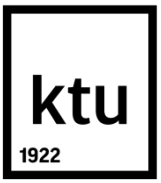

KAUNAS UNIVERSITY OF TECHNOLOGY

ELECTRICAL AND ELECTRONICS ENGINNERING FACULTY

SUBHASH PANDA SRINIVASAN

"INVESTIGATION OF MBED OPERATING SYSTEM WITH TRUSTZONE AS SECURITY SOLUTION FOR CELLULAR IOT"

Master's Degree Final Project

Supervisor

Prof. Dr. Zilvinas Nakutis

KAUNAS, 2017

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

"INVESTIGATION OF MBED OPERATING SYSTEM WITH TRUSTZONE AS SECURITY SOLUTION FOR CELLULAR IOT"

Master's Degree Final Project

Electronics Engineering (621H61002)

Supervisor

(signature) Prof. Dr. Zilvinas Nakutis

(date)

Reviewer

(signature) Assoc. Prof. Dr. Marius Saunoris

(date)

Project made by

(signature) Subhash Panda Srinivasan

(date)

KAUNAS, 2017

KAUNAS UNIVERSITY OF TECHNOLOGY

Electrical and Electronics Engineering Faculty
(Faculty)

Subhash Panda Srinivasan
(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, **Subhash Panda Srinivasan**, on the subject "**Investigation of MBED Operating System with Trustzone as Security Solution for Cellular IoT**" 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)*

Panda Srinivasan Subhash. Investigation of MBED Operating System with Trustzone as Security Solution for Cellular IoT. Final project of Electronics Engineering *Master's* degree / supervisor Prof. Dr. Zilvinas Nakutis. 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, Trustzone

Kaunas, 2017. 70 p.

SUMMARY

The master degree final project deals with Embedded Operating System blended with Security Solution for a Wireless Cellular IoT System. In this thesis, the new cellular standards like Narrow Band IoT and LTE Cat M which will replace the available GSM communication technology are analysed. In the next decade, there are more number of devices which are connected through cellular system. In this project, initially evaluate the MBED OS using MPS2+ prototyping board and measure the performance analysis like memory, heap, stack. Also perform MBED with TrustZone operations and system performance. On the security side trust zone is to be investigated to see if it is suitable to be used in cellular IoT system. The outcome would be to understand how a cellular IoT system would be affected in the relation to adding this trust zone features.

Panda Srinivasan Subhash. Saugiems Cellular IoT taikymams skirtos operacinės sistemos MBED su Trustzone posisteme tyrimas. Elektronikos inžinerijos in *Magistro* baigiamasis projektas / vadovas Prof. Dr Zilvinas Nakutis. 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, Trustzone

Kaunas, 2017. 70 p.

SANTRAUKA

Šis baigiamasis projektas magistro laipsniui apginti yra skirtas įterptinių operacijų sistemų ir saugos sprendimų bevielėms korinio ryšio daiktų internet sistemoms tyrimui. Tezėse nagrinėjami nauji standartai, tokie kaip siaurajuostis IoT (angl. Narrow Band IoT) ir LTE Cat M, kurie tikimasi pakeis šiuo metu naudojamus GSM komunikacijų standartus. Prognozuojama, kad aseinenti dešimtmetį vis daugės įrenginių prijungtų per korines sistemas. Šiame projekte, MBED operacijų sistemos našumo analizė, apimanti atminties, dinaminės atminties (angl. Heap), dėklo (angl. stack) dydziy priklausomybių nuo aparatinių ir programinių modulių panaudojimo, atlikta naudojant MPS2+ prototipavimo plokštę. Taip pat atlikti MBED ir TrustZone operacijų sistemų našumo. Saugos funkcijų realizavimo TrustZone posisteme yra analizuoja, siekiant nustatyti jos tinkamumą korinio ryšio daiktų interneto sistemų kūrimui. Tuo siekiama nustatyti, kaip korinio ryšio daiktų interneto sistema yra įtakojama TrustZone posistemės savybių.

CONTENT

INTRODUCTION 10
Research Objective 11
Thesis Structure 12
1 RELATED WORK REVIEWS 13
1.1 Cellular IoT 13
1.2 Survey of Embedded Operating Systems for the IoT Environment 14
1.2.1 Contiki OS 15
1.2.2 Apache Mynewt OS 15
1.2.3 RIOT OS 15
1.3 Internet of Things Security 16
1.3.1 Embedded Security for IoT 16
1.3.2 Security threats against IoT embedded devices and systems 16
1.3.3 IoT security impacts on mobile networks 16
1.3.4 Trusted computing blocks for embedded Linux-based ARM Trustzone Platforms 17
1.3.5 ARM TrustZone Devices in Restricted Spaces 17
1.4 Problem Analysis in Existing Methods 17
1.5 Proposed System 18
2 METHODOLOGIAL INVESTIGATION OF MBED OS LINKED WITH TRUSTZONE FOR SECURITY SOLUTIONS 19
2.1 Hardware Description – V2M MPS2+ board 20
2.2 Introduction to MBED OS 20
2.2.1 OS Platform requirements 21
2.2.2 MBED OS Core 22
2.3 Cortex Microcontroller Software Interface Standard (CMSIS) 23
2.3.1 Overview of CMSIS RTOS v2 24
2.4 Functions defined within CMSIS RTOS v2 25
2.4.1 CMSIS RTOS Thread 26
2.4.2 CMSIS RTOS Semaphore 26
2.4.3 CMSIS RTOS Mutex 27
2.5 Introduction to Trustzone 27
2.5.1 Programmer’s Model for ARM v8 M 28
2.5.2 Registers 29
2.5.3 Memory Map 30
2.5.4 RTOS Thread Context Management 32

2.5.5 Trustzone Security Requirements 32
2.6 Security for IoT Devices 33
3 RESULTS AND DISCUSSIONS 35
3.1 Comparison of ARM v7 M and ARM v8 M Architecture 35
3.1.1 Cortex M Processors 36
3.2 Memory Management on MBED Operating System 36
3.3 Simulation Results 38
3.3.1 Thread Analysis on Cortex M4 and Cortex M33 39
3.3.2 Experimental results on Cortex M4 and Cortex M33 42
3.3.3 Stack Management on Cortex M33 45
3.3.4 Performance Analysis 46
3.3.5 Trustzone Results 47
CONCLUSIONS AND SUGGESTIONS 53
INFORMATION SOURCE LIST 55
APPENDIXES 57
APPENDIX 1 Testing Results 57
APPENDIX 2 Main Programs 61

List of Figures

Figure 1 Billion global connections, 2015 – 2025 11
Figure 2 Trustzone without operating system 17
Figure 3 Embedded Operating System with TrustZone 18
Figure 4 MBED OS Stack 21
Figure 5 MBED OS platform for internet of things 22
Figure 6 MBED OS core layers 22
Figure 7 CMSIS Structure 23
Figure 8 CMSIS RTOS API Structure 25
Figure 9 Thread State and State Transition 26
Figure 10 CMSIS-RTOS Semaphore 27
Figure 11 CMSIS-RTOS Mutex 27
Figure 12 ARM Trustzone security concept 28
Figure 13 Secure Memory Map 28
Figure 14 Non-secure Memory Map 29
Figure 15 Register in ARM v8 M 29
Figure 16 Memory map model 30
Figure 17 RTOS Thread Context Management for ARMv8-M TrustZone 32
Figure 18 Trustzone Security address 33
Figure 19 Trustzone for IoT Devices 33
Figure 20 ARM v8 M structure 35
Figure 21 Comparison between Cortex M processors 36
Figure 22 Memory Organization in MBED OS 37
Figure 23 Thread Analysis on Cortex M4 39
Figure 24 Thread Analysis on Cortex M33 41
Figure 25 Experimental results on Cortex M4 43
Figure 26 Experimental results on Cortex M33 44
Figure 27 Stack consuming using Semaphore and Mutex Functions 45
Figure 28 Kernel Information of semaphore and mutex 46
Figure 29 Trustzone Secure and Non-Secure Functions 48
Figure 30 Thread Values 49
Figure 31 Security Transition States 49
Figure 32 Start of Non-secure mode 51
Figure 33 Non-secure to secure state 51
Figure 34 Secure to non-secure state 52

List of Tables

Table 1 Comparison of Cellular IoT Systems 14
Table 2 ARM v8 M Default Memory Map 31
Table 3 Memory Model of MBED OS 38
Table 4 Stack size of the threads 38
Table 5 Thread Execution timings 46
Table 6 Memory Management in Trustzone 47
Table 7 Registers Address Point Corresponding Regions 50

INTRODUCTION

The mobile ecosystem is the developing technology in Internet of Things field. The cellular systems are driven by the internet of things. The world leading organization CISCO that estimates nearly 30 billion devices should be connected by cellular IoT in 2025. The technologies used in cellular systems are 2G, 3G, and 4G for the internet of things. These are not specifically for IoT and also for Low Power Wide Area Network. The cellular IoT that provides many services, including utility meters, medical, machines, and automotive fields. Long Term Evolutions required enhanced services to enable the device in the network with more lifetime, improving coverage area, the large number of supported devices and low deployment cost. The network needs very simple and less number of devices to work. Rural area networks that required extended coverage in transmission and the higher layer protocols which help to consume less power by achieving long life over a decade to the device.

Today LTE supports internet of thing with so-called Cat 1 device, for utilizing coverage and massive devices should be connected to the technologies. The enhanced Machine Type Communication referred as NB-IoT and LTE-M. The data rates should be scalable for all the systems. The solutions can be increased in spectrum together with LTE. The bandwidth in GSM carrier as narrow for NB-IoT. Remaining networks are legacy cellular networks. The networks are updated with software to get a long life of the device, more coverage and low cost along with spectrum benefits. The internet of things device is interconnected to exchange data between them. The system will bring huge improvements in user experience and efficiency. The total IoT domain that estimates a large number of devices are connected to the network. The most usage connections are fixed and short-range communications. It will significantly for all connections expected through cellular IoT.

The ARM develops NB-IoT system by using Core-Link SSE 200 IoT subsystem. It is the fastest way to create secure IoT chips, which integrates the core components of your system. Security in embedded systems with all components should take a large amount of time and effort. But Core-Link SSE 200 IoT subsystem makes the processor become easier. It contains two core structures and features for IoT chips. Thus, it consists of two Cortex M33 ARM v8 processors and trust zone with crypto security for implementing a secure solution. The internet of things devices needs an operating system to perform the task scheduled in it. So, that it makes the system more efficient. The embedded system devices support a lot of operating systems, particularly internet of things devices demands high configured and exact platform operating systems.

The devices that are connected to the networks need to secure the internet of things systems. ARM trust zone technology is a system on chip and system-wide approach to security. Hardware-based security that provides secure end points and trusted device root. The secure and non-secure methods are done by Trustzone on a system on chip. Trust zone in cellular IoT comes with trust opportunity. It built into the number of connected devices in LPWAN, we ensure that the data is secure and scalable. This operating system and security in an IoT device make the system very effective. Thus, the NB IoT using core link SSE 200 subsystem along with ARM CORDIO Radio core IP obtains best cellular IoT network with security solutions in the wireless system. Core link SSE 200 subsystem is the developing system in an internet of things field. By comparing with other devices, it will become more professional and cost effective. Figure 1 shows billions of devices are connected through cellular IoT in the next decade. It describes the fixed and short-range communications will be increase gradually.

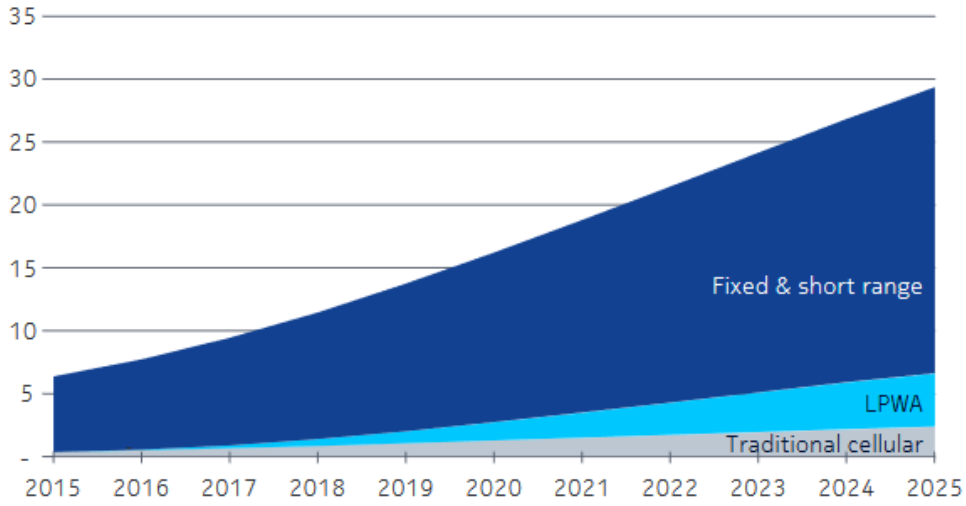

Figure 1 Billion global connections, 2015 – 2025 [1]

Research Objective

The main objectives of this thesis are,

A. To investigate/evaluate/research/study of MBED operating system.

B. To evaluate Trustzone security solution along with MBED OS for a wireless cellular system.

In this project, the wireless cellular IoT system considered as a target.

The tasks to be carried out for this thesis includes:

1. Investigating MBED OS as RTOS and to research the possibilities of enabling MBED OS on ARMv8 Cortex-M architecture, which is on the latest available processor like Cortex M33. Cortex M33 has properties for trust zone, where both MBED OS and trust zone will be implemented on same cortex M33. To estimate the properties like performance, memory consumption and stack usage will be investigated to get an understanding of new RTOS in cellular IoT system.

2. On the security side, trust zone will be implemented on ARM cortex M33 and analysed to see if it is suitable for cellular IoT system. Properties like hardware impact, software impact, performance on security related parts will be investigated.

The main goal of this thesis is to evaluate MBED OS with trust zone for security solution in the cellular IoT system.

Thesis Structure

This thesis structure contains five chapters, all the chapters will describe the project in detail.

To target the above-mentioned goal, the following chapters must be achieved.

Initially, an introduction of cellular it is discussed which includes the goal and the structure of the thesis. This gives an overview representation of embedded OS and trust zone technology.

Chapter 1 mainly discusses related work reviews which may include the existing system developed in the early days and the mythological investigation for the security solutions with standard IoT protocols application in cellular IoT.

Chapter 2 gives a brief description of MBED OS and the trust zone security solution which is including structure and replacement of current Cellular IoT system. In addition, this chapter describes Trustzone for a cellular network.

Chapter 3 discusses various methods, and testing of different operating system with trust zone. This chapter concludes the impact of trust zone on MBED operating system.

Finally, giving a brief note on the conclusion and suggestions for implementation of MBED OS linked with trust zone as security solution that satisfies the requirements of the Narrow Band IoT system.

1 RELATED WORK REVIEWS

In this literature survey, the basic process involved in cellular IoT and the trust zone security have been outlined. The basic ideas of embedded operating system and security solution for internet of things have been discussed. In conclusion of this section, the reasons for MBED OS and security solution of trust zone and comparison of their results have been discussed.

1.1 Cellular IoT

Popularity of IoT devices has guide to the rise of low-power wide-area network (LPWAN) options such as 6LowPAN and LoRa. Traditional cellular networks consume more energy. It can able to transmit small of data occasionally.

LTE Machine Type Communication (CAT 0, CAT 1, CAT M1)

From 3GPP Release 8, several categories of user equipment are available. The lowest performance is called CAT 1 which is LTE category, that means it gives both duplex methods. It is fully commercial and used in many internet of things deployments. In 3GPP Release 12, user equipment of CAT 0 is specified to reduce device complexity. It supports simplifications such as operating device with one receiver chain, allowing half duplex and minimize peak rate. This category is in limited commercial availability. In 3GPP Release 13, CAT M1 having three objectives. They are reducing complexity from CAT 0 with increased coverage area and improved battery life. The main thing of CAT M1 is the reduction in cost from CAT 0 with reduced bandwidth to six physical resource blocks and referred to bandwidth limited. Due to this limitation, a new control channel and frequency hopping mechanism were specified. However, legacy LTE signalling broadcast for system information is sent in six physical resource blocks. These channels did not need to be re broadcast for CAT M1 user equipment, thus it reduces signal overhead. CAT M1 allows an extended battery life of more than decade for wide range of communications. Even with complexity reduction, CAT M1 still provide many features to LTE, such as connected mode mobility and hand offs, frequency packets are in scheduled through semi persistent scheduling, and low latency packet while in connection. These are all the features for CAT M1 equipment to voice in internet of things application in coverage mode. Cat M1 targeting LPWA applications where small amount of data transfer is required. For example, smart metering that communicates with small amount of data. It simply needs to upload new software for the devices to operate within its LTE network.

NB-IoT/Cat-M2 NB-IoT is also known as CAT M2 which has similar goal to Cat-M, but it uses a different technology. NB-IoT is potentially less expensive which eliminates the extra gateways, so it is hyped. Main server connects sensor gateway for the communication purpose. This advantage makes the networks efficiency, so that the bigger companies like Qualcomm, Ericsson are trying to achieve commercialize NB-IoT [2].

EC-GSM it is an improved network, where the protocol in GSM network can be used by 80% of smart phones universally [2]. It can be positioned in available GSM networks. Leading network companies have completed trials of extended communication earlier, but still it doesn't generate as much buzz as NB-IoT

Table 1 Comparison of Cellular IoT Systems [1]

Parameters	**LTE Rel-8 Cat-1**	**LTE Rel-12 Cat-0**	**LTE Rel-13 Cat-M1**	***NB-IoT Rel-13***	**EC-GSM-IoT Rel-13**
DL peak rate	10 Mbps	1 Mbps	1 Mbps	~0.2 Mbps	~0.5 Mbps
UL peak rate	5 Mbps	1 Mbps	1 Mbps	~0.2 Mbps	~0.5 Mbps
Duplex mode	Full	Half or Full	Half or Full	Half	Half
UE bandwidth	20 MHz	20 MHz	1.4 MHz	0.18 MHz	0.2MHz
Max transmit power	23 dBm	23 dBm	20 or 23 dBm	23 dbm	23 or 33 dBm
Relative modem complexity	100%	50%	20-25%	10%	Not evaluated

1.2 Survey of Embedded Operating Systems for the IoT Environment

Internet of things environment can be performed based on the operating system contain in IoT devices. Therefore, the operating systems analyse each process that occurs in the systems. Operating systems helps to run the services to other applications in the system. The programs provide functionality that the user of the computer needs. The operating systems provide services to make application faster, easier and sustainable. Most operating systems performs multi-tasking. The scheduler in operating system that responsible for selecting program to run and execute by rapid switching between the processor and kernel. The scheduler in real time operating system is designed to provide excepted execution pattern. This assurance meet real time requirements can only be made in the scheduler of operating system behaviour [3]. There are many existing embedded operating systems are available. The operating system classifies and controls the hardware and it is that piece of software that turns hardware into computing tool. The main task of operating systems defines the functionality of process processor management which ensuring that each process and applications receives enough of the processor function time, using maximum processor cycles for work and switch between processes in multi threads. Memory management includes enough memory for each process to execute and use different types of memory in the system. Device management manages all hardware not on motherboard through driver programs. Driver provides applications for hardware without having to know details of hardware.

