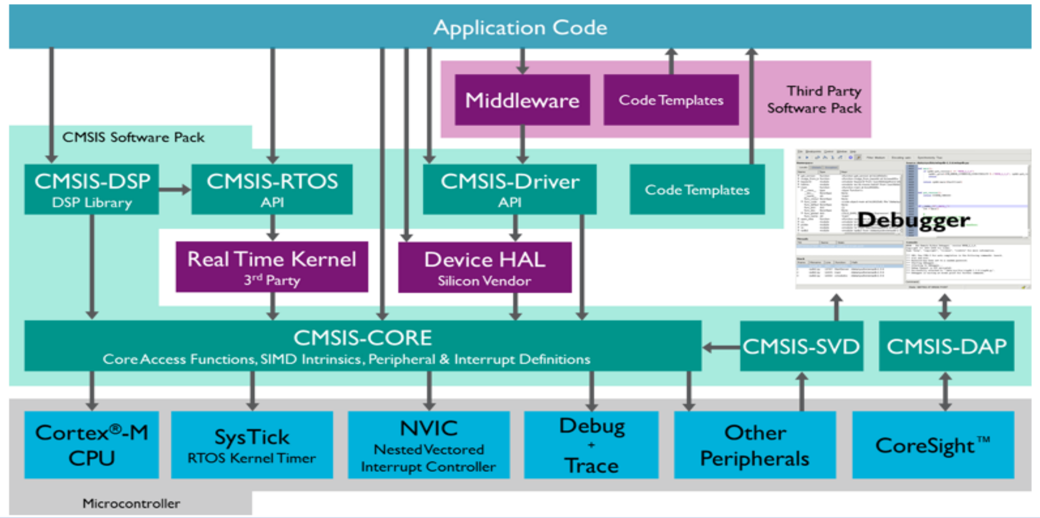


Figure 18: CMSIS Components [13]

The CMSIS consists of main components as

a) **CMSIS core –** This is main component of the project and includes API for the ARM Cortex-M processor core and peripherals. It contributes a standardized interface for ARM Cortex M processor series as well as SIMD intrinsic functions for Cortex-M4, Cortex-M7, and Cortex-M33 SIMD instruct ions. The project targets on CMSIS core for Cortex M4 and M33 processors, where CMSIS core operates a simple run-time system for a Cortex-M processor and enables accessibility to the processor core and the device peripherals.

b) **CMSIS Driver -** This illustrates universal peripheral driver connection for middleware enabling its reusability across affordable devices [13]. The API is RTOS which doesn't depend upon other layers and interfaces microcontroller components with middleware which enables communication stacks, file systems and other structures [13].

c) **CMSIS DSP –** This defines DSP Library with nearly 60 Functions for different data types (for instance fixed and single precision floating point). The library is accessible for all Cortex-

M processor cores. The performance which are stabilized for the SIMD instruction set are shared by Cortex-M4, Cortex-M7, and Cortex-M33 processors.

d) **CMSIS-RTOS v1** – It describes common API for RTOS together with reference operations based on Keil RTX. It offers a stable programming connection which is compact to various RTOS and provides software peripherals to be accessible across various RTOS systems.

e) **CMSIS-RTOS v2 -** This is another main component of the project. This is upgraded version of CMSIS-RTOS v1 with reference operations based on RTX-5, which is latest version of RTX kernel. As stated earlier, it also affords for ARMv8-M architecture, dynamic object creation, accessibility for multi-core systems, and binary optimized interface across ABI compliant compilers [13].

2.7.1 Introduction of CMSIS-RTOS-2

The CMSIS-RTOS API Version 2 also known as CMSIS-RTOS2 is a generic RTOS interface designed mainly for ARM Cortex-M processor-based devices. It offers a recognised API for software peripherals which needs RTOS functionality and contributes dedicated benefits to the users and software company. It should be noted that middleware components which needs CMSIS-RTOS2 refers to RTOS agnostic and are simpler to adapt. This latest CMSIS RTOS handles the sources of microcontroller system and performs the operation of parallel threads which operates concurrently. Any application that needs various concurrent operations, CMSIS-RTOS2 performs multiple concurrent operations quickly. Every operation has a separate thread which performs an individual task and this minimizes the overall structure of program. The CMSIS-RTOS2 system is ascendable and later more threads can be summed quickly, where threads with high priority is first executed.

The CMSIS-RTOS2 contributes multiple services required in various applications, for instance, periodical triggering of timer functions, memory consumption and other applications.

The CMSIS-RTOS2 focusses the below new needs:

a) Dynamic object generation does not need static memory as static memory buffers are currently optional for CMSIS-RTOS2.

b) Affords for ARMv8-M architecture that contributes a secure and non-secure mode of code execution.

c) Dedicated message passing in multi-core systems.

d) Full afford of C++ run-time environments.

A main thing that should be noted that CMSIS-RTOS2 offers a translation layer for the CMSIS-RTOS API v1. Hence there is also way to couple CMSIS-RTOS API Version 2 and CMSIS-RTOS API Version 1 within the same application.

2.7.1.1 Generic RTOS Interface

CMSIS-RTOS2 refers to a generic API which is agnostic of an elemental RTOS kernel. Application programmers needs CMSIS-RTOS2 API functions in the user code to offer superlative portability from one RTOS to other. Middleware using CMSIS-RTOS2 API prefer this superior approach by averting needless porting efforts, which is picturized in figure 19.

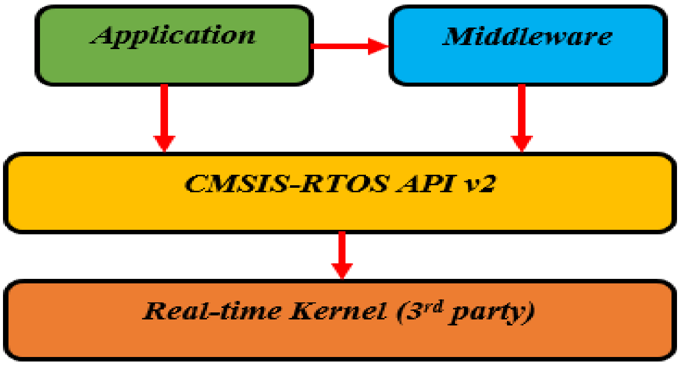


Figure 19: CMSIS-RTOS API structure

A classic CMSIS-RTOS2 API implementation connects to a current real-time kernel. The CMSIS-RTOS2 API contributes below characteristics and functionalities:

a) Function names, identifiers, and parameters are definitive and simple to grasp. The functions are forceful and have greater flexibility which minimizes the number of functions defined to the user.

b) Thread management which enables users to define, create and control threads.

c) Interrupt Service Routines (ISR) need some CMSIS-RTOS functions. If a specific CMSIS-RTOS function cannot be called from an ISR context, it dismisses the invocation and sends an error code.

d) There are three main event types that affords communication between multiple threads and/or ISR:

   1) Thread Flags: This may be used to define specific conditions to a thread.

   2) Event Flags: It refers to describe events to a thread or even ISR.

   3) Messages: This can be transmitted to a thread or an ISR, where messages are buffered in a queue.

e) It is important to note that mutex and semaphore management are integrated.

f) CPU time can be scheduled with below various functionalities:

   1) A timeout parameter is integrated in different CMSIS-RTOS functions to minimize system lockup. When a timeout is defined, the system waits [14] until a resource is accessible or an event exists. Meanwhile during hold time other threads are simultaneously scheduled.

2) osDelay and osDelayUntil functions forces a thread into the WAITING state for a defined amount of time.

3) osThreadYield offers co-operative thread switching and sends execution to other thread with equal priority.

g) Timer Management functions are required to trigger the execution of various functions.

2.8 Functionalities of CMSIS-RTOS2

The list of parameters which are described briefly below (i.e. Thread, Mutex, Semaphore and Message Queue) plays a major role in the project. Depending on these functions defined within CMSIS- RTOS2, the operation of MBED OS and FreeRTOS is performed and compared, which are represented via simulation results discussed in next chapter. Also, these functions decide which OS is superior and satisfies the requirements of NB-IoT system.

2.8.1 Thread Management

This function group enables to define, create, and control thread functions present in the system. It can be noted that thread functions will not be called from ISR (Interrupt Service Routines). The threads can be involved in below states as described, which is depicted in figure 20:

1) **Running state -** The thread which is initially running is present in the RUNNING state. It can also be observed that only one thread at a time can be involved in this state.

2) **Ready state** – Threads that are ready to run will be in the READY state. Once the RUNNING thread has stopped, or is BLOCKED, the next READY thread with the highest priority will simultaneously become the RUNNING thread.

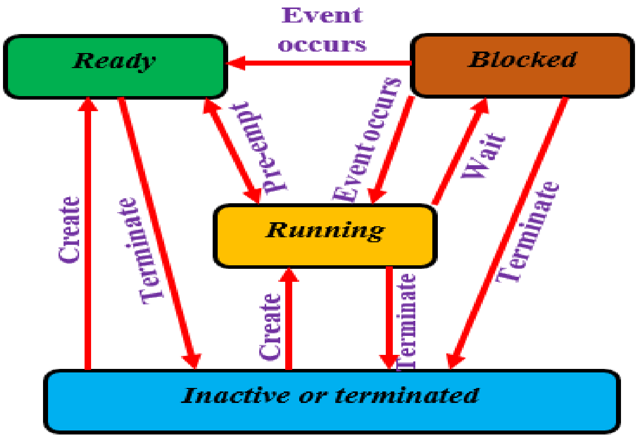


Figure 20: Thread State and State Transitions

3) **Blocked state -** Threads which are blocked either delayed, waiting for an event to occur or suspended will be present in the BLOCKED state.

4) **Terminated state –** When osThreadTerminate function is called, threads are automatically terminated with resources not being released.

5) **Inactive state -** Threads which are not created or have been terminated with all resources released will be in the INACTIVE state.

2.8.2 Semaphore Management

Semaphores are mainly used to handle and secure access to contributed resources. Semaphores are identical to Mutexes. It can be noted that mutex allows only single thread to approach a shared resource at a given time whereas, a semaphore is used to access a settled number of threads or ISRs to use a shared resource fully. Having semaphores, permit to a chain of similar peripherals can be handled (for instance various DMA channels), which is picturized in figure 21.

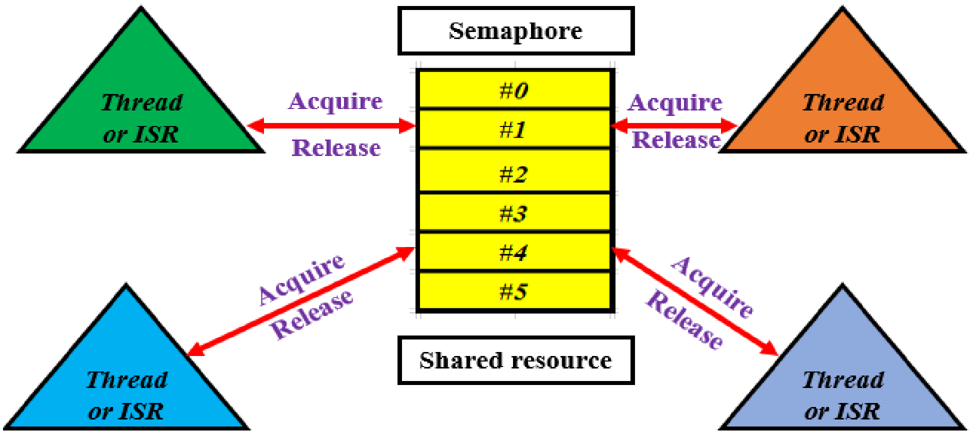


Figure 21: CMSIS-RTOS2 Semaphore Management

It can be observed that a semaphore object is necessary to be started to the maximal number of accessible tokens. These accessible resources are defined as parameter of osSemaphoreNew function. Every period a semaphore token is produced with osSemaphoreAcquire, where the count of semaphore is decreased. When the count of semaphore is zero, semaphore token is not produced. The thread or ISR takes effort to produce semaphore token that is necessary to wait until another token is available. It is to view that semaphores functions are released using osSemaphoreRelease upgrading the count of semaphore.

2.8.3 Mutex Management

Mutex, also known as mutual exclusion is widely used in different OSes for resource management. Various resources involved in a microcontroller device can be used continuously, however only by single thread at a time (for instance communication channels, memory, and files). Mutexes are mainly to secure access to a shared resource. A mutex is created and then sent between the threads, where they acquire and release the mutex, which is depicted in figure 22.

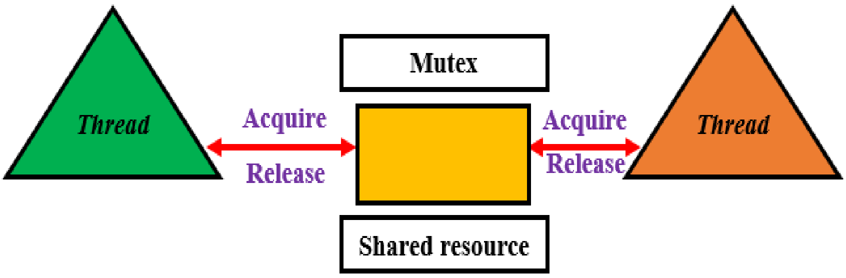


Figure 22: CMSIS-RTOS2 Mutex Management

It should be observed that mutex is a latest version of a semaphore. As semaphore, mutex acts as container for tokens. However, in order to not have multiple tokens, a mutex can handle only one token serving the resource. Thus, a mutex token refers to be a binary and bounded. The main feature of a mutex is that it involves thread ownership. If a thread acquires a mutex and acts as its owner, next mutex acquired from that thread will succeed suddenly without any delay (when osMutexRecursive function is defined). Hence, acquiring or releasing of mutex can be nested. It can be clearly noted that as thread functions, mutex functions cannot be called from ISR. On other hand, as compared to mutex, which is special kind of semaphore, binary semaphore is possible to release from ISR (Interrupt Service Routines).

2.8.4 Message Queue Management

Passing of message is next simple communication method between threads. In the message passing method, one thread transmits data explicitly, while other thread accepts it. The performance is like some type of I/O unlike a direct permit to information to be contributed. In CMSIS-RTOS2, this operation is referred to message queue. The data is transmitted from one thread to other in a FIFO fashion. Having message queue functions, one can regulate, transmit, accept, or hold for messages, which is picturized in figure 23. The data to be transmitted can be of integer or pointer style.

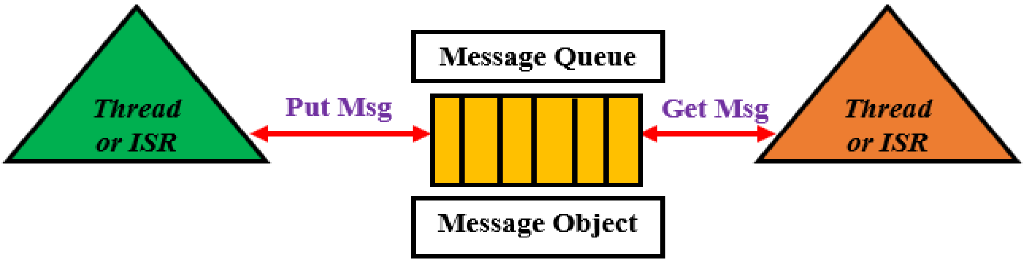


Figure 23: CMSIS-RTOS2 Message Queue Management

With respect to memory pool, message queues are quite inefficient in general, however it handles a wider range of defects such as if threads have invalid address space or the use of shared memory creates errors like mutual exclusion. It can be noted that like semaphores, message queue functions can be called from ISR (Interrupt Service Routines).

2.9 Keil RTX v5 Implementation

Keil RTX version 5 (RTX5) performs the CMSIS-RTOS2 as a native RTOS interface for ARM Cortex-M processor-based devices. It is also noted that a translation layer to CMSIS-RTOS API v1 is offered. Therefore, RTX5 offers both API layers: CMSIS-RTOS v1 and CMSIS-RTOS v2 and can be used in applications that were previously based on RTX version 4 and CMSIS-RTOS version 1 with minimal effort. A figure 24 picturizes the migration of Keil RTX-5, where one can intermix both API layers (CMSIS-RTOS v1 and CMSIS-RTOS v1). It can also be viewed from figure 24 that the thesis focusses on highlighted box, where MBED OS is implemented using CMSIS-RTOS2 with Keil RTX-5 as kernel and FreeRTOS using CMSIS-RTOS2 with FreeRTOS as kernel on same MPS2+ hardware board for comparison of MBED OS and FreeRTOS.

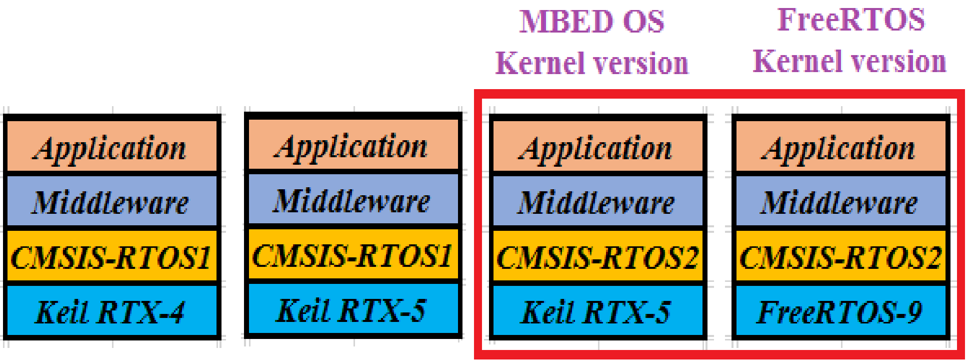


Figure 24: RTX v5 Migration

2.9.1 Operation of RTX v5

a) System Start-up

It can be noted that as main function is not available RTX5 thread does not involve with the system start-up until main() is reached. Once the execution attains main section there is an option to activate the hardware and begin the kernel. This is also replicated in the user code template file "CMSIS-RTOS2 'main' function" provided with the RTX5 component.

Any specific application’s main section must implement the following needs, which is described below [40]:

1. Initialization and configuration of hardware.
2. To update the system core clock using appropriate CMSIS-CORE function.
3. Initialization of CMSIS-RTOS kernel using osKernelInitialize function.

4. This is optional case, where one can create a new thread (for instance, app_main), which is decided as a main thread using osThreadNew function defined within CMSIS RTOS. However, threads can be created directly in main section.
5. Initiate the RTOS scheduler using osKernelStart function which does not return under successful execution. The required operations are predefined and programmed before osKernelStart function and once the kernel is activated, the codes that starts after osKernelStart function will not be executed unless kernel is suspended.

b) Memory Allocation

RTX5 objects such as thread, mutex, semaphore, message queue [40] and other functions needs facilitated RAM memory. Objects can be created using osobjectNew() calls and removed by osobjectDelete() calls. The related object memory required to be accessible during the lifetime of the object. RTX5 provides three various memory allocation methods for objects, which is described below:

1) **Global Memory Pool:** This refers to use a single global memory pool for all objects. It is simple to configure, but also have defects for memory fragmentation when objects with various sizes are created and killed. This allocates all objects from a memory area. This kind of memory allocation is the default configuration setting of RTX5.
2) **Object-specific Memory Pools:** It uses a stable-size memory pool for each object type. Its features are time deterministic and minimizes memory fragmentation, which refers to object creation and deletion requires exactly same period. As a stabilized-size memory pool is appropriate to an object type, errors like out-of-memory is minimized easily.
3) **Static Object Memory:** This method resumes memory during compilation time and absolutely dismisses the system with out of memory. This method is popular as it is needed for safety critical systems.

2.10 Description of stack size approximations in ARM Cortex-M adopted applications

This is a frequent inquiry - “How much stack memory needed for specific application?” which arises among various software builders investigating on applications which operate on microcontroller devices. If the allotted stack size is inadequate, the stack memory adopted automatically end up overflowing into memory spaces allotted for another storage of data. Thus, a program gets affected, where it produces incorrect results. For systems which have security needs, stack overflow also produces security susceptibilities. Therefore, corrective remedy should be done, which is explained in upcoming sections and also how MBED OS handles stack overflow and minimize it is described.

2.10.1 Stack Memory Layout

It should be noted that for ARM Cortex-M processors, the stack operates by “Full-Descending” model, which refers to pointing the stack pointers to last occupied stack area address. Also, if a new data is to be forced into stack, the simultaneous stack pointer is lessened and the data is stored in new memory address pointed to by stack pointer, which is depicted in figure 25. Hence, the first value for each stack pointer is filled to top of each stack area.