Application program interfaces uses functions of computer and operating system without having track of all details mainly CPU operation. Providing common user interface brings formal structure to the interaction between user and the computer. Real Time computing operating system should have features to support this critical requirement to reduce it. The RTOS should have certain behaviour to unpredictable events. A good RTOS should bounded under all system load scenario.

1.2.1 Contiki OS

Contiki is open source operating system that connects microcontroller to the internet of things. It makes application that should be efficient use of hardware platforms. Contiki is used in many systems. It has some standards for internet and developments. It supports both internet protocol fully along with wireless standards. It has rapid development in IDE simulator. The main features of Contiki OS are memory allocation for tiny systems, IP networking with standard IP protocols such as UDP, TCP and HTTP, module loading for loading and linking of modules at run time, protothreads to save memory and provide flow control and build systems makes it easy compile for any available Contiki platforms.

1.2.2 Apache Mynewt OS

Aache Mynewt is a real-time operating system that needs to perform for a long-time memory and other constrictions to the IoT devices. It gives a complete system for prototyping, managing and development. Microcontroller environments have number of characteristics that makes unique system. It has low memory footprint in the system range, reduced code size from 64-128KB to 16-32MB. Processing speeds of operating system is low and conserve maximum power usage. It will become more complex when more number of devices are connected. To perform many functions, it should have networking stacks, peripherals, and scheduled process. Benefits of using this operating system helps developers from other application code being written. It provides features to create complete operating system for controlled devices. Apache mynewt OS contains scheduler, time and tasks, semaphore, mutex, memory pools and heap.

1.2.3 RIOT OS

RIOT OS helps at bridging the gap we observed for sensor networks and traditional fledged running on host. It is based on objectives including efficiency, memory footprint, and API access, of hardware. RIOT implements kernel that supports multi-threading. The main features of RIOT make it robust against error in single components. It allows developers to create many threads and distributed systems can be implemented by kernel API. The amount of threads is only limited by memory and stack size for each thread. The requirements for real time process ensure RIOT constant periods for kernel tasks like scheduler run and timing operations. The runtimes of OS are exclusive use of static memory in kernel. RIOT switching context performs two cases [4].

From the above operating systems, they are all performed in all embedded devices without any security. The embedded devices for NB-IoT needs operating system with security solution. So the security and operating system will combine each other.

1.3 Internet of Things Security

A device that is connected to data communication networks needs security. Thus, the internet of things security gives safeguard to all connected devices. IoT involves objects provided with unique identifiers and ability to transfer data over a network.

1.3.1 Embedded Security for IoT

Internet of things imposes abnormal restrictions of computational power, connectivity, energy and number of devices which makes difference from authorized policy of security in distributed systems. To overcome the problem of security in internet domain that form ubiquity in IoT domain which are vulnerable to security attacks. In this work, embedded security required solution to resist different attacks and temper proofing of devices by concept of computing trusted platform. This issue addresses problems in hardware platform. Our work also partially helps in addressing securing data in transit [5].

1.3.2 Security threats against IoT embedded devices and systems

Security and privacy are two main challenges of the IoT, particularly due to the emerging threats embedded devices face due to their unique limitations in terms of connectivity, computational power and energy budget. Providing secure communications among M2M devices over cellular networks are an emerging research area, with divergent approaches being adopted. On one hand, efforts aim to secure the device itself and, on the other hand, network/provider-based architectures that benefit from the existing authentication methods of a cellular telecom operator are being proposed. In parallel, privacy is increasingly becoming one of the major concerns in these kinds of systems, especially given the surge of applications handling critical information. This is a particularly crucial area in certain IoT system categories, such as the case of network-enabled medical environments [6].

1.3.3 IoT security impacts on mobile networks

From the security and privacy of IoT devices, the deployment of M2M wireless system on mobile networks also having important security implication on the networks itself. The cellular network has big challenge to provide resource allocation for embedded devices in mobile infrastructures. Beyond the network operation challenge under such a large load of IoT traffic, M2M traffic is considered as one of the main factors within the overall LTE network security framework. Industry and standardization forums defining the main security threats and requirements for mobile network security are indeed, highlighting the IoT and its potential impact [6].

1.3.4 Trusted computing blocks for embedded Linux-based ARM Trustzone Platforms

Embedded security is an emerging topic in the field of mobile. Mobile trusted modules with trusted computing has outlined possible approach to mobile platform security. The TCG is a platform independent approach to trusted computing explicitly allowing for a wide range of implementations. Extending platforms to hardware with ARM support TrustZone security mechanism. ARM follows different approach to platform security to build embedded trusted computing platform [33].

1.3.5 ARM TrustZone Devices in Restricted Spaces

Some devices equipped with wide range of peripheral can potentially be misused in various environments. They can be used to get sensitive information from other sources. One way to prevent these situations to regulate smart devices in restricted spaces. ARM Trustzone in restricted space hosts use memory operations to analyse and regulate devices within the space. It shows that TrustZone to obtain strong security for small trusted computing base to execute on guest device [34].

1.4 Problem Analysis in Existing Methods

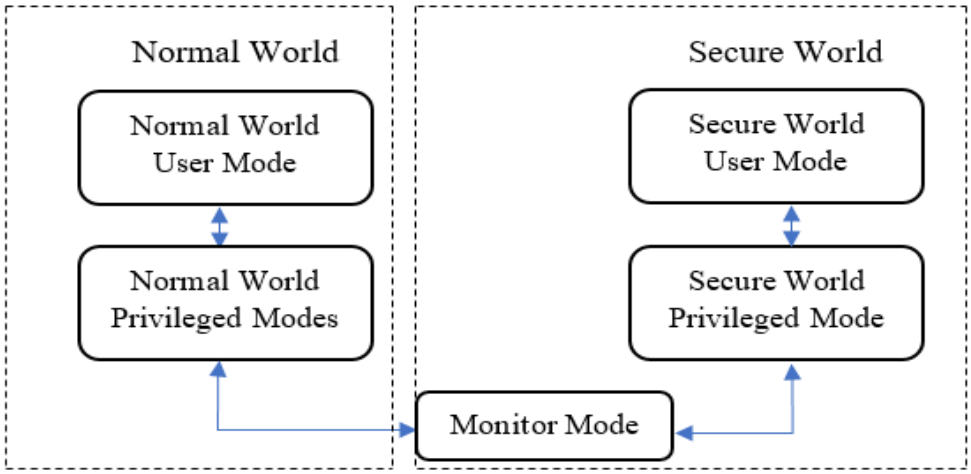

Figure 2 Trustzone without operating system

From the above survey, OS for IoT environment are well equipped with major networks and communication protocols, security features as well as optimized for efficient usage of computing power in constraint environment. The operating system does not have any secure information. So, embedded systems need security solution for Trustzone implementation. But in the embedded Linux based on arm completes simple security on operating system. In ARM-FreeRTOS of IoT platform they developed operating system between user and network without any security solutions. It explained communication between two nodes/paths. Security and privacy on embedded system explained about the emerging of connectivity and energy. This needs a crucial environment for Trustzone in cellular networks. IoT impact realize that connectivity between any nodes should have secure path to exchange data or communication. Overall cellular system requires better operating region in embedded system with secure zone.

To get a better network we need to find performance based systems and memory allocation of the embedded device. Trustzone is developed on hardware by ARM, that is built in system on chip semiconductor which gives secure end points on the trusted root.

1.5 Proposed System

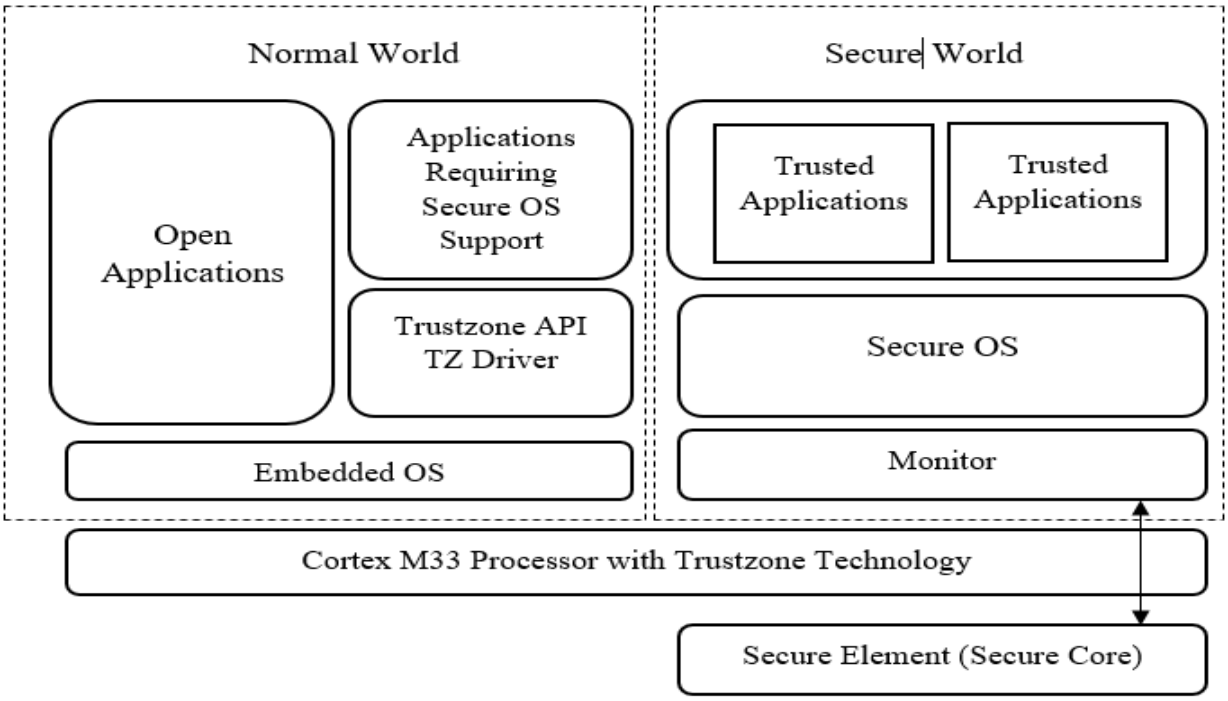

Figure 3 Embedded Operating System with TrustZone

Cellular IoT is an emerging technology that will have a huge impact on the world in many aspects. In an older technology, cellular networks do not have any special requirements like coverage, bandwidth, lower power consumption, cost and linear functionality. The newer outcome of NB-IoT in cellular networks will overcome the disabilities. Thus, it supports on bandwidth, spectrum and maximum data rates in the IoT network. LTE M that supports both frequency and time division duplex modes. Therefore, problems in older methods are performance, bandwidth and duplex methods can be reduced by enhanced machine type communication and narrowband internet of things.

ARM has software tool to make IoT deployment faster and easier, so called MBED IoT device platform. It is primarily MBED operating system built open standards, that claims security and standards based manageability in single tool. MBED OS that supports cortex M processor based devices. It will support standards such as cellular technologies, thread, 6LowPAN and other protocols. It acts as bridge between protocol and APIs for IoT devices. Improved efficiency and security are available in MBED device server. MBED OS can be implemented on Cortex M33 by using CMSIS RTX 5 kernel. The main reason for choosing MBED operating system is due to energy efficiency, RAM requirements, memory footprint and security. These are explaining how it is suitable for narrow band internet of things in Core Link SSE 200 IoT subsystem with Trustzone.

2 METHODOLOGIAL INVESTIGATION OF MBED OS LINKED WITH TRUSTZONE FOR SECURITY SOLUTIONS

The latest cellular standard is Narrow Band IoT (NB-IoT) mainly for Long Term Evolution (LTE). Long and wide range cellular systems are connected by Narrow Band IoT which has Low Power Wide Area Network technology. It is mainly designed for the Internet of Things. Cellular IoT technology is standardized by the 3rd Generation Partnership Project (3GPP). NB-IoT is specifically designed for large number of connected devices, low power consumption and low cost. In this proposed system, NB-IoT can be implemented in Core-Link SSE 200 IoT sub-system. IoT sub system can be done with V2M-MPS2+ board. The board contains Cortex M prototyping system, which is mainly designed for evaluation and prototyping of ARM v8 architecture. The MBED OS that only supports on Cortex M series like M0, M0+, M3, M4, M7. Also, ARM develops cortex M23 and M33 for security purposes.

The focus of this project is to implement MBED OS on Cortex M Prototyping Board. The board that initially build with Cortex M series in it, the MBED OS performance can be evaluated on Cortex M33. The Cortex M33 is implemented on V2M-MPS2+ board utilizing FPGA configuration. This makes that cortex M prototyping board will act with CM33 processor. ARM Keil provides the software pack for CMSIS RTOS to build kernel in CM33. This includes software components to development tools. Source modules, config files, user code templates and header are available in software components. Using kernel, we can develop MBED OS with corresponding drivers, peripherals, and secure connectivity.

The CMSIS defines generic tools interfaces and hardware abstraction layer for Cortex M processors. The CMSIS is intended to establish the software combination from multiple vendors. It supports ARMv8 architecture including trust-zone for hardware security extensions and CM23 and CM33 processors. The CMSIS contains CMSIS core which provide standardized interface for cortex M processor core and peripherals. CMSIS peripherals driver's interfaces for supported devices. The application peripheral interface in RTOS that connects the peripherals in microcontroller that implements for system stack and user interfaces. DSP library available in CMSIS for all cortex M cores, which implements all instruction sets available for M4, M7 and M33. Core implements the run time system for Cortex M devices that gives access to the core and devices by the user.

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.

2.1 Hardware Description – V2M MPS2+ board

It can be noted that both MBED operating system and Trustzone are implemented on ARM Cortex M prototyping system (V2M MPS2+). It is mainly designed for Cortex M processors prototyping and evaluation purposes. The system includes latest cortex M7 processor with cheap motherboard. It provides useful peripherals with encrypted FPGA for all Cortex M processors. In this system cortex M33 can be implemented using IoT kit image file. Thus, the prototyping system with FPGA will act as cortex M33 processor. Using IoTkit_CM33_FP Keil pack we can configured peripherals for the design. It is an ideal platform used for FPGA prototyping and evaluation of the operating systems. Additionally, this hardware board has a functionality of going back to previous ARM Cortex series especially ARM cortex M4 by basic booting process, thereby MPS2+ purely provides great support for ARM Cortex M processors. MBED OS is performed by using CMSIS RTX v5 kernel. RTX kernel having some functions in run time environment with kernel configuration that describes running task, systick timer as kernel timer, clock and round robin switching methods. RTX kernel functions that runs and execute main thread, it has some external function to initialize kernel and suspend the kernel. Both initialize, running and suspend are performed under main function. Thus, the threads are decelerated with some attributions, otherwise it will take as default value.

2.2 Introduction to MBED OS

The MBED OS is an operating system for the IoT devices. It is mainly designed for low cost energy environment, connectivity, security, and device management that required by IoT devices. It gives application framework for development and supports for all standard connectivity. The MBED OS is an operating system mainly created for MBED enabled devices. It allows the application to control the hardware of the boards by providing APIs. MBED OS intended specifically for controlled devices, the hardware and networks which are limited resources that works automatically behind the system.

The single thread, hardware abstract and OS efficiency that manages power and schedules tasks, manages device, and securing communication by using MBED OS uvisor and MBED TLS supports multiple networking options. The MBED OS core provides application portability through kernel and hardware abstraction layer. MBED connectivity develops APIs to increase portability and productivity of the device. This connectivity implements choice of low level communication stack. For comprehensive lifecycle, device communication and security framework can be performed by MBED security. The MBED tools manage configuration, built and testing.

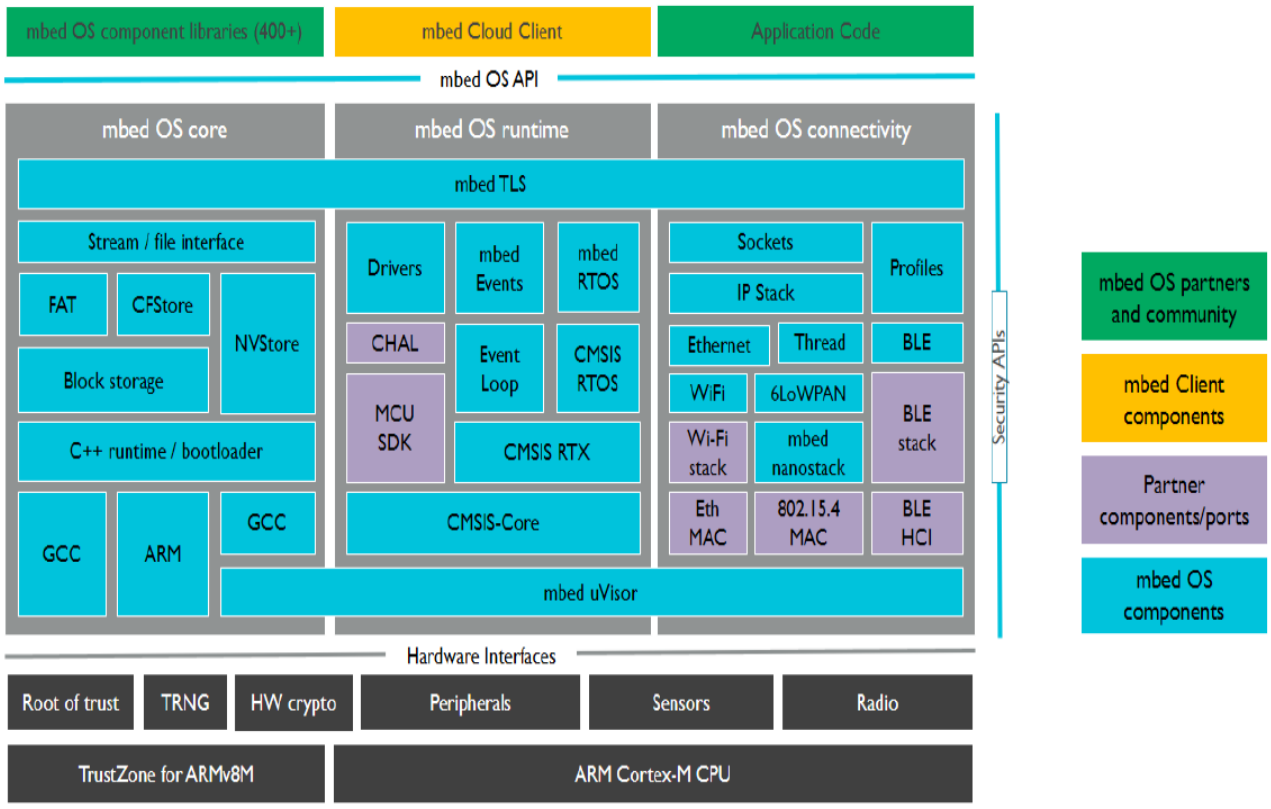

Figure 4 MBED OS Stack [7]

2.2.1 OS Platform requirements

It requires mainly connectivity, security, and management for the embedded systems. IoT products requires device management services which will become device essential components with platform OS.

1. To accelerate the development of IoT devices that needs pre-integrated to all necessary connectivity and software components which provide across many hardware solutions, modern development methodologies, choice to microcontroller units and to improve productivity. It provides operating system components and application peripherals across many vendor solutions.
2. The deployment of IoT devices providing standard connectivity to difference transports and manage cloud to open opportunities and minimize cost Solve the device management problem to deployment of the IoT devices.
3. Ecosystem scale provide maximum gearing and pace in IoT platform. Open source to remove barriers, collaboration with partners to provide maximum gearing of investment for everyone.

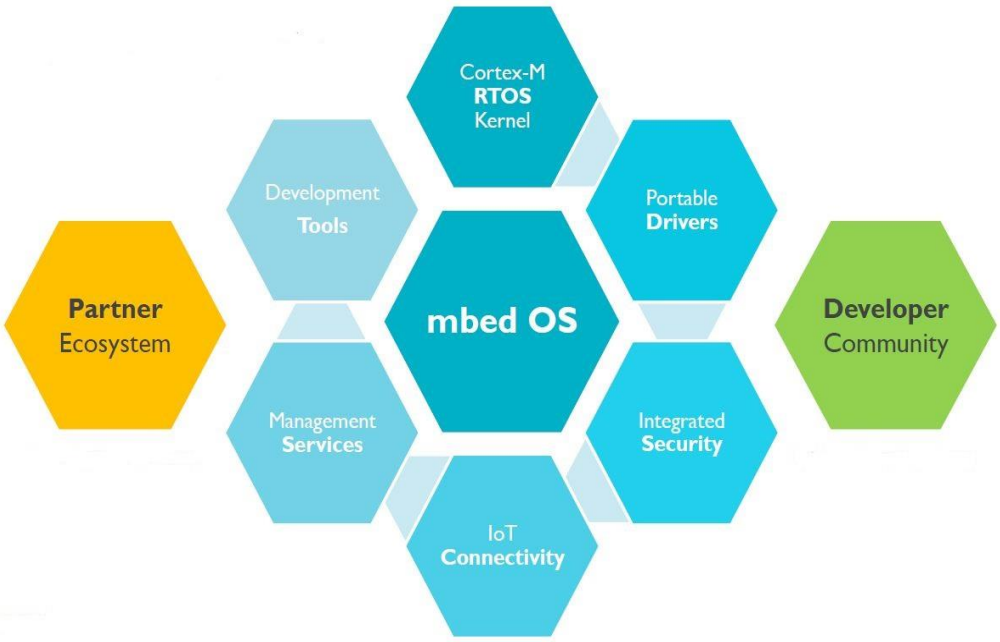

Figure 5 MBED OS platform for internet of things [8]

2.2.2 MBED OS Core

MBED OS core enables application and component libraries to work unchanged across microcontroller units. It provides portability for developers. Consistent boot and C/C++ runtime across microcontroller units includes support over different toolchains and standard library integrations. RTOS kernel is built on the established, widely used and open source CMSIS RTOS RTX. Memory devices optimized small kernel. Common peripheral driver APIs for supported across all MCUs. It is helpful to start up and initialization, memory maps and cross toolchain integration. In MBED OS core defines that initially start up with boot CMSIS with corresponding runtime events to the networking. In figure 6 the OS core layer defines the events, threads and CMSIS RTOS RTX of the MBED OS.

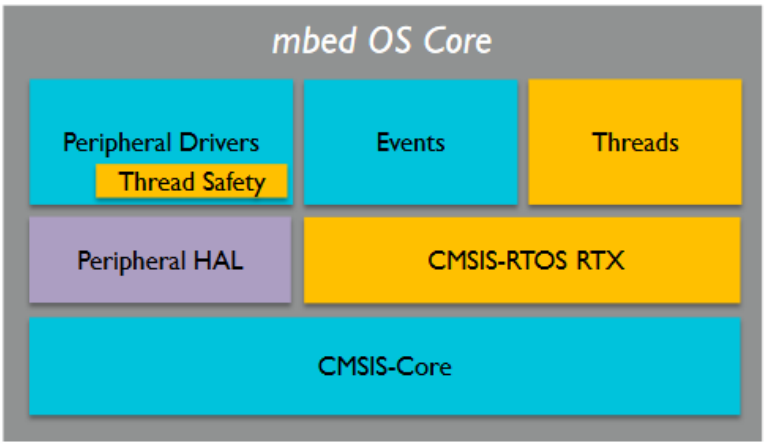

Figure 6 MBED OS core layers [8]

2.3 Cortex Microcontroller Software Interface Standard (CMSIS)

Cortex M processor series are supported by hardware abstraction layer given by CMSIS Keil pack which describes tool interfaces. It offers device affordability and software connection to the peripherals, software reusability to the processors for effortless learning in microcontroller develops and save time for latest devices. Figure 7 shows CMSIS structure, it describes the components as follows. In this project, the latest version of CMSIS v5 which is suitable for ARM v8 M architecture (Cortex M33) and Trustzone for ARMv8-M hardware security solutions, which is other main part of the project. CMSIS is described with different silicon vendors and provides unique decision for peripherals to connect RTOS and middleware components [9].

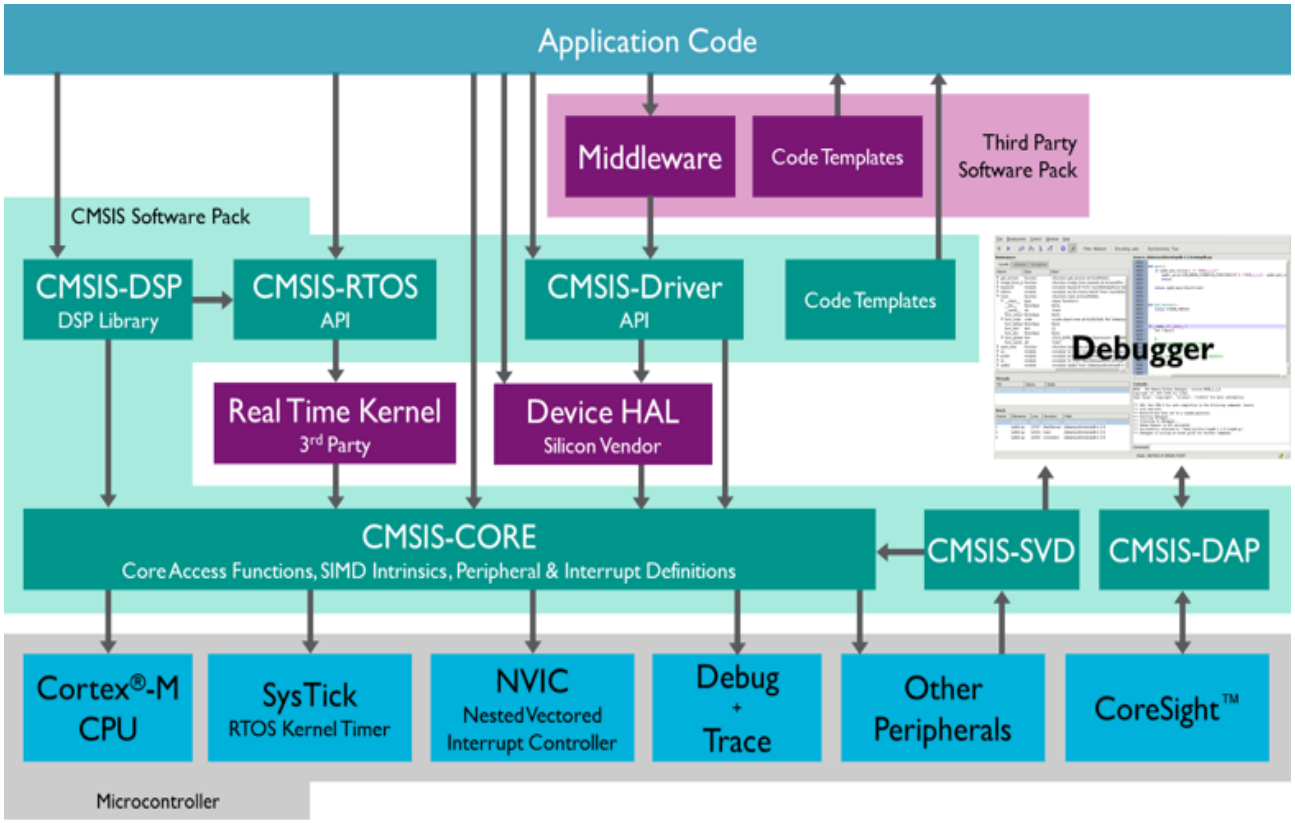

Figure 7 CMSIS Structure [9]

The CMSIS consists of main components as

CMSIS core – This is a main part of the structure that includes APT for Cortex M processor core and its peripherals. It provides a standardized interface for Cortex M series as well as Single Input Multiple Output function for all Cortex M series. In this project, the CMSIS core operates a runtime system for Cortex M33 and Cortex M4 processor and allow accessibility to the core and the peripherals of the device. CMSIS core describe how the security extensions which are available in Cortex M33 processor. Thus, MBED OS will be implemented using kernel RTX 5 core.

CMSIS Driver – To enable connection between middleware peripheral driver for reusability through the devices. The microcontroller with middleware interfaces allows communication stacks, structures and file systems in RTOS API [9].

CMSIS DSP – The DSP library with different data type for single precision and fixed floating points are defined by CMSIS DSP. The Cortex M processor access all library cores. It is stabilized for performance single instruction multiple data instructions are handled by ARM v7 M and ARM v8 M architectures [9].

CMSIS DAP – Standard firmware for debug unit that connects to access port. It is distributed as package and will suitable for evaluation boards. Debug access port is provided as separate download.

CMSIS-RTOS v2 – CMSIS RTOS2 is an upgrade version of CMSIS RTOS with kernel based operations in RTX. ARM v8 M suitable with dynamic object, binary enhanced interface between API compilers and multi core accessibility for system [9].

2.3.1 Overview of CMSIS RTOS v2

The CMSIS RTOS2 is common API interface mainly for Cortex M processor devices. It provides a standard API for software peripherals which needs functionality and provide dedicated benefits to the users. The middleware components need CMSIS RTOS2 to refer undecided RTOS and then will easy to adapt firmware structure. The source of microcontroller system handled by CMSIS RTOS to perform parallel thread operation concurrently. CMSIS RTOS2 perform several operations quickly and concurrently. Thread operations performs various task to reduce the program structure. The system is ascendable and more threads are summed rapidly, it executes high priority thread initially. The CMSIS RTOS2 gives several services in various application for periodical timer function with triggering and memory consumption with applications.

The CMSIS-RTOS2 concentrates the following needs:

1. CMSIS RTOS v2 does not need memory buffers for dynamic objects. It is optional for CMSIS RTOS
2. Secure and non-secure modes are implement in ARM V8 M architecture
3. Multi core systems allows to pass messages between two layers.
4. CMSIS RTOS fully based on C++ run time environments.

ARM Cortex Microcontroller Software Interface Standard (CMSIS) motivate to provide a standardized API for software components. It enables standard project templates, stimulate middleware development, and simplify the usage of CMSIS DSP library. It is used to simplifying programming models of cortex M devices. It provides signal protocols and middleware industry with standard RTOS APIs. Software ecosystem allows application sharing to the API design.

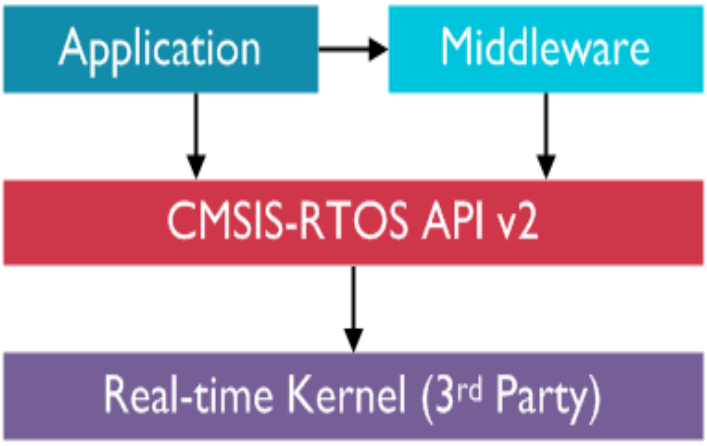

Figure 8 CMSIS RTOS API Structure [10]

The CMSIS-RTOS API v2 is a generic RTOS interface for Cortex-M devices. CMSIS-RTOS2 provides features that are required in many applications. Middleware components that use the CMSIS-RTOS API are RTOS agnostic. CMSIS-RTOS compliant middleware is easier to adapt. API design consideration is suitable for small memory foot print, scalable functionality, and multi-processor systems. CMSIS RTOS that requires pre-emptive context switching with multiple priorities. It mainly required functionalities like mutex, semaphore, time management, queue, and event signal. Particularly message queue and mailing queue that works with interrupt signals.

Features that allows generic wait function with support to time intervals. Zero mail queue is used to support multi-processor systems. Priority inversion is done using deterministic context switching and round robin context switching. CMSIS RTOS2 that preforms inter-process communications to exchange data between two or more separate independent threads. Operating system provide resources for inter process communication, such as message queues, semaphores, memory map, and time management. Inter process communications are programmed in high level of abstraction that gives application facilities to the distributed system.

2.4 Functions defined within CMSIS RTOS v2

MBED OS embedded operating system is implemented in cortex M prototyping system (V2M MPS2+). MBED OS is performed by using CMSIS RTX v5 kernel. RTX kernel having some functions in run time environment with kernel configuration that describes running task, systick timer as kernel timer, clock and round robin switching methods. RTX kernel functions that runs and execute main thread, it has some external function to initialize kernel and suspend the kernel. Both initialize, running and suspend are performed under main function. Thus, the threads are decelerated with some attributions, otherwise it will take as default value.

2.4.1 CMSIS RTOS Thread

The threads have scheduling unit, it performs under some priority functions too. Thread will be generated by osThreadCreate() by defining with priority, instances, and stack size. Thread defines an object leaves it as not schedulable. It has some states like running, ready, waiting and inactive to perform declared function in that thread.

- Running state execute only one thread at that time.
- Running thread has terminated, the next thread in ready position with highest priority become running thread.
- Waiting state that perform to wait event in thread functions.
- Thread are not created in Inactive state.

Thread priorities levels are osPriorityIdle, osPriorityLow, osPriorityBelowNormal, osPriorityNormal , osPriorityAboveNormal, osPriorityHigh, osPriorityRealTime.. The priority level is set when thread object defined. The thread priority level change be changed by osThreadSetPriority and osThreadGetPriority for new priority and return the current task priority. Thread states and transition function are shown in figure 9.

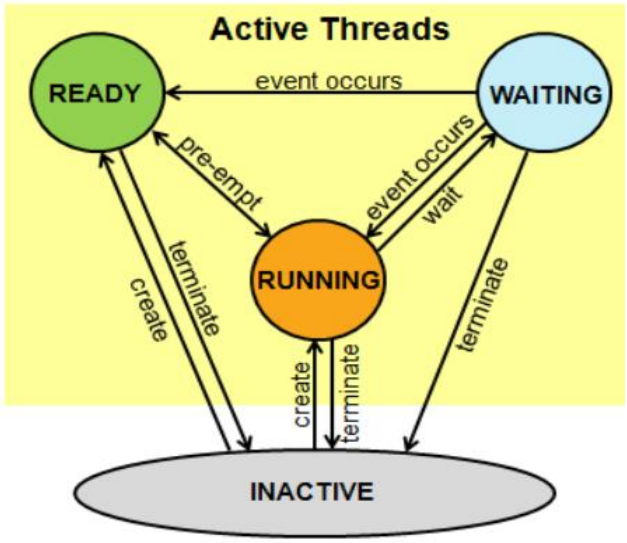

Figure 9 Thread State and State Transition [11]

2.4.2 CMSIS RTOS Semaphore

Semaphore management function is used to protect and managed access to shared resources. It accesses to the group of identical peripherals which can be managed. Each time a semaphore token is obtained with osSemaphoreWait the semaphore count is decreased. The no token can be acquired, when the count is zero. Semaphores are released with osSemaphoreRelease functions which increase the count [12].

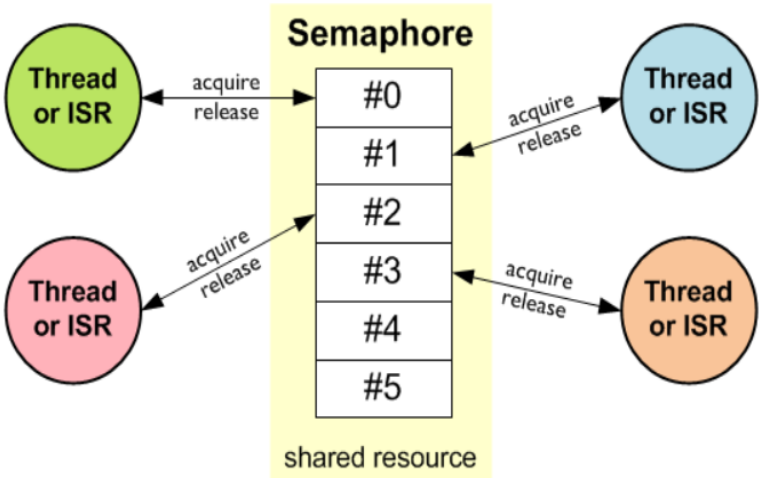

Figure 10 CMSIS-RTOS Semaphore [12]

2.4.3 CMSIS RTOS Mutex

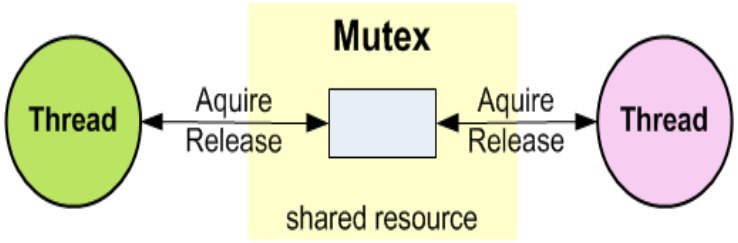

Figure 11 CMSIS-RTOS Mutex [13]

For resource management, the operating systems can be done using mutual exclusion. Microcontroller devices having resources which may use often, but only one thread at time. Shared resources protected by using mutex function. It is created and then passed between the threads. Like semaphore mutex also contains token, it is special version of semaphore. Comparing to semaphore mutex perform only one thread at the time. Mutex having thread ownership thus it has token is in binary and bounded. When a thread acquires a mutex and becomes its owner, subsequent mutex acquires from that thread will succeed immediately without any latency [13].

2.5 Introduction to Trustzone

Embedded system products require sensitive hardware, real time operation, less power and security protection. To accelerate the system design, the modern applications need security. The protection of assets requires device communication using crypto and authentication methods, firmware against IP theft, secret data such as personal information and operation to maintain services. Trustzone for ARM v8 M includes:

- Secure and non-secure domain conserves low interrupt latency
- The complexity of solution does not code overhead
- The secure domain has minimal call instructions.