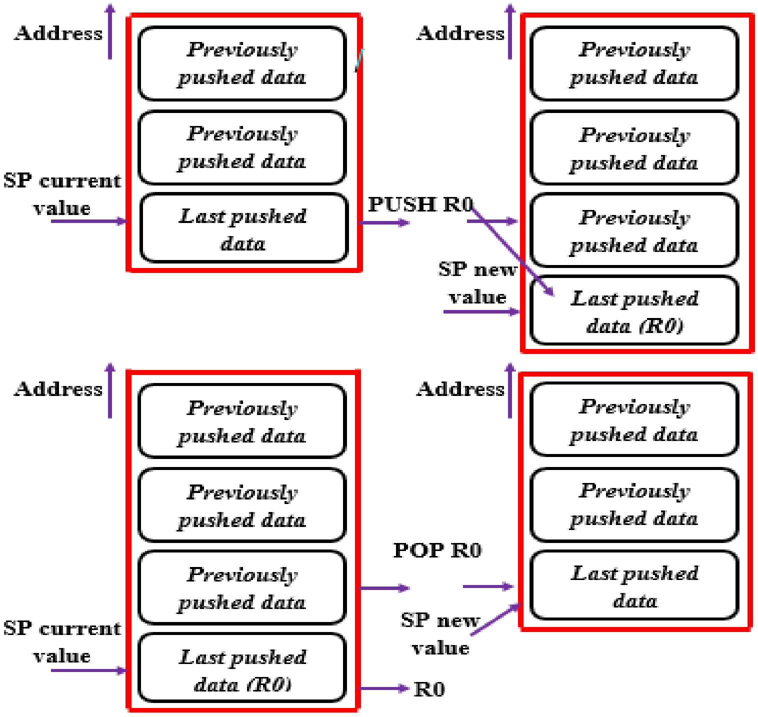


Figure 25: Stack Memory Layout

It can be noted that there are two stack pointers in processors which are adopted on ARMv6-M and ARMv7-M architectures. In the newest ARMv8-M architecture, the highest number of stack pointers is upgraded to 4 (Table 3) when an elective Security expansion is developed. The ARMv6-M architecture handles the Cortex-M0, Cortex-M0+ and Cortex-M1 processors, and ARMv7-M architecture handles the Cortex-M3, Cortex-M4 and Cortex-M7 processors, while ARMv8-M architecture governs Cortex-M23 and Cortex-M33 processors.

This is to observe that in many basic applications, which doesn’t adopt an RTOS, one can handle the MSP for whole operations. This describes that PSP remains unused and omitted. In this situation, one is permitted to have only single stack area in specific application. Basically, there is single stack memory space for the main stack, and several process stack spaces, one for each application thread. Accordingly, the PSP is dynamically migrated to each of these stack areas when the OS transfers between various threads.

Table 3: Stack Pointers available on ARM Cortex-M processors [14]

Kind of Stack	**Stack pointers on ARMv6-M, ARMv8-M architectures**	**Stack pointers on ARMv8-M architectures**
Main stack (default stack) for applications without RTOS, exception handlers	Main Stack Pointer (MSP)	Secure Main Stack Pointer (MSP_S) used only for security purposes
Process Stack for tasks within RTOS structure	Process Stack Pointer	Secure Process Stack Pointer (PSP_S) used for security reasons. Non-secure Process Stack Pointer (PSP_NS)

2.10.2 Determination of maximum stack consumption – RTOS Scheme

Considering an application adopting RTOS, it’s mandatory to assign stack area for main stack and for processor stack area for each of the threads. In addition, stack space needed for exception handlers is to be considered. In many situations, the RTOS vendors offers evaluation of main stack needs, but the main thing is that the need of stack size is highly dependent upon OS properties being used. From the figure 26, it is viewed that for an application with RTOS, Program Stack Pointer (PSP) is to be considered while executing application threads. Thus, the needed stack size composed the stack sections for an application threads as follows:

a) Greatest stack size needed by the application thread (collected from the stack consumption report via compilation tools), and circled to multiple of eight (double word alignment).

b) Stack size for the exception stack structure (8 words if FPU is not used in this thread, or 26 words if FPU is used).

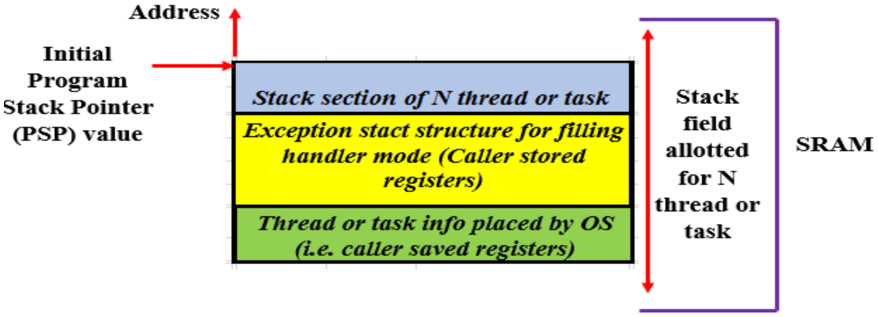

Address
Initial Program Stack Pointer (PSP) value
Stack section of N thread or task
Exception stact structure for filling handler mode (Caller stored registers)
Thread or task info placed by OS (i.e. caller saved registers)
Stack field allotted for N thread or task
SRAM

Figure 26: Stack Size required by specific application thread / task with RTOS

c) Further, stack size is needed for supplementary data storing work, which is needed by the OS. During context switching, the OS will adopt processor stack for each thread to store additional data. The correct size required depends upon OS, however this also provides area needed for stored registers (R4 to R12, and S16 to S31 if the FPU is adopted in the thread).

3. TESTING AND RESULTS

This chapter discuss about various testing and results that are conducted to each module of the project. This chapter also concludes performance analysis of MBED OS and FreeRTOS.

3.1 Memory Management on MBED Operating System - Overview

mbed OS offers memory allocation benefits which are based on a defined memory model, explained below. The memory allocation benefits contribute for many cases in memory allocation, which also heap allocation, pool allocation, and extendable pools.

Thus, in a typical embedded system, there are four kinds of memory like Code, Global Data, the heap, and the stack. Subsequently, the heap and the stack are coordinated, hence they fill equal section of memory. In mbed OS, we need two additional sections of memory which are uVisor memory and the never free heap. Memory is systematized as shown in figure 31. It should be noted that code memory normally located in ROM, hence it is not included in the figure 31.

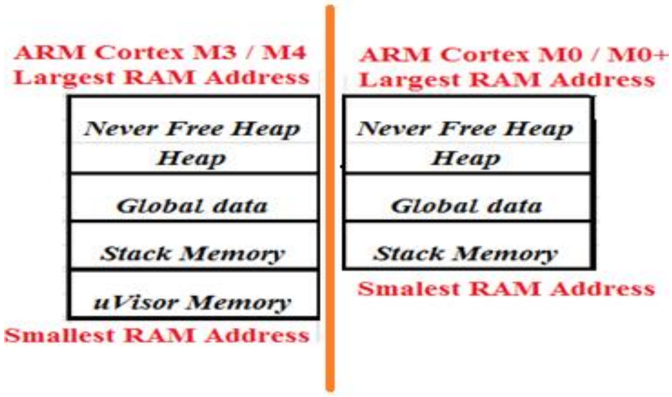


Figure 27: Memory Organization in MBED OS

a) uVisor Memory

It can be viewed from the figure 34 that on ARM Cortex-M3/M4, the uVisor occupies a less memory at the initial stage of RAM for its own purpose and for protected features (i.e. boxes). The uVisor protects this space adopting the MPU.

b) Stack

In MBED OS, the stack is located at the bottom of the memory, and it develops downwards. This address is purely taken as it allows access for overflows on stack to be identified automatically. In a Cortex-M3 or M4 system, where the uVisor is in use [15], the initial permit under the bottom of the stack will regulate a MemManage exception, handled by the uVisor. In a Cortex-M0 or M0+ system, it will regulate a HardFault. This allows accessibility for application to prevent from stack overflows which is basically done through reset operation..

This organization really mean that a stack must be evaluated to meet the needs of an application. Presently, this is a value set in the target, but a future version will expose stack configuration through yotta config.

c) **Global data**

Global Data is the conventional .bss and .data regions developed by the compiler. This size of this section is highly dependent upon application without any configuration.

d) **Heaps**

There are two types of heap in mbed OS. The standard heap develops upwards from the bottom of the heap region using the sbrk function, which is a trivial allocator developed in the core-util module. It affords linear allocation and deallocation, which is absolutely lock free. On the other hand, the never free heap develops downwards from the top. The never free heap is basically to be used with data which need not be freed like memory pools. Memory is allocated from the never free heap using the reverse sbrk function (krbs) provided by the core-util module [15].

3.2 Memory Management on FreeRTOS - Overview

It can be noted that in a tiny embedded system, adopting malloc() and free() to allocate memory for tasks, queues or semaphores can cause different defects like preemption while allocating some memory, memory allocation and free can be a nondeterministic operation, when compiled, they consume a large space or affects from memory fragmentation.

Thus, FreeRTOS contributes various possibilities to allocate memory, each adapted to a various situation however, all attempts to offer a solution adapted to tiny embedded systems. Once the correct situation is examined, the programmer can select an appropriate memory management scheme for which the kernel activity to be included. It can be noted that FreeRTOS maintains the memory allocation inside its portable layer. The portable layer is located outside of the source files which executes the core RTOS services thereby, permitting an application that fits an appropriate operation for the real-time system being developed. When the RTOS kernel needs RAM, instead of calling malloc(), it automatically calls pvPortMalloc(). When RAM is being freed, the RTOS kernel calls vPortFree() instead of free().

FreeRTOS contributes various heap management methods which varies in terms of difficulty and properties. Using two heap implementations obviously allows task stacks and other RTOS objects to be located in fast internal RAM, and application data to be located in slower external RAM. It can be noted that this project uses heap 5 for its memory allocation, which is picturized in figure 32 and further it is explained in next section.

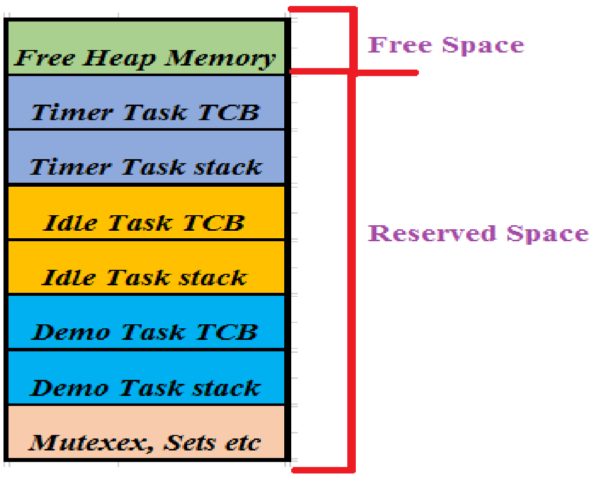


Figure 28: FreeRTOS Heap Layout

It can be noted that once the scheduler starts and run the whole application, there is no chance to reallocate any structures which are allocated earlier and to allocate new structures additionally. Thus, FreeRTOS basically includes TCB (Task Control Block), which is the structure used by FreeRTOS to govern the tasks Figure 33 depicts description about how the memory is handled. This memory management allocates a basic array sized after the constant configTOTAL_HEAP_SIZE in FreeRTOSConfig.h, and separates it in smaller parts to allocate for memory, where all tasks needs. This makes the application to appear to consume a lot of memory, even before any memory allocation.

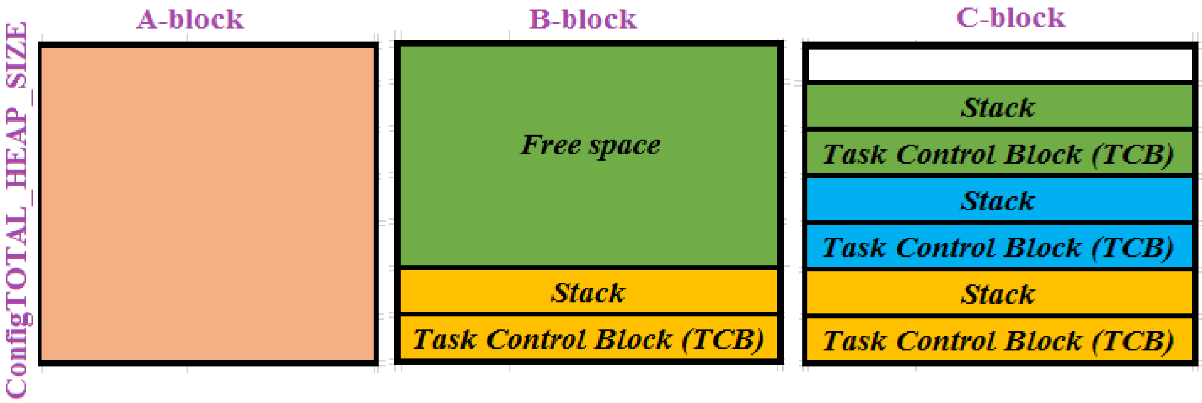


Figure 29: In A, no memory is allocated; in B, memory have allocated for blue task; in C, all required memory is allocated

3.3 Comparison of MBED OS and FreeRTOS with respect to Memory footprint

Initially, there is an investigation of what type of memory we need to consider. Widely talking, there is ROM memory, also referred as Program Memory which is used for stable saving of programs being executed and Data Memory (RAM), which is used for saving data temporarily and maintaining intermediate results and variables.

It can be noted that the executable code, constants and other read-only data comes under category called "RO" (for read-only), which is saved in the FLASH memory of the device. Initialised static and global variables pushes into a category called "RW" (read-write), and the uninitialized ones in to one called "ZI" (Zero Initialise). These Read -Write (RW) and Zero Initialized data comes under RAM memory, which is depicted in figure 34.

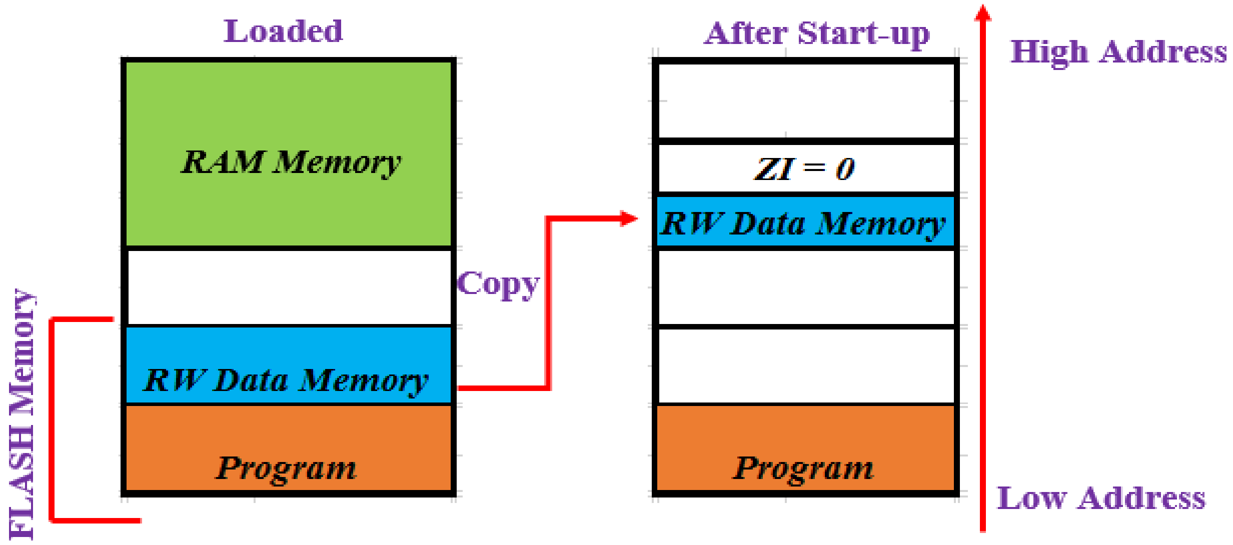


Figure 30: Typical Memory Model of OS

This should be noted that MBED OS uses Keil RTX-5 as kernel, while FreeRTOS uses FreeRTOS kernel. However, both the kernels are available with a CMSIS-RTOS v2 API interface. Also, CMSIS-RTOS v1 based applications may use the compatibility layer. Therefore, the same program (i.e. main program coded for Semaphore, Mutex and Message Queue functions) can be used for both MBED OS and FreeRTOS for comparison, which mainly differs based on kernel designated. In addition, the results are conducted on both ARM Cortex M4 as well as on its successor ARM Cortex M33.

3.3.1 Why ARM Compiler claims that “FreeRTOS is consuming all the available RAM”

From MBED OS point of view, it is observed from figure 35 that local variables that exists on stack and dynamically developed data that exists on heap differs during program execution and have steady initial points. Thus, MBED OS uses single memory area shared stack/heap model which enables flexibility in the size of each, restricted only by our available RAM. This means that the heap initiate at the start address after the end of ZI, developing up into higher memory addresses, and the stack begins at the last memory address of RAM, and grows downwards into lower memory addresses, thereby consuming smaller RAM memory.

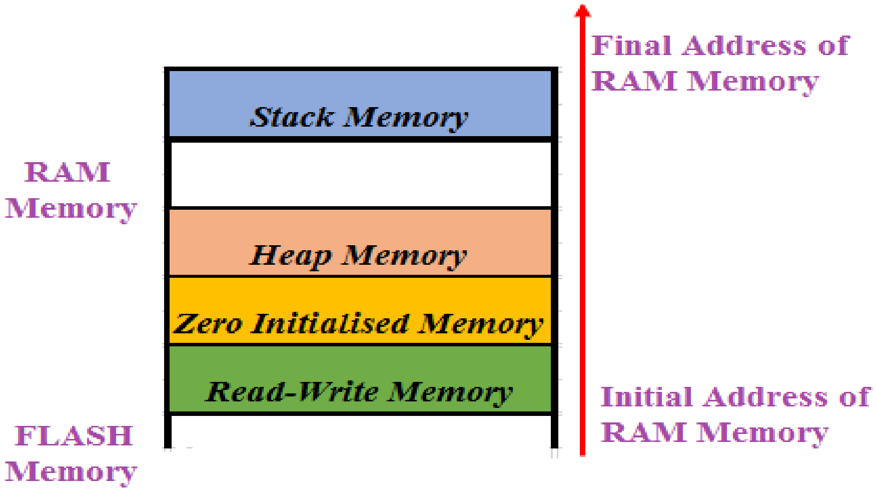


Figure 31: MBED OS RAM and Flash Memory

Taking FreeRTOS into account for contrast, there are different memory allocation methods provided with FreeRTOS which allocate memory from a statically allocated array which is structured by the configTOTAL_HEAP_SIZE constant in FreeRTOSConfig.h as stated earlier. These are just regular statically allocated arrays, and hence occurs in the RAM usage figures afforded by various tool chains. This project uses ARM Complier 6 toolchain for both MBED OS and FreeRTOS. The tool chain is not fully supportive as it represents the heap as RAM memory consumed, all though the heap is fully empty as no memory is still allocated.

C scripted applications needs some RAM to handle static variables, buffers, etc. but FreeRTOS randomly consumes entire RAM on a microcontroller. Most of the FreeRTOS applications shape the heap to use up all the RAM that is left over, thus confirming that the application is using entire RAM available. Due to this reason MBED OS is superior than FreeRTOS in terms of memory consumption. A table 4 shows the memory requirements of both MBED OS and FreeRTOS, where MBED OS represents good considerations in terms of memory.

Table 4: Comparison of MBED OS and FreeRTOS in terms of Memory Model

Parameters	**MBED OS – Memory**	**FreeRTOS – Memory**
Code Size	< 4 KBytes	5 to 10 KBytes
RAM space for Kernel	< 300 Bytes + 128 Bytes User Stack	Nearly 350 to 450 Bytes + 128 Bytes User Stack
RAM space for Task	52 Bytes + TaskStackSize	64 bytes (includes 4 characters for the task name) + the task stack size.
RAM space for Semaphore	8 Bytes	16 Bytes
RAM space for Mutex	12 Bytes	8 Bytes
RAM space for user timer	8 Bytes	24 Bytes

3.4 Simulation Results for Memory consumption

Based on above discussion about comparison between MBED OS and FreeRTOS in terms of memory consumption, the following tests are conducted on ARM Cortex M4 and M33, which is divided into 2 sections as follows. Before conducting the tests, which needs to be done using Keil environment, the required configuration files (i.e. CMSIS files, driver's files and other functions) for MBED OS and FreeRTOS are setup on Keil for a stable operation.

a) **Initial Testing -** This is carried out without thread and with single thread to examine the memory consumption (mainly code Memory) of specific operating system (say MBED OS or FreeRTOS). This is done to verify the functionality of CMSIS RTOS v2 API to be fit enough for MBED OS or FreeRTOS. Also, the test is conducted with Semaphore and Mutex functions.

b) **Final Testing –** This is performed to investigate the memory consumption when the program size is increased (i.e. main program). This is done with following functionalities within CMSIS RTOS v2 API on section 2.8:

   1) **Semaphore Structure -** used 5 threads and 2 Semaphore functions
   2) **Mutex Structure -** Used 5 Threads and 2 Mutex functions
   3) **Message Queue Structure -** Used 5 Threads and 2 Message Queue functions

Table 5: Thread with Stack Size designation

Name of Thread	**Stack Size for each thread given**
Thread_1	512 Bytes
Thread_2	96 Bytes
Thread_3	200 Bytes (Default Stack Size)
Thread_4	128 Bytes
Thread_5	96 Bytes

The above details are configured in kernel configuration (i.e. RTX_Config.h file for MBED OS) and minimal stack size of 128 words (i.e. 128*4 = 512 Bytes) needed for FreeRTOS is configured in FreeRTOSConfig.h file. Each thread has given its own stack size, which is tabulated in table 5.