The confidential and integral part of system is developed by Trustzone. The processors application protects high value code of authentication and enterprise. On the application processor, it is used to provide security boundary for global platform environment. This is depicted on figure 12, where the processor family features contribute equal security approaches, but an entirely contrasting operation. Trustzone system offers a foundation for system-broad security and the development of a trusted platform.

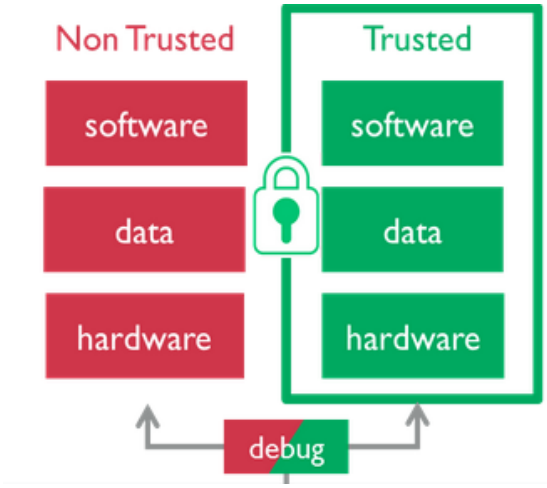

Figure 12 ARM Trustzone security concept [15]

2.5.1 Programmer's Model for ARM v8 M

In the secure state, it accesses all the peripherals and the memory. The memory alias that mirrored all secure and non-secure peripherals in the system control and debug area. Secure code in secure region that access memory in both regions. The secure peripherals are assessable during program execution. The Security Attribution Unit configures non-secure memory, interrupt and peripheral access. A memory protection unit and system control block and systick are also available in secure state. The two interrupt vectors for secure and non-secure supports system execution. This assignment is controlled during secure state execution through nested vector interrupt controller [14].

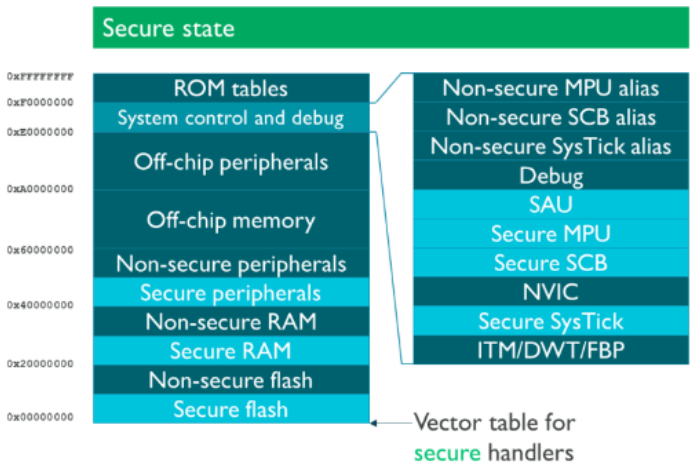

Figure 13 Secure Memory Map [14]

The memory view is same as cortex M memory map. It accesses to secure memory that triggers security exception in handler state. Code that from non-secure is executes in non-secure and only access memory in that region. The secure regions from non-secure code that is executed and security state of the system in fault exception. CMSIS core defines additional file that is used in secure attribution unit.

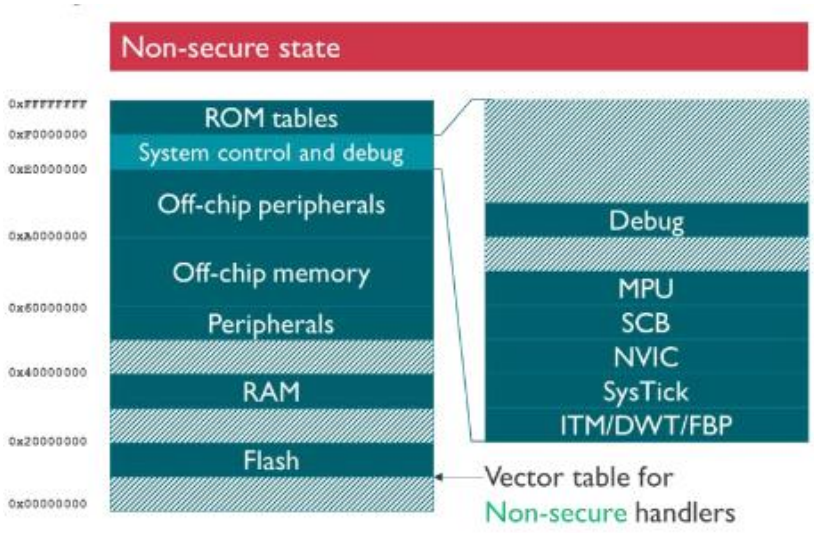

Figure 14 Non-secure Memory Map [14]

2.5.2 Registers

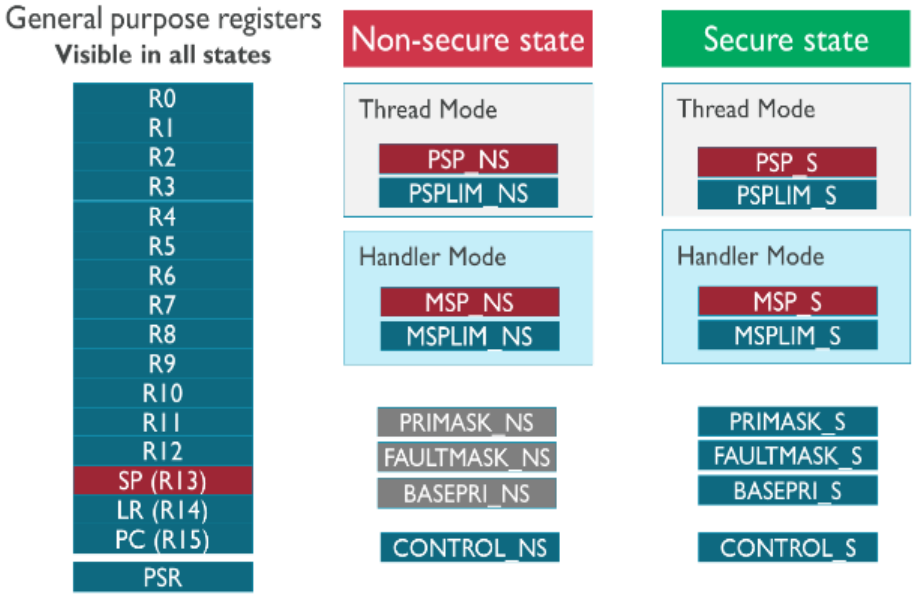

Figure 15 Register in ARM v8 M [14]

Above figure shows register view of ARM v8 M system, the general-purpose registers can be accessed from any state, callable function use registers for parameter and return values. R13 stack pointer register alias (PSP_NS, MSP_NS, PSP_S, MSP_S) accessed depends on state and mode of thread. Each stack pointer having optional limit registers for stack overflows.

The system has independent CONTROL register for both state. The interrupt control registers are banked between the states. The interrupt priority for the non-secure state can be lowered so that secure interrupt has always higher priority. The core registers of the current state are accessed using standard core registers functions. Registers are accessible in both secure and non-secure states [14].

Summary of Register

In figure 15 the general-purpose registers are shown. Registers that perform different operations and functions in it. General-purpose registers for data operations can be done by using Registers from R0-R12. Stack pointer is in register R13, it is used to indicate CONTROL registers and having Main Stack Pointer (MSP) and Process Stack Pointer (PSP). For the security extension can be implemented using MSP_NS for Non-secure and MSP_S for secure state. Similarly, for PSP registers. Link Register is register R14 to stores return values form function calls. Register R15 is used for Program Counter contains address of program. Program Status Register is the combination of Application, Interrupt and Execution Registers. PRIMASK protect all exception with priority. For the security extension both non-secure and secure states are implemented. If it is not need means, CONTROL registers control in stack that used in Thread mode.

2.5.3 Memory Map

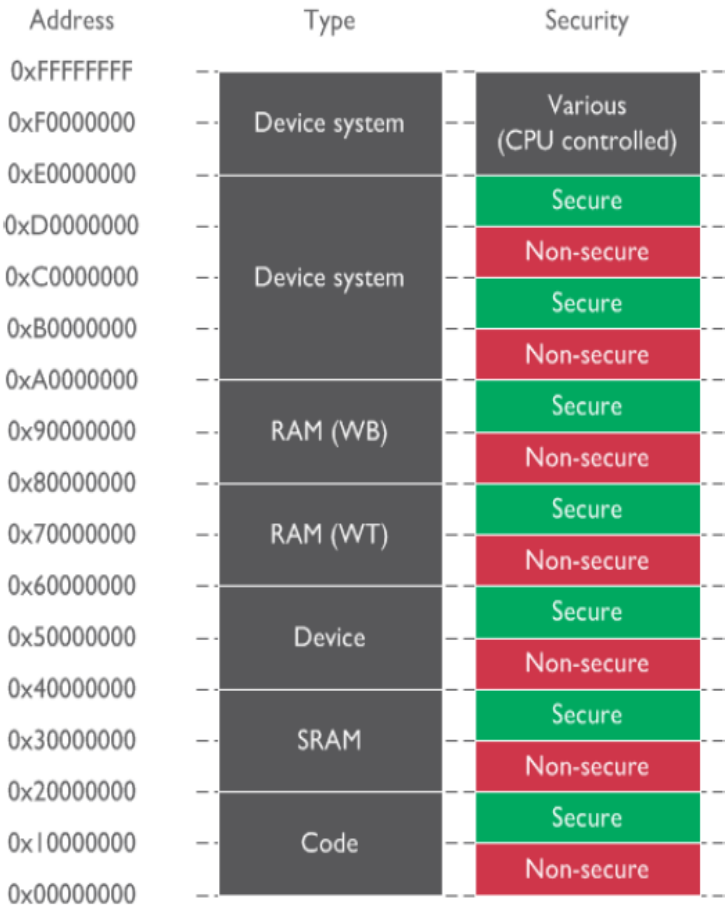

Figure 16 Memory map model [15]

Memory map of Trustzone boundaries is divided into 512MB. It adds security support by aliasing each boundary at intermediate point. The lower part access to 256MB non-secure state and the upper part provides secure state of 256MB. Each secure window and non-secure window are accessible through control points. The user could use 28-bit address to define secure and non-secure memory regions [15].

Table 2 ARM v8 M Default Memory Map [16]

Address Region	**Region Name**	**Memory Type**	**Description**
0x00000000 - 0x1FFFFFFF	Code	Normal	Typically, ROM/Flash vector table that is required for boot up
0x20000000 - 0x3FFFFFFF	SRAM	Normal	On chip RAM
0x40000000 - 0x5FFFFFFF	Peripheral	Device	On chip peripherals
0x60000000 - 0x9FFFFFFF	RAM	Normal	Supports code
0xA0000000 - 0xDFFFFFFF	Device	Device	Expansion memory
0xE0000000 - 0xE003FFFF	PPB	-	NVIC, MPU and SAU registers
0xE0040000 - 0xE004FFFF	Device	Device	ETM, MTB configuration registers
0xE0050000 - 0xE00EFFFF	PPB	-	Reserved memory
0xE00F0000 - 0xE00FFFFF	Device	Device	MCU ROM
0xE0100000 - 0xFFFFFFFF	Vendor_SYS	Device	Core sight ROM

2.5.4 RTOS Thread Context Management

To contribute a stable RTOS thread context management for TrustZone system based on ARMv8-M across the different RTOS, the CMSIS-CORE provides header file called as TZ_context.h including API definitions, which is picturized in figure 17 . A non-secure application adopts an RTOS and access library modules defined under secure mode needs an authority the secure stack area. It can be noted that RTOS which operates in non-secure mode don't have permissions for accessing registers under Secure mode. Thus, Secure functions offers a consistent thread context switch.

As the non-secure and secure mode parts of an application are splitted, the API for governing the secure stack area should be stabilized. If not the secure library modules automatically push an application under non – secure mode to adopt an identical RTOS implementation. It should be noted that to allocate the context memory for threads, an RTOS kernel that operates under non-secure mode calls the interface functions prescribed by the header file TZ_context.h. The TZ_context functions itself are sections of the application running under secure mode. A minimum implementation is offered as part of RTOS2 and needs to manage the secure stack for execution of the thread. But, there is also way to realise the context memory management system with supplementary features like access control to protect state memory regions using an MPU.

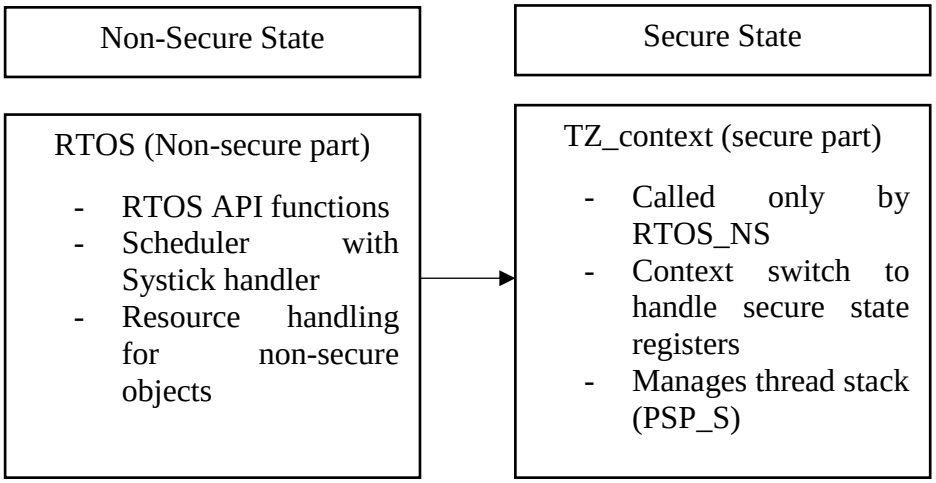

Figure 17 RTOS Thread Context Management for ARMv8-M TrustZone

2.5.5 Trustzone Security Requirements

In embedded system security can mean many different things which are not limited which is shown in figure 18. It requires the followings need,

1. Communication protection which means protection of data transfers from being visible to unauthorized parties and other like cryptography.
2. The data protection prevents data from third parties and that is stored inside the devices.
3. firmware protection secures on chip from reverse engineering.
4. Operation protection secures operations from threats.

5. Many security products are required protection mechanism of devise from being overridden.

6. Firmware in secure memories are preloaded to prevent attacks. Trustzone technology for can also work with protection techniques.

7. Critical operations of software can be preloaded to permit access from secure state. Thus, operations can be protected from non-secure state.

8. Secure boot enables platform and it will boot from secure state.

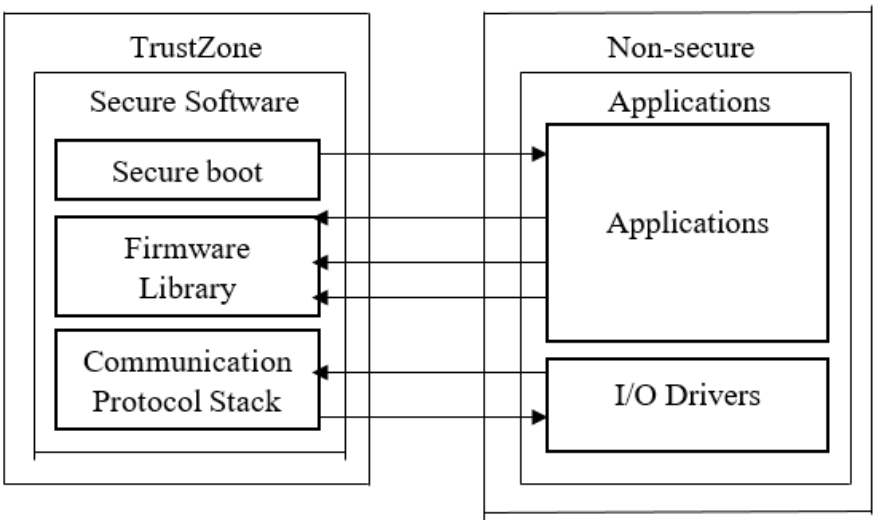

Figure 18 Trustzone Security address

2.6 Security for IoT Devices

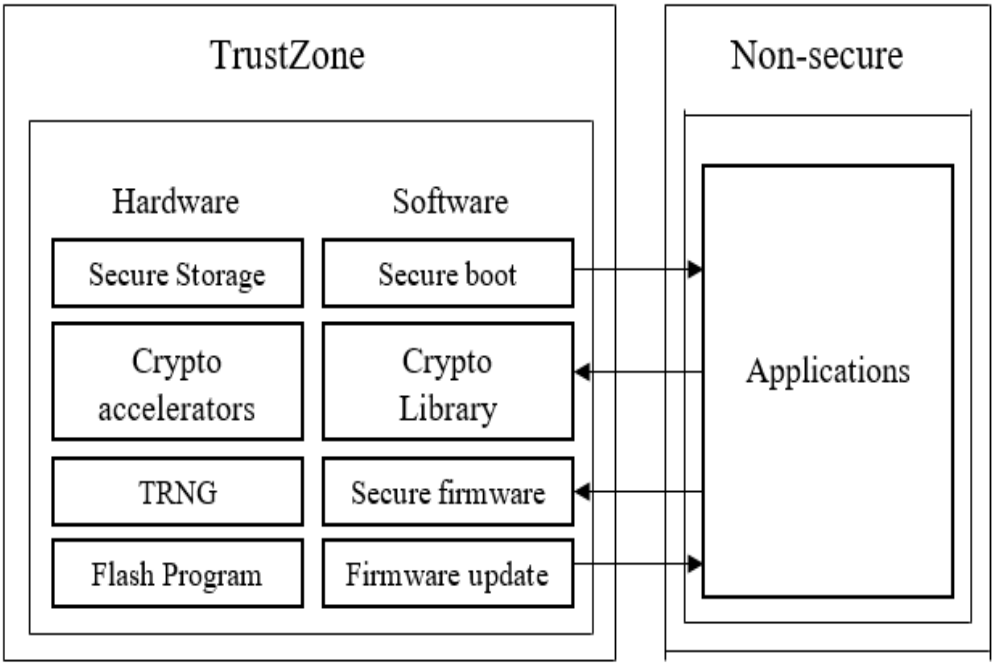

Figure 19 Trustzone for IoT Devices

Figure 19 describes Trustzone technology also used for addition protection features in microcontroller that targeting internet of things products. The Cortex M33 microcontroller is developed for cellular IoT which include a long range of security features. The use of TrustZone can ensure that all features only accessed using APIs with entry points. To use Trustzone for safe guard security features user must prevent untrusted applications from directly accessing resources. Ensure reprogramed flash image for checking and prevent being from reverse engineered.

3 RESULTS AND DISCUSSIONS

This chapter explains about the results which are taken from Cortex M33 and Cortex M4 processors using MBED OS. TrustZone with MBED OS On Cortex M33 performance can be discussed.

3.1 Comparison of ARM v7 M and ARM v8 M Architecture

It is focused to bring security and productivity on embedded applications in an IoT field. The architecture that reduces complexity and develops secure solutions that fits to SoCs. ARM v8 M is the successor of ARM v6 M and ARM v7 M. It adds fast, low overhead security in hardware. Breakpoints and watchpoints that enhance the trace flexibility. Product performance that improves solutions in memory protection unit. It has two sub profiles such as ARM v8 M Baseline and ARM v8 M Mainline. Baseline is used for power and area constrained devices. In addition, C11 atomic data types instructions enhance the system support. Mainline that is used for featured and capable applications. Trustzone for ARM v8 M optimised for affordable security, restriction to secure memory and I/O paths. Removing code in virtualization solution. It introduces secure gateway SG instructions where the domain call from the instruction.

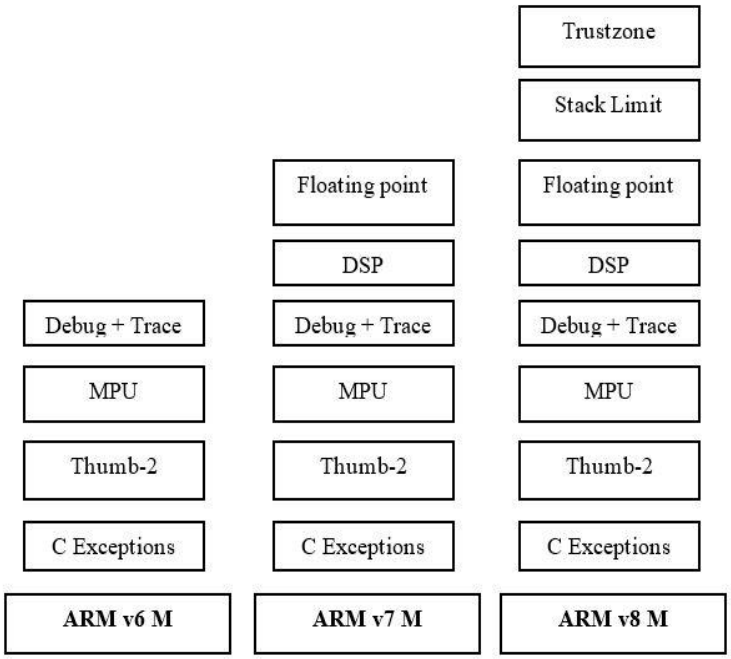

Figure 20 ARM v8 M structure

ARM v8 M that supports Cortex M33 and ARM v7 M supports Cortex M4, where Cortex M33 is the successors of Cortex M4. Core-Link SSE 200 is compactable with M33 processor. The results that are compared between memory consumptions in both M4 and M33 processors by developing MBED OS. M33 gives all security solution with Trustzone and it is improved real time operations. The same process happened M4 without Trustzone isolation.

In Cortex M4 both operating systems are supported, but in Cortex M33 MBED OS supports easily. Cortex M33 performs with ARM CLANG Compiler. But Cortex M4 compile using ARM CC. In this project port file for M4 was developed by changing ARM CC to ARM CLANG compiler.

3.1.1 Cortex M Processors

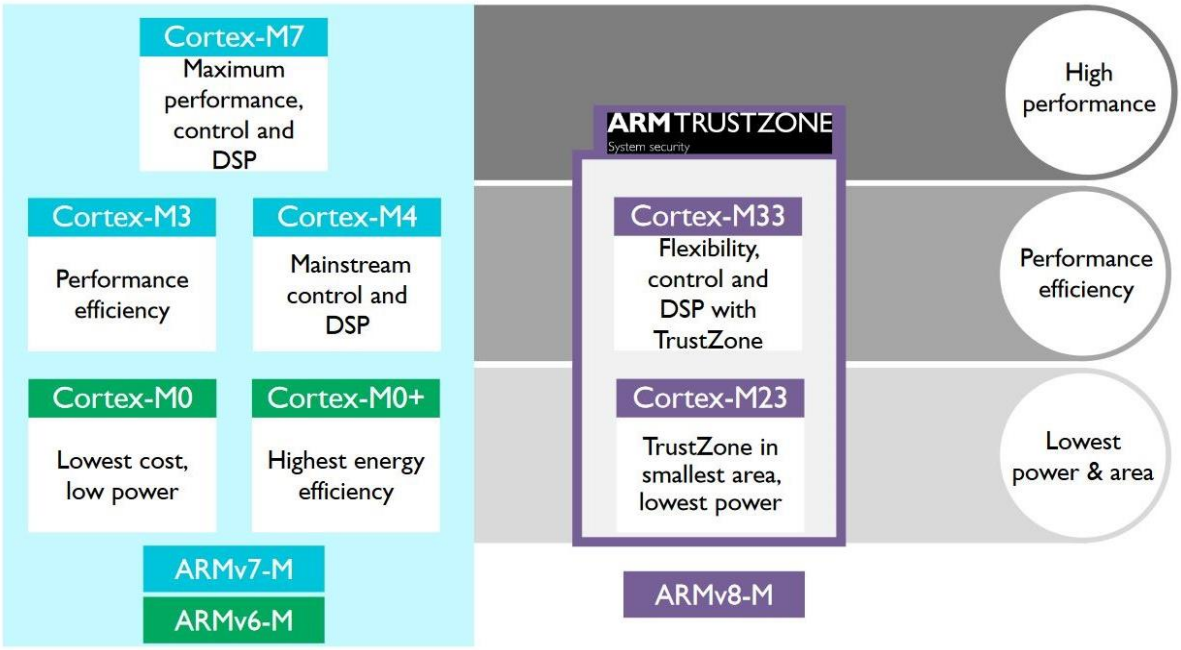

Figure 21 Comparison between Cortex M processors [35]

Cortex M33 is an application of ARM v8 M, it uses same debugging interface for all Cortex M processors. M33 offers wakeup interrupt controller and DSP/SMD instruction for low power devices. M4 having same DSP instruction. Both M4 and M33 has floating point unit which adds more instructions. AHB5 extend security and memory to the whole system. Using memory protection specification can be setup in both regions. It should extend up to 480 interrupts. Embedded Trace Macro cell fits to the design. It buffers as option to trace memory instead of trace-out. Co-processor in M33 supports 8 co-processors. Hardware that check stack limit often. Trustzone gives secure isolation for whole system. The specifications of architecture of processor from debug points having instruction set, exception and program model, debug registers are defined by architecture specifications. ARM v8 architectures as 32 bit which is highly compatible with existing ARM v6 M and ARM v7 M to enable migration within Cortex M processor.

3.2 Memory Management on MBED Operating System

Memory management of an operating system manages primary memory and it process in main memory during execution. It keeps tracks of location, that check how much it consumes during process. It decides the process to get memory on time. MBED OS having memory allocation which are based on defined memory model. Memory allocation contributes many cases like pool allocation, heap allocation and extendable pools.

In typical embedded system, there are four different kinds of memory like code, global data, heap and stack. Consequently, heap and stack area coordinated to fill equal section of memory. In MBED OS, we need two additional memory which are uvisor memory and free heap which are normally located in ROM.

uVisor memory

On Cortex M3/M4, initially it occupies small part of memory in RAM and the protected features. The uvisor protects space adoption in the microprocessor unit.

Stack

In the memory organization of MBED OS that contains stack at the bottom of memory. This address is selected absolutely since it permits stack overflows to be found easily. In Cortex M3 or Cortex M4 memory management is regulated by uvisor which is initially permits by the stack handled by the uvisor memory. In Cortex M0 and M0+ will regulate hardfault. This permits application to rescue from stack overflows, normally via a reset.

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.

Heaps

Never free heap and standard heap are the types of MBED OS. The data can be used by never free heap which is on the top of the address and memory pools are not to be freed. Sbrk function used standard heap which is on the bottom of the address. The core util module developed by trivial allocator for linear allocation and deallocation of memory. This memory allocations are lock free.

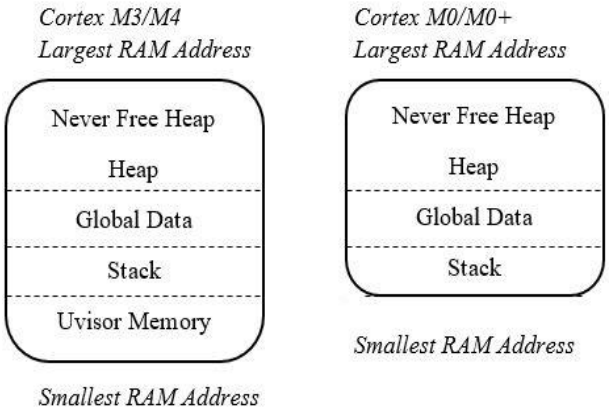

Figure 22 Memory Organization in MBED OS

Table 3 Memory Model of MBED OS [21]

Parameters	**MBED OS – Memory**
Code Size	< 4 KBytes
RAM space for Kernel	< 300 Bytes + 128 Bytes User Stack
RAM space for Task	52 Bytes + TaskStackSize
RAM space for Semaphore	8 Bytes
RAM space for Mutex	12 Bytes
RAM space for user timer	8 Bytes

3.3 Simulation Results

To implement MBED OS together with TrustZone for security solutions on NB-IoT, the thread analysis and experimental results were experimented on ARM Cortex M4 and ARM Cortex M33 processors. These experiments are mainly done to verify the memory consumption on MBED OS to be linked with TrustZone, since memory plays major role on security reasons for NB-IoT. In addition, memory consumption is verified on both the processors (Cortex M4 and M33) to verify the stability of memory mainly on Cortex M33, since it has a property of TrustZone system rather than on ARM Cortex M4.

Thread Analysis - It is conducted to verify whether thread supports for MBED OS, as thread is important parameter for functioning different operations on RTX-5 kernel within CMSIS-RTOS2.

Experimental results – It is conducted mainly to check stability of MBED OS with respect to memory for TrustZone system. It is performed with below implementations with allocated stack sizes tabulated on table-4:

1) **Semaphore function -** Used 3 threads and 1 Semaphore functions
2) **Mutex function -** Used 3 Threads and 1 Mutex functions

Table 4 Stack size of the threads

Thread Name	**Assumed stack size**
Thread_1	256 Bytes
Thread_2	128 Bytes
Thread_3	200 Bytes (Default)

3.3.1 Thread Analysis on Cortex M4 and Cortex M33

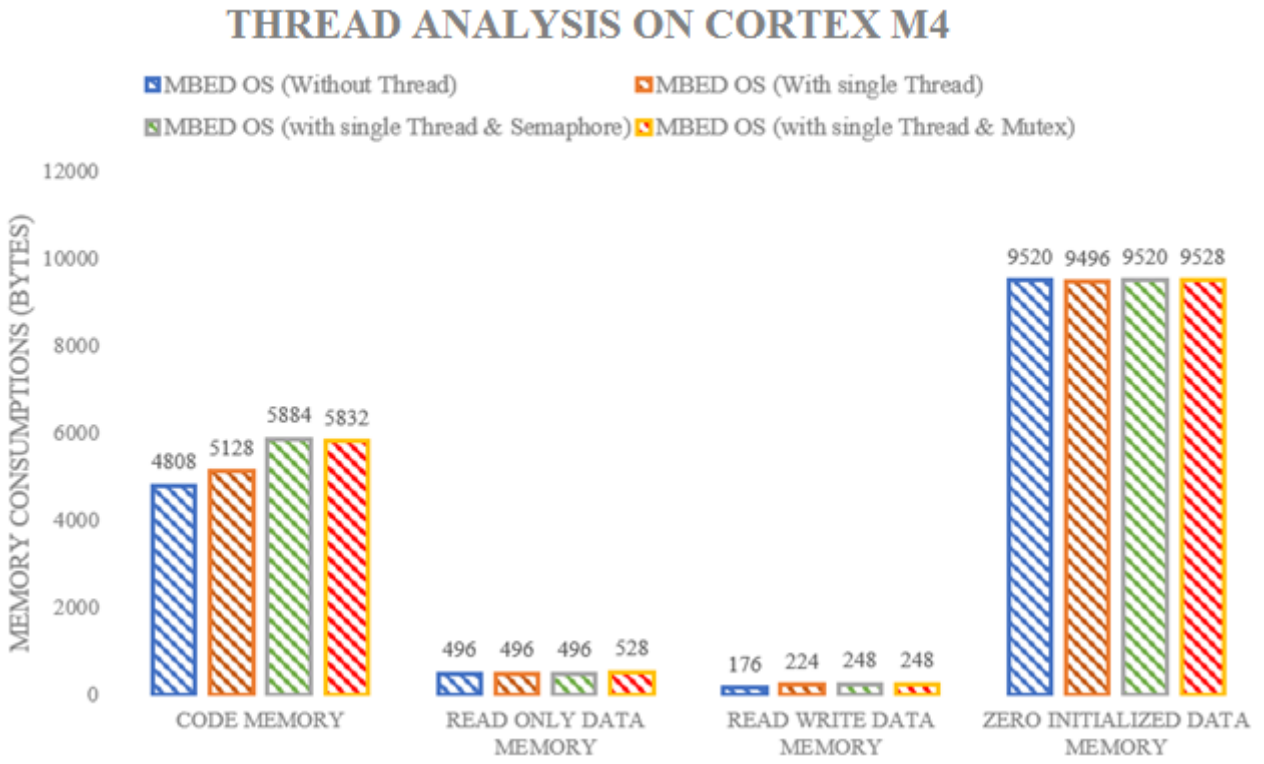

Figure 23 Thread Analysis on Cortex M4

An operating system – MBED offers services to both the users and programs to execute, which follows sequences of operation known as process, which requires certain resources adopting CPU time, memory, files and input and output devices to complete a specific task. Main memory also known as Program memory is responsible for accessing the data quicker, which is shared by CPU. This program or main memory consists of following memories:

a) **Code Memory** – This is mainly dependent on program coding that is viewed on-chip as ROM or EPROM. This code memory can also be saved fully off-chip in external ROM. It can be observed that flash RAM can also be used to saving a program.

b) **Read only data Memory –** When the required program code or data is written onto ROM memory, it can't be eliminated and can be viewed or read only. Compared to RAM memory, ROM restores its contents even though hardware is switched off. Thus, RAM is volatile, whereas ROM is non-volatile memory.

c) **Read write data Memory –** It is type of memory that is basically written into chip as well as read from it, which is normally linked with running software and not requiring physical processes unlike ROM memory. Thus, RAM is often referred to as Read write memory.

Without Thread

MBED OS operating systems are initially performed without threads. RTX 5 kernel on MBED OS that is no longer available when there is no thread, so the system does not interface until the main reached. Once it reaches, the hardware will initialize and starts the kernel. The kernel will initialize when the main execute. The types of memory without thread consume memory size based on the defined kernel configuration.

1. To calculate used ROM space adding code memory, RO data and RW data, the results are obtained.
2. To calculate used RAM space adding RW data and ZI data, the results are obtained.

MBED OS ROM & RAM usage without Thread:

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

With one Thread

Thread mainly refers to performing scheduled tasks sequentially which is contained within a process and various threads created on same process or operation share memory, which is described on calculation. It is to note that thread, which is basic unit of CPU utilization contains PC (Program Counter), stack, registers with corresponding thread ID for checking which thread is performing allotted task or operation. Additionally, on multiprocessor or multicore, threads designed on RTX-5 kernel for MBED OS has a functionality of running at same time, thus saving the execution time. Threads has two types, which are user threads and idle thread designated within kernel itself. It is to view that if kernel is single thread, the user thread handling blocked threads will affects entire process to block, where the idle thread plays its role of performing the remaining task. However rather than having single thread of handling various process, it is essential to increase the number of threads for handling multitasks and thereby saving execution time, which is tested on experimental results discussed on next section. The types of memories are obtained by using one thread function in MBED OS.

MBED OS ROM & RAM usage with one Thread:

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

By comparing with and without thread the memory consumptions of the operating systems are approximately same. The thread function that reduce the processing time in an operating system

With one Thread and Semaphore

As stated earlier on section 2.4.2 regarding functionality of Semaphore defined within CMSIS-RTOS2 is mainly used for secured variables which adopts shared resources on multi-processing environment. It can be noted that semaphore functions don't operate without thread functionality. Semaphore acquire and release values by using thread.

MBED OS ROM & RAM usage with one Thread and one Semaphore:

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

With one Thread and Mutex

It was described previously on section 2.4.3 that mutex defined within CMSIS-RTOS2 is a program object which allows multiple threads to share resources but not simultaneously. It creates resource with unique name.

MBED OS ROM & RAM usage with one Thread and one Mutex:

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

By comparing thread with semaphore and mutex the memory consumptions of MBED operating system consume the values which are predefined in table 4.

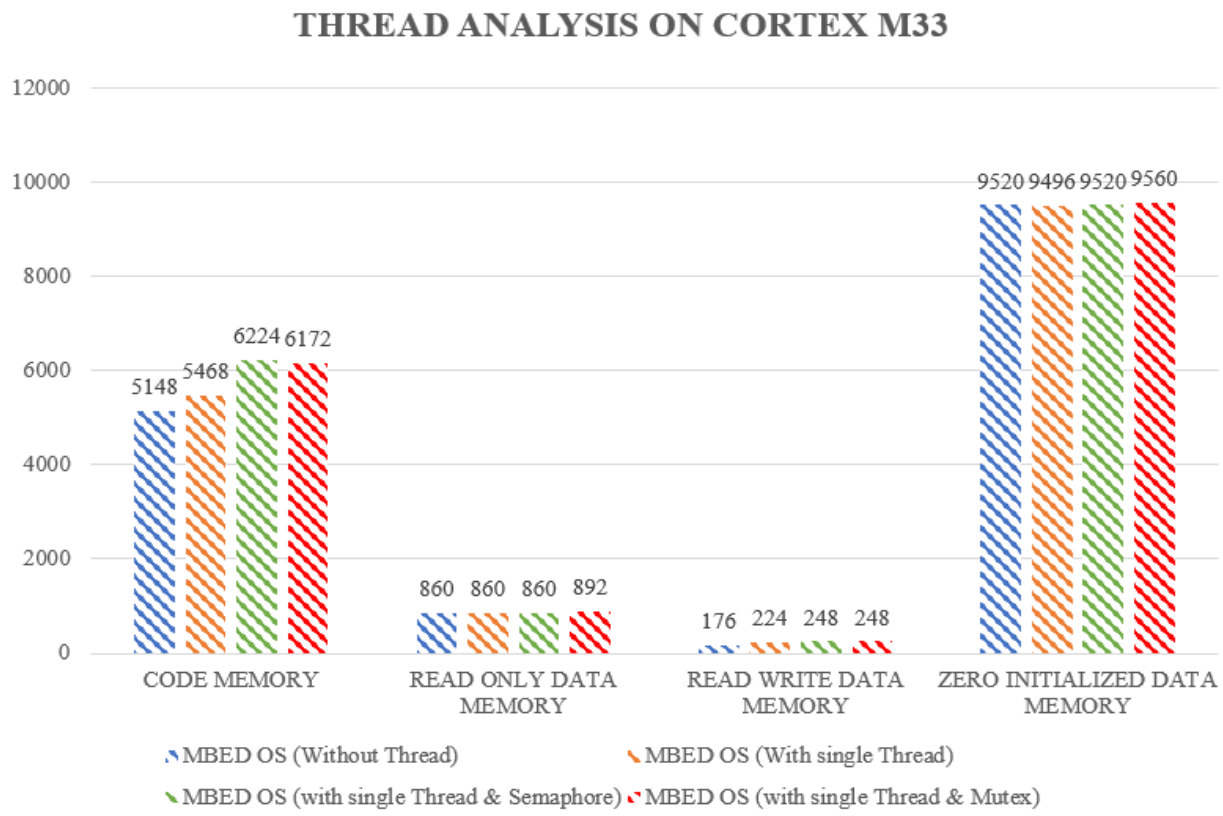

Figure 24 Thread Analysis on Cortex M33

Cortex M33 is the developer of Cortex M4 processor. Both are having same properties but Cortex M33 has different register address and have secure gate through Trustzone. The initial same testing also done on Cortex M33 which is same as M4. Thus, we calculate consumed ROM and RAM memory in M33 using different memory types.