3.4.1 Initial testing on ARM Cortex M4 and M33

An initial testing is conducted for different cases as stated above on ARM Cortex M4 which is depicted in figure 32, where the different memory types (for instance code memory, read only data memory etc.) are plotted against the memory consumed by specific Operating System (MBED OS or FreeRTOS). An initial testing is performed using Keil environment, which is user-friendly product of ARM and has quick response. An execution of program is referred as process, which differs from program, thus I created a program (described on Appendix-B) for different cases to be tested on ARM Cortex M4. A process has two entities which are active entity and passive entity - both are held by process that contains memory, state and CPU. Program memory is splitted into four various types. They are code memory which is made to compile the program code, reads from non-volatile storage. To execute main global and static variables that are allocated, dynamic memory allocate heap to manage functions. Local variables are used by stack when they are declared. Operating system which is defined by resources allocator allocates and manage memory. Control program controls and execute the user program and IO device applications. Kernel, which is heart of OS handles running programs (application programs). The results estimated for different cases are as follows:

1) Without Thread on MBED OS and FreeRTOS

An initial testing where without considering thread is conducted on Cortex M4 for MBED OS and FreeRTOS, thus both OS having different kernels. A program is compiled without thread, where RTX 5 kernel on MBED OS does not interface until the main reached. Once the main() is reached, the OS kernel will initialize and starts the kernel. FreeRTOS kernel also does the same function. Once the compilation is successful the Keil environment generates the memory map file which describes different types of memory consumed for MBED OS and FreeRTOS, thereby calculating used RAM and ROM sizes which is a key-note as follows, whose values are tabulated on Appendix-A:

Total RAM Size = RW Data + ZI Data   ***Total ROM Size = Code + RO Data + RW Data***

a) Without Thread - ROM & RAM usage (MBED OS)

ROM = 4808+496+176 = 5480 Bytes   RAM = 176+9520 = 9696 Bytes

b) Without Thread - ROM & RAM usage (FreeRTOS)

ROM = 5636+224+24 = 5884 Bytes   RAM = 24+14880 = 14904 Bytes

2) With one Thread on MBED OS and FreeRTOS

Thread consists of own stack, program counter and set of registers for execution. It is known as parallelism to improve application. The process execution of threads has own independent resources. Number of threads correspondingly increase process to execute parallelly. The CPU switches often among the threads which are running in parallel. There are two types of threads they are user and kernel threads. The user threads are used for application program by the user. Kernel threads are supported by operating system, it allows multiple tasks that calls simultaneously. Accordingly, the program is compiled with thread having default stack size (200 Bytes) and the Keil environment generates the memory map file, thus the total RAM and RAM Sizes are below:

a) With one Thread - ROM & RAM usage (MBED OS)

ROM = 5128+496+224 = 5848 Bytes RAM = 224+9496 = 9720 Bytes

b) With one Thread - ROM & RAM usage (FreeRTOS)

ROM = 6420+224+24 = 6668 Bytes RAM = 24+14880 = 14904 Bytes

By comparing with and without thread the memory consumptions (RAM and ROM) of two operating systems are producing nearly identical values. Hence the thread function affords greatly for MBED OS and FreeRTOS which reduces the execution time in an operating system. But the memory consumption which decides stable operating system differs greatly for MBED OS and FreeRTOS, which is explained later within this section.

3) With one Thread and Semaphore on MBED OS and FreeRTOS

Semaphore function in an operating system is used for protected variables that can facilitate shared resources multi-processing environment. Semaphores are performed with thread functions, where it acts as token passing, thus performing Semaphore acquire and release operations by using thread. Like previous cases, the compilation is done with one thread and semaphore, where the calculations are as follows:

a) With one Thread and Semaphore - ROM & RAM usage (MBED OS)

ROM = 5884+496+248 = 6628 Bytes RAM = 248+9520 = 9768 Bytes

b) With one Thread and Semaphore - ROM & RAM usage (FreeRTOS)

ROM = 7200+224+24 = 7448 Bytes RAM = 24+14896 = 14920 Bytes

4) With one Thread and Mutex on MBED OS and FreeRTOS

Mutex also known as mutex exclusion is mainly related to resource management, where various resources will be contained on device and unlike semaphore mutex handles only single thread at a time by finding corresponding resources with exclusive name allotted to each resource. The calculations are as follows:

a) With one Thread and Mutex - ROM & RAM usage (MBED OS)

ROM = 5832+528+248 = 6608 Bytes RAM = 248+9528 = 9776 Bytes

b) With one Thread and Mutex - ROM & RAM usage (FreeRTOS)

ROM = 7060+256+24 = 7340 Bytes RAM = 24+14896 = 14920 Bytes

It can be noted that ROM and RAM memory consumption plays important role in deciding stable operating system for NB-IoT, hence the obtained data are plotted for comparison between MBED OS and FreeRTOS, which are depicted on figure 33 for ARM Cortex M4.

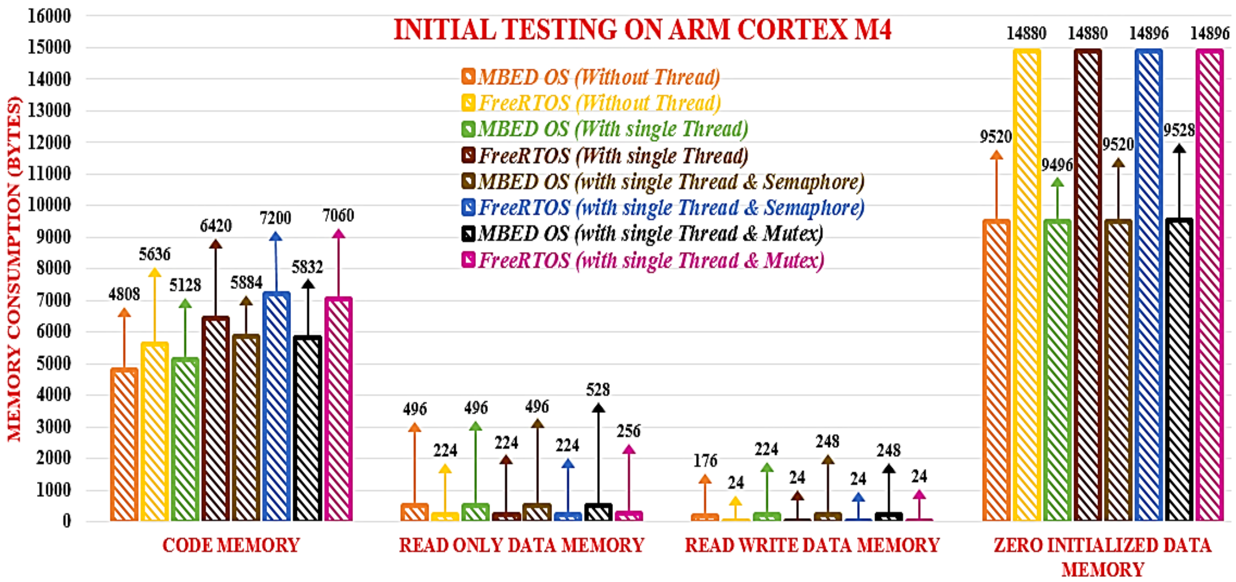


Figure 32: Initial Testing on ARM Cortex M4

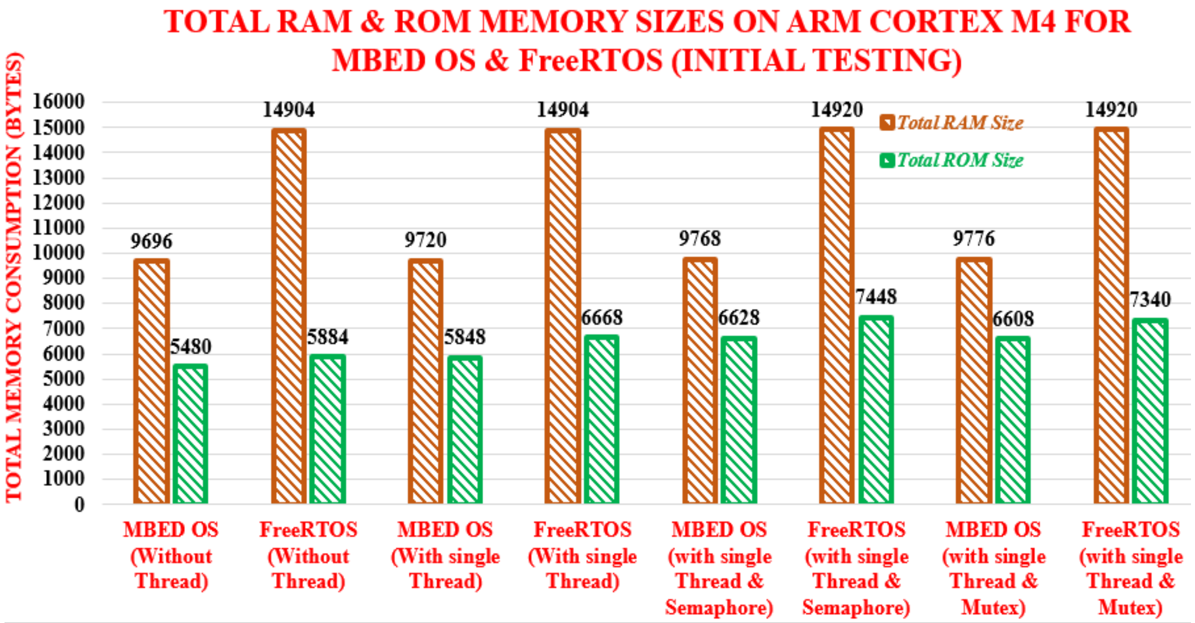


Figure 33: Total RAM and ROM Memory on ARM Cortex M4 - Initial Testing

As it is known that ARM Cortex M33 is successor of M4. Hence an initial testing is also conducted on M33, where the memory management slightly differs between two processors due to CMSIS Core registers and peripherals defined within them.

A calculated RAM and RAM Sizes on Cortex M33 are as follows and plot for different memory types is depicted on figure 34 and the below calculated data for total RAM and ROM sizes is plotted, which is depicted on figure 35:

1) **Without Thread on MBED OS and FreeRTOS**

a) **Without Thread - ROM & RAM usage (MBED OS)**

ROM = 5148+860+176 = 6184 Bytes RAM = 176+9520 = 9696 Bytes

b) Without Thread - ROM & RAM usage (FreeRTOS)

ROM = 5668+588+24 = 6280 Bytes RAM = 24+14976 = 15000 Bytes

2) With one Thread on MBED OS and FreeRTOS

a) With one Thread - ROM & RAM usage (MBED OS)

ROM = 5468+860+224 = 6552 Bytes RAM = 224+9496 = 9720 Bytes

b) With one Thread - ROM & RAM usage (FreeRTOS)

ROM = 6452+588+24 = 7064 Bytes RAM = 24+14976 = 15000 Bytes

3) With one Thread and Semaphore on MBED OS and FreeRTOS

a) With one Thread and Semaphore - ROM & RAM usage (MBED OS)

ROM = 6224+860+248 = 7332 Bytes RAM = 248+9520 = 9768 Bytes

b) With one Thread and Semaphore - ROM & RAM usage (FreeRTOS)

ROM = 7236+588+24 = 8096 Bytes RAM = 24+14992 = 15016 Bytes

4) With one Thread and Mutex on MBED OS and FreeRTOS

a) With one Thread and Mutex - ROM & RAM usage (MBED OS)

ROM = 6172+892+248 = 7312 Bytes RAM = 248+9560 = 9808 Bytes

b) With one Thread and Mutex - ROM & RAM usage (FreeRTOS)

ROM = 7108+620+24 = 7752 Bytes RAM = 24+14992 = 15016 Bytes

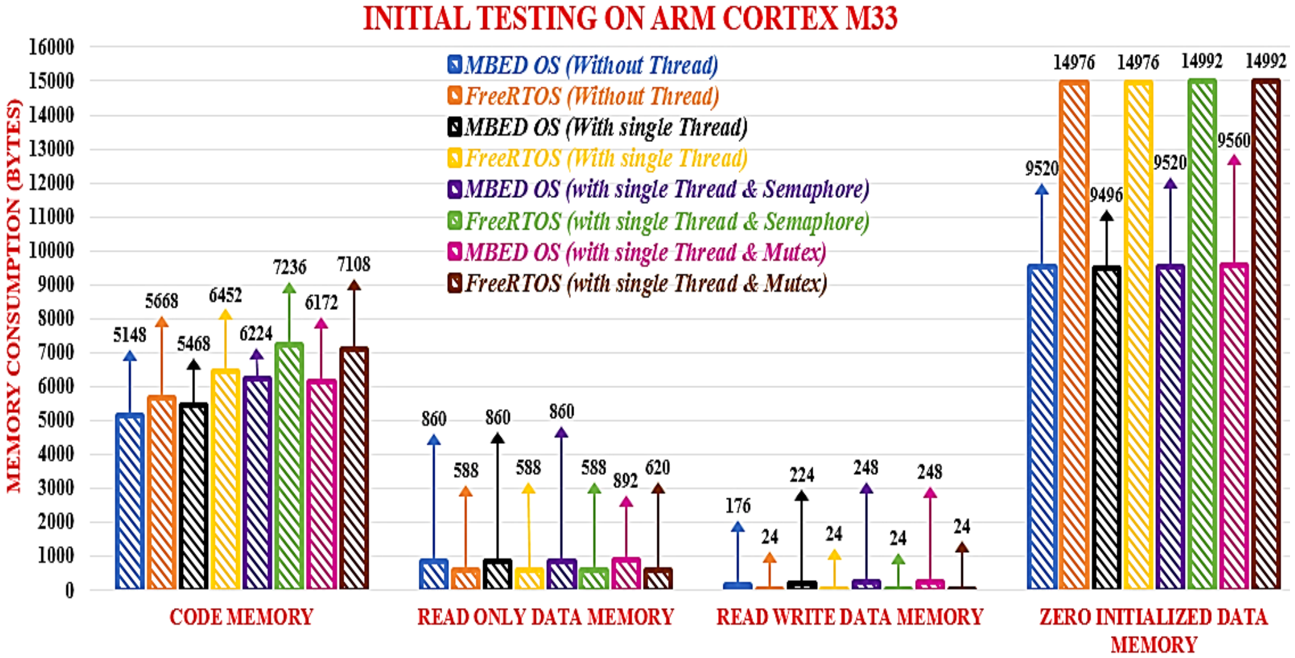


Figure 34: Initial Testing on ARM Cortex M33

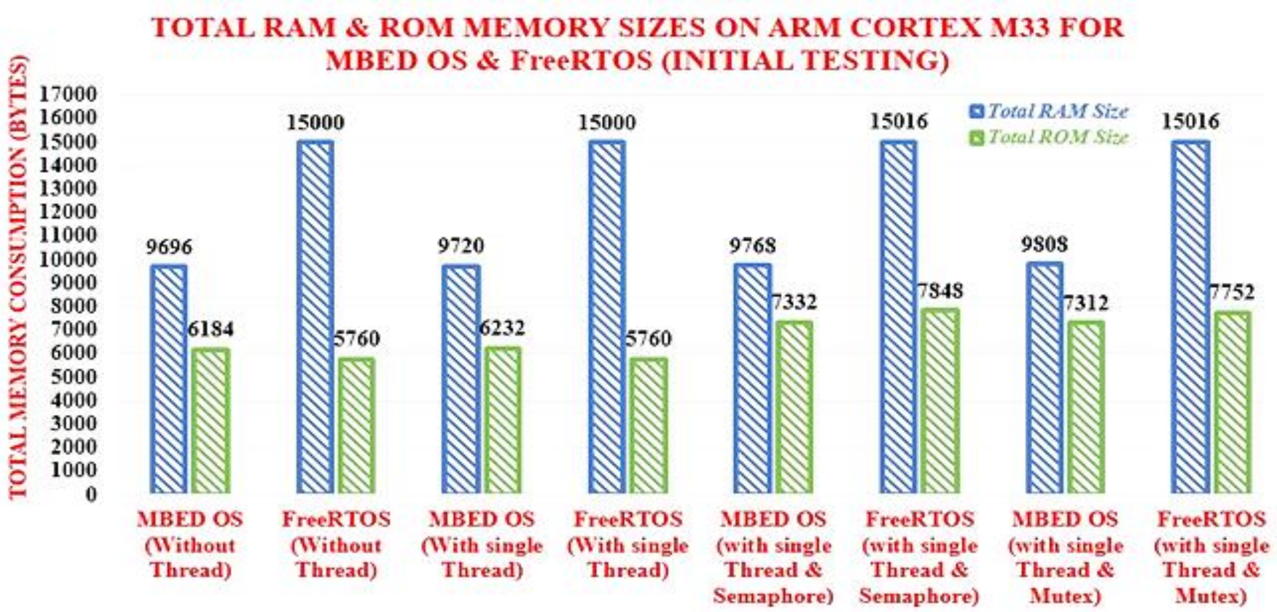


Figure 35: Total RAM and ROM Memory on ARM Cortex M33 - Initial Testing

3.4.2 Final Testing on ARM Cortex M4 and M33

A final testing is performed by increasing program size using management function to verify the memory consumptions for MBED OS and FreeRTOS. By increasing number of threads with different priority and different stack size as tabulated on table-5, we can calculate the memory management on both operating systems. Multi thread modes defines all user threads to be mapped into single kernel thread. The thread management is handled by thread library in user space, which is efficient in nature. Using this we can compare the memory consumption between less thread and more thread functions which decides multi-tasking operation for specific Operating System. The calculation of RAM and ROM sizes are as follows and the corresponding plots between MBED OS and FreeRTOS for final testing on M4 is depicted in figure 36 and the below calculated values for RAM and ROM sizes are depicted on 37 for comparison between MBED OS and FreeRTOS.

1) With 5 Threads and 2 Semaphores on MBED OS and FreeRTOS

a) MBED OS

ROM = 6716+732+248 = 7696 Bytes RAM =248+11704 = 11952 Bytes

b) FreeRTOS

ROM = 8508+520+24 = 9052 Bytes RAM = 24+15008 =15032 Bytes

2) With 5 Threads and 2 Mutexes on MBED OS and FreeRTOS

a) MBED OS

ROM = 6656+740+248 = 7644 Bytes RAM =248+11728 = 11976 Bytes

b) FreeRTOS

ROM = 8356+528+24 = 8908 Bytes RAM = 24+15008 =15032 Bytes

3) With 5 Threads and 2 Message Queue on MBED OS and FreeRTOS

With message queue, it's different process compared to Semaphore or Mutex function, since it does two operations simultaneously (send and receive), which is explained clearly on thread context switching on section 3.6.1. The calculation includes:

a) MBED OS

ROM = 7868+676+272 = 8816 Bytes RAM =272+11752 = 12024 Bytes

b) FreeRTOS

ROM = 8980+464+24 = 9468 Bytes RAM = 24+15024 =15048 Bytes

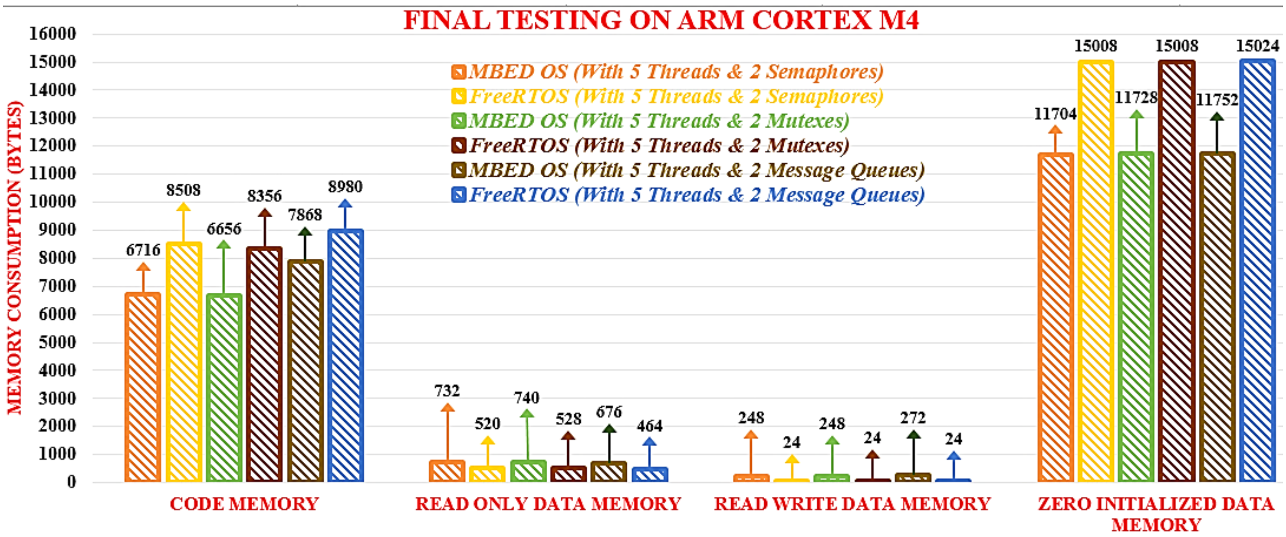


Figure 36: Final Testing on ARM Cortex M4

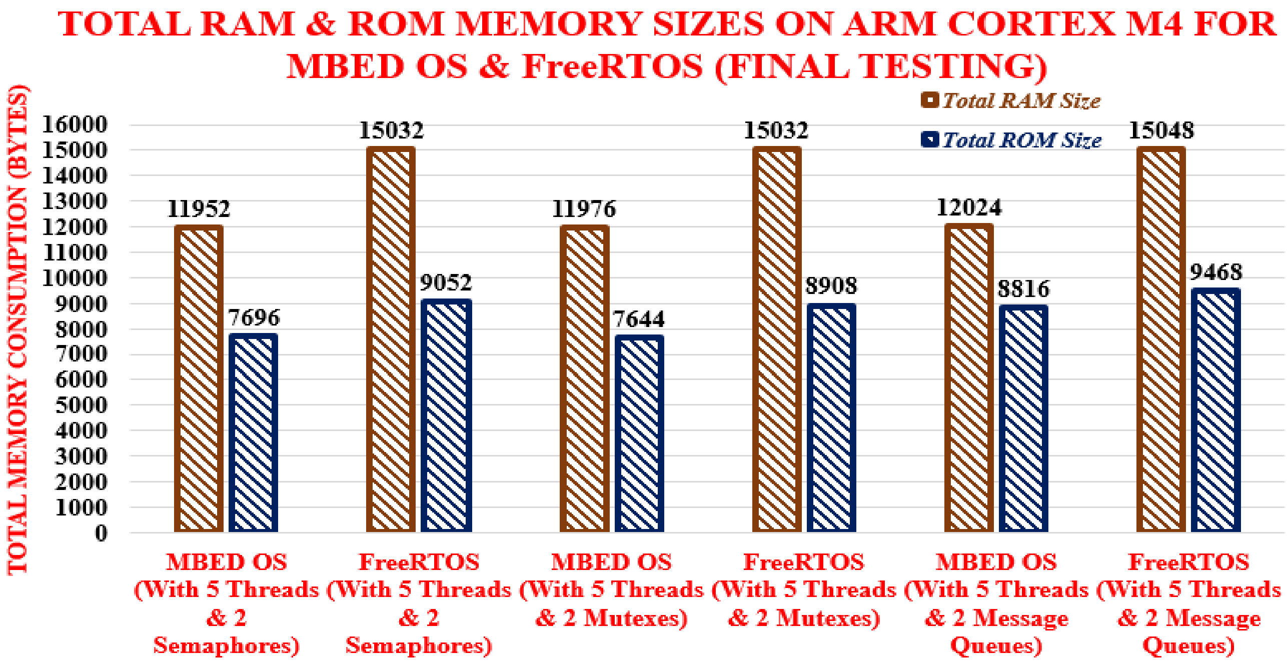


Figure 37: Total RAM and ROM Memory on ARM Cortex M4 - Final Testing

Similarly, the final testing is conducted on ARM cortex M33, since it is built from M4 as a foundation for Cortex M33, which is main factor to be considered for NB-IoT. The corresponding plots between MBED OS and FreeRTOS for final testing on M33 is depicted in figure 38 and the below calculated values for RAM and ROM memory sizes for MBED OS and FreeRTOS is depicted on figure 39.

1) **With 5 Threads and 2 Semaphores on MBED OS and FreeRTOS**

a) **MBED OS**

ROM = 7056+1096+248 = 8400 Bytes RAM = 248+11704 = 11952 Bytes

b) **FreeRTOS**

ROM = 8548+884+24 = 9456 Bytes RAM = 24+15008 =15032 Bytes

2) **With 5 Threads and 2 Mutexes on MBED OS and FreeRTOS**

a) **MBED OS**

ROM = 6996+1104+248 = 9900 Bytes RAM =248+11728 = 11976 Bytes

b) **FreeRTOS**

ROM = 8412+892+24 = 9328 Bytes RAM = 24+15008 =15032 Bytes

3) With 5 Threads and 2 Message Queue on MBED OS and FreeRTOS

a) **MBED OS**

ROM = 8216+1040+272 = 9528 Bytes RAM =272+11752 = 12024 Bytes}

b) **FreeRTOS**

ROM = 9012+828+24 = 9864 Bytes RAM = 24+15024 =15048 Bytes

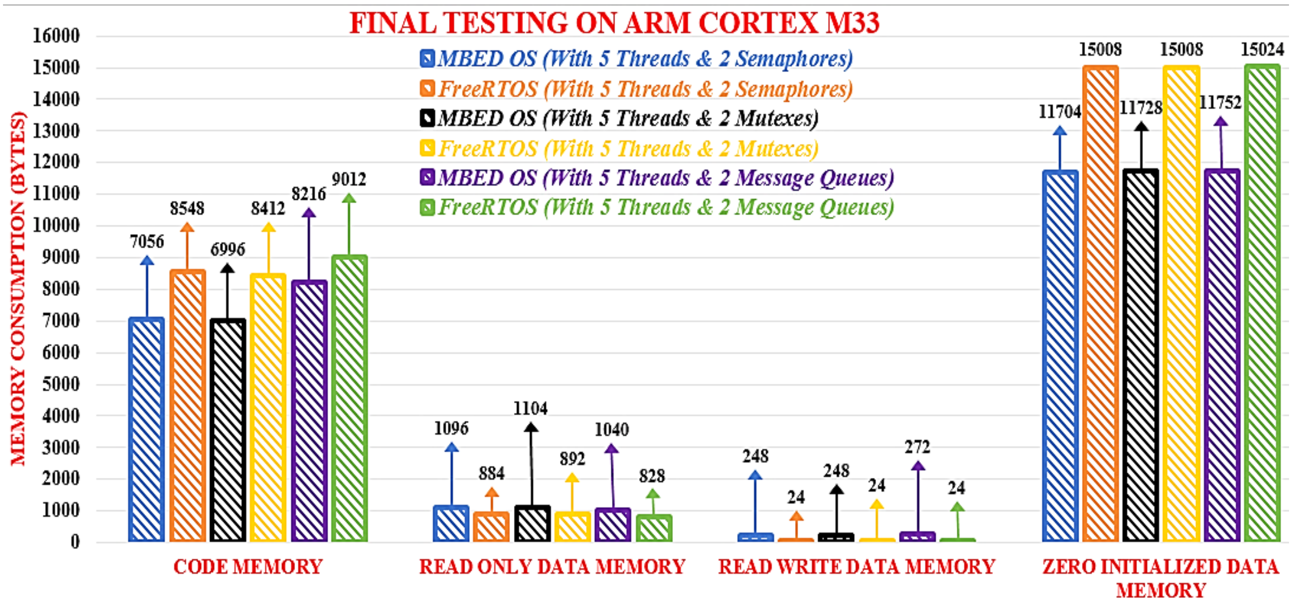


Figure 38: Final Testing on ARM Cortex M33

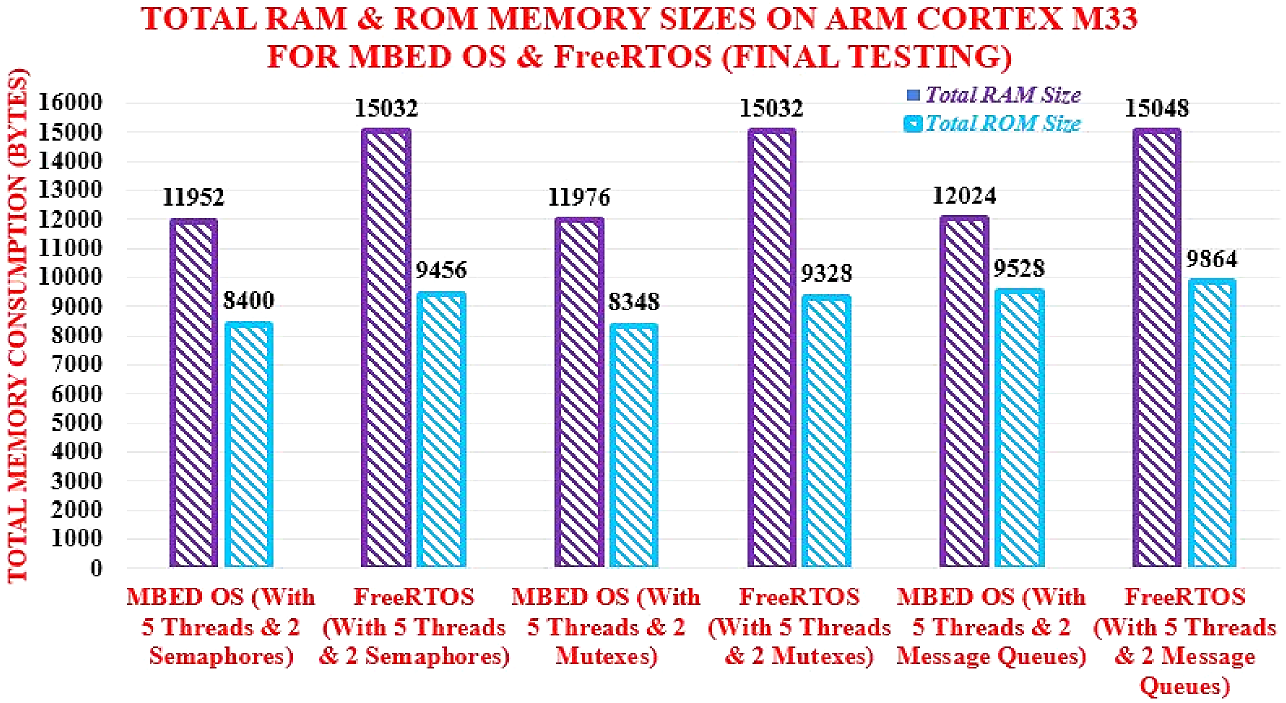


Figure 39: Total RAM and ROM Memory on ARM Cortex M33 - Final Testing

Deciding stable operating system for NB-IoT through Initial and Final Testing

It can be noted from all the above cases (i.e. Comparison between MBED OS and FreeRTOS) that Code Memory (which decides ROM memory) is most important factor in memory consumption, which prescribes flexibility and robustness of specific Operating System. By viewing all the cases (i.e. with and without Thread, with Semaphore or Mutex), MBED OS dominates FreeRTOS and plays a significant role, which is clearly explained below.

From FreeRTOS point of view, the defined ROM/Flash footprint structures are genuine. If we design a small FreeRTOS test program and it causes to consume more ROM than expected then it is mainly because of the libraries that are being added in our build, apart from defects on FreeRTOS Kernel. In general, GCC string handling and any floating-point library will bloat our code. This can be minimized by adding various string handling functions in a file known as printf-stdarg.c to the project. This method drastically degrades both the content of ROM used by the build, and the size of the stack needed to be allocated to any task making a string handling library call (i.e. sprintf()). It should be noted that even though printf-stdarg.c is open source; this file cannot be added to FreeRTOS as it is not adopted by the FreeRTOS license. On other hand, MBED OS has great support with many string handling functions (also printf-stdarg.c). On final testing by increasing thread and other functions to investigate multitask operations, any specific operating system consumes more memory due to more functions. Thus, FreeRTOS is consuming large RAM, which affects threads or tasks to be performed on time and it takes more time which is clearly explained via performance analysis. Hence MBED OS is becoming superior than FreeRTOS, which is consuming less memory and performing multiple tasks on time, thereby fulfilling the needs of NB-IoT.

3.5 Comparison of MBED OS and FreeRTOS in terms of Stack consumption

It should be accounted that depending upon type of toolchain used, the memory structure in SRAM is designed, which is depicted in figure 40, considering two cases for memory structure – one arrangement, where stack field is located at the end of used SRAM area and another case, where stack field is imported at the end of unused SRAM area. The project uses ARMCLANG toolchain (ARM Compiler 6.6 for both MBED OS and FreeRTOS.

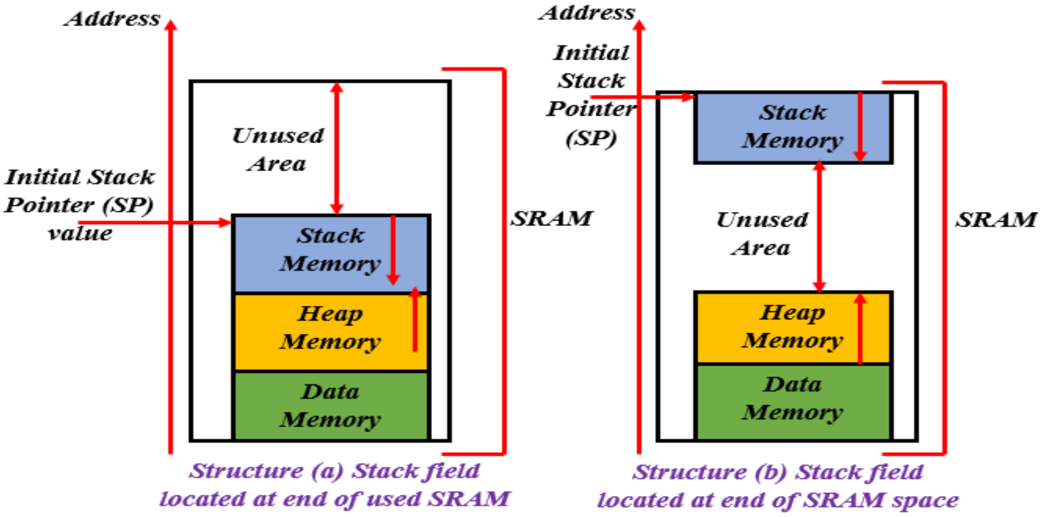

Address
Initial Stack Pointer (SP) value
Unused Area
Stack Memory
Heap Memory
Data Memory
SRAM
Structure (a) Stack field located at end of used SRAM
Address
Initial Stack Pointer (SP)
Stack Memory
Unused Area
Heap Memory
Data Memory
SRAM
Structure (b) Stack field located at end of SRAM space

Figure 40: Memory Layout for SRAM

Compared to FreeRTOS, there is a special feature of second level of stack pointer computation before arriving the main () function in MBED OS. This property dominates the value of stack pointer that was entered in vector table although the recent value can be like one occurred in vector table in various situations. Thus, this structure is more effective for applications which consumes external memory for stack memories and the external memory connection required to be setup before being adopted. With this structure, the initial stack (inside the vector table) can be fixed to an internal SRAM space to access the reset handler to work and compute the external memory connection, before the external memory is consumed for saving stack inside the main application program.

With respect to FreeRTOS, there is no possibility to provide supplementary stack areas for exception handlers in contrast to MBED OS. The main reasons for defects in FreeRTOS in terms of stack consumption are as follows:

a) An adoption of function pointers within an application defines that tool is unable to generate a call tree.

b) In various tools, the stack consumption defined for functions in C runtime libraries are difficult to predict.

c) An application within FreeRTOS produces repeated function calls or adjusting the program by itself.

However, these errors within FreeRTOS can be reduced by estimating greatest stack consumption for specific functions defined within an application manually or by conducting tests. For instance, debugger can be used to enter the stack memory field with certain data structures before executing a program and investigate the stack memory consumed with respect to stack memory values entered initially from the program execution. But this method also can't able to withstand by FreeRTOS, which is explained later within same chapter.

3.6 Simulation results for Stack consumption via Thread Management

The operation of program is setup with characteristics of CMSIS RTOS v2 API (i.e. Semaphore, Mutex and Message Queue) as configured earlier in section 3.4 for both MBED OS and FreeRTOS. Further the control block for each object (i.e. Thread, semaphore etc.) is allocated which depends upon the number of objects used and stack size allocated. This is tabulated in table 6. It should be noted that there is no any special description for event control block within FreeRTOS. Also, FreeRTOS kernel that is used within CMSIS RTOS v2 API shows only stack consumed (in Bytes) by each task or thread with corresponding thread states, priority levels and top and limit address of stack rather than representing stack consumption with watermarks and maximum stack size that will be taken by each specific thread (different from used / consumed stack memory) as these features are incorporated within Keil RTX-5 kernel used by MBED OS. In addition, FreeRTOS kernel doesn't represent the stack size which is originally allocated within a program code as this feature is inbuilt in RTX-5 kernel. Thus, it's really complicated to handle FreeRTOS kernel in contrast to Keil RTX-5 kernel. These details along with kernel structures for comparison between MBED OS and FreeRTOS are designated in Thread Management table, which is described in Appendix-A

Table 6: Control Block Size of CMSIS RTOS v2 API Objects

CMSIS RTOS v2 API Objects	Control Block Size (in Bytes)
Threads (used 5 Threads)	340 Bytes
Semaphores (used 2 Semaphores)	32 Bytes
Mutexes (used 2 Mutexes)	56 Bytes
Event Flags [Within Message Queue] (used 2 Event Flag)	32 Bytes
Message Queues (used 1 Message Queue)	52 Bytes

3.6.1 Results of Stack Consumption for Semaphore, Mutex and Message Queue functions

It can be observed from figure 41 that each thread has allocated different stack sizes and if certain threads are in ready or blocked mode they will consume same stack sizes except in running mode, since it differs with respect to operation. This is similar case even for threads with same stack sizes, which consumes same amount of memory in ready and blocked mode except in running mode. Also, it should be noted from thread management table described in Appendix-A that **'maximum'** tab defines maximum stack size to be accepted by each thread and **'used'** tab defines how much amount of stack memory (in bytes) to be consumed.

This is similar case for Mutex, which is picturized in same figure 41, except that the stack consumption changes for running Thread due to Mutex functionality.

a) Operation of Thread Context Switching

This is explained with respect to program coded for each function (i.e. Semaphore, Mutex and Message Queue) as follows:

The corresponding programs are described in Appendix-B for reference. As described earlier in section 2.8.3 that Semaphore handles multiple tokens while Mutex handles only single token at a time. Apart from this, the operation of Semaphore and Mutex are almost identical. The program is designed considering 5 Threads or tasks with different priorities (with Thread 5 having low priority and Thread 3 with highest priority and operations. The remaining details like which Thread acquires which semaphore or Mutex function is explained in Thread Management table described briefly in Appendix-A.

It can be noted that as threads are coded with delays, the operation of specific thread performs later although it has highest priority. Likewise, taking Semaphores and Mutexes functions into account, each thread operates until it is preempted by thread with highest priority. Thus, initially Thread 5 runs until it is preempted by Thread 3, while all other Threads remains in ready or blocked state. During this time, Semaphore or Mutex function defined within this thread (say Thread 5) is initially created and passed between different Threads, where they acquire and release semaphore or Mutex functions (like token passing scheme) in a cyclic manner.

With respect to Message Queue, it’s totally different operation for all 5 Threads as compared to Semaphore or Mutex function. It can be noted that Thread 1 sends message queue, where Thread 2 receives it. Next, Thread 3 sends two event flags, where Thread 4 receives one event flag and Thread 5 receives another. Likewise, Thread 3 starts initially as it doesn’t contain any delays in a program and sends event flags, while all other Threads will be in ready mode to send or receive messages.