Without Thread

MBED OS ROM & RAM usage without Thread:

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

With one Thread

MBED OS ROM & RAM usage with one Thread:

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

With one Thread and Semaphore

MBED OS ROM & RAM usage with one Thread and one Semaphore:

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

With one Thread and Mutex

MBED OS ROM & RAM usage with one Thread and one Mutex:

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

3.3.2 Experimental results on Cortex M4 and Cortex M33

As described earlier that with use of single thread, it is unable to handle multitasks and it takes more time to execute, where more memory will be consumed, hence the experimental results were carried out for multitasking by increasing the number of threads and other functions within CMSIS-RTOS2 stated on section 3.5. This is mainly done to investigate the memory consumptions on both ARM cortex M4 and M33 processors.

Thus, the threads are increased by allocating different stack sizes and giving different priority for each thread, so that each thread performs its own operation on priority basis and simultaneously the memory consumed by each thread and release and acquire operations of Semaphore and Mutex functions are noted, which is tabulated on Thread Management table on Appendix. The plots are correspondingly described on figure 25.

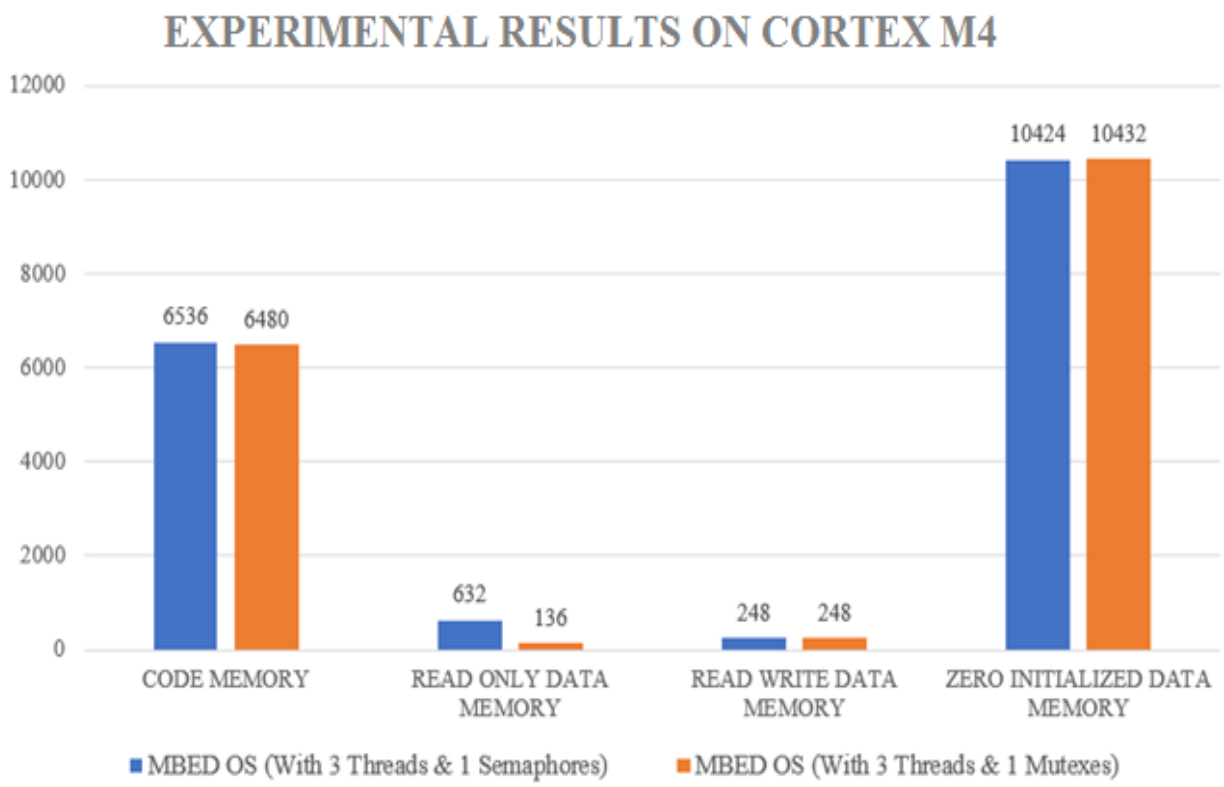

Figure 25 Experimental results on Cortex M4

With 3 Threads and 1 Semaphores

MBED OS

ROM = 6536+632+248 = 7416 Bytes

RAM =248+10424 = 10672 Bytes

With 3 Threads and 1 Mutexes

MBED OS

ROM = 6480+136+248= 6864 Bytes

RAM =248+10432 = 10680 Bytes

Final testing on M33 same as M4, memory management in M33 gives the performance analysis of operating system. It will decide that which operating system having better function. Core Link SSE 200 prototyping processor depends on operating system which will be implemented on it. Therefore, the memory consumption of M33 should be lower while comparing with others. It will also have depicted the operating system.

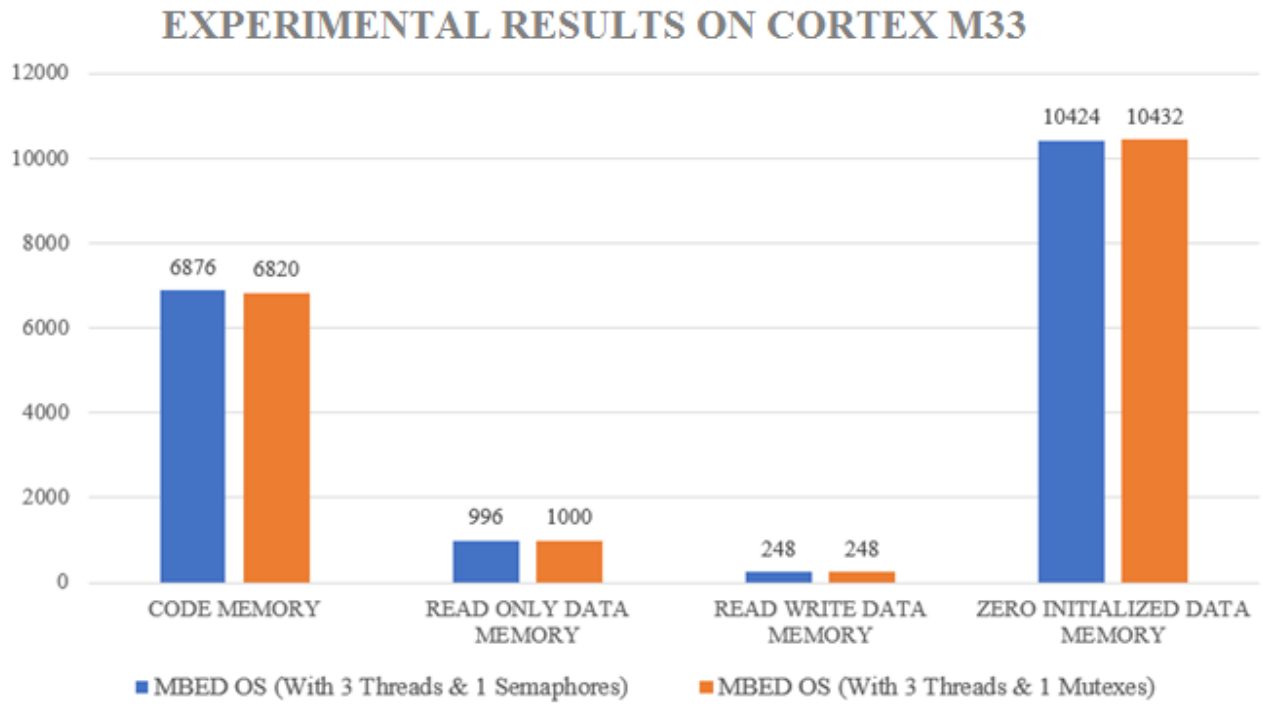

Figure 26 Experimental results on Cortex M33

With 3 Threads and 1 Semaphores

MBED OS

ROM = 6876+996+248 = 8120 Bytes

RAM = 248+10424 = 10672 Bytes

With 3 Threads and 1 Mutexes

MBED OS

ROM = 6820+1000+248 = 8068 Bytes

RAM =248+10432 = 10680 Bytes

Comparing thread analysis and experimental results on Cortex M4 and Cortex M33

Based on above conducted results it can be observed that thread supports greatly on MBED OS for performing multitasks by consuming less memory, which is satisfying the needs of security solutions to be fit into TrustZone system. MBED OS supports string functions and has well defined libraries, which is main point for having less ROM consumption. Thus, the initial testing on both processors performs well. On the final testing of MBED OS, by increasing thread function and corresponding thread managements using semaphore and mutex. The RAM consumed by these values are higher than the initial values. Because increasing functions in operating system also increase the RAM memory. If it is less number of functions means it will consume less memory otherwise it will increase.

According to the result the memory consumption of MBED OS is moderate and nearly coping with ARM Cortex M4, thus it will suitable for Cortex M33 processor to be implement MBED OS together with TrustZone on this latest processor which is later integrated into Core Link SSE 200.

3.3.3 Stack Management on Cortex M33

The operation of program is done by using semaphore and mutex structures. Kernel configured with semaphore and mutex functions, where one thread is defined as default stack size 200 bytes and remaining threads have specific stack size. The performance based on MBED OS stack size and corresponding address are tabulated in the Appendix. The table in the appendix depicts thread management using semaphore, mutex which can be observed that each thread has different stack size and if certain threads are in ready or blocked mode they will consume same memory in ready and blocked mode. But in the running mode it will consume different memory size. It should be noted the maximum stack size of each thread and define used size of memory in bytes. Although stack consumption will be performed on Cortex M4, the deciding processor for MBED OS together with TrustZone for NB-IoT is ARM cortex M33. Thus, the Stack consumption using thread function of semaphore and mutex are shown in figure 27. This is mainly done based on memory consumption between ARM Cortex M4 and M33, since the memory consumption plays major role on security solutions for protecting the registers with less memory handling.

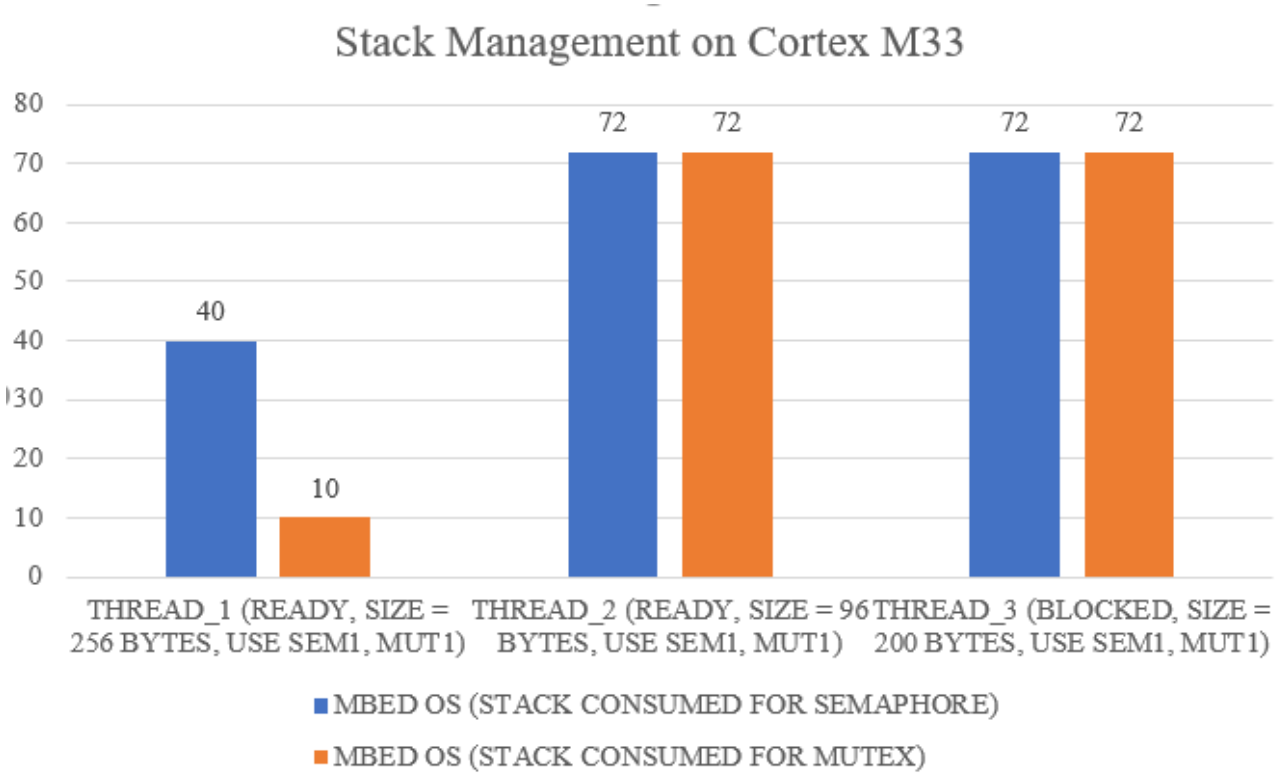

Figure 27 Stack consuming using Semaphore and Mutex Functions

RTX kernel information

It gives kernel ID and kernel state. State define that the kernel is in running or ready mode. The frequency of kernel is 1000 Hz for all thread managements. Kernel tick count defines the running time of kernel. Therefore, the thread objects provide control block size, default size and user stack size. Control block size define process table, Task Strut or switch frame in the operating system. The semaphore consumes 16 bytes in control block, and mutex consumes 28 bytes.

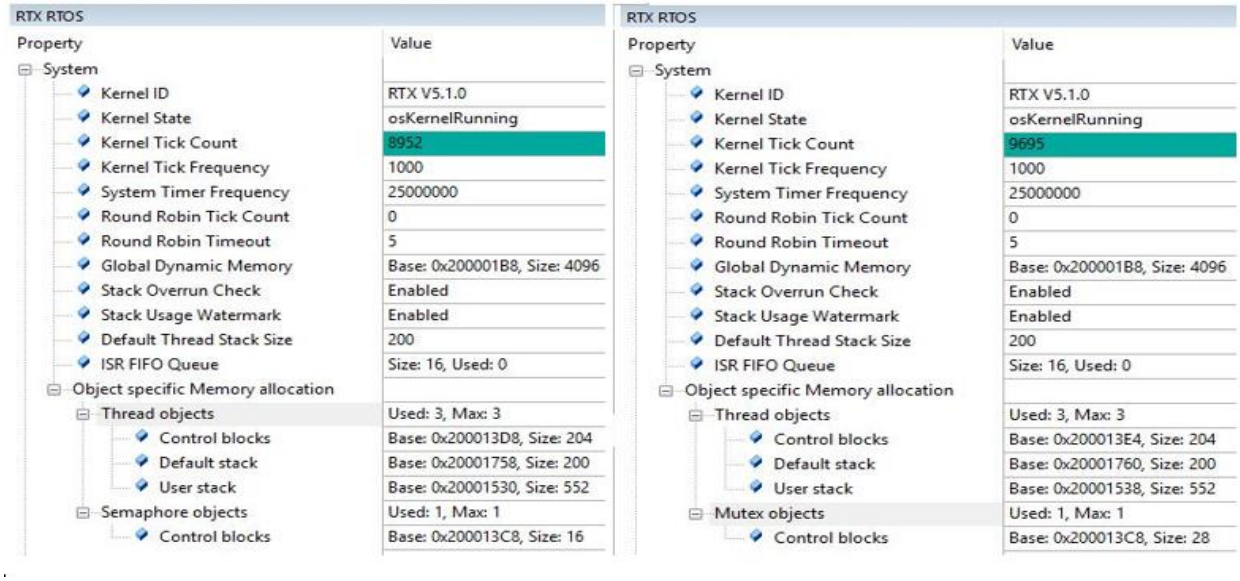

Figure 28 Kernel Information of semaphore and mutex

3.3.4 Performance Analysis

The operating systems executes the program in certain time which is dependent on kernel. The main defines with thread having different execution time based on the priority and task on it. The thread managements with semaphore and mutex of both MBED OS with each thread timings are tabulated in table 5. From the table, we conclude that the running thread takes little bit time comparing with other threads. The total execution time is obtained by adding all thread timings.

Table 5 Thread Execution timings

Functions	Threads	Execution time on MBED OS
Semaphore	**Thread 1**	1.098 sec
	Thread 2	2.083 us
	Thread 3	26.858 ms
Total execution time (in sec)		1.124 Sec
Mutex	**Thread 1**	1.012 us
	Thread 2	2.083 us
	Thread 3	29.087 ms
Total execution time (in sec)		1.041 sec

3.3.5 Trustzone Results

Based on the MBED OS memory consumption TrustZone will be implemented on Cortex M33. It can be noted that memory consumed by different registers on MBED operating system is simultaneously secured using TrustZone system. Figure 29 defines secure and non-secure functions of the thread which is pointing different regions of memory. The workflow of Trustzone in this method has five different functions with callable function. It gives integer data type output values. The program is structured with three threads each performing specific operations.

Function - 1: Non-secure callable function

Function - 2: Non-secure callable function calling a non-secure callback function [Function - 6]

Function - 3: Non-secure callable function

Function - 4: Non-secure callable function calling a non-secure callback function [Function - 6]

Function - 5: Non-secure callable function calling a non-secure callback function [Function - 6]

The three different threads contain different functions. They are,

Thread A – uses value 1 and 2 [Val1_ThreadA and Val2_Thread2]

Thread B – uses value 3 and 4 [Val3_ThreadB and Val4_ThreadB]

Thread C – uses value 5 [Val5_ThreadC].

Workflow of Trustzone describes, initially the thread is in secure mode when the breakpoint is placed at specific value, it changes to non-secure mode. It refers that the non-secure start and to perform the operation needed for the specific value which is defined under the thread. For example, if the breakpoint is placed in val1_ThreadA the system that switches to non-secure to perform the value. Thus, the ThreadA is in running mode and remaining threads are in ready mode.