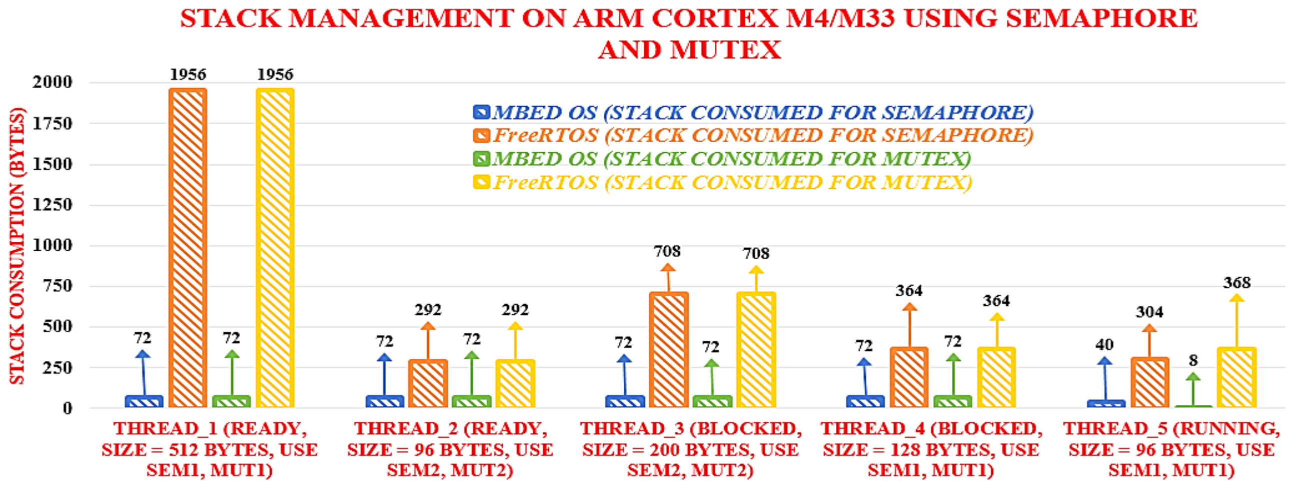


Figure 41: Stack Management on ARM Cortex M4 / M33 using Semaphore and Mutex functions

It should be noted from figure 42 that if the operation of specific thread is different compared to other threads, it will consume different stack sizes even though they are in ready or blocked mode, which is a case for Message Queue, since Threads 1 and 2 handles Message Queue functionality and Threads 3, 4 and 5 handles event flags functionality, although programmed within Message queue. The corresponding values are tabulated in table [Thread Management using Message Queue] on Appendix-A.

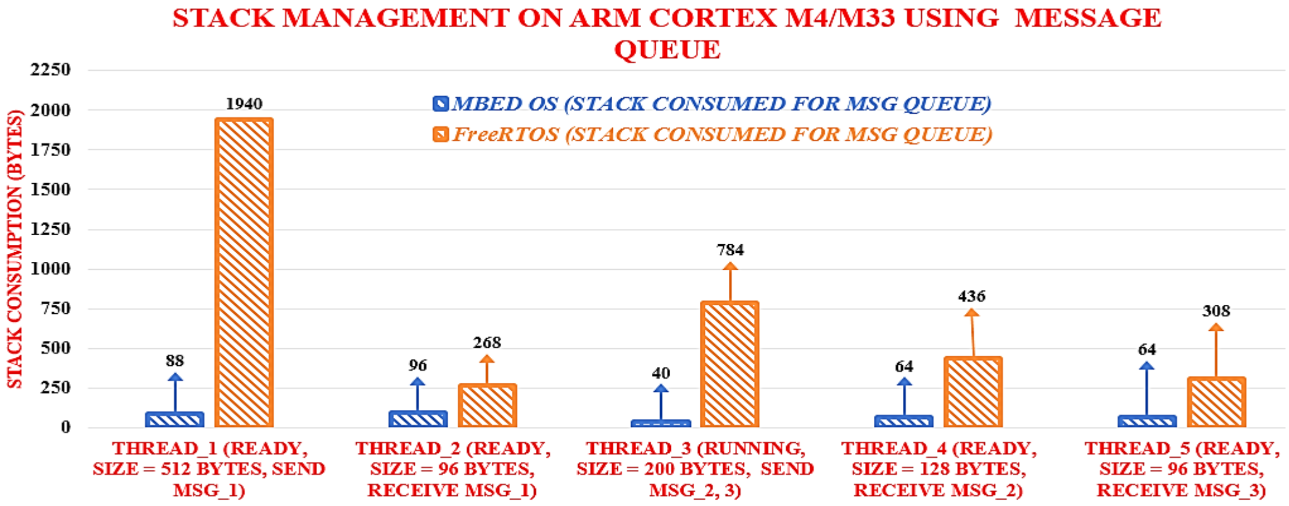


Figure 42: Stack Management on ARM Cortex M4 / M33 using Message Queue function

Deciding stable operating system for NB-IoT through Stack Management

Taking FreeRTOS into account, it can be noted from Thread management table described in Appendix-A, where the threads allocated with different stack sizes consumes different sizes in every mode (ready, running, suspended or delayed mode), which is nearly 3 times greater than actual stack size. This is mainly caused where the parameter **"minimal stack size"** configured in FreeRTOS Configuration table (Default Size = 128 Words [128 * 4 = 512 Bytes]). It was stated earlier in section

3.5 that one can use Debugger for reducing defects in stack consumption, however it's not possible as due to minimal stack size, total stack size consumed (considering 5 Threads or tasks) is higher (nearly 4Kbytes) than actual values (832 Bytes). This is because there is no way to allocate stack size for specific thread smaller than minimal stack size and this minimal stack size deals with idle thread that comes within FreeRTOS kernel. Thus, if the stack size for user Thread is smaller than minimal stack size, it affects the whole memory structure on FreeRTOS kernel, which represents unstable allocation of memory for all threads or tasks created within a program. On contrast, there is no minimal stack size incorporated in MBED OS and the stack memory consumption is less than actual values, which is tabulated in table 7 and the corresponding plot is depicted in figure 43. Apart from this, the main thing is to note that event flags are incorporated within Message Queue functionality, which has greater affordability for MBED OS, whereas with FreeRTOS, the queue functionality is not fully supportive since the queue data structures consumes whole RAM memory than other RTOS systems allocate individually. Thus, considering all the points from the beginning MBED OS is still dominating and becomes preferable for Cellular IoT system.

Table 7: Total Stack Consumption – MBED OS and FreeRTOS

Actual Stack Size (in Bytes, considering all 5 Threads)	**Total Consumed Stack Size (in Bytes, considering all 5 Threads) – MBED OS**	**Total Consumed Stack Size (in Bytes, considering all 5 Threads) – FreeRTOS**
Semaphore Function (832 Bytes)	328 Bytes	3624 Bytes
Mutex Function (832 Bytes)	296 Bytes	3688 Bytes
Message Queue Function (832 Bytes)	352 Bytes	3736 Bytes

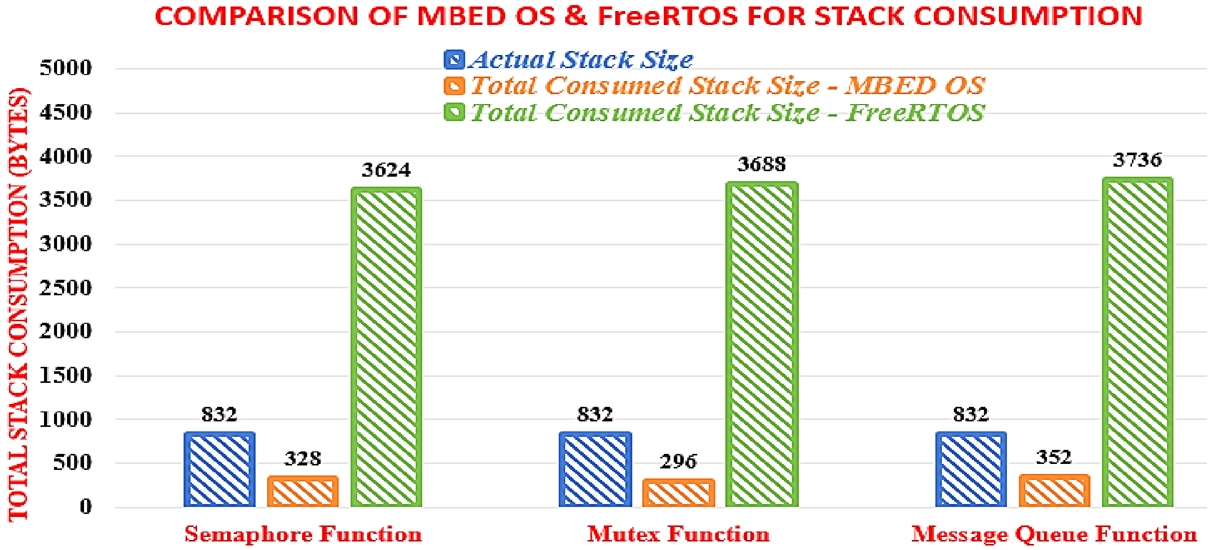


Figure 43: Comparison of MBED OS and FreeRTOS for total Stack consumption

3.7 Performance Analysis of MBED OS and FreeRTOS

The performance Analyzer incorporated in Keil product represents execution statistics such as execution time and number of calls within a function or module. The test is conducted considering all the five threads or tasks for each CMSIS RTOS-2 functions (i.e. Semaphore, Mutex and Message Queue). The corresponding values are tabulated on table 8, where the execution time for threads in blocked or ready mode has same time to execute for both MBED OS and FreeRTOS. With respect to Semaphore and Mutex functions, Thread 5 runs until it is preempted by Thread 3 which was stated earlier in section 3.6.1, hence the execution time differs for both MBED OS and FreeRTOS, which is observed from table 8 Also, it is to note that tasks under MBED OS executes faster than under FreeRTOS because of less memory consumption explained earlier and it has good stability and robustness, thus again agrees that FreeRTOS is still far away from MBED OS in development towards NB-IoT system. Similarly, for Message Queue, Thread 3 is in running mode, whose execution time differs for both MBED OS and FreeRTOS, where still MBED OS performs faster. The corresponding plot for performance analysis on MBED OS and FreeRTOS is depicted on figure 44.

Table 8: Execution time of Threads under different CMSIS RTOS 2 functions

Functions	**Threads**	**Execution time on MBED OS**	**Execution time on FreeRTOS**
Semaphore	Thread 1	2.583 us	2.583 us
	Thread 2	1.167 us	2.417 us
	Thread 3	43.500 ms	56.217 ms
	Thread 4	1.667 us	1.667 us
	Thread 5	1.778 sec	2.03 sec
Total execution time (in sec)		1.821 sec	2.086 sec
Mutex	Thread 1	2.583 us	2.583 us
	Thread 2	1.167 us	2.417 us
	Thread 3	46.311 ms	51.837 ms
	Thread 4	1.667 us	1.667 us
	Thread 5	1.620 sec	2.726 sec
Total execution time (in sec)		1.666 sec	2.777 sec
Message Queue	Thread 1	3.917 us	3.917 us
	Thread 2	3.167 us	3.167 us
	Thread 3	1.360 sec	1.476 sec
	Thread 4	0 us	0 us
	Thread 5	0 us	0 us
Total execution time (in sec)		1.36 sec	1.476 sec

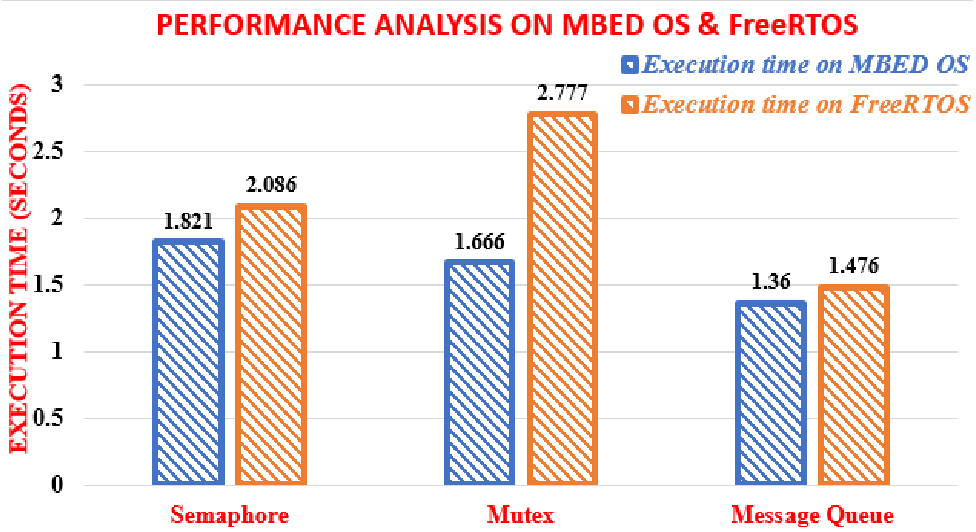


Figure 44: Performance Analysis on MBED OS and FreeRTOS

CONCLUSIONS AND SUGGESTIONS

There are various conclusions that can be laid down based on analysis and testing results conducted on both ARM Cortex M4 and M33 processors for investigating suitable Operating System and security solution needed for world wide deployment of NB-IoT, which are as follows:

1) There are different features and properties for NB-IoT, however the main key-note where IoT market demands NB-IoT to be preferable solution on cellular domain is low power consumption, where it has long battery life (~10 years of battery life, which is never seen on other Cellular IoT system). Although other Cellular IoT systems discussed earlier offers low cost, there is reason for NB-IoT to be cost effective because other systems lag between features or services and cost (i.e. Best performance, but high cost). On the other hand, NB-IoT contributes emerging features and services with cost effective, thereby shaping the business model and fulfilling the needs of LPWA technology.
2) Traditional cellular modem designs are built for high throughput and as such cannot simply 'shrink' to meet the new standards, instead they require a redesign and a new approach. Thus, NB-IoT have significantly lower signal processing needs and in turn allow the selection of low power optimized architectures.
3) ARM Cortex-M family of processors is extremely well positioned to enable the emerging family of low power, always connected IoT devices. The ability to implement the protocol software as well as many elements of the lower level DSP code brings considerable benefits including cost reduction and power saving. In addition, this approach offers the benefit of a unified architecture for development and debug. This forces ARM Cortex M33 to be preferable architecture, which is configurable of all Cortex -M processors and has greater flexibility. It is to note that silicon for M33 core is still in development, which is building from ARM Cortex M4, but still the peripherals and other structures are accessible from Cortex M4.
4) Next, the operating system plays a major role in NB-IoT system, which is main section of this project. The reason why MBED OS is chosen is clearly explained in section 1.1.2, where the main point that forces MBED OS is open source, which again satisfies the vision of NB-IoT and it provides greater affordability for Cortex-M33 processor. This MBED OS was compared with another open source OS [FreeRTOS] to investigate their functionalities. The key parameter which decides OS is kernel structure, which acts as heart of operating System. Considering Keil RTX-5 kernel (for MBED OS) and FreeRTOS kernel (for FreeRTOS), RTX-5 dominates FreeRTOS kernel, where it has flexible scheduling. This scheduler can be configured in three different ways – Preemptive [each task has different priority and runs on high priority basis], Round-Robin [each task runs for fixed CPU run time] and Co-operative [each task runs till it told to pass control to next task]. On the other hand, FreeRTOS supports

only pre-emptive scheduling, where it fails to handle multitasking operations and the kernel takes more time to execute, which is shown via performance Analysis on section 3.7.

5) Although FreeRTOS kernel has pre-emptive scheduler for doing tasks on priority basis, it does not implement latest mechanism like priority ceilings to avoid priority inversion, which means a high priority task is indirectly pre-empted by a lower priority task effectively "inverting" the relative priorities of the two tasks. On contrast MBED OS has feature inbuilt on RTX-5 to avoid priority inversion, where it can be clearly seen from Thread management table on Appendix-A that even though thread is given high priority, it is pre-empted invisibly by low priority task and starts running, where RTX kernel prevents priority inversion from other idle threads and makes both low priority and high priority to perform simultaneously, thereby saving execution time and also RTX has two more schedulers to adopt it. But if FreeRTOS kernel is affected by priority inversion, the kernel does not know which task to be given high priority to run and the kernel will be automatically suspended.
6) It can be noted that programming model is nearly related to selected programming language, since it affects the implementation of operating system itself and determine which programming language to be used by developers when working with specific OS. Thus, MBED OS becomes superior, since it is developed by C++ programming language, supporting facilities for low-level memory manipulations, whereas FreeRTOS supports only basic C language leading to large memory usage.
7) There are many comparisons that were discussed between MBED OS and FreeRTOS in terms of memory as it plays main role in selecting stable operating system. Based on discussions, the main factor that affects the memory on FreeRTOS than on MBED OS is minimal stack size that handles only idle thread. Due to this unstable allocation of memory, the running tasks are consuming more stack size than defined (nearly three times greater). Also, it is to note that FreeRTOS allocate fixed memory to each thread or tasks, but do not deallocate it, thus the memory is wasted and can’t be reused.
8) ARM Cortex M33 accepts only ARMCLANG toolchain (ARM Compiler 6) for compilation, which is optimized. Thus, MBED OS on M33 works well with this compiler optimizations, consuming less code memory. Also, Cortex M4 initially had ARMCC toolchain (ARM Compiler 5), but I converted from ARMCC to ARMCLANG toolchain, where MBED OS on M4 also shows good optimizations with less code memory consumption. On the other hand, when I used ARMCLANG toolchain for both ARM Cortex M4 and M33 on FreeRTOS, it is not fully optimized, thereby consuming more code memory than MBED OS.

Thus, by considering all the above discussions and comparison between MBED OS and FreeRTOS, I conclude that MBED OS dominates FreeRTOS in all aspects and becomes preferred solution to be used along with TrustZone system for security solutions, since security is another important factor to be considered within NarrowBand IoT. Thus, this process is currently in development to be integrated with ARM CORDIO Radio core IP on CoreLink SSE 200 Subsystem complete NB-IoT Modem.

In addition, the future development is going on with the extension of CoreLink SSE 200 Subsystem to contribute more services and productivity towards Cellular IoT market, which is referred as Caldesi NB-IoT test chip, where it is planning to implement three ARM Cortex M33 processors – two for CoreLink SSE 200 Subsystem and another M33 processor for Baseband architecture.

INFORMATION SOURCE LIST

[1] ARM, “Whitepaper: NB – IoT: Connecting the IoT with ARM,” 2017.

[2] LinkLabs, “Blog: What is M2M?,” 10 November 2015. [Online]. Available: https://www.link-labs.com/blog/what-is-m2m.

[3] Chantal Polsonetti , Vice President, ARC Advisory Group, “Article: Know the Difference Between IoT and M2M,” 15 July 2014. [Online]. Available: https://www.automationworld.com/article/topics/cloud-computing/know-difference-between-iot-and-m2m.

[4] ROHDE & SCHWARZ, “Article: eMTC and NB-IoT pave the way to 5G/IoT,” [Online]. Available: https://www.rohde-schwarz.com/us/solutions/wireless-communications/lte/in-focus/emtc-and-nb-iot-pave-the-way-to-5g-iot_230416.html?rusprivacypolicy=0.

[5] Colin Walls, “Article: Selecting an operating system for an embedded application,” *embedded,* p. 1, 25 October 2014.

[6] ERIC BROWN, “Article: Open Source Operating Systems for IoT,” *Linux.com - "News for the Open Source Professional",* p. 1, 26 October 2016.

[7] ARROW, “Article: IoT Operating Systems,” 13 September 2016. [Online]. Available: https://www.arrow.com/en/research-and-events/articles/iot-operating-systems#.

[8] GSMA, “3GPP LOW POWER WIDE AREA TECHNOLOGIES,” 2016.

[9] Nandan Nayampally,VP Marketing and Strategy,ARM, “ARM Cortex-M23 and Cortex-M33 processors for billions of securely connected devices,” 7 December 2016. [Online]. Available: https://www.arm.com/files/event/2016_ATS_India_A4_Nandan_Nayampally.pdf.

[10] ARM , “Overview of Cortex-M Prototyping System +,” [Online]. Available: https://developer.arm.com/products/system-design/development-boards/cortex-m-prototyping-system.

[11] Mihail Stoyanov , “mbed Connect Asia 2016 Intro to mbed OS,” slideshare, 2016.

[12] ARM mbed, “Features and benefits of mbed OS,” [Online]. Available: https://www.mbed.com/en/platform/mbed-os/#features.

[13] Keil, “Introduction,” [Online]. Available: http://www.keil.com/pack/doc/CMSIS/General/html/index.html.

[14] Joseph Yiu, “Blog:How much stack memory do I need for my ARM Cortex-M applications,” 21 March 2016. [Online]. Available: https://community.arm.com/processors/b/blog/posts/how-much-stack-memory-do-i-need-for-my-arm-cortex--m-applications?pi353792392=2.

[15] ARM mbed, “Memory in mbed OS,” [Online]. Available: https://docs.mbed.com/docs/getting-started-mbed-os/en/latest/Full_Guide/memory/.

[16] ARM Limited, “CoreLink SSE-200 Subsystem,” [Online]. Available: https://www.arm.com/products/internet-of-things-solutions/corelink-sse-200-subsystem.php.

[17] cobham, “Blog: The NB-IoT standard — its benefits and test challenges,” 16 November 2016. [Online]. Available: http://cobhamwireless.com/resources/blog/nb-iot-standard-benefits-test-challenges/.

[18] eleven-x, “Whitepaper: The Need for a Dedicated Cellular IOT Solution,” 2015.