Both MBED OS supporting on ARM v8 M architecture. But CMSIS RTOS v2 gives standard interface for RTOS running on M33. Thus, MBED OS is the better solution for the Trustzone. The RTX 5 runs in non-secure state, but the call functions from the secure state. Memory management of Trustzone of secure and non-secure is shown in table 6.

Table 6 Memory Management in Trustzone

MEMORY MANAGEMENT		
	SECURE	**NON-SECURE**
Code Memory	2432 Bytes	5812 Bytes
Read Only Data Memory	588 Bytes	968 Bytes
Read Write Data Memory	8 Bytes	224 Bytes
Zero Initialized Data Memory	6336 Bytes	11144 Bytes

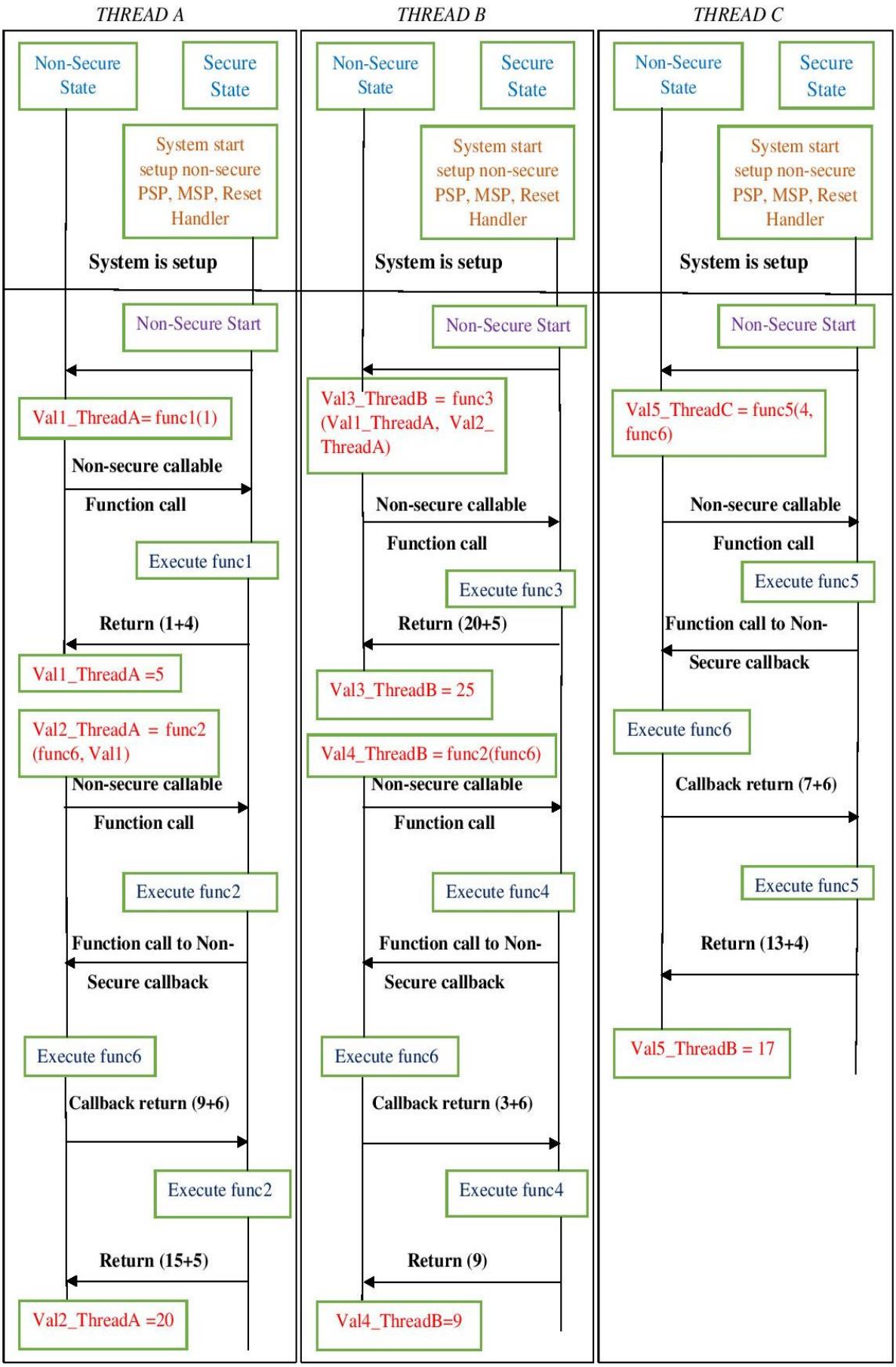

Figure 29 Trustzone Secure and Non-Secure Functions

The corresponding values of each thread is shown in figure 30

Watch 1

Name	Value	Type
val1_ThreadA	5	uint
val2_ThreadA	20	int
val3_ThreadB	25	int
val4_ThreadB	9	int
val5_ThreadC	17	int

Figure 30 Thread Values

Switching between Secure and Non-Secure Modes

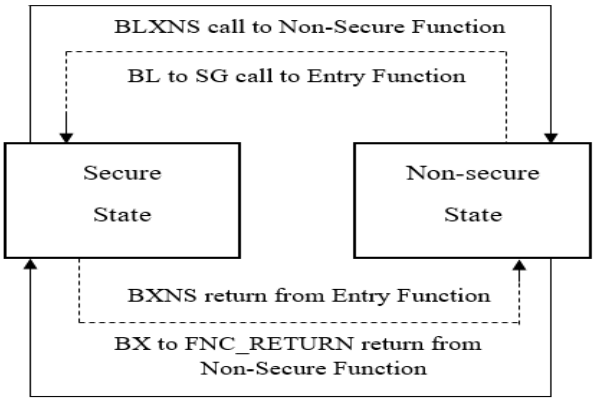

Figure 31 Security Transition States

The processor that handles state transition exchange from Secure Gateway to Non-secure state BXNS and the branch with link exchange to BLXNS. Secure gateway (SG) is used for switching from Non-secure to Secure state at the first instruction of Secure entry point. Branch with exchange to Non-secure state (BXNS) is used by Secure software to branch or return to Non-secure program. Branch with link and exchange to Non-secure state (BLXNS) is used by Secure software to call Non-secure functions. A direct API function call from Non-secure to Secure software entry points is allowed if the first instruction of the entry point is SG, and it is in a Non-secure callable memory location [17]. The corresponding registers with address in secure and non-secure mode with stack where the address in both secure and non-secure modes that protects different regions in system memory. It shows that stack point process and limit function with control registers in the states. The workflow of Trustzone with breakpoints values and corresponding address and stack size are tabulated in Appendix. The table 7 shows how the functions that initially change from non-secure to secure state.

Table 7 Registers Address Point Corresponding Regions

Non-secure start Registers		**Address From**	**Address To**	**Size**	**Memory Region**	**Security**
R1	*0x00000028*	0x00000000	0x0DFFFFFF	224MB	Code Memory	NS
R8 - R11	*0x00000000*					
R12	*0x002017E5*					
R14(LR)	*0x002006C5*					
R15(PC)	*0x002006C4*					
R13(SP)	*0x28200358*	0x28000000	0x2FFFFFFF	128MB	Expansion 0	NS
Secure Registers		**Address From**	**Address To**	**Size**	**Memory Region**	**Security**
R13(SP)	*0x380001C8*	0x38000000	0x30FFFFFF	128MB	Expansion 0	S
R14(LR)	*0x002006CA*	0x10000000	0x1DFFFFFF	224MB	Code Memory	S
R15(PC)	*0x10000B44*					
Non-Secure Registers		**Address From**	**Address To**	**Size**	**Memory Region**	**Security**
R13(SP)	*0x28200358*	0x28000000	0x2FFFFFFF	128MB	Expansion 0	NS
R15(PC)	*0x002006CA*	0x00000000	0x0DFFFFFF	224MB	Code Memory	NS

Detailed Description state of transition

When non-secure program calls secure state, that API completes by returning to NS state using BXNS instruction. The program in NS attempts to call a secure program address without using entry point otherwise it generates fault event. The architecture contains hard fault to handles fault events in secure state. The security extension in an architecture that allow a secure program to call non-secure state. Therefore, BLXNS instruction is used to call non-secure program.

The return address and some processor are pushed onto secure stack, while return address on LR with a special value known as FNC_RETURN, the address should be zero. The non-secure function performing branch address which automatically trigger unstacking of return address from secure stack to call function. Secure state selects to transfer some register values to non-secure and clear other data from register banks before function call.

a. Non-secure Start

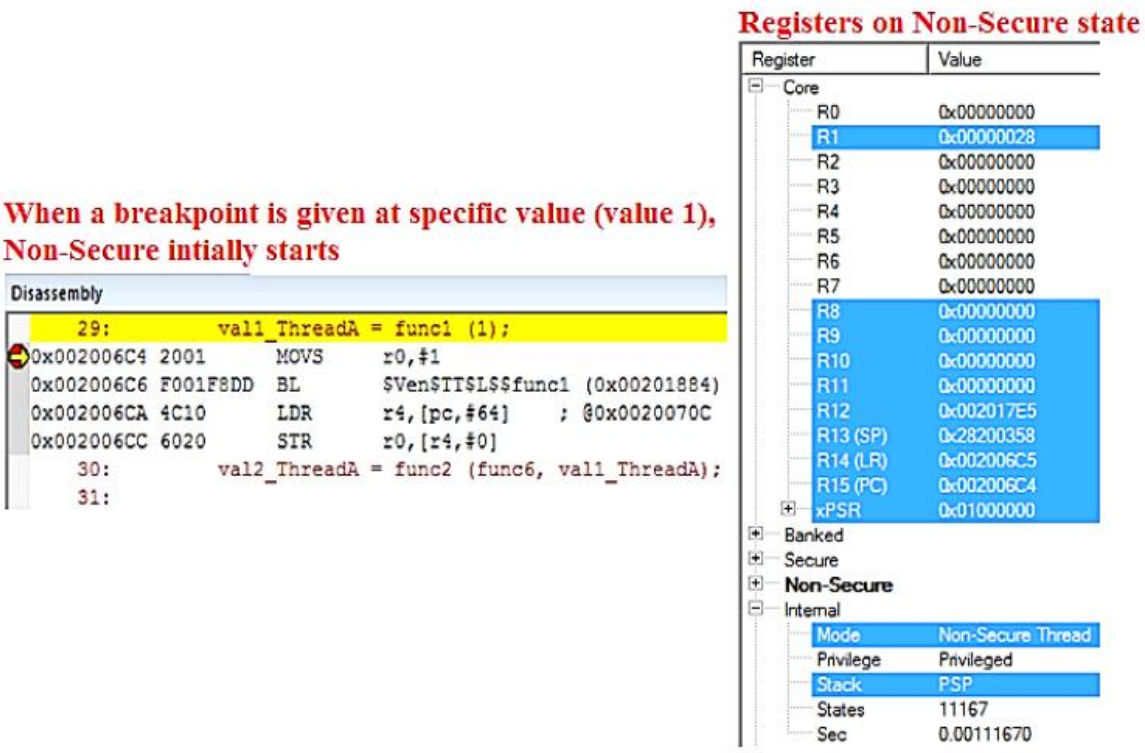

Figure 32 Start of Non-secure mode

b. Switching from non-secure to secure state

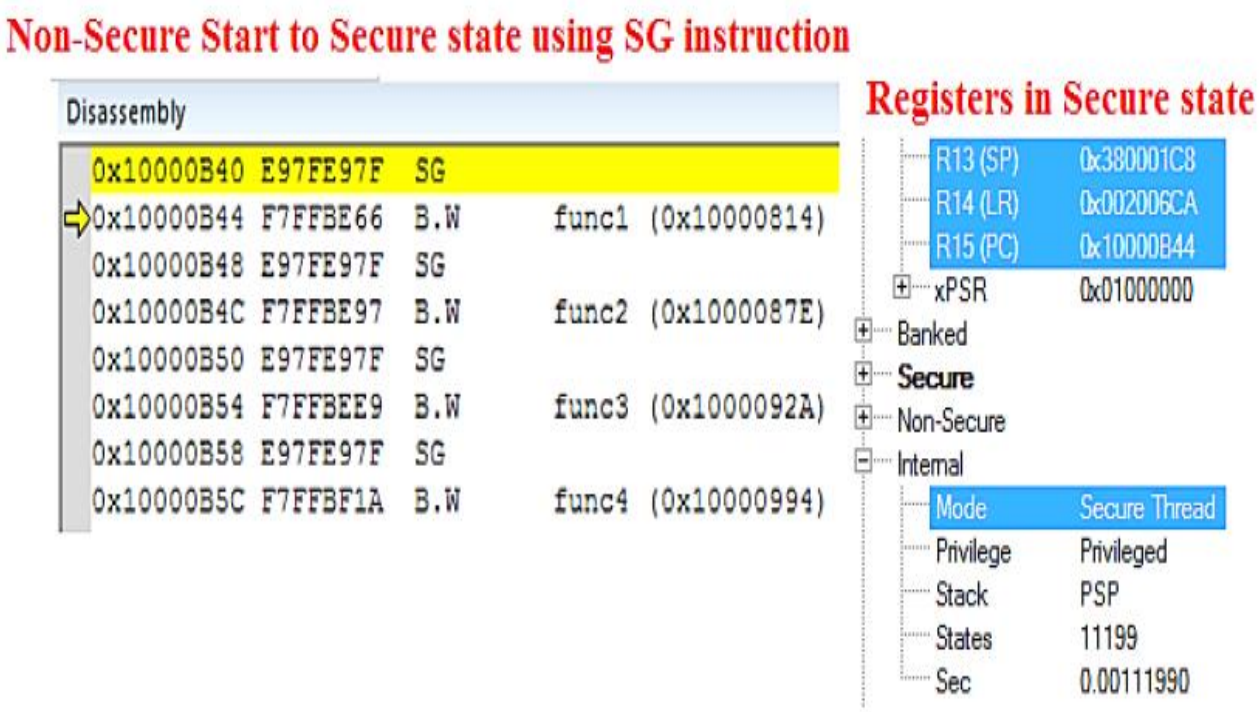

Figure 33 Non-secure to secure state

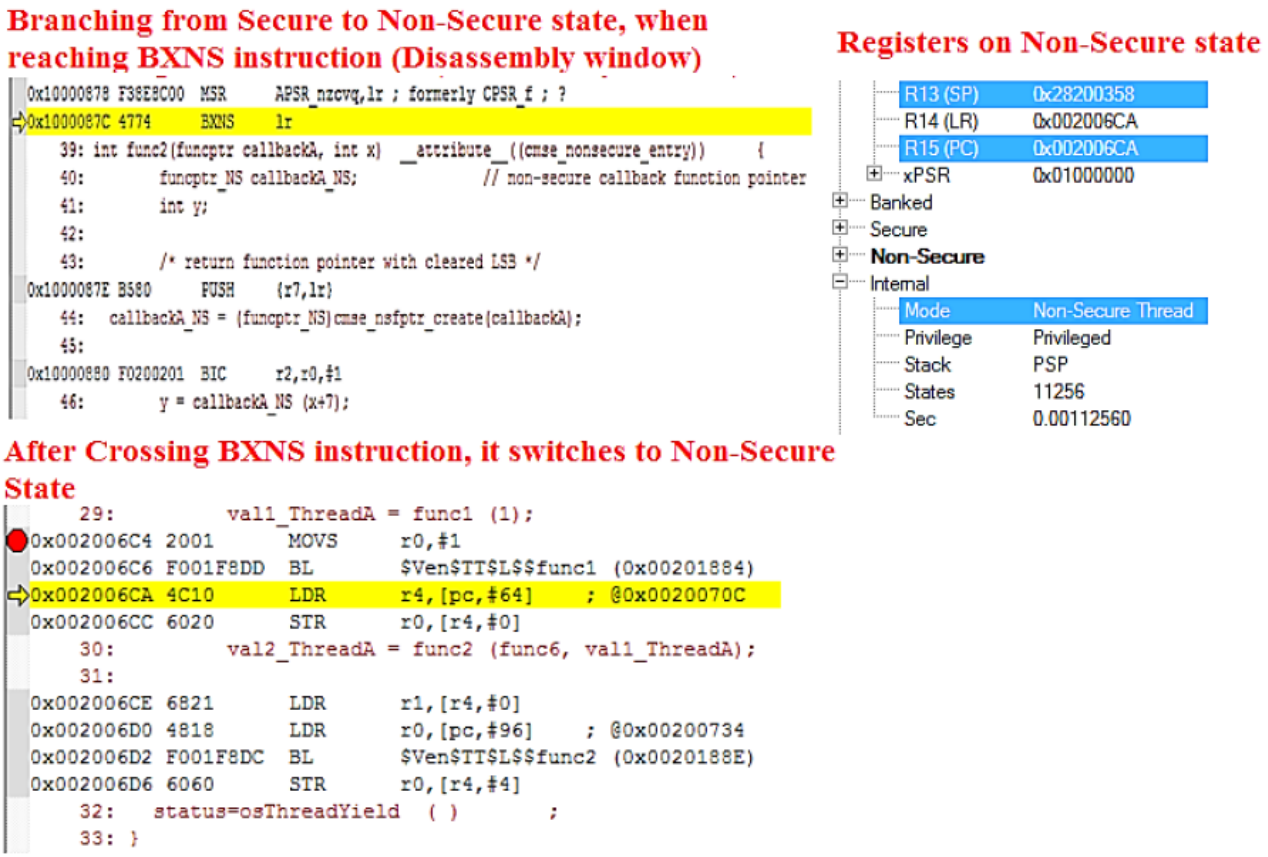

Figure 34 Secure to non-secure state

Table 7 describes the transition state with corresponding addresses. Initially, when the breakpoint set to the value. It changes the state from secure to non-secure start, that was shown in above figures. thus, the values in non-secure call secure state to execute the function. It is a cyclic process that changes from secure to non-secure, some functions are defined with callable function which is taken from non-secure itself. The code memory and expansion 0 memory address change from non-secure to secure vice versa. The details that contain in secure mode is always protected. But in non-secure it needs call function to protect the data. Trustzone that protect whole Core Link SSE 200 system.

CONCLUSIONS AND SUGGESTIONS

From the results and discussions, I conclude the project “Investigation of MBED operating system with TrustZone as security solutions for a cellular IoT” by the followings.