[19] Tarinder Sandhu, “Article: ARM looks to supercharge Internet of Things with mbed OS,” 1 October 2014. [Online]. Available: http://hexus.net/tech/news/software/75357-arm-looks-supercharge-internet-things-mbed-os/.

[20] Tuomas Tirronen, “Research-blog: Cellular IoT alphabet soup,” 29 Febuary 2016. [Online]. Available: https://www.ericsson.com/research-blog/internet-of-things/cellular-iot-alphabet-soup/.

[21] Yitaek Hwang, “Cellular IoT Explained – NB-IoT vs. LTE-M vs. 5G and More,” *iot-for-all,* p. 1, 30 December 2016.

[22] Aleksandar Milinković, Stevan Milinković, Ljubomir Lazić, “Choosing the right RTOS for IoT platform,” *INFOTEH-JAHORINA,* vol. 14, pp. 1-6, 2015.

[23] “CONTIKI OS - THE OPEN SOURCE OPERATING SYSTEM,” [Online]. Available: https://komalbattula.wordpress.com/architecture/.

[24] RIOT - The friendly Operating System for the Internet of Things, “RIOT Documentation,” [Online]. Available: http://riot-os.org/api/.

[25] FreeRTOS, [Online]. Available: http://www.freertos.org/.

[26] Thang Vu Chien,Hung Nguyen Chan, “A Comparative Study on Operating System for Wireless Sensor Networks,” in *International Conference on Advanced Computer Science and Information Systems,* Vietnam, 2011.

[27] Mahdi Amiri Kordestani,Hadj Bourdoucen, “A Survey On Embedded Open Source System Software For The Internet Of Things,” in *Free And Open Source Software Conference 2017 (FOSSC'17),* MUSCAT, 2017.

[28] Sheetal Kumbhar, “Blog: CAT-M1 vs NB-IoT – examining the real differences,” *IoT Now - How to run an IoT enabled business,* p. 1, 21 June 2016.

[29] QUALCOMM Technologies, “Whitepaper: Paving the path to Narrowband 5G,” 2016.

[30] Michal Stala, CEO, Mistbase; Magnus Midholt, Co-Founder, Mistbase, “LTE Cat-M A Cellular Standard for IoT,” ARM Limited, 2016.

[31] ARM mbed, “Introduction to the mbed OS 5 Handbook,” 2016. [Online]. Available: https://docs.mbed.com/docs/mbed-os-handbook/en/5.4/.

[32] William Albano, “Blog: A Chat with David Pan on IoT Device Development with ARM mbed,” *HWTrek - the collaborative platform for hardware innovation,* p. 1, 7 February 2017.

[33] Sam Grove, Principal software engineer, “mbed OS Technical Overview,” slideshare, Las Vegas, 2016.

[34] Keil, “CMSIS-RTOS2 Documentation,” [Online]. Available: http://www.keil.com/pack/doc/CMSIS/RTOS2/html/index.html.

[35] Keil, “RTX v5 Implementation,” [Online]. Available: http://www.keil.com/pack/doc/CMSIS/RTOS2/html/rtx5_impl.html.

[36] Nicolas Melot, “Study of an operating system: FreeRTOS”.

APPENDIXES

Appendix 1. Tested Data Results for Memory Management between MBED OS and FreeRTOS

Initial Testing on Cortex M4

MBED OS without Thread:

Program Size:

Code	4808 Bytes
RO Data	496 Bytes
RW Data	176 Bytes
ZI-Data	9520 Bytes

FreeRTOS without Thread:

Program Size:

Code	5636 Bytes
RO Data	224 Bytes
RW Data	24 Bytes
ZI-Data	14880 Bytes

MBED OS with one Thread:

Program Size:

Code	5128 Bytes
RO Data	496 Bytes
RW Data	224 Bytes
ZI-Data	9496 Bytes

FreeRTOS with one Thread:

Program Size:

Code	6420 Bytes
RO Data	224 Bytes
RW Data	24 Bytes
ZI-Data	14880 Bytes

MBED OS with 1 Thread & 1 Semaphore

Program Size:

Code	5884 Bytes
RO Data	496 Bytes
RW Data	248 Bytes
ZI-Data	9520 Bytes

FreeRTOS with 1 Thread & 1 Semaphore:

Program Size:

Code	7200 Bytes
RO Data	224 Bytes
RW Data	24 Bytes
ZI-Data	14896 Bytes

MBED OS with 1 Thread & 1 Mutex:

Program Size:

Code	5832 Bytes
RO Data	528 Bytes
RW Data	248 Bytes
ZI-Data	9528 Bytes

FreeRTOS with 1 Thread & 1 Mutex:

Program Size:

Code	7060 Bytes
RO Data	256 Bytes
RW Data	24 Bytes
ZI-Data	14896 Bytes

Initial Testing on Cortex M33

MBED OS without Thread:

Program Size:

Code	5148 Bytes
RO Data	860 Bytes
RW Data	176 Bytes
ZI-Data	9520 Bytes

FreeRTOS without Thread:

Program Size:

Code	5668 Bytes
RO Data	588 Bytes
RW Data	24 Bytes
ZI-Data	14976 Bytes

MBED OS with one Thread:

Program Size:

Code	5468 Bytes
RO Data	860 Bytes
RW Data	224 Bytes
ZI-Data	9496 Bytes

FreeRTOS with one Thread:

Program Size:

Code	6452 Bytes
RO Data	588 Bytes
RW Data	24 Bytes
ZI-Data	14976 Bytes

MBED OS with 1 Thread & 1 Semaphore:

Program Size:

Code	6224 Bytes
RO Data	860 Bytes
RW Data	248 Bytes
ZI-Data	9520 Bytes

FreeRTOS with 1 Thread & 1 Semaphore:

Program Size:

Code	7236 Bytes
RO Data	588 Bytes
RW Data	24 Bytes
ZI-Data	14992 Bytes

MBED OS with 1 Thread & 1 Mutex:

Program Size:

Code	6172 Bytes
RO Data	892 Bytes
RW Data	248 Bytes
ZI-Data	9560 Bytes

FreeRTOS with 1 Thread & 1 Mutex:

Program Size:

Code	7108 Bytes
RO Data	620 Bytes
RW Data	24 Bytes
ZI-Data	14992 Bytes

Final Testing on Cortex M4

MBED OS with 5 Thread & 2 Semaphore:

Program Size:

Code	6716 Bytes
RO Data	732 Bytes
RW Data	248 Bytes
ZI-Data	11704 Bytes

FreeRTOS with 5 Thread & 2 Semaphore:

Program Size:

Code	8508 Bytes
RO Data	520 Bytes
RW Data	24 Bytes
ZI-Data	15008 Bytes

MBED OS using 5 Threads & 2 Mutex:

Program Size:

Code	6656 Bytes
RO Data	740 Bytes
RW Data	248 Bytes
ZI-Data	11728 Bytes

FreeRTOS using 5 Threads & 2 Mutex:

Program Size:

Code	8356 Bytes
RO Data	528 Bytes
RW Data	24 Bytes
ZI-Data	15008 Bytes

MBED OS using 5 Threads & 2 Message Queue:

Program Size:

Code	7868 Bytes
RO Data	676 Bytes
RW Data	272 Bytes
ZI-Data	11752 Bytes

FreeRTOS using 5 Threads & 2 Message Queue:

Program Size:

Code	8980 Bytes
RO Data	464 Bytes
RW Data	24 Bytes
ZI-Data	15024 Bytes

Final Testing on Cortex M33

MBED OS with 5 Thread & 2 Semaphore:

Program Size:

Code	7056 Bytes
RO Data	1096 Bytes
RW Data	248 Bytes
ZI-Data	11704 Bytes

FreeRTOS with 5 Thread & 2 Semaphore:

Program Size:

Code	8548 Bytes
RO Data	884 Bytes
RW Data	24 Bytes
ZI-Data	15008 Bytes

MBED OS using 5 Threads & 2 Mutex:

Program Size:

Code	6996 Bytes
RO Data	1104 Bytes
RW Data	248 Bytes
ZI-Data	11728 Bytes

FreeRTOS using 5 Threads & 2 Mutex:

Program Size:

Code	8412 Bytes
RO Data	892 Bytes
RW Data	24 Bytes
ZI-Data	15008 Bytes

MBED OS using 5 Threads & 2 Message Queue:

Program Size:

Code	8216 Bytes
RO Data	1040 Bytes
RW Data	272 Bytes
ZI-Data	11752 Bytes

FreeRTOS using 5 Threads & 2 Message Queue:

Program Size:

Code	9012 Bytes
RO Data	828 Bytes
RW Data	24 Bytes
ZI-Data	15024 Bytes

MBED OS TEST RESULTS USING FUNCTIONALITIES OF CMSIS RTOS-2

MBED OS ON M33 (KERNEL - RTX 5 (CMSIS API v2) [SEMAPHORE]

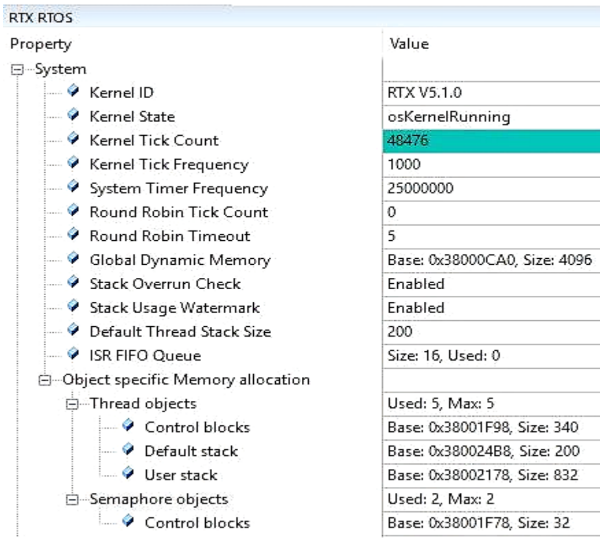
Initial Condition	Semaphore_1	Max_Count_Token - 4	Initial_Count_Token - 2
	Semaphore_2	Max_Count_Token - 5	Initial_Count_Token - 3

Table 1: Thread Management using Semaphore Function

Parameters	***Thread_1 (Use Semaphore_1)***	***Thread_2 (Use Semaphore_2)***	***Thread_3 (Use Semaphore_2)***	***Thread_4 (Use Semaphore_1)***	***Thread_5 (Use Semaphore_1)***
State	osThreadReady	osThreadReady	osThreadBlocked	osThreadBlocked	osThreadRunning
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x38000380	0x38001988	0x38001D20	0x38000400	0x380019F0
Stack Limit (address)	0x38000180	0x38001928	0x38001C58	0x38000380	0x38001990
Stack Available	*Stack (used 14% Max 14%)*	*Stack (used 75% Max 75%)*	*Stack (used 36% Max 36%)*	*Stack (used 56% Max 56%)*	*Stack (used 40% Max 75%)*
	Used:72	Used:72	Used:72	Used:72	Used:40
	Max:72	Max:72	Max:72	Max:72	Max:72
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

Running Condition	Semaphore_1	Available Tokens – 1	Max Tokens - 4
	Semaphore_2	Available Tokens – 3	Max Tokens - 5

RTX-5 Kernel Information



RTX RTOS

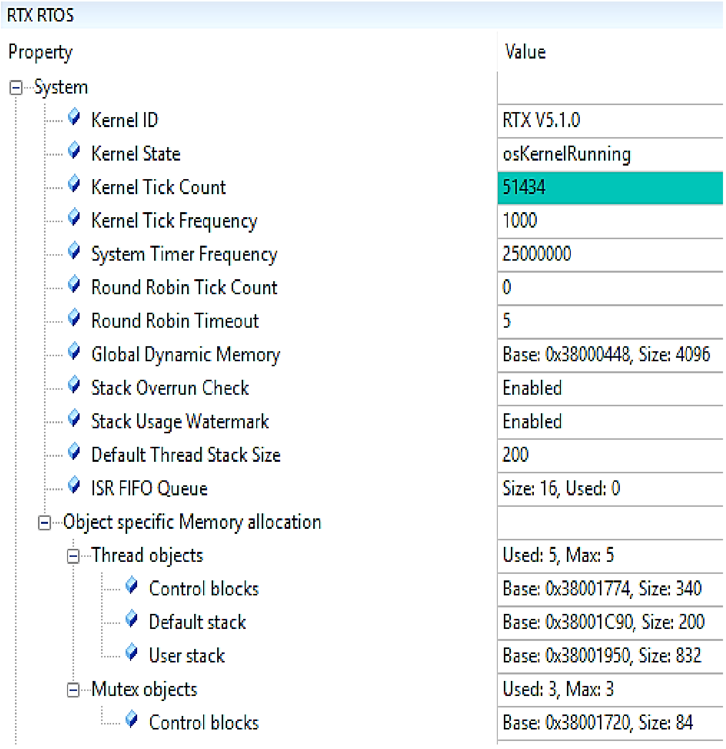
Property	Value
System	
Kernel ID	RTX V5.1.0
Kernel State	osKernelRunning
Kernel Tick Count	48476
Kernel Tick Frequency	1000
System Timer Frequency	25000000
Round Robin Tick Count	0
Round Robin Timeout	5
Global Dynamic Memory	Base: 0x38000CA0, Size: 4096
Stack Overrun Check	Enabled
Stack Usage Watermark	Enabled
Default Thread Stack Size	200
ISR FIFO Queue	Size: 16, Used: 0
Object specific Memory allocation	
Thread objects	Used: 5, Max: 5
Control blocks	Base: 0x38001F98, Size: 340
Default stack	Base: 0x380024B8, Size: 200
User stack	Base: 0x38002178, Size: 832
Semaphore objects	Used: 2, Max: 2
Control blocks	Base: 0x38001F78, Size: 32

Table 2: Thread Management using Mutex Function

Parameters	***Thread_1 (Use Mutex_1)***	***Thread_2 (Use Mutex_2)***	***Thread_3 (Use Mutex_2)***	***Thread_4 (Use Mutex_1)***	***Thread_5 (Use Mutex_1)***
State	osThreadReady	osThreadReady	osThreadBlocked	osThreadBlocked	osThreadRunning
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x38000388	0x380019C0	0x38001D58	0x38000408	0x38001A28
Stack Limit (address)	0x38000188	0x38001960	0x38001C90	0x38000388	0x380019C8
Stack Available	*Stack (used 14% Max 14%)*	*Stack (used 75% Max 75%)*	*Stack (used 36% Max 36%)*	*Stack (used 56% Max 56%)*	*Stack (used 8% Max 75%)*
	Used:72	Used:72	Used:72	Used:72	Used:8
	Max:72	Max:72	Max:72	Max:72	Max:72
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

States	**MUTEX 1**	**MUTEX 2**
OS Mutex Receive	True	True
OS Mutex PrioInherit	True	False
OS Mutex Robust	False	True
Owner Thread	Thread 5	Thread 3

RTX Kernel Information



RTX RTOS

Property	Value
⊟ System	
Kernel ID	RTX V5.1.0
Kernel State	osKernelRunning
Kernel Tick Count	51434
Kernel Tick Frequency	1000
System Timer Frequency	25000000
Round Robin Tick Count	0
Round Robin Timeout	5
Global Dynamic Memory	Base: 0x38000448, Size: 4096
Stack Overrun Check	Enabled
Stack Usage Watermark	Enabled
Default Thread Stack Size	200
ISR FIFO Queue	Size: 16, Used: 0
⊟ Object specific Memory allocation	
⊟ Thread objects	Used: 5, Max: 5
Control blocks	Base: 0x38001774, Size: 340
Default stack	Base: 0x38001C90, Size: 200
User stack	Base: 0x38001950, Size: 832
⊟ Mutex objects	Used: 3, Max: 3
Control blocks	Base: 0x38001720, Size: 84

MBED OS ON M33 (KERNEL - RTX 5 (CMSIS API v2) [MESSAGE QUEUE]

Table 3: Thread Management using Message Queue Function

Parameters	***Thread_1 (Send Msg_Q_1)***	***Thread_2 (Receive Msg_Q_1)***	***Thread_3 (Send Msg_Q_2&3)***	***Thread_4 (Receive Msg_Q_2)***	***Thread_5 (Receive Msg_Q_3)***
State	osThreadReady	osThreadReady	osThreadRunning	osThreadReady	osThreadReady
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x38000398	0x380019D0	0x38001D68	0x38000418	0x38001A38
Stack Limit (address)	0x38000198	0x38001970	0x38001CA0	0x38000398	0x380019D8
Stack Available	*Stack (used 17% Max 17%)*	*Stack (used 100% Max 100%)*	*Stack (used 20 % Max 32%)*	*Stack (used 50% Max 50%)*	*Stack (used 66% Max 66%)*
	Used:88 Bytes	Used:96 Bytes	Used:40 Bytes	Used:64 Bytes	Used:64 Bytes
	Max:88 Bytes	Max:96 Bytes	Max:64 Bytes	Max:64 Bytes	Max:64 Bytes
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

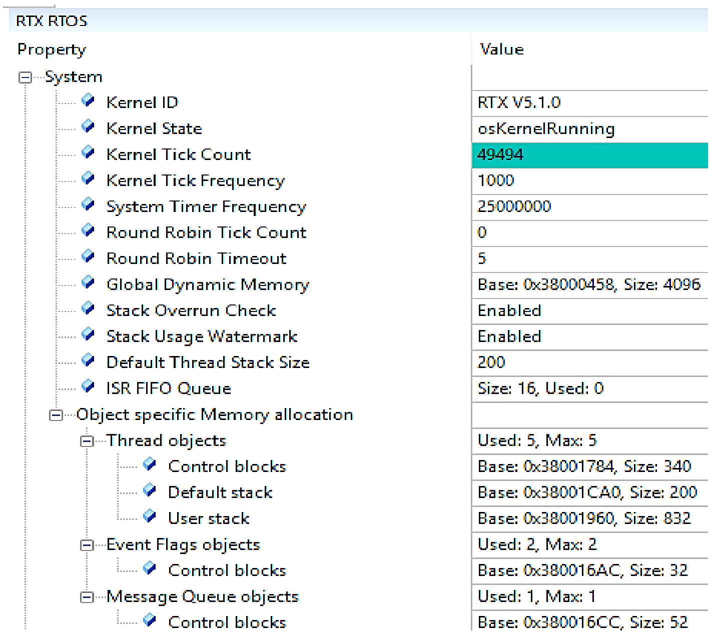
Event Flag – Flag Received = 0x20000002

Message Queue – Available Msg = 16, Maximum Msg = 16, Message Size = 36 Bytes

Msg Queue [15]	**Address**	**Priority**
Queue [0]	0x38000574	5
Queue [1]	0x380005A4	5
Queue [2]	0x380005D4	5
Queue [3]	0x38000604	5
Queue [4]	0x38000634	5
Queue [5]	0x38000664	5
Queue [6]	0x38000694	5
Queue [7]	0x380006C4	5
Queue [8]	0x380006F4	5
Queue [9]	0x38000724	5
Queue [10]	0x38000454	5
Queue [11]	0x38000484	5
Queue [12]	0x380004B4	5
Queue [13]	0x380004E4	5
Queue [14]	0x38000514	5
Queue [15]	0x38000544	5

Thread waiting -Thread 1 Timeout = 10ms

RTX Kernel Information



RTX RTOS

Property	Value
System	
Kernel ID	RTX V5.1.0
Kernel State	osKernelRunning
Kernel Tick Count	49494
Kernel Tick Frequency	1000
System Timer Frequency	25000000
Round Robin Tick Count	0
Round Robin Timeout	5
Global Dynamic Memory	Base: 0x38000458, Size: 4096
Stack Overrun Check	Enabled
Stack Usage Watermark	Enabled
Default Thread Stack Size	200
ISR FIFO Queue	Size: 16, Used: 0
Object specific Memory allocation	
Thread objects	Used: 5, Max: 5
Control blocks	Base: 0x38001784, Size: 340
Default stack	Base: 0x38001CA0, Size: 200
User stack	Base: 0x38001960, Size: 832
Event Flags objects	Used: 2, Max: 2
Control blocks	Base: 0x380016AC, Size: 32
Message Queue objects	Used: 1, Max: 1
Control blocks	Base: 0x380016CC, Size: 52

MBED OS ON M4 (KERNEL - RTX 5 (CMSIS API v2) [SEMAPHORE]

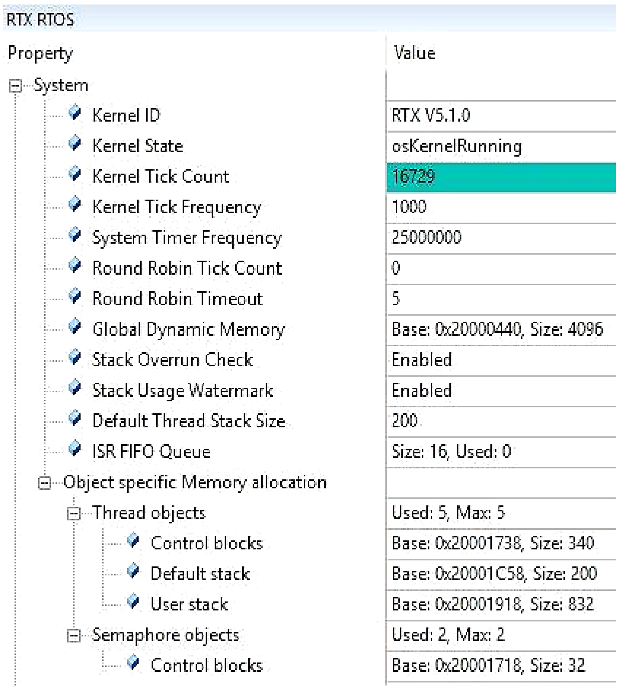
Initial Condition	Semaphore_1	Max_Count_Token - 4	Initial_Count_Token - 2
	Semaphore_2	Max_Count_Token - 5	Initial_Count_Token - 3

Table 1: Thread Management using Semaphore Function

Parameters	***Thread_1 (Use Semaphore_1)***	***Thread_2 (Use Semaphore_2)***	***Thread_3 (Use Semaphore_2)***	***Thread_4 (Use Semaphore_1)***	***Thread_5 (Use Semaphore_1)***
State	osThreadReady	osThreadReady	osThreadBlocked	osThreadBlocked	osThreadRunning
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x20000380	0x20001988	0x20001D20	0x20000400	0x200019F0
Stack Limit (address)	0x20000180	0x20001928	0x20001C58	0x20000380	0x20001990
Stack Available	*Stack (used 14% Max 14%)*	*Stack (used 75% Max 75%)*	*Stack (used 36% Max 36%)*	*Stack (used 56% Max 56%)*	*Stack (used 41% Max 75%)*
	Used:72 Bytes	Used:72 Bytes	Used:72 Bytes	Used:72 Bytes	Used:40 Bytes
	Max:72 Bytes	Max:72 Bytes	Max:72 Bytes	Max:72 Bytes	Max:72 Bytes
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

Running Condition	Semaphore_1	Available Tokens - 2	Max Tokens - 4
	Semaphore_2	Available Tokens - 3	Max Tokens - 5

RTX Kernel Information



RTX RTOS

Property	Value
System	
Kernel ID	RTX V5.1.0
Kernel State	osKernelRunning
Kernel Tick Count	16729
Kernel Tick Frequency	1000
System Timer Frequency	25000000
Round Robin Tick Count	0
Round Robin Timeout	5
Global Dynamic Memory	Base: 0x20000440, Size: 4096
Stack Overrun Check	Enabled
Stack Usage Watermark	Enabled
Default Thread Stack Size	200
ISR FIFO Queue	Size: 16, Used: 0
Object specific Memory allocation	
Thread objects	Used: 5, Max: 5
Control blocks	Base: 0x20001738, Size: 340
Default stack	Base: 0x20001C58, Size: 200
User stack	Base: 0x20001918, Size: 832
Semaphore objects	Used: 2, Max: 2
Control blocks	Base: 0x20001718, Size: 32

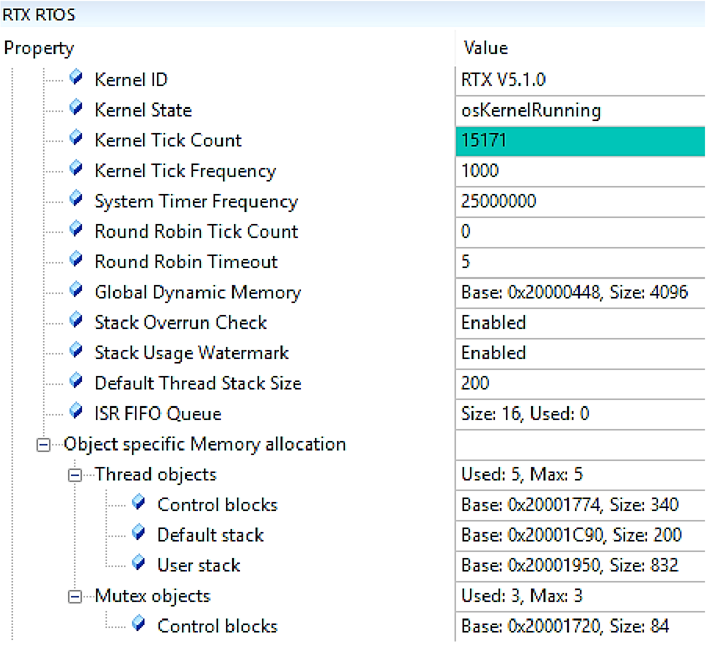
MBED OS ON M4 (KERNEL - RTX 5 (CMSIS API v2) [MUTEX]

Table 2: Thread Management using Mutex Function

Parameters	***Thread_1 (Use Mutex_1)***	***Thread_2 (Use Mutex_2)***	***Thread_3 (Use Mutex_2)***	***Thread_4 (Use Mutex_1)***	***Thread_5 (Use Mutex_1)***
State	osThreadReady	osThreadReady	osThreadBlocked	osThreadBlocked	osThreadRunning
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x20000388	0x200019C0	0x20001D58	0x20000408	0x20001A28
Stack Limit (address)	0x20000188	0x20001960	0x20001C90	0x20000388	0x200019C8
Stack Available	*Stack (used 14% Max 14%)*	*Stack (used 75% Max 75%)*	*Stack (used 36% Max 36%)*	*Stack (used 56% Max 56%)*	*Stack (used 8% Max 75%)*
	Used:72 Bytes	Used 72 Bytes	Used 72 Bytes	Used 72 Bytes	Used 8 Bytes
	Max:72 Bytes	Max:72 Bytes	Max:72 Bytes	Max:72 Bytes	Max:72 Bytes
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

States	**MUTEX 1**	**MUTEX 2**
OS Mutex Receive	True	True
OS Mutex PrioInherit	True	False
OS Mutex Robust	False	True
Owner Thread	Thread 5	Thread 3

RTX Kernel Information



RTX RTOS

Property	Value
Kernel ID	RTX V5.1.0
Kernel State	osKernelRunning
Kernel Tick Count	15171
Kernel Tick Frequency	1000
System Timer Frequency	25000000
Round Robin Tick Count	0
Round Robin Timeout	5
Global Dynamic Memory	Base: 0x20000448, Size: 4096
Stack Overrun Check	Enabled
Stack Usage Watermark	Enabled
Default Thread Stack Size	200
ISR FIFO Queue	Size: 16, Used: 0
Object specific Memory allocation	
Thread objects	Used: 5, Max: 5
Control blocks	Base: 0x20001774, Size: 340
Default stack	Base: 0x20001C90, Size: 200
User stack	Base: 0x20001950, Size: 832
Mutex objects	Used: 3, Max: 3
Control blocks	Base: 0x20001720, Size: 84

Table 3: Thread Management using Message Queue Function					
Parameters	***Thread_1 (Send Msg_Q_1)***	***Thread_2 (Receive Msg_Q_1)***	***Thread_3 (Send Msg_Q_2&3)***	***Thread_4 (Receive Msg_Q_2)***	***Thread_5 (Receive Msg_Q_3)***
State	osThreadReady	osThreadReady	osThreadRunning	osThreadReady	osThreadReady
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x20000398	0x200019D0	0x20001D68	0x20000418	0x20001A38
Stack Limit (address)	0x20000198	0x20001970	0x20001CA0	0x20000398	0x200019D8
Stack Available	*Stack (used 17% Max 17%)*	*Stack (used 100% Max 100%)*	*Stack (used 20% Max 32%)*	*Stack (used 50% Max 50%)*	*Stack (used 66% Max 66%)*
	Used:88 Bytes	Used:96 Bytes	Used:40 Bytes	Used:64 Bytes	Used:64 Bytes
	Max:88 Bytes	Max:96 Bytes	Max:64 Bytes	Max:64 Bytes	Max:64 Bytes
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

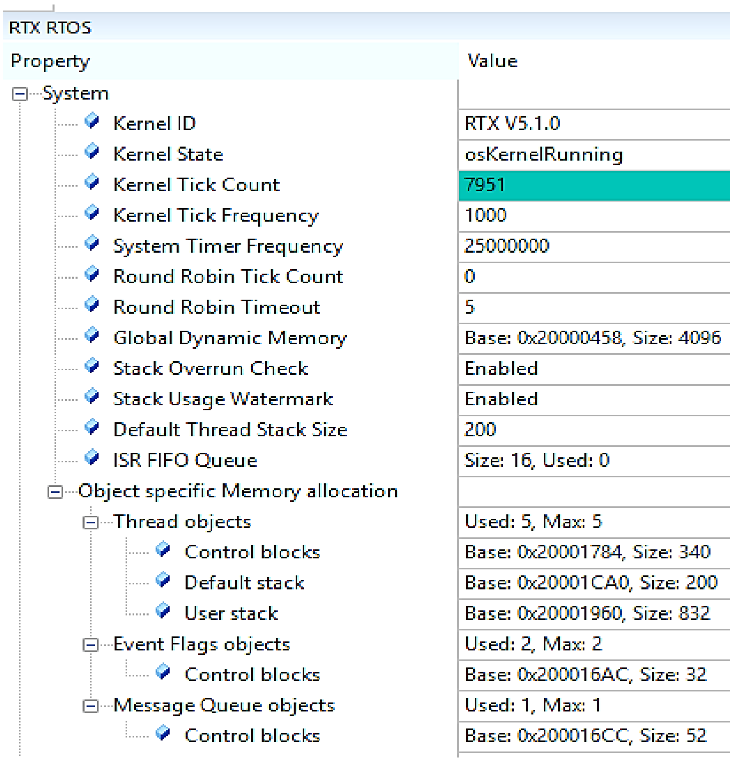
Event Flag – Flag Received = 0x20000002

Message Queue – Available Msg = 15 Maximum Msg = 16 Message Size = 36 Bytes

Msg Queue [15]	Address	Priority
Queue [0]	0x20000604	5
Queue [1]	0x20000634	5
Queue [2]	0x20000664	5
Queue [3]	0x20000694	5
Queue [4]	0x200006C4	5
Queue [5]	0x200006F4	5
Queue [6]	0x20000724	5
Queue [7]	0x20000454	5
Queue [8]	0x20000484	5
Queue [9]	0x200004B4	5
Queue [10]	0x200004E4	5
Queue [11]	0x20000514	5
Queue [12]	0x20000544	5
Queue [13]	0x20000574	5
Queue [14]	0x200005A4	5

Thread waiting - Thread 1 Timeout = 10ms

RTX Kernel Information



RTX RTOS

Property	Value
System	
Kernel ID	RTX V5.1.0
Kernel State	osKernelRunning
Kernel Tick Count	7951
Kernel Tick Frequency	1000
System Timer Frequency	25000000
Round Robin Tick Count	0
Round Robin Timeout	5
Global Dynamic Memory	Base: 0x20000458, Size: 4096
Stack Overrun Check	Enabled
Stack Usage Watermark	Enabled
Default Thread Stack Size	200
ISR FIFO Queue	Size: 16, Used: 0
Object specific Memory allocation	
Thread objects	Used: 5, Max: 5
Control blocks	Base: 0x20001784, Size: 340
Default stack	Base: 0x20001CA0, Size: 200
User stack	Base: 0x20001960, Size: 832
Event Flags objects	Used: 2, Max: 2
Control blocks	Base: 0x200016AC, Size: 32
Message Queue objects	Used: 1, Max: 1
Control blocks	Base: 0x200016CC, Size: 52

FREERTOS ON M33 (KERNEL - FreeRTOS) [SEMAPHORE]

Table 1: Thread Management using Semaphore Function					
Parameters	***Thread_1 (Use Sem_1)***	***Thread_2 (Use Sem_2)***	***Thread_3 (Use Sem_2)***	***Thread_4 (Use Sem_1)***	***(Thread_5 Use Sem_1)***
State	*Ready Task*	*Ready Task*	*Delayed Task*	*Suspended Task*	*Running Task*
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x3800130C	0x380014FC	0x3800176C	0x380019A4	0x38001BD4
Stack Limit (address)	0x38000B68	0x380013D8	0x380015C8	0x38001838	0x38001AA8
Stack Available	Used:1956 Bytes	Used:292 Bytes	Used:708 Bytes	Used:364 Bytes	Used:304 Bytes
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

FREERTOS ON M33 (KERNEL - FreeRTOS) [MUTEX]

Table 2: Thread Management using Mutex Function					
Parameters	***Thread_1 (Use Mut_1)***	***Thread_2 (Use Mut_2)***	***Thread_3 (Use Mut_2)***	***Thread_4 (Use Mut_1)***	***Thread_5 (Use Mut_1)***
State	*Ready Task*	*Ready Task*	*Delayed Task*	*Suspended Task*	*Running Task*
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x3800130C	0x380014FC	0x3800188C	0x38001AC4	0x38001CF4
Stack Limit (address)	0x38000B68	0x380013D8	0x380015C8	0x38001958	0x38001BC8
Stack Available	Used:1956 Bytes	Used:292 Bytes	Used:708 Bytes	Used:364 Bytes	Used:368 Bytes
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

FREERTOS ON M33 (KERNEL - FreeRTOS) [MESSAGE QUEUE]

Table 3: Thread Management using Message Queue Function

Parameters	***Thread_1 (Send Msg_Q_1)***	***Thread_2 (Receive Msg_Q_1)***	***Thread_3 (Send Msg_Q_2&3)***	***Thread_4 (Receive Msg_Q_2)***	***Thread_5 (Receive Msg_Q_3)***
State	*Ready Task*	*Ready Task*	*Running Task*	*Ready Task*	*Ready Task*
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x380014C4	0x380016AC	0x38001A5C	0x38001CD4	0x38001EC4
Stack Limit (address)	0x38000D30	0x380015A0	0x38001790	0x38001B20	0x38001D90
Stack Available	Used:1940 Bytes	Used:268 Bytes	Used:784 Bytes	Used:436 Bytes	Used:308 Bytes
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

FREERTOS ON M4 (KERNEL - FreeRTOS) [SEMAPHORE]

Table 1: Thread Management using Semaphore Function

Parameters	***Thread_1 (Use Sem_1)***	***Thread_2 (Use Sem_2)***	***Thread_3 (Use Sem_2)***	***Thread_4 (Use Sem_1)***	***Thread_5 Use Sem_1)***
State	*Ready Task*	*Ready Task*	*Delayed Task*	*Suspended Task*	*Running Task*
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x2000130C	0x200014FC	0x2000176C	0x200019A4	0x20001BD4
Stack Limit (address)	0x20000B68	0x200013D8	0x200015C8	0x20001838	0x20001AA8
Stack Available	Used:1956 Bytes	Used:292 Bytes	Used:708 Bytes	Used:364 Bytes	Used:304 Bytes
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

FREERTOS ON M4 (KERNEL - FreeRTOS) [MUTEX]

Table 2: Thread Management using Mutex Function

Parameters	***Thread_1 (Use Mut_1)***	***Thread_2 (Use Mut_2)***	***Thread_3 (Use Mut_2)***	***Thread_4 (Use Mut_1)***	***Thread_5 (Use Mut_1)***
State	*Ready Task*	*Ready Task*	*Delayed Task*	*Suspended Task*	*Running Task*
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x2000130C	0x200014FC	0x2000188C	0x20001AC4	0x20001CF4
Stack Limit (address)	0x20000B68	0x200013D8	0x200015C8	0x20001958	0x20001BC8
Stack Available	Used:1956 Bytes	Used:292 Bytes	Used:708 Bytes	Used:364 Bytes	Used:368 Bytes
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

FREERTOS ON M4 (KERNEL - FreeRTOS) [MESSAGE QUEUE]

Table 3: Thread Management using Message Queue Function

Parameters	***Thread_1 (Send Msg_Q_1)***	***Thread_2 (Receive Msg_Q_1)***	***Thread_3 (Send Msg_Q_2&3)***	***Thread_4 (Receive Msg_Q_2)***	***Thread_5 (Receive Msg_Q_3)***
State	*Ready Task*	*Ready Task*	*Running Task*	*Ready Task*	*Ready Task*
Priority	Low	Idle	High	Above Normal	Low2
Stack Top (address)	0x200014C4	0x200016AC	0x20001A5C	0x20001CD4	0x20001EC4
Stack Limit (address)	0x20000D30	0x200015A0	0x20001790	0x20001B20	0x20001D90
Stack Available	Used:1940 Bytes	Used:268 Bytes	Used:784 Bytes	Used:436 Bytes	Used:308 Bytes
Stack Size	512 Bytes	96 Bytes	200 Bytes	128 Bytes	96 Bytes

Appendix 2. Main programs on functionalities of CMSIS RTOS 2 needed for MBED OS and FreeRTOS