1. Operating system plays a major role in Cortex M processors. The Core Link SSE 200 IoT subsystem contains two ARM Cortex M33 processors. Based on the operating system the device should perform well. On the other side, the operating system needs security purpose to secure the memory which contains all the data and registers values. Thus, the processor needs better operating system with security solution. In section 1.4 it is clearly explained about the reason of choosing MBED OS, it also explains MBED OS with TrustZone security in the next section 1.5. Depending on the functionalities of CMSIS RTOS v2 the processors were tested using MBED OS in both Cortex M33 and Cortex M4 processors. Based on the performance, memory consumption and stack values of both processors, it clearly shows that Cortex M33 has better solution than Cortex M4 which are proven in section 3.3.
2. Memory consumption of MBED operating system is done with thread management using semaphore and mutex functions. Initially, testing was carried out with and with single thread and thread using one semaphore and one mutex. Thus, the kernel is performed better solution for memory management. The memory consumption of initial testing values is acceptable which are already predefined in kernel functions. Finally, thread management is increased based on the user defined application. In this project, final testing is done with 3 threads, 1 semaphores and 1 mutex function. Even the number of threads can be increased by user, the cellular IoT has more operations and functions. So, the number of threads and thread managements are increased. Consumption of memory is also increased due to the larger number of thread.
3. On the security side, Trustzone parameters are tested on Cortex M33 processor. Cortex M33 is only having security layer, whereas it is not in Cortex M4. The main advantage of Trustzone is that it protects AHB5 on Core Link SSE 200 IoT subsystem. It has two modes secure and non-secure. Both modes are having same handler and ISR function. From the non-secure point of view, the registers are protected within secure state via callable function. The memory based calculation on MBED OS are used to protect the register function in the property of TrustZone. It will explain the secure and non-secure function of register with corresponding memory and address. It clearly shows that how the stack limit is consumed and stack pointer in the registers which are mentioned in the section 3.3.5.

4. Registers in ARM v8 M architecture are protected by MBED OS based Trustzone function. While RTX 5 supports Trustzone without any fails. But in other RTOS it needs some internal functions to call secure state. Therefore, Trustzone using MBED OS on Cortex M33 gives best solution for cellular IoT.

Overall, the conclusion of embedded operating system with security solution can be done using MBED operating system with Trustzone security on Cortex M33. Therefore, MBED OS supports for Core Link SSE 200 IoT subsystem.

In future scope ARM CORDIO Radio Core IP is to be integrated with Core Link SSE 200 subsystem that provides more services and productivity towards the cellular IoT which is referred as Cadelsi NB-IoT.

INFORMATION SOURCE LIST

[1] LTE evolution for IoT connectivity, Nokia, 2017, pp. 1-18.

[2] “Cellular IoT,” [Online]. Available: https://iot-for-all.com/cellular-iot-explained-nb-iot-vs-lte-m/.

[3] “FreeRTOS,” [Online]. Available: http://www.freertos.org/about-RTOS.html.

[4] Hahm, Emmanuel Baccelli and Oliver; W¨ahlisch, Mesut G¨unes and Matthias; Schmidt, Thomas C., *RIOT OS: Towards an OS for the Internet of Things*, pp. 1-2, 2013.

[5] Arijit, Ukil; Jaydip, Sen; Sripad, Koilankonda, *Embedded security for Internet of Things*, 2011.

[6] Roger Piqueras Jover, *Security and impact of the IoT on LTE mobile networks*, pp. 1-25, 2015.

[7] Mihail Stoyanov,, “Introduction to mbed OS,” ARM, china, 2016.

[8] Sam Grove, “mbed OS technical overview,” ARM, Las Vegas, 2016.

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

[10] “CMSIS RTOS API,” [Online]. Available: https://www.keil.com/pack/doc/CMSIS/RTOS2/html/genRTOS2IF.html.

[11] “CMSIS Thread,” [Online]. Available: https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__ThreadMgmt.html.

[12] “CMSIS Semphore,” [Online]. Available: https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__SemaphoreMgmt.html.

[13] “CMSIS Mutex,” [Online]. Available: https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__MutexMgmt.html.

[14] Using TrustZone on ARMv8, ARM Keil, 2016.

[15] TrustZone® technology for ARM®v8-M Architecture Version 1.1, ARM, 2016.

[16] ARM® Cortex®-M23 Processor Technical Reference Manual, ARM, 2016.

[17] TrustZone®technology for ARM®v8-M Architecture Version 1.0, ARM, 2016.

[18] “Embedded Know How,” [Online]. Available: http://www.embedded-know-how.com/article/2425/iot-automation-platform-based-on-arm-cortex-m-and-freertos.

[19] “Development Board,” [Online]. Available: https://developer.arm.com/products/system-design/development-boards/cortex-m-prototyping-system.

[20] “CMSIS Message Queue,” [Online]. Available: https://www.keil.com/pack/doc/CMSIS/RTOS2/html/group__CMSIS__RTOS__Message.html.

[21] “RTX memory requirements,” [Online]. Available: http://www.keil.com/rl-arm/rtx_size.asp.

[22] Ö. Fredrik, *A Sensor Network Simulator for the Contiki OS*, pp. 1-40, 2006.

[23] K. Mahdi Amiri and B. Hadj , *A Survey On Embedded Open Source System Software For The Internet Of Things*, pp. 27-32, 2017.

[24] M. Nicolas, “Study of an operating system: FreeRTOS”.*Operating systems for embedded devices* .

[25] 3GPP Low Power wide Area Technologies, Svetlana Gran (GSMA), 2016.

[26] Cortex-M33 processor ARMv8-M IoT Kit FVP User Guide Version 2.0, ARM, 2017.

[27] L. Miller, M2M/IoT Cellular Data Security, TELIT, 2015.

[28] “ARM TrustZone,” [Online]. Available: https://www.arm.com/products/security-on-arm/trustzone.

[29] Diaa Jadaan, “Memory management and error handling in FreeRTOS for a CubeSat project”.

[30] Rich Goyette, *An Analysis and Description of the Inner Workings of the FreeRTOS Kernel*, pp. 1-46, 2011.

[31] Sagar P M, “Embedded Operating Systems for Real,” 2002.

[32] ARM, “TrustZone,” [Online]. Available: https://developer.arm.com/technologies/trustzone.

[33] Johannes Winter, “Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms,” pp. 1-10.

[34] Ferdinand Brasser; Daeyoung Kim; Christopher Liebchen; Vinod Ganapathy; Liviu Iftode; Ahmad-Reza Sadeghi, “Regulating Smart Personal Devices in Restricted Spaces,” pp. 1-14, 2015.

[35] T. Sandhu, “ARM unveils new IoT platform,” *Hexus*, 25 October 2016.

APPENDIXES

APPENDIX 1 Testing Results

Thread Analysis on Cortex M4

MBED OS without Thread:

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

MBED OS with one Thread:

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

MBED OS with 1 Thread & 1 Semaphore:

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

MBED OS with 1 Thread & 1 Mutex:

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

Thread Analysis on Cortex M33

MBED OS without Thread:

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

MBED OS with one Thread:

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

MBED OS with 1 Thread & 1 Semaphore:

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

MBED OS with 1 Thread & 1 Mutex:

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

Experimental Results on Cortex M4

MBED OS with 3 Thread & 1 Semaphore:

Code	6536 Bytes
RO Data	632 Bytes
RW Data	248 Bytes
ZI-Data	10424 Bytes

MBED OS with 3 Thread & 1 Mutex:

Code	6480 Bytes
RO Data	136 Bytes
RW Data	248 Bytes
ZI-Data	10432 Bytes

Experimental Results on Cortex M33

MBED OS with 3 Thread & 1 Semaphore:

Code	6876 Bytes
RO Data	996 Bytes
RW Data	248 Bytes
ZI-Data	10424 Bytes

MBED OS with 3 Thread & 1 Mutex:

Code	6820 Bytes
RO Data	1000 Bytes
RW Data	248 Bytes
ZI-Data	10432 Bytes

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

Initial Condition	Semaphore_1	Max_Count_Token - 3	Initial_Count_Token - 2

Parameters	***Thread_1 (Use Semaphore_1)***	***Thread_2 (Use Semaphore_2)***	***Thread_3 (Use Semaphore_2)***
State	osThreadRunning	osThreadReady	osThreadBlocked
Priority	Low	Idle	High
Stack Top (address)	0x38001640	0x380016A8	0x38001820
Stack Limit (address)	0x38001540	0x38001648	0x38001758
Stack Available	*Stack (used 15% Max 28%)*	*Stack (used 75% Max 75%)*	*Stack (used 36% Max 36%)*
	Used:40	Used:72	Used:72
	Max:72	Max:72	Max:72
Stack Size	256 Bytes	96 Bytes	200 Bytes

Running Condition	Semaphore_1	Max_Count_Token - 3	Initial_Count_Token - 3

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

Parameters	***Thread_1 (Use Mutex_1)***	***Thread_2 (Use Mutex_2)***	***Thread_3 (Use Mutex_2)***
State	osThreadRunning	osThreadReady	osThreadBlocked
Priority	Low	Idle	High
Stack Top (address)	0x38001648	0x380016B0	0x38001828
Stack Limit (address)	0x38001548	0x38001650	0x38001760
Stack Available	*Stack (used 4% Max 28%)*	*Stack (used 75% Max 75%)*	*Stack (used 36% Max 36%)*
	Used:10	Used:72	Used:72
	Max:72	Max:72	Max:72
Stack Size	256 Bytes	96 Bytes	200 Bytes

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

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

Initial Condition	Semaphore_1	Max_Count_Token - 3	Initial_Count_Token - 2

Parameters	***Thread_1 (Use Semaphore_1)***	***Thread_2 (Use Semaphore_2)***	***Thread_3 (Use Semaphore_2)***
State	osThreadRunning	osThreadReady	osThreadBlocked
Priority	Low	Idle	High
Stack Top (address)	0x20001640	0x200016A8	0x20001820
Stack Limit (address)	0x20001540	0x20001648	0x20001758
Stack Available	*Stack (used 15% Max 28%)*	*Stack (used 75% Max 75%)*	*Stack (used 36% Max 36%)*
	Used:40	Used:72	Used:72
	Max:72	Max:72	Max:72
Stack Size	256 Bytes	96 Bytes	200 Bytes

Running Condition	Semaphore_1	Max_Count_Token - 3	Initial_Count_Token - 3

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

Parameters	***Thread_1 (Use Mutex_1)***	***Thread_2 (Use Mutex_2)***	***Thread_3 (Use Mutex_2)***
State	osThreadRunning	osThreadReady	osThreadBlocked
Priority	Low	Idle	High
Stack Top (address)	0x20001648	0xx200016B0	0x20001828
Stack Limit (address)	0x20001548	0x20001650	0x20001760
Stack Available	*Stack (used 4% Max 28%)*	*Stack (used 75% Max 75%)*	*Stack (used 36% Max 36%)*
	Used:10	Used:72	Used:72
	Max:72	Max:72	Max:72
Stack Size	256 Bytes	96 Bytes	200 Bytes

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

Trustzone

Table 1: Thread Management (Break point at Value1 / Value2)

	Thread_A	***Thread_B***	***Thread_C***
State	osThreadRunning	osThreadReady	osThreadReady
Priority	High	Normal	Above Normal
	Stack (used 1% Max 12%)	*Stack (used 50% Max 50%)*	*Stack (used 32% Max 32%)*
Used	Used:8 Bytes	Used:64 Bytes	Used:64 Bytes
Max	Max:64 Bytes	Max:64 Bytes	Max:64 Bytes
Top	0x28200360	0x282003E0	0x28201AD0
Limit	0x28200160	0x28200360	0x28201A08
Size	512 Bytes	128 Bytes	200 Bytes

Table 2: Thread Management (Break point at Value3 / Value4)

	Thread_A
State	osThreadRunning
Priority	Low
	Stack (used 6% Max 56%)
Used	Used:8
Max	Max:72
Top	0x282003E0
Limit	0x28200360
Size	128 Bytes

Table 3: Thread Management (Break point at Value5)

	Thread_B	***Thread_C***
State	osThreadReady	osThreadRunning
Priority	Normal	Above Normal
	Stack (used 56% Max 56%)	*Stack (used 4% Max 32%)*
Used	Used:64 Bytes	Used:8 Bytes
Max	Max:64 Bytes	Max:64 Bytes
Top	0x282003E0	0x28201AD0
Limit	0x28200360	0x28201A08
Size	128 Bytes	200 Bytes

Note: Stack Configuration (in bytes) - 0x0000 0400

Heap Configuration (in bytes) - 0x0000 0C00

APPENDIX 2 Main Programs

Semaphore

```
#include "cmsis_os2.h"
#include “mbed.h”                    // mbed

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

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
}
}

/* ---------- THREAD_1 STRUCTURE---------- */

const osThreadAttr_t Thread_Semaphore1_attr = {
"Thread_Sem1",
.stack_size = 256
};

/* ----------- 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, osPriorityBelowNormal); // 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
}
}


/* -------------- THREAD_2 STRUCTURE----------------- */
const osThreadAttr_t Thread_Semaphore2_attr = {
"Thread_Sem2",
.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
}
}
/* -------------- THREAD_3 STRUCTURE----------------- */
const osThreadAttr_t Thread_Semaphore3_attr = {
"Thread_Sem3",
.stack_size = 200
};
/* ---------- 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);
}
if (osKernelGetState() == osKernelReady) {
status=osKernelStart();                 // Start thread execution
}
for (;;) {}
}
```

Mutex

```
#include "cmsis_os2.h"
#include “mbed.h”                      // mbed

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
osMutexId_t mutex3_id;                        // Mutex id
```

```
osStatus_t status;


/* ----------- THREAD_FUNCTION_1 ---------------- */
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
  }
}

/* -------------- THREAD_1 STRUCTURE----------------- */

const osThreadAttr_t Thread_Mutex1_attr = {
 "Thread_Mutex1",
          .stack_size = 256
          };
/* ----------- 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
}
```

```
}

/* -------------- THREAD_2 STRUCTURE----------------- */

const osThreadAttr_t Thread_Mutex2_attr = {
 "Thread_Mutex2",
          .stack_size = 96
          };

/* ----------- Mutex-2 Structure --------------- */
const osMutexAttr_t Thread_Mutex2a_attr = {
  "myThreadMutex2",                     // human readable mutex name
  osMutexRecursive | 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
}
}

/* -------------- THREAD_3 STRUCTURE----------------- */

const osThreadAttr_t Thread_Mutex3_attr = {
 "Thread_Mutex3",
          .stack_size = 200
          };

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
}
mutex3_id = osMutexNew(&Thread_Mutex3a_attr);
if (mutex3_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);
}
if (osKernelGetState() == osKernelReady) {
status=osKernelStart();                    // Start thread execution
}
for (;;) {}
}
```

Trustzone
Secure Program

```
#include <arm_cmse.h>

#include "RTE_Components.h"
#include CMSIS_device_header

/* TZ_START_NS: Start address of non-secure application */
#ifndef TZ_START_NS
#define TZ_START_NS (0x200000U)
#endif

/* typedef for non-secure callback functions */
typedef void (*funcptr_void) (void) __attribute__((cmse_nonsecure_call));

/* Secure main() */
int main(void) {
  funcptr_void NonSecure_ResetHandler;
```

```
  /* Add user setup code for secure part here*/

  /* Set non-secure main stack (MSP_NS) */
  __TZ_set_MSP_NS(*((uint32_t *)(TZ_START_NS)));

  /* Get non-secure reset handler */
  NonSecure_ResetHandler = (funcptr_void)(*((uint32_t *)((TZ_START_NS) + 4U)));

  /* Start non-secure state software application */
  NonSecure_ResetHandler();

  /* Non-secure software does not return, this code is not executed */
  while (1) {
    __NOP();
  }
}
```

Non-secure Program

```
#include "cmsis_os2.h"
#include "mbed.h"
#include "interface.h"
#include <stdio.h>
extern volatile int val1, val2, val3, val4, val5;
volatile int val1, val2, val3, val4, val5;

void ThreadA (void *argument);
static osThreadId_t tid_ThreadA;

void ThreadB (void *argument);
static osThreadId_t tid_ThreadB;

void ThreadC (void *argument);
static osThreadId_t tid_ThreadC;

static osStatus_t status;

int func6(int x);

int func6(int x) {
        return (x+6);
}

void ThreadA (void *argument){

        tid_ThreadA = osThreadGetId();
        status = osThreadSetPriority(tid_ThreadA, osPriorityHigh);

        val1 = func1 (1);
        val2 = func2 (func6, val1);

  status=osThreadYield        ( )       ;
}
```

```
void ThreadB (void *argument){

        tid_ThreadB = osThreadGetId();
        status = osThreadSetPriority(tid_ThreadB, osPriorityLow);
        val3 = func3 (val1, val2);
        val4 = func4 (func6);
        status=osThreadYield ( )        ;
}

void ThreadC (void *argument){

        tid_ThreadC = osThreadGetId();
        status = osThreadSetPriority(tid_ThreadC, osPriorityAboveNormal);
        val5 = func5 (4, func6);
        status=osThreadYield ( )        ;
}

static uint64_t Thread_1_stk_1[64];

static const osThreadAttr_t Thread_1_attr = {
   .tz_module = 1U,
  .stack_mem  = &Thread_1_stk_1[0],
  .stack_size = sizeof(Thread_1_stk_1)
};

static uint32_t Thread_2_stk_1[32];

static const osThreadAttr_t Thread_2_attr = {
   .tz_module = 1U,
  .stack_mem  = &Thread_2_stk_1[0],
  .stack_size = sizeof(Thread_2_stk_1)
};



static const osThreadAttr_t Thread_3_attr = {
   .tz_module = 1U,
   .stack_size = 0
};


int main(void) {

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

        tid_ThreadA = osThreadNew(ThreadA, NULL, &Thread_1_attr);

  tid_ThreadB = osThreadNew(ThreadB, NULL, &Thread_2_attr);
        tid_ThreadC = osThreadNew(ThreadC, NULL, &Thread_3_attr);
```

```
if (osKernelGetState() == osKernelReady) {
status=osKernelStart();
}
for (;;) {}
}
```

Interface.c

```
#include <arm_cmse.h>     // CMSE definitions
#include "interface.h"    // Header file with secure interface API

/* typedef for non-secure callback functions */
typedef funcptr funcptr_NS __attribute__((cmse_nonsecure_call));

/* Non-secure callable (entry) function */
int func1(int x) __attribute__((cmse_nonsecure_entry)) {
  return x+4;
}

/* Non-secure callable (entry) function, calling a non-secure callback function */
int func2(funcptr callbackA, int x)  __attribute__((cmse_nonsecure_entry))          {
          funcptr_NS callbackA_NS;              // non-secure callback function pointer
          int y;

          /* return function pointer with cleared LSB */
  callbackA_NS = (funcptr_NS)cmse_nsfptr_create(callbackA);

          y = callbackA_NS (x+7);

          return (y+2);
}
/* Non-secure callable (entry) function */
int func3(int x, int y) __attribute__((cmse_nonsecure_entry)) {
          return (x+y);
}

/* Non-secure callable (entry) function, calling a non-secure callback function */
int func4(funcptr callbackB) __attribute__((cmse_nonsecure_entry)) {
          funcptr_NS callbackB_NS;                                                    //       non-
secure callback function pointer
          int y;


          /* return function pointer with cleared LSB */
  callbackB_NS = (funcptr_NS)cmse_nsfptr_create(callbackB);

          y = callbackB_NS(3);

          return y;
}
```

```
/* Non-secure callable (entry) function, calling a non-secure callback function */
int func5(int x,funcptr callbackC) __attribute__((cmse_nonsecure_entry)) {
        funcptr_NS callbackC_NS;
        int y;

        /* return function pointer with cleared LSB */
  callbackC_NS = (funcptr_NS)cmse_nsfptr_create(callbackC);

         y = callbackC_NS (x+5);

        return (2+y);
}
```