Using Semaphore Function

```
/*-------------MBEDOS and FreeRTOS on Cortex M4 and M33 using Semaphore--------------------*/

#include "mbed.h"                    // MBED OS
#include "FreeRTOS.h"                  // ARM.FreeRTOS::RTOS:Core
#include "cmsis_os2.h"

void Thread_Semaphore1 (void *argument);                // function prototype for thread_1
osThreadId_t tid_Thread_Semaphore1;                   // thread id_1
osSemaphoreId_t sid_Thread_Semaphore1;                  // semaphore id_1

void Thread_Semaphore2 (void *argument);                // function prototype for thread_2
osThreadId_t tid_Thread_Semaphore2;                   // thread id_2
osSemaphoreId_t sid_Thread_Semaphore2;                  // semaphore id_2

void Thread_Semaphore3 (void *argument);                // function prototype for thread_3
osThreadId_t tid_Thread_Semaphore3;                   // thread id_3

void Thread_Semaphore4 (void *argument);                // function prototype for thread_4
osThreadId_t tid_Thread_Semaphore4;                   // thread id_4

void Thread_Semaphore5 (void *argument);                // function prototype for thread_5
osThreadId_t tid_Thread_Semaphore5;                   // thread id_5

osStatus_t status;

/* ----------- THREAD_FUNCTION_1 --------------- */
void Thread_Semaphore1 (void *argument)                 // thread function_1
{
osThreadFlagsSet(tid_Thread_Semaphore1,0x00000004U);       // Sets the thread flags for a thread
specified by parameter thread_1
osThreadFlagsWait (0x00000006U, osFlagsWaitAny, 1);        // Wait forever until thread flag 1 is
set.

while (1) {
tid_Thread_Semaphore1 = osThreadGetId ();  // Obtain ID of current running thread
osThreadSetPriority (tid_Thread_Semaphore1, osPriorityLow); // Set thread priority

status = osSemaphoreAcquire (sid_Thread_Semaphore1, 10);    // wait 10 mSec to acquire a
Semaphore token
status = osSemaphoreRelease (sid_Thread_Semaphore1);       // Return a token back to a semaphore

status = osThreadYield ();                          // suspend thread
}
}

/* -------------- STACK SIZE FOR THREAD_1----------------- */
static uint64_t Thread_Semaphore1_stk_1[64];
```

```
const osThreadAttr_t Thread_Semaphore1_attr = {

  .stack_mem  = &Thread_Semaphore1_stk_1[0],
  .stack_size = sizeof(Thread_Semaphore1_stk_1)
};

/* ----------- THREAD_FUNCTION_2 --------------- */
void Thread_Semaphore2 (void *argument)                   // thread function_2
{
osThreadFlagsSet(tid_Thread_Semaphore2,0x00000001U);       // Sets the thread flags for a thread specified by parameter thread_2


while (1) {
tid_Thread_Semaphore2 = osThreadGetId ();  // Obtain ID of current running thread
osThreadSetPriority (tid_Thread_Semaphore2, osPriorityIdle); // Set thread priority

status = osSemaphoreAcquire (sid_Thread_Semaphore2, osWaitForever);    // Wait indefinitely to acquire a Semaphore token
status = osSemaphoreRelease (sid_Thread_Semaphore2);        // Return a token back to a semaphore

status = osThreadYield ();                               // suspend thread
}
}

/* -------------- STACK SIZE FOR THREAD_2----------------- */

const osThreadAttr_t Thread_Semaphore2_attr = {
  .stack_size = 96
};


/* ----------- THREAD_FUNCTION_3 --------------- */
void Thread_Semaphore3 (void *argument)                   // thread function_3
{
osThreadFlagsSet(tid_Thread_Semaphore3,0x00000003U);       // Sets the thread flags for a thread specified by parameter thread_3
osThreadFlagsWait (0x00000002U, osFlagsWaitAny,  osWaitForever);        // Wait forever until thread flag is set.

while (1) {
tid_Thread_Semaphore3 = osThreadGetId ();  // Obtain ID of current running thread
osThreadSetPriority (tid_Thread_Semaphore3, osPriorityHigh); // Set thread priority

osDelay(1); // Pass control to other tasks for 1ms
status = osSemaphoreAcquire (sid_Thread_Semaphore2, 2);    // wait 2 mSec to acquire a Semaphore token
status = osSemaphoreRelease (sid_Thread_Semaphore2);        // Return a token back to a semaphore

status = osThreadYield ();                               // suspend thread
} }
```

```
/* -------------- STACK SIZE FOR THREAD_3----------------- */

const osThreadAttr_t Thread_Semaphore3_attr = {
.stack_size = 0
};


/* ----------- THREAD_FUNCTION_4 --------------- */
void Thread_Semaphore4 (void *argument)                 // thread function_4
{
osThreadFlagsWait (0x00000005U, osFlagsWaitAll, 5);        // Wait for 5 timer ticks until thread
flags are set. Timeout afterwards.
osThreadFlagsWait (0x00000005U, osFlagsWaitAll | osFlagsNoClear, osWaitForever);   // Same as
the above, but the flags will not be cleared.
while (1) {
tid_Thread_Semaphore4 = osThreadGetId ();  // Obtain ID of current running thread
osThreadSetPriority (tid_Thread_Semaphore4, osPriorityAboveNormal); // Set thread priority

status = osSemaphoreAcquire (sid_Thread_Semaphore1, osWaitForever);    // wait indefinitely to
acquire a Semaphore token
status = osSemaphoreRelease (sid_Thread_Semaphore1);       // Return a token back to a semaphore

status = osThreadYield ();                            // suspend thread
}
}

/* -------------- STACK SIZE FOR THREAD_4----------------- */
static uint32_t Thread_Semaphore4_stk_1[32];

const osThreadAttr_t Thread_Semaphore4_attr = {

  .stack_mem  = &Thread_Semaphore4_stk_1[0],
  .stack_size = sizeof(Thread_Semaphore4_stk_1)
};


/* ----------- THREAD_FUNCTION_5 --------------- */
void Thread_Semaphore5 (void *argument)                 // thread function_5
{
osThreadFlagsSet(tid_Thread_Semaphore5,0x00000007U);       // Sets the thread flags for a thread
specified by parameter thread_5

while (1) {
tid_Thread_Semaphore5 = osThreadGetId ();  // Obtain ID of current running thread
osThreadSetPriority (tid_Thread_Semaphore5, osPriorityLow2); // Set thread priority

status = osSemaphoreAcquire (sid_Thread_Semaphore1, 0);    // wait 0 mSec to acquire a
Semaphore token
status = osSemaphoreRelease (sid_Thread_Semaphore1);       // Return a token back to a semaphore

status = osThreadYield ();                            // suspend thread
```

```
}}
/* -------------- STACK SIZE FOR THREAD_5----------------- */

const osThreadAttr_t Thread_Semaphore5_attr = {
.stack_size = 96
};

/* ---------- SEMAPHORE STRUCTURE-------------- */
const osSemaphoreAttr_t Thread1_semaphore_attr = {
  "Semaphore1",                          // human readable semaphore name
  };

const osSemaphoreAttr_t Thread2_semaphore_attr = {
  "Semaphore2",                          // human readable semaphore name
  };

int main(void) {

// System Initialization
SystemCoreClockUpdate();

if(osKernelGetState() == osKernelInactive) {
status=osKernelInitialize();
}

sid_Thread_Semaphore1 = osSemaphoreNew(4, 2, &Thread1_semaphore_attr);
if (!sid_Thread_Semaphore1) {
; // Semaphore object not created, handle failure
}

sid_Thread_Semaphore2 = osSemaphoreNew(5, 3, &Thread2_semaphore_attr);
if (!sid_Thread_Semaphore2) {
; // Semaphore object not created, handle failure
}

tid_Thread_Semaphore1 = osThreadNew (Thread_Semaphore1, NULL,
&Thread_Semaphore1_attr);
if (!tid_Thread_Semaphore1) {
return(-1);
}

tid_Thread_Semaphore2 = osThreadNew (Thread_Semaphore2, NULL,
&Thread_Semaphore2_attr);
if (!tid_Thread_Semaphore2) {
return(-1);
}

tid_Thread_Semaphore3 = osThreadNew (Thread_Semaphore3, NULL,
&Thread_Semaphore3_attr);
if (!tid_Thread_Semaphore3) {
return(-1);
```

```
}

tid_Thread_Semaphore4 = osThreadNew (Thread_Semaphore4, NULL,
&Thread_Semaphore4_attr);
if (!tid_Thread_Semaphore4) {
return(-1);
}

tid_Thread_Semaphore5 = osThreadNew (Thread_Semaphore5, NULL,
&Thread_Semaphore5_attr);
if (!tid_Thread_Semaphore5) {
return(-1);
}

if (osKernelGetState() == osKernelReady) {
status=osKernelStart();                    // Start thread execution
}
for (;;) {}
}
```

Using Mutex Function

```
/* -------------MBED OS and FreeRTOS on Cortex M4 and M33 using Mutex--------------- */

 #include "mbed.h"                      // MBED OS
#include "FreeRTOS.h"                   // ARM.FreeRTOS::RTOS:Core
 #include "cmsis_os2.h"

void Thread_Mutex1 (void *argument);                  // thread function
osThreadId_t tid_Thread_Mutex1;                    // thread id
osMutexId_t mutex1_id;                          // Mutex id

void Thread_Mutex2 (void *argument);                  // thread function
osThreadId_t tid_Thread_Mutex2;                    // thread id
osMutexId_t mutex2_id;                          // Mutex id

void Thread_Mutex3 (void *argument);                  // thread function
osThreadId_t tid_Thread_Mutex3;                    // thread id


void Thread_Mutex4 (void *argument);                  // thread function
osThreadId_t tid_Thread_Mutex4;                    // thread id

void Thread_Mutex5 (void *argument);                  // thread function
osThreadId_t tid_Thread_Mutex5;                    // thread id

osStatus_t status;


void Thread_Mutex1 (void *argument)
{
```

```
osThreadFlagsSet(tid_Thread_Mutex1,0x00000004U);        // Sets the thread flags for a thread
specified by parameter thread_1
osThreadFlagsWait (0x00000006U, osFlagsWaitAny, 1);     // Wait forever until thread flag 1 is set.
while (1) {

tid_Thread_Mutex1 = osThreadGetId ();                  // Obtain ID of current running thread
osThreadSetPriority (tid_Thread_Mutex1, osPriorityLow); // Set thread priority
status = osMutexAcquire(mutex1_id, 10);                // Waits 10ms for a mutex object specified by
parameter mutex_id becomes available
status = osMutexRelease(mutex1_id);                  //  Releases a mutex specified by parameter
mutex_id.
status=osThreadYield ();                          //  Suspend thread
  }
}
/* -------------- STACK SIZE FOR THREAD_1----------------- */
static uint64_t Thread_Mutex1_stk_1[64];
const osThreadAttr_t Thread_Mutex1_attr = {

  .stack_mem  = &Thread_Mutex1_stk_1[0],
  .stack_size = sizeof(Thread_Mutex1_stk_1)
   };
/* ----------- Mutex-1 Structure --------------- */
const osMutexAttr_t Thread_Mutex1a_attr = {
  "myThreadMutex1",                        // human readable mutex name
  osMutexRecursive | osMutexPrioInherit,    // attr_bits
  NULL,                               // memory for control block
  NULL                                // size for control block
  };

/* ----------- THREAD_FUNCTION_2 --------------- */
void Thread_Mutex2 (void *argument)
{
osThreadFlagsSet(tid_Thread_Mutex2,0x00000001U);        // Sets the thread flags for a thread
specified by parameter thread_2
while (1) {

tid_Thread_Mutex2 = osThreadGetId ();                  // Obtain ID of current running thread
osThreadSetPriority (tid_Thread_Mutex2, osPriorityIdle); // Set thread priority
status  = osMutexAcquire(mutex2_id, osWaitForever);     // Waits until a mutex object specified by
parameter mutex_id becomes available
status = osMutexRelease(mutex2_id);                  //  Releases a mutex specified by parameter
mutex_id.
status=osThreadYield ();                          // suspend thread
}
}
/* -------------- STACK SIZE FOR THREAD_2----------------- */

const osThreadAttr_t Thread_Mutex2_attr = {
  .stack_size = 96
};
/* ----------- Mutex-2 Structure --------------- */
```

```
const osMutexAttr_t Thread_Mutex2a_attr = {
  "myThreadMutex2",                         // human readable mutex name
  osMutexPrioInherit & osMutexRobust,    // attr_bits
  NULL,                                 // memory for control block
  NULL                                  // size for control block
  };

/* ----------- THREAD_FUNCTION_3 --------------- */
void Thread_Mutex3 (void *argument)                  // thread function_3
{
osThreadFlagsSet(tid_Thread_Mutex3,0x00000003U);        // Sets the thread flags for a thread
specified by parameter thread_3
osThreadFlagsWait (0x00000002U, osFlagsWaitAny,  osWaitForever);  // Wait forever until thread
flag is set.

while (1) {
tid_Thread_Mutex3 = osThreadGetId ();  // Obtain ID of current running thread
osThreadSetPriority (tid_Thread_Mutex3, osPriorityHigh); // Set thread priority

osDelay(1); // Pass control to other tasks for 1ms
status  = osMutexAcquire(mutex2_id, 2);               // Waits 2ms for a mutex object specified by
parameter mutex_id becomes available
status = osMutexRelease(mutex2_id);                  //  Releases a mutex specified by parameter
mutex_id.
status = osThreadYield ();                          // suspend thread
}
}

/* -------------- STACK SIZE FOR THREAD_3----------------- */

const osThreadAttr_t Thread_Mutex3_attr = {
.stack_size = 0
};


/* ----------- THREAD_FUNCTION_4 --------------- */
void Thread_Mutex4 (void *argument)                  // thread function_4
{
osThreadFlagsWait (0x00000005U, osFlagsWaitAll, 5);        // Wait for 5 timer ticks until thread
flags are set. Timeout afterwards.
osThreadFlagsWait (0x00000005U, osFlagsWaitAll | osFlagsNoClear, osWaitForever);   // Same as
the above, but the flags will not be cleared.
while (1) {
tid_Thread_Mutex4 = osThreadGetId ();  // Obtain ID of current running thread
osThreadSetPriority (tid_Thread_Mutex4, osPriorityAboveNormal); // Set thread priority

status  = osMutexAcquire(mutex1_id, osWaitForever);               // Waits until a mutex object
specified by parameter mutex_id becomes available
status = osMutexRelease(mutex1_id);                  //  Releases a mutex specified by parameter
mutex_id.
status = osThreadYield ();                             // suspend thread
}
}

/* -------------- STACK SIZE FOR THREAD_4----------------- */
static uint32_t Thread_Mutex4_stk_1[32];

const osThreadAttr_t Thread_Mutex4_attr = {

  .stack_mem  = &Thread_Mutex4_stk_1[0],
  .stack_size = sizeof(Thread_Mutex4_stk_1)
};

/* ----------- THREAD_FUNCTION_5 --------------- */
void Thread_Mutex5 (void *argument)                  // thread function_5
{
osThreadFlagsSet(tid_Thread_Mutex5,0x00000007U);       // Sets the thread flags for a thread
specified by parameter thread_5

while (1) {
tid_Thread_Mutex5 = osThreadGetId ();  // Obtain ID of current running thread
osThreadSetPriority (tid_Thread_Mutex5, osPriorityLow2); // Set thread priority

status  = osMutexAcquire(mutex1_id, 0);               // Don't wait for a mutex object specified by
parameter mutex_id becomes available
status = osMutexRelease(mutex1_id);                 //  Releases a mutex specified by parameter
mutex_id.
status = osThreadYield ();                           // suspend thread
}
}

/* -------------- STACK SIZE FOR THREAD_5----------------- */


const osThreadAttr_t Thread_Mutex5_attr = {
.stack_size = 96
};

int main(void) {

// System Initialization
SystemCoreClockUpdate();

if(osKernelGetState() == osKernelInactive) {
status=osKernelInitialize();
}

mutex1_id = osMutexNew(&Thread_Mutex1a_attr);
if (mutex1_id != NULL)  {
// Mutex object created
}
mutex2_id = osMutexNew(&Thread_Mutex2a_attr);
```

```
if (mutex2_id != NULL)  {
// Mutex object created
}

tid_Thread_Mutex1 = osThreadNew (Thread_Mutex1, NULL, &Thread_Mutex1_attr);
if (!tid_Thread_Mutex1) {
return(-1);
}
tid_Thread_Mutex2 = osThreadNew (Thread_Mutex2, NULL, &Thread_Mutex2_attr);
if (!tid_Thread_Mutex2) {
return(-1);
}
tid_Thread_Mutex3 = osThreadNew (Thread_Mutex3, NULL, &Thread_Mutex3_attr);
if (!tid_Thread_Mutex3) {
return(-1);
}
tid_Thread_Mutex4 = osThreadNew (Thread_Mutex4, NULL, &Thread_Mutex4_attr);
if (!tid_Thread_Mutex4) {
return(-1);
}
tid_Thread_Mutex5 = osThreadNew (Thread_Mutex5, NULL, &Thread_Mutex5_attr);
if (!tid_Thread_Mutex5) {
return(-1);
}
if (osKernelGetState() == osKernelReady) {
status=osKernelStart();                    // Start thread execution
}
for (;;) {}
}
```

Using Message Queue Function

```
/*-------- MBEDOS and FreeRTOS on Cortex M4 and M33 using Message Queue----------- */

 #include "mbed.h"                        // MBED OS
#include "FreeRTOS.h"                    // ARM.FreeRTOS::RTOS:Core
 #include "cmsis_os2.h"                  // CMSIS RTOS header file
 /*----------------------------------------------------------------------------
 *     Message Queue creation & usage
 *---------------------------------------------------------------------------*/

void Thread_MsgQueue1 (void *argument);                    // thread function 1
void Thread_MsgQueue2 (void *argument);                    // thread function 2
osThreadId_t tid_Thread_MsgQueue1;                       // thread id 1
osThreadId_t tid_Thread_MsgQueue2;                       // thread id 2
osThreadId_t tid_Thread_MsgQueue3;                       // thread id 3
osThreadId_t tid_Thread_MsgQueue4;                       // thread id 4
osThreadId_t tid_Thread_MsgQueue5;                       // thread id 5

#define MSGQUEUE_OBJECTS      16                         // number of Message Queue Objects
```

```
osStatus_t status;

osEventFlagsId_t evt1_id;                       // event flag id1
#define FLAGS_MSK1 0x00000002ul

osEventFlagsId_t evt2_id;                       // event flag id2
#define FLAGS_MSK2 0x00000005ul

typedef struct {                              // object data type
        uint8_t Buf[32];
  uint8_t Idx;

} MSGQUEUE_OBJ_t;

uint32_t flags;
osMessageQueueId_t mid_MsgQueue;                       // message queue id

/* ----------- THREAD_FUNCTION_1 --------------- */
void Thread_MsgQueue1 (void *argument) {

  MSGQUEUE_OBJ_t *pMsg = 0;

        osThreadFlagsSet(tid_Thread_MsgQueue1,0x00000003U);     // Sets the thread flags for a
thread specified by parameter thread_1
  osThreadFlagsWait (0x00000002U, osFlagsWaitAny,  osWaitForever);      // Wait forever until
thread flag is set.

  while (1) {
                tid_Thread_MsgQueue1 = osThreadGetId ();  // Obtain ID of current running thread
    osThreadSetPriority (tid_Thread_MsgQueue1, osPriorityLow); // Set thread priority


                pMsg->Buf[0] = 0x65;                            // do some work...
    pMsg->Idx    = 10;


                osMessageQueuePut (mid_MsgQueue, &pMsg, 5, 10); //puts a message into the the
message queue specified by parameter mq_id
                osThreadYield ();                            // suspend thread
  }
}

/* -------------- STACK SIZE FOR THREAD_1----------------- */
static uint64_t Thread_queue1_stk_1[64];

const osThreadAttr_t Thread_queue1_attr = {

  .stack_mem  = &Thread_queue1_stk_1[0],
  .stack_size = sizeof(Thread_queue1_stk_1)
};
```

```
/* ----------- THREAD_FUNCTION_2 --------------- */

void Thread_MsgQueue2 (void *argument) {

            uint8_t msg_prio;

  MSGQUEUE_OBJ_t *pMsg = 0;

 osThreadFlagsSet(tid_Thread_MsgQueue2,0x00000007U);    // Sets the thread flags for a thread
specified by parameter thread_2

  while (1) {
                        msg_prio=8;
    tid_Thread_MsgQueue2 = osThreadGetId ();  // Obtain ID of current running thread
    osThreadSetPriority (tid_Thread_MsgQueue2, osPriorityIdle); // Set thread priority
    status = osMessageQueueGet (mid_MsgQueue, &pMsg, &msg_prio, 2);  // wait 2m Sec for
message
                        status=osThreadYield (); // suspend thread
                         }
}
/* -------------- STACK SIZE FOR THREAD_2----------------- */

const osThreadAttr_t Thread_queue2_attr = {

  .stack_size = 96
};
/* ----------- THREAD_FUNCTION_3 --------------- */
void Thread_MsgQueue3 (void *argument) {

  evt1_id = osEventFlagsNew(NULL);
            evt2_id = osEventFlagsNew(NULL);

  while (1) {
                        tid_Thread_MsgQueue3 = osThreadGetId ();  // Obtain ID of current running thread
    osThreadSetPriority (tid_Thread_MsgQueue3, osPriorityHigh); // Set thread priority

                        osEventFlagsSet(evt1_id, FLAGS_MSK1);
                        osEventFlagsSet(evt2_id, FLAGS_MSK2);
    osThreadYield ();                                       // suspend thread
  }
}

/* -------------- STACK SIZE FOR THREAD_3----------------- */

const osThreadAttr_t Thread_queue3_attr = {

  .stack_size = 0
};

/* ----------- THREAD_FUNCTION_4 --------------- */
```

```
void Thread_MsgQueue4 (void *argument) {

  while (1) {

    tid_Thread_MsgQueue4 = osThreadGetId ();  // Obtain ID of current running thread
    osThreadSetPriority (tid_Thread_MsgQueue4, osPriorityAboveNormal); // Set thread priority
 flags = osEventFlagsWait (evt1_id,FLAGS_MSK1,osFlagsWaitAll, osWaitForever);
                        status=osThreadYield (); // suspend thread
                        }}
/* -------------- STACK SIZE FOR THREAD_4----------------- */

static uint32_t Thread_queue4_stk_1[32];

const osThreadAttr_t Thread_queue4_attr = {

  .stack_mem  = &Thread_queue4_stk_1[0],
  .stack_size = sizeof(Thread_queue4_stk_1)
};
/* ----------- THREAD_FUNCTION_5 --------------- */
void Thread_MsgQueue5 (void *argument) {
while (1) {
tid_Thread_MsgQueue5 = osThreadGetId ();  // Obtain ID of current running thread
    osThreadSetPriority (tid_Thread_MsgQueue5, osPriorityLow2); // Set thread priority


     flags = osEventFlagsWait (evt2_id,FLAGS_MSK2,osFlagsWaitAny, 5);
                        status=osThreadYield (); // suspend thread
                        }
}
/* -------------- STACK SIZE FOR THREAD_5----------------- */
const osThreadAttr_t Thread_queue5_attr = {
  .stack_size = 96
};

int main(void) {
// System Initialization
SystemCoreClockUpdate();

if(osKernelGetState() == osKernelInactive) {
osKernelInitialize();
}

     mid_MsgQueue = osMessageQueueNew(MSGQUEUE_OBJECTS,
sizeof(MSGQUEUE_OBJ_t), NULL);
  if (!mid_MsgQueue) {
    ; // Message Queue object not created, handle failure
  }
tid_Thread_MsgQueue1 = osThreadNew (Thread_MsgQueue1, NULL, &Thread_queue1_attr);
  if (!tid_Thread_MsgQueue1) return(-1);

  tid_Thread_MsgQueue2 = osThreadNew (Thread_MsgQueue2, NULL, &Thread_queue2_attr);
  if (!tid_Thread_MsgQueue2) return(-1);

            tid_Thread_MsgQueue3 = osThreadNew (Thread_MsgQueue3, NULL,
&Thread_queue3_attr);
  if (!tid_Thread_MsgQueue3) return(-1);

            tid_Thread_MsgQueue4 = osThreadNew (Thread_MsgQueue4, NULL,
&Thread_queue4_attr);
  if (!tid_Thread_MsgQueue4) return(-1);

            tid_Thread_MsgQueue5 = osThreadNew (Thread_MsgQueue5, NULL,
&Thread_queue5_attr);

 if (!tid_Thread_MsgQueue5) return(-1);
if (osKernelGetState() == osKernelReady) {
  osKernelStart();    // Start thread execution
}
for (;;) {}
}
